A. A. Prihozhy

ANALYSIS, TRANSFORMATION AND OPTIMIZATION FOR HIGH PERFORMANCE PARALLEL COMPUTING

Minsk
BNTU
2019

UDC 004.272.2

Prihozhy, A. A. Analysis, transformation and optimization for high perfomance parallel computing / A. A. Prihozhy. – Minsk: BNTU, 2019. – 229 p. – ISBN 978-985-583-366-7.

This book studies hardware and software specifications at algorithmic level from the point of measuring and extracting the potential parallelism hidden in them. It investigates the possibilities of using this parallelism for the synthesis and optimization of high-performance software and hardware implementations. The basic single-block flow model is a result of the algorithm transformation, and is a basis for developing efficient methods of synthesis and optimization of parallel implementations. It supports the generation and optimization of computational pipelines and concurrent net algorithms, which lead to higher performances of the computing systems.

This book is useful in training of scientific researchers and graduate students. It is also useful for teaching students and undergraduates in computer science at universities.

Tables 34. Figures 117. Bibliography 91.

Recommended for publication by the Scientific
and Technical Council of Belarusian national technical university
(Record № 1, 18.01.2019)

Reviewers:
Head of the Department of information management systems,
Belarusian national university, Dr. Prof. *V. V. Krasnoproshin*;
Professor of the Info-communication technologies department,
Belarusian university of informatics and radio-electronics,
Dr. Prof. *A. A. Boriskevich*

ISBN 978-985-583-366-7

CONTENT

PREFACE

An algorithmic description of a solution of an applied, scientific or technical problem is given, for which it is necessary to obtain a high-performance software implementation on a multiprocessor or multi-core system, or a hardware implementation on an FPGA or ASIC. How to perform the mapping of the source algorithm onto an efficient parallel implementation? How to discover, measure, extract and optimally implement the hidden parallelism is the main objective of this book. This book develops a technology for analyzing, transforming, optimizing and mapping hard-to-parallel algorithms and programs to pipeline and network implementations. The algorithm analysis is based on the profiling of the program in order to measure the computational complexity, the critical path and the potential parallelism on realistic input data. The transformation performs parallelism extraction from the program, preserving the original functionality. The synthesis and optimization improves the structure and parameters of computational pipelines and network computing schedules.

This book will be useful for scientific researchers, engineers, PhD students and undergraduates. It is mostly based on author’s publications written during more than 25 years period and presents state of the art in scientific direction under consideration. The author’s works have been published by such well known publishers as IEEE, Kluwer Academic Publishers, Springer and many others.

My gratitude is large to my partners and friends Dr. Jean Mermet and Dr. Bernard Courtois (France), as well as to Dr. Marco Mattavelli, Dr. Daniel Mlynek, Dr. Alain Vachoux, Dr. Masimo Ravasi, Dr. Ab Al Hadi Bin Ab Rahman, Dr. Simone Casale-Brunet and Dr. Endri Bezati (Switzerland) for collaboration, joint work and joint publications.

The author acknowledges the colleagues of computer and system software department, the colleagues of the faculty of information technologies and robotics, and the scientific research sector of the Belarusian National Technical University, who have helped in preparing the manuscript and in getting approval to publish this book.

Anatoly Prihozhy
Minsk, December, 2018
prihozhy@yahoo.com

INTRODUCTION

This book has a coherent logical thread, revealing the topic stated in the title. It outlines the models and methods of analysis, transformation and optimization of algorithms and programs for effective high-performance parallel execution on multiprocessor systems.

Firstly, we explore the metrics of algorithms and programs, providing the estimation of the hidden, but potentially retrievable and implementable dataflow parallelism. We consider three metrics, namely, computational complexity, critical path and the parallelization factor on the graph of program execution. This graph is formed dynamically during the execution of the program code on specific source data. These data reflect the most realistic conditions for the application of the algorithm, and not the conditions of the “worst case”, which is crucial for an objective and reliable assessment. In order to measure the values of metrics, we develop a model and method for instrumenting and extending the program code of an algorithm, create appropriate tools, and perform measurements on a number of important algorithms for processing video and audio information, as well as on cryptographic algorithms.

After the potential parallelism has been measured, and the expediency of paralleling the algorithm is justified, the program code transformation stage begins in order to extract the data flow. This book describes a set of rules for the transformation of various kinds of statements, control structures of the programming language and super positions of them, to the basic single-block flow model that is built on a single loop. One part of the control instructions is deleted, the other part is split. As a result, the data flow becomes extremely dominant in the presentation of algorithm. The proposed method provides the extraction of a dataflow from difficultly parallelizable algorithms.

The basic single-block flow model makes operators less dependent on one another and freer with respect to the permutation. However, it makes it difficult to analyze the transformed code with respect to identifying mutually exclusive branches, compared to the source code. The analysis method proposed in the book uses the theory of Boolean functions and formal logic. The search for mutually exclusive operators that are under conditional if-then instructions is equivalent to checking the orthogonality of Boolean conditional variables, which in turn is equivalent to checking the tautology of Boolean expressions. Analysis of algorithms with feedback in the control flow due to presence of loop statements is performed by the method of mathematical induction.

The basic single-block flow model of an algorithm is an efficient source of synthesis and optimization of computational pipelines. It is a source of generating a series of relations and graphs on sets of operators and variables. The graph of operator conflicts that arise when operators are assigned to pipeline stages, allows us to solve the problem of minimizing the number of stages for a given constraint on the operation time of one pipeline stage. Another important parameter is the size of the buffers used to push data through the pipeline. The total size of the buffers should be minimized. Two algorithms optimize the pipeline: accurate and heuristic. The exact algorithm is capable of finding the global optimum for a small-size pipeline. The heuristic algorithm is capable of finding a near-optimal solution for a large-size pipeline. The developed software has allowed to synthesize and optimize pipelines used in practically significant applied areas. Experiments performed on large algorithmic descriptions taken from real practice and generated by random number generators showed that the proposed heuristic pipeline optimization algorithm yields significantly better results than such algorithms like ASAP and ALAP.

The quality of the optimization results obtained by the heuristic algorithm substantially depends on the composition of the heuristics and the weight of each of them in the integrated heuristic, which is used to select the preferred solution when searching for the optimal parallel implementation of the algorithm. The problem of setting up heuristics for a specific paralleling problem and a specific paralleling algorithm is solved in this book by using a genetic algorithm. We construct chromosomes, a fitness function, a generation and population of chromosomes, selection, and crossover and mutation genetic operations over chromosomes to determine the significance of each heuristic in the paralleling algorithm. The use of the genetic algorithm is illustrated by the example of a heuristic algorithm for optimizing computational pipelines. Experiments have shown that the genetic algorithm can significantly improve the quality of synthesized pipelines.

The basic single-block flow model of the algorithm is a source of synthesis and optimization of network computing schedules and network algorithms. We give a definition of a network schedule, describes a method of estimating the execution time and implementation cost of the schedule over the cliques of the graph of sequential execution of operators and over the cliques of the graph of parallel execution of operators. Further, we formulates and solve the problem of the existence of a network schedule for a given level of parallelism. The schedule is optimized on the minimum of execution time, or on the minimum of consumed computing resources. The book describes an instrumental system that supports a graphs-description language, allows to interactively create network algorithm graph-descriptions, performs the simulation of the graph-descriptions, and optimizes the network schedules.

All models and methods of analysis, transformation and optimization of algorithms for parallel efficient execution are illustrated with a number of examples. The developed software tools are applied to several practically significant hardware and software applications. In particular, they are applied to two-dimensional WAVELET codec, RSAREF cryptographic toolkit, MPEG-4 video codec, Bayer filter, 8×8FDCT, and middle-size and large-size random designs.

1. EVALUATION OF COMPUTATIONAL COMPLEXITY, CRITICAL PATH AND PARALLELIZATION POTENTIAL OF ALGORITHMS

1.1. Metrics of algorithms

This chapter presents metrics for evaluating the computational complexity, critical path and parallelization potential of algorithms that are represented and executed as a computer program. The model metrics aim at the estimation and increase of the upper bound of the algorithm execution speed on a parallel computing platform. They are particularly tailored for application to network, multimedia, cryptographic, scientific and other complex algorithms.

1.1.1. Computational complexity of algorithm

The computational complexity theory [31] classifies computational problems according to their difficulty, and relating the complexity classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by an algorithm, and therefore may be solved by a computer. The theory introduces mathematical models of computation to study the computational problems and quantifying their computational complexity, i.e., the amount of resources needed to solve them, such as time and storage.

The analysis of computational complexity of an algorithm aims at analyzing the amount of resources needed by a particular algorithm. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps the algorithm takes, or the number of storage locations it uses. An algorithm is said to be efficient when this function's values grow slowly compared to a growth in the size of the input. Different inputs of the same length may cause the algorithm to have different behavior. Best, worst and average case trends are often of practical interest. The function describing the performance of an algorithm is usually an upper bound, which is determined from the worst case inputs to the algorithm.

The term *analysis of algorithms* was introduced by Donald Knuth [32-34]. Algorithm analysis provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

The theoretical analysis of algorithms determines the complexity function for arbitrarily large input, and uses big O notation (big-omega notation) and Big-theta notation. Asymptotic estimates are preferable because different implementations of the same algorithm may differ in efficiency. Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation.

Nowadays, processing and compression algorithms, communication protocols and multimedia systems have reached an extremely high level of sophistication. Architectural implementation choices based on designer feeling or intuition without objective measures and verifications become extremely difficult or impossible tasks.

The increasing complexity of the algorithms has lead to the need of specifications and more intensive validations of such system descriptions by means of C/C++ software implementations. These implementations are often huge and virtually impossible to be analyzed and manipulated without the aid of automated tools and appropriate methodologies. In many cases the understanding of the algorithms and the evaluation of their complexity and parallelization potential, are fundamental steps for correct architectural implementation choices.

Another important issue is that the interest in the network, multimedia, cryptographic and other fields is restricted to evaluations and measures under real input conditions, and not through strict worst case analysis that would lead to consider pathological cases far from the interest of real efficient implementation solutions.

It is also desirable to understand and measure the algorithm complexity and the parallelization potential at the highest possible algorithmic level. Such understanding at the very early stage is fundamental in order to be able to take meaningful and efficient partitioning decisions and bring them to actual efficient parallel implementations.

1.1.2. Critical path analysis problem

The problem of identifying one of the longest paths in a circuit or in a program is called a critical path problem [41]. The critical path analysis is an efficient mechanism used at several levels of system design, including circuit-, logic-, architecture-, algorithmic-, and system-levels. At system level, the process of hardware/software partitioning is a complex optimization problem [21]. The final solution depends on a variety of design constraints/goals like performance, power consumption, implementation cost. The critical path analysis can be used for the detection of more efficient hardware/software partitions and their implementation parameters.

At circuit level, the length of critical path plays a key role in setting the clock cycle time and improving the architecture performance [41]. The critical path length is computed as the longest signal propagation delay in the circuit. In [41] the speed up of the critical path computation is achieved by means of parallel processing.

In designing VLSI or systems on chips architectures a complex computational task is represented as a directed task graph. The concept of critical path on the graph is used in [8] for solving the optimal buffer assignment problem by means of formulation of integer linear programming problem and decomposing the graph into a number of sub-graphs.

In high-level synthesis [43], the static data flow graph model is widely used for solving such tasks as scheduling, allocation and binding. During scheduling, the achievable iteration period is limited by the critical path time on the graphs. Some transformations are proposed in [43] on the static graphs in order to reduce the critical path time.

The idea described in [91] is to insert parallelism analysis code into the sequential simulation program. The execution of a discrete event simulation follows causality constraints, and the relationships between the events can be described by an event precedence graph. When the modified sequential program is executed, the time complexity of the parallel simulation is computed.

In [5] the critical path analysis is presented as a method for detailed understanding of when and how delays are introduced in data transfers in the Internet. By constructing and profiling the critical path, it is possible to determine what fraction of the total transfer latency is due to packet propagation, delays at the server and at the client, network variation etc.

In message passing and shared-memory parallel programs [22], communication and synchronization events result in multiple paths through a program's execution. The critical path of the program is simply defined as the longest time-weighted sequence of events from the start of the program to its termination. The parallel computations are described by the program activity graph. The critical path of a parallel program is the longest path through the graph.

The critical path profiling is a metrics explicitly developed for parallel programs [22] and proved to be useful for several optimization goals. The critical path profile is a list of procedures and the time each procedure contributed to the length of the critical path. Critical path profiling is a way to identify the component in a parallel program that limits its performance. It is an effective metric for tuning parallel programs and is especially useful during the early stages of tuning a parallel program when load imbalance is a significant bottleneck. It also helps to find out, which components should be prioritized to terminate the program in time. Where an operation has to be completed on time, critical path analysis helps us to focus on the essential activities to which attention and resources should be devoted. Work [22] describes a runtime non-trace-based algorithm to compute the critical path profile of the execution of message passing and shared-memory parallel programs. This work also presents an online algorithm to compute a variant of critical path, called critical path zeroing, which measures the reduction in application's execution time after the elimination of a selected procedure.

The critical path analysis also gives an effective basis for the scheduling of computations. On multiprocessor system, task scheduling is important to achieve good performance. The work presented in [36] proposes a task scheduling algorithm that allocates tasks followed by correcting the critical path. The technique described in [38] schedules non-cyclic non-branching task graphs, analyzing dynamically the critical paths in current schedule. Papers [61, 71] define the net schedule concurrency level with a set of pairs of operations to be executed in parallel. The techniques based on the minimization of critical path length that is estimated as the maximum clique weight of the sequential and parallel operator graphs constitute the most efficient approach to the generation of concurrent schedules.

1.1.3. Parallelization potential of algorithm

The algorithmic complexity and parallelization potential that is hidden in an algorithm does not depend on the type of underlying hardware architecture and compiler technology which are used for the complexity evaluation. It depends on the algorithm itself and on input data that has to be processed to output data. This book focuses on the methodology for the measure of the critical path as evaluation of the parallelization potential of algorithms / architectures that are described /modeled using a high level programming or hardware description language. Despite the approach could in principle be applied to any programming language, the implementation using an automatic instrumentation stage presented here has been studied and developed for C language.

1.2. Methodology of evaluating algorithm critical path

This chapter presents a methodology for evaluating the critical path on the Data Flow Execution Graph (DFEG) of algorithms specified as C programs. It proposes an efficient dynamic critical path evaluation approach that generates dynamically a data flow execution graph. Such an approach includes two key stages: (1) the instrumentation of the C code and mapping it into a C++ code version, (2) the execution of the C++ code under real input data and dynamically evaluating the actual critical path. The methodology and tools of analyzing algorithms / programs aim at the estimation and increase of the upper bound of the execution speed and parallelization potential of algorithms. The methodology is particularly tailored for application to multimedia, cryptographic and other complex algorithms. Critical path analysis and the subsequent algorithmic development stage is a fundamental methodological preliminary step for the efficient definition of architectures when the objective is the implementation of the multimedia algorithms on parallel homogeneous and heterogeneous platforms.

Summarizing the previous results, we can conclude that the majority of already developed methodologies and tools aim at the critical path profiling for tuning existing parallel programs executed on basic machines (Fig. 1a). In this paper, the objective is to propose a critical path model metrics that can be obtained using automatic evaluation tools such as the one described in [75, 83] in order to be able to find out in which degree a given algorithm described in the C language satisfies the parallel implementation conditions (Fig. 1b). Analyzing the measures of the critical path obtained from simulation results using an automatic instrumentation tool, the most promising algorithms, from the parallelization point of view, can be selected among many alternatives. Moreover, the equivalent transformation of algorithms reducing the critical path and increasing the possible acceleration of the future parallel architecture can be performed.

The following principles constitute a basis for the methodology of the critical path evaluation:

1. The critical path is defined on the C-code's data flow execution without taking into account the true control flow;
2. The critical path length and the system parallelization potential are defined in terms of the complexity of C language basic operations (including *read* and *write* operations). The parameters of the machine executing the instrumented C-code during evaluating the critical path are not taken into account
3. In the definition of the critical path, the Data Flow Execution Graph results from the partial computation of the C-code using true input data. Therefore, such Data Flow Execution Graph is used for the critical path definition instead of the traditional static Data Flow Graph.

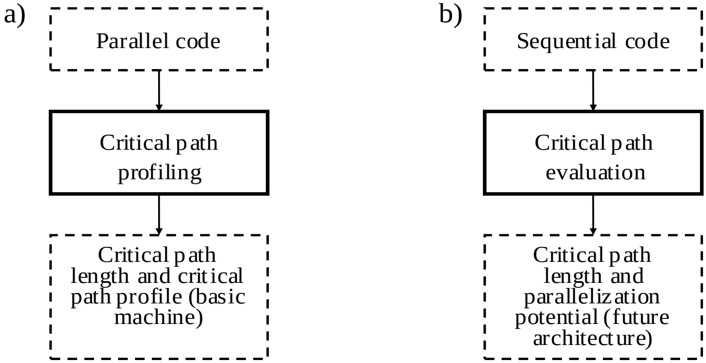

a)
Parallel code
Critical path profiling
Critical path length and critical path profile (basic machine)
b)
Sequential code
Critical path evaluation
Critical path length and parallelization potential (future architecture)

Figure 1.1. Critical path profiling (a) of parallel code on event graphs versus critical path evaluation (b) of sequential code on data dependences graphs

1.2.1. Data Flow Execution Graph

The DFEG is represented as a finite non-cyclic directed weighted graph constructed on the two types of node. The first type includes name-, address-, and scalar value-nodes. The second type includes operator-nodes. The name- and address-nodes are represented as $\boxed{i}$ and the value-nodes are represented as $\boxed{i}$. The operator-nodes are denoted using the usual C-language notation: =, [], ++, --, +, *, **%**, ==, **/**=,<, >, +=, /=, *read* (*r*), *write* (*w*) and others. The graph nodes may be connected by two types of arc: the data dependence arc denoted → and the conditional dependence arc denoted ⇢. The data dependence arc connects input names, addresses and values with an operator and connects an operator with its output value or address. The conditional dependence arc connects a test value with an operator or value covered by a conditional instruction. A graph node without incoming arcs is called an initial node and a graph node without outgoing arcs is called a final node. A DFEG fragment for *if (c) d*=2;* C-code is shown in Fig. 1.2. It contains four value-nodes, one operator-node, three data and one conditional dependence arcs.

An example C-code for recurrent computations is presented in Fig. 1.3. The static DFG for the code is shown in Fig. 1.4. The corresponding dynamic DFEG for the first iteration of the loop is shown in Fig. 1.5. The array components are treated as separate scalar elements. The DFEG includes all types of name-, address-, and value-nodes as well as the various operator-nodes (*deref* is an implicit dereference operator).

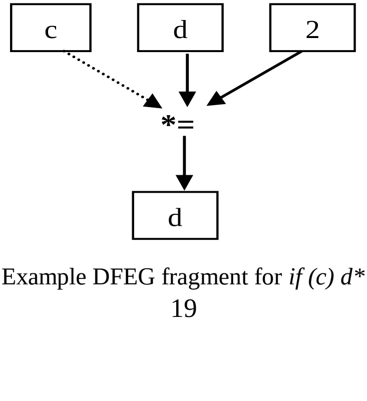


Figure 1.2. Example DFEG fragment for *if (c) d*=2;* C-code

```
#define L 10
void main () {
   float X[L] = {0.6F,0.1F,0.9F,0.3F,0.8F,0.5F,0.7F,0.2F,0.4F,0.7F};
   float Y[L] = {0.3F,0.8F,0.4F,0.2F,0.1F,0.9F,0.5F,0.7F,0.1F,0.6F};
   float Z[L];    Z[0] = 0.5F;
   for ( int i=1;  i < L;  i++)    { int  i1 = i - 1;
      float& Z0=Z[i1];    float& X0=X[i1];    float& Y0=Y[i1];
      float& Z1=Z[i];      float& X1=X[i];      float& Y1=Y[i];
      if ( (Z0+X0*0.7F+Y0*0.3F+0.5F) < ((X1/X0)*0.1F+(Y1/Y0)*0.9F) )
           Z1=X0*0.4F-Y0*0.3F+X1*0.2F-Y1*0.1F+Z0;     else
           Z1=X0*0.1F-Y0*0.2F+X1*0.3F-Y1*0.4F+Z0;
   }
}
```

Figure 1.3. An example C-code for recurrent computations

1.2.2 Computational complexity of data flow execution graph

The complexity of static DFG is traditionally evaluated as a sum of node (in particular operator-node) weights. If we assume that the weight of each operator-node equals 1, then the static complexity of the DFG presented in Fig. 1.4 equals 81. It is obvious this is the complexity of the algorithm description rather than the computational complexity of the algorithm.

The computational complexity of the algorithm can be evaluated on DFEG. The DFEG is weighted with the node complexities. All the complexities are accumulated at the operator-nodes and represent each C-operator by a fragment in the DFEG as shown in Fig. 1.6. A *read* operator is associated with each incoming arc of the operator-node and a *write* operator is associated with its outgoing arc. The complexity of the fragment in Fig. 1.6 is equal to 4 basic operators.

Similarly, assume that each basic operator complexity be equal to 1. Table 1.1 represents the C-language operator complexities. When the basic operator complexities are different, the table can be easily modified to map the critical path length on any target architecture.

The results of the complexity evaluation of the algorithm DFEG fragment shown in Fig. 1.5 are reported in Table 1.2. The overall complexity is of 97 basic operators. The implicit dereference operator complexity is assumed here to be equal to 0.

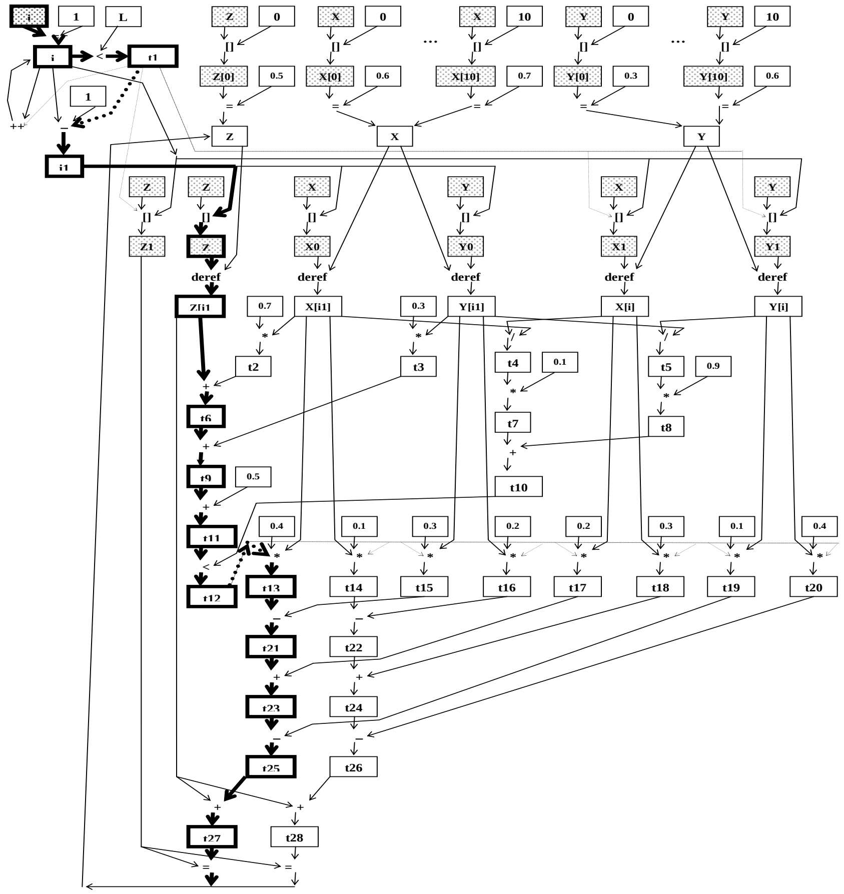


Figure 1.4. The static DFG for the C-code shown in Fig. 1.3. The algorithm description complexity equals 81 operator-nodes. The static critical path is shown in bold. The critical path length equals 14 operator-nodes

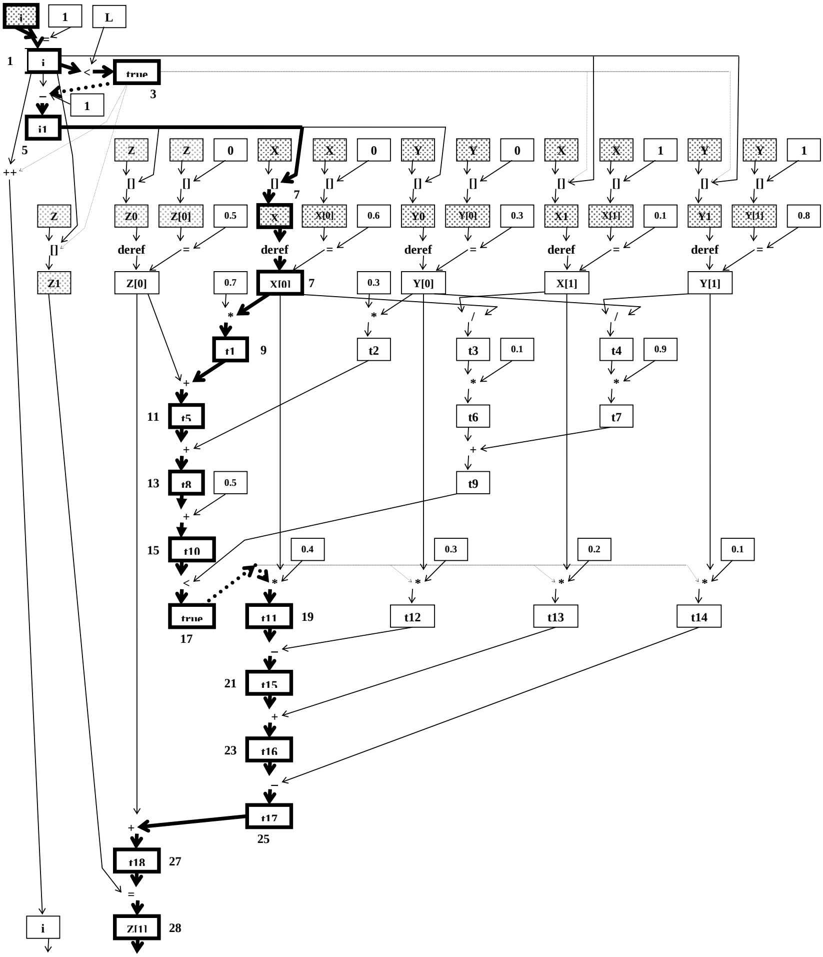


Figure 1.5. The DFEG fragment for C-code shown in Fig. 1.3. The nodes generated during first iteration of the loop are presented. The critical path is in bold. The computational complexity of C-code is estimated through the number of nodes in DFEG. These are name-, value-, and operator-nodes. The data dependences are represented by lines and the conditional dependences are represented by dashed lines

1.2.3 Critical path on data flow execution graph

In literature [43, 52], the concept of static critical path is defined on DFG as a simple critical path and a loop critical path. The objective is to minimize the iteration period during scheduling and resource allocation in high-level VLSI synthesis by means of unfolding, retiming, and pipelining transformations.

The essential drawback of this concept is that the minimum execution time and computational complexity of the overall algorithm cannot be estimated and exploited.

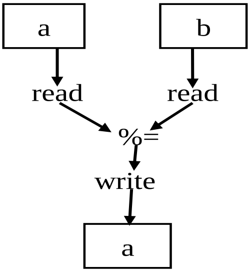


Figure 1.6. Evaluation of the complexity of *a%=b;* C-code

Table 1.1

Complexity and critical path length of C language operators

Operation	Operator	Complexity	Critical path
Assignment	=	1	1
Reference	&	2	2
Dereference	*	2	2
Arithmetic	+, -, *, /, %	3	2
Arithmetic-assignment	+=, -=, *=, /=, %=	4	3
Subscript	[]	3	2
Increment (decrement)	++, --	3	3
Unary minus and others	-	2	2

Table 1.2

Evaluation of the complexity of graph shown in Fig. 1.5

Operator	Operator complexity	Number of operators	Total complexity
=	1	7	7
[]	3	8	24
<	3	2	6
++	3	1	3
*	3	8	24
/	3	2	6
–	3	3	9
+	3	6	18

$\sum = 97$

The critical path on the DFEG is defined as a sequence of the graph nodes with the maximal sum of weights connecting an initial node with a final node. The internal critical path length on the graph fragment shown in Fig. 1.7 equals 3 because two *read* operations are executed in parallel. Similarly, Table 1.1 represents internal critical path lengths of the C-language operators.

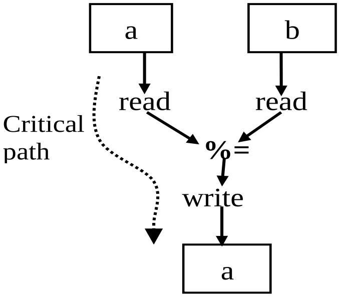


Figure 1.7. Evaluation of the critical path on *a%=b;* C-code

In Fig. 1.5, address- and value-nodes are weighted with external critical path lengths. The critical path on the DFEG for the first iteration of the loop is shown in bold.

The critical path length equals 28. The maximum path length between Z[0] and Z[1] value-nodes equals 19. The nodes describe data dependences between neighbor iterations of the loop and influence of the overall critical path length on the C-code that allows several iterations of the loop. It should be mentioned that the portion of the overall critical path in the DFEG fragment is not the same as shown in bold.

1.2.4 Parallelization potential and feasible acceleration

The C-code computational complexity together with the critical path length in DFEG defines the parallelization potential of the algorithm:

Parallelization_Potential = Complexity / Critical_Path_Length.

The algorithm parallelization potential aims at searching for an efficient parallel implementation of the C-code. It describes the possible acceleration that can be achieved due to a parallel implementation of the algorithm instead of the sequential implementation, but it does not describe the way of construction of such a parallel architecture. Various parallel implementations are possible for the same C-code. The acceleration due to parallelization depends on input data. Different input data imply different possible acceleration of the C-code. The parallelization potential is an upper bound for a non-pipelined architecture. Intermediate parallelization can be considered depending on the constraints on computing resources.

The estimated acceleration can be used in two ways:

- For creating a parallel version of the algorithm
- For reducing the C-code complexity and its critical path or developing a better C-code (algorithm)

The parallelization potential estimates the degree of possible reduction of the execution time due to parallelization of the overall computations associated with the given C-code and input data. Note that the definition of parallelization potential becomes illegal in case the description of complexity and critical path on static DFG such as defined in [43, 52] is used:

- The static DFG in [43, 52] is a very specific model not capable of representing most of algorithm descriptions in C-codes, assuming in particular there are no mutually exclusive branches on DFG
- The description complexity of DFG can be a basis for the estima-

tion of the algorithm computational complexity in specific non numerous applications

- The static critical path time represents the iteration period and is not capable of evaluating and reducing the overall execution time of algorithms in the general case.

For the C-code shown in Fig. 1.3 and for the DFEG's fragment presented in Fig. 1.5, the algorithm complexity for the first iteration is equal to 97 and the critical path length is equal to 28. Therefore, the parallelization potential of the C-code portion is equal to 3.5.

1.3 Evaluation of computational complexity and critical path on data flow execution graph

1.3.1 Explicit evaluation of critical path

One approach to evaluating the critical path length consists in the preliminary generation of the DFEG by means of performing partial computations on the C-code's DFG (Fig. 1.8) under certain meaningful input data. The partially computed flow graph finally contains the operators associated with the scalar operands (values, addresses and variables) and does not contain elements associated with the true control structures. All scalar operands and operators remain in the DFEG.

Given the complexity and internal critical path length of each operator-node in the DFEG, we can evaluate the external critical path for each address-, value- and operator-node in DFEG using the following simple recursive technique:

1. If *val* is an initial name-, address- or value-node then its external critical path length *cpl(val)=0*.
2. If *val* is a value- or address-node and *op_1,…,op_r* are operator-predecessors of *val* (Fig. 1.9b), then its critical path length *cpl(val) = max(cpl(op_1),…, cpl(op_r))*.
3. If *op* is an operator-node and *val_1,…,val_k* are value-address-predecessors of *op* (Fig. 1.9a), then the operator critical path length is *cpl(op) = cplint(op)+ max(cpl(val_1),…,cpl(val_k))*, where *cplint(op)* is the *op* operator's internal critical path length.

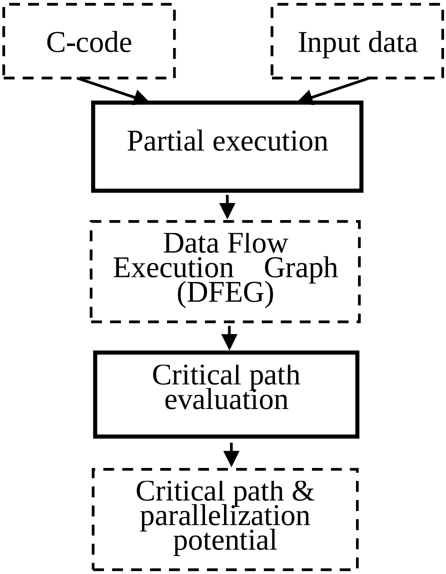

C-code
Input data
Partial execution
Data Flow Execution Graph (DFEG)
Critical path evaluation
Critical path & parallelization potential

Figure 1.8. Critical path evaluation by means of explicit generation of DFEG

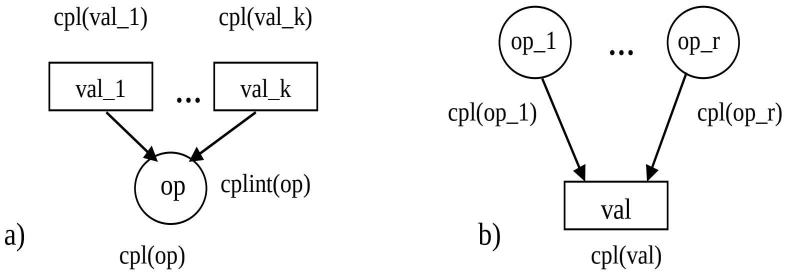


Figure 1.9. The graph fragment (a) for evaluating the critical path for an operator and the graph fragment (b) for evaluating the critical path for a value (address)

The technique itself is very efficient, although it cannot be practically used. Its drawback is that the DFEG can result to a large graph that is difficult to handle. Fig. 1.5 illustrates the evaluation of critical path on the explicitly generated DFEG. The evaluation procedure starts at the initial nodes and step by step using the above listed rules computes the external critical path length for each address- and value-node. The critical path for the Z[1] value-node is the overall critical path on this DFEG.

1.3.2 Dynamic evaluation of critical path

Since the number of nodes in the DFEG is equal to the number of operation calls during the program's execution, explicitly building the graph is not practical for long running programs. One way to overcome this limitation is to develop a technique that does not require building the graph. Such a technique is based on the flow shown in Fig. 1.10. Firstly, the C-code is instrumented by overloading all explicit and implicit operators [83] and is transformed into an equivalent C++-code, in terms of the operators applied to the input data. Secondly, the C++-code is executed under the given input data, computing output data and evaluating the complexity, critical path and parallelization potential of the algorithm.

In the C++-code, an additional *cpl* variable is associated with each actual scalar *var* variable (a separate variable, a scalar element of an array, a scalar element of a structure and so on) of the C-code (Fig. 1.11).

The execution of a C-code operation also results in computing a new value of the associated variable. The *cpl* variable describes the external critical path length for the main *var* variable. The computation of *cpl* is coupled with the computation of *var*. The performance of *op* operator results in computing the value of *var*, re-computing the algorithm complexity, and computing the *cpl* for *var*.

1.4 Tool for estimation of algorithm parallelization potential

1.4.1 Instrumenting and mapping the C-code onto a C++-code

The dynamic evaluation of the critical path as described in the previous section is useful if the program can be appropriately instrumented and mapped using automatic tools into an equivalent version of the code, thus avoiding annoying and resource consuming code rewriting.

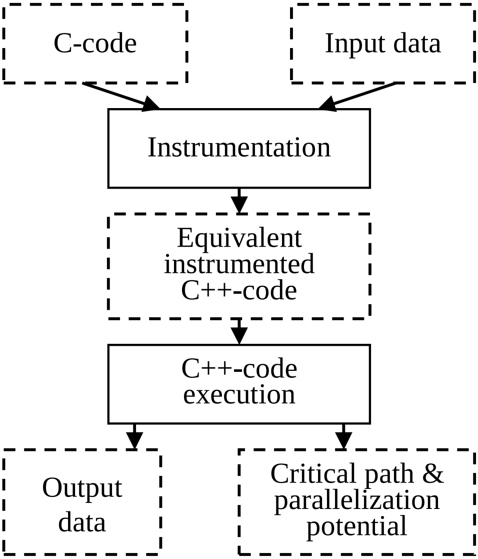

C-code
Input data
Instrumentation
Equivalent instrumented C++-code
C++-code execution
Output data
Critical path & parallelization potential

Figure 1.10. Dynamic evaluation of the critical path by means of instrumenting and executing the C-code

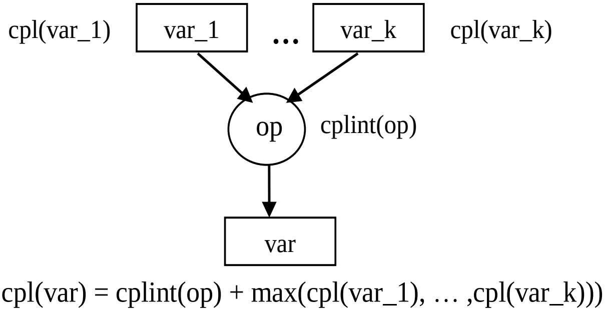


Figure 1.11. General scheme for the dynamic evaluation of the critical path

This section provides an example of how such mapping can be implemented. More details of one possible implementation of such non trivial mapping can be also found in [83]. During the mapping of the source C-code into a C++-code version, the following parts of the C-code have to be instrumented to evaluate the parallelization potential of the algorithm:

- Data types and data objects
- Operators
- Control structures
- Functions.

So as to correctly accomplish the evaluation, global and local additional variables and objects can be used in the instrumented C++-code. Global variable declarations can be as follows:

static unsigned long Algorithm_Complexity = 0;
static unsigned long Critical_Path_Length = 0;
static Critical_Path_Stack   _CPS_;

where *Critical_Path_Stack* is a class implementing the mechanism of processing of conditional dependences associated with the nested control structures. An additional class object and its internal data elements can be associated with each scalar variable of the C-program. The C++-code in Fig. 1.12 will be used in this Section to illustrate and explain the key solutions taken during mapping the C-code into an equivalent C++-code.

The basic types of the C language such as *char*, *int*, *float*, *double*, *signed char*, *unsigned char*, *short int*, *long int*, *unsigned short int*, and others can be mapped into the classes with similar names CHAR, INT, FLOAT, DOUBLE, SIGNEDCHAR, UNSIGNEDCHAR, SHORTINT, LONGINT, UNSIGNED SHORTINT and others in the C++ language. The structure of the INT class in C++ for the *int* basic type of C is shown in Fig. 1.13. The *val* data element of the *int* type represents a variable in the source C-code. The *cpath* data element of the *unsigned long (double)* type describes the external critical path length for the *val* variable.

The class functions overload the operators on the data elements. Fig. 1.12 illustrates the way in which variables *i* and *i1* of type *int* (Fig. 1.3) can be replaced with the same name objects of class INT.

```
#define L 10
void main()    {
         CRITICAL_PATH_TURN_ON
   FLOAT X[L]={0.6F, 0.1F, 0.9F, 0.3F, 0.8F, 0.5F, 0.7F, 0.2F, 0.4F, 0.7F};
   FLOAT Y[L]={0.3F, 0.8F, 0.4F, 0.2F, 0.1F, 0.9F, 0.5F, 0.7F, 0.1F, 0.6F};
   PointerPrih<FLOAT> Xp=X;           PointerPrih<FLOAT> Yp=Y;
   FLOAT Z[L];        PointerPrih<FLOAT> Zp=Z;          Zp[0]=0.5F;
   for(INT i=1;   PUSH_LOOP(i<L);   i++,   POP1)   {     INT i1=i-1;
        FLOAT& Z0=Zp[i1];    FLOAT& X0=Xp[i1];    FLOAT& Y0=Yp[i1];
        FLOAT& Z1=Zp[i];      FLOAT& X1=Xp[i];      FLOAT& Y1=Yp[i];
        if(PUSH_IF((Z0 + X0*0.7F + Y0*0.3F + 0.5F) < ((X1/X0)*0.1F + (Y1/Y0)*0.9F)))
                  Z1=Z0 + X0*0.4F - Y0*0.3F + X1*0.2F - Y1*0.1F;     else
                  Z1=Z0 + X0*0.1F - Y0*0.2F + X1*0.3F - Y1*0.4F;      POP1;
   }
         CRITICAL_PATH_TURN_OFF
}
```

Figure 1.12. Example of a possible C++ instrumentation of the C-code shown in Fig. 1.3

```
class INT {
    int val;
    unsigned long cpath;
  public:
    constructor & destructor functions
    functions for overloading operators
    critical path stack functions
    other functions
};
```

Figure 1.13. Example of possible mapping of the C’s *int* basic data type to the INT class in C++

```
template <class IT> class PointerPrih {
    IT * val;
    unsigned long cpath;
  public:
    constructor & destructor functions
    functions for overloading operators on pointers
    critical path stack functions
    other functions
};
```

Figure 1.14. Example of a possible instrumentation of the C’s pointers in C++

Declarations of pointers to basic types *char*, *int*, *float*, *double*, etc. in C-code can be replaced by the *PointerPrih* classes defined for CHAR, INT, FLOAT, DOUBLE, and other instrumented types. The single template presented in Fig. 1.14 can generate all the classes, where IT denotes an instrumented type.

An array of elements of a basic type in the C-code can be mapped to an array of objects of the corresponding instrumented class in the C++-code. In order to be able to count operations on the arrays including the [] subscript operation, a mechanism of instrumented pointers can be used. An appropriate instrumented pointer can be introduced for each array in the C++-code. All the operations to be executed on the array in the C-code are associated with the pointer in the C++-code. For example, the X, Y, and Z arrays of *float* type in Fig. 1.3 can be replaced with the X, Y, and Z arrays of objects of the FLOAT class in Fig. 1.12. Moreover, the Xp, Yp, and Zp instrumented pointers of the Pointer-Prih<FLOAT> class are introduced in the C++-code. After that, all array operations are executed on the pointers. Other composite types of C language can be instrumented in the similar way in C++ language.

All the operations on addresses and values that will be performed during the C-code execution stage are instrumented during transition from the C-code to the C++-code. Each operator in the C-code is overloaded by an appropriate class function in the C++-code (Fig. 1.15). The operators on the C-types are replaced with operators on the C++-classes. The overloading functions are defined for groups of close operators.

The true control structures are not taken into account during evaluating the critical path. The only influence of the structures on the DFEG is through the conditional dependences. A critical path stack is introduced in the instrumented C++-code in order to find out the dependences. The external critical path length of the declared or temporary T test variable is an element of the stack record.

A new record is added to the stack by the functions PUSH_LOOP (T), PUSH_IF(T), and PUSH_SWITCH(T) presented in Table 1.3 and overloaded for each instrumented basic type by means of the member function *push(cpath)* of the _CPS_ object of the Critical_Path_Stack class. Functions PUSH_LOOP and PUSH_IF return a value of the *bool* type. The difference between the functions is that PUSH_IF adds a record to the stack in any case not depending on its return value.

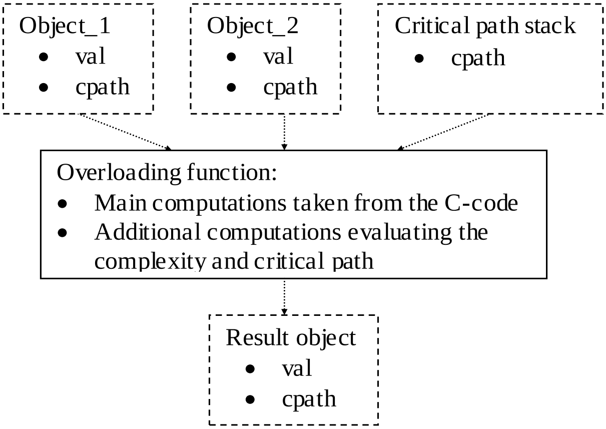

Object_1
val
cpath
Object_2
val
cpath
Critical path stack
cpath
Overloading function:
Main computations taken from the C-code
Additional computations evaluating the complexity and critical path
Result object
val
cpath

Figure 1.15. Overloading a binary operator by a class function

Table 1.3

PUSH and POP macros/functions on the critical path stack

N	Function/Macro	Return type	Description
1	PUSH_IF(Test)	bool	push in _CPS_
2	PUSH_SWITCH(Se)	type of Se	push in _CPS_
3	PUSH_LOOP(Test)	bool	push in _CPS_ when *true*
4	PUSH(CPlen)	void	push CPlen in _CPS_
5	POP(N)	void	pop N records of _CPS_
6	POP1	void	pop 1 record POP(1)
7	POP_(Expr)	type of Expr	POP(1) and transmit Expr
8	POP_(N, Expr)	type of Expr	POP(N) and transmit Expr

The PUSH_LOOP function updates the stack when the return value equals *true* and does not update the stack when the value equals *false*. The return value type of PUSH_SWITCH function is the same as the basic type of T argument. The function always adds a record to the stack.

The top records are removed from the stack by the macros/functions presented in Table 1.3. The macros/function POP(N) belongs to the critical path stack object _CPS_, where N is the number of records to be removed. The value of N equals 1 for *loop*- *if*- and *switch*-statements. It can be greater than 1 for *break*- *continue*- and *return*-statements. A conditional ternary (T ? TE : FE) operator is instrumented as POP_(PUSH_IF(T) ? TE : FE) where TE and FE are expressions executed when test expression T is evaluated to *true* and *false* respectively, and POP_ removes exactly one record from the stack and transmits the operator value. In general case, a *goto* statement makes the use of PUSH and POP functions illegal. The *goto* statements can be eliminated from the C/C++-code by equivalently transforming the unstructured program to a structured one. The mapping rules between C and C++ code versions for control structures are shown in Table 1.4, where *Stat* is a statement.

Fig. 1.16 presents an example of the mechanism of interaction of the instrumented control structures (Fig. 1.12) and the overloaded operators by means of the critical path stack. It is easy to see that the top *cpath* value is always larger than the previous ones in the stack.

The C-function bodies do not constitute a boarder for the data and conditional dependences among external and internal variables. The dependences are transmitted from the external environment to the function body and from the function body to the external environment by means of function's arguments and the return value of instrumented types.

The critical path can be evaluated for any part (parts) of the C-code. They should be described as a separated region by two macros: CRITICAL_PATH_TURN_ON and CRITICAL_PATH_TURN_OFF.

The C++-code that is out of the region simply transmits the variable critical path lengths. Thus, the critical path on the key functions of C code can be evaluated.

1.5 Reduction of critical path and increase of parallelism

1.5.1 Reduction by transformation of C/C++-code

The true control structures of the C-code are an obstacle in the direct implementation of parallelization potential and possible acceleration [69, 70]. The transformation methodology is a mechanism of searching for an appropriate architectural implementation [7, 16, 19]. It allows the reduction of execution time (iteration period, control steps and clock cycles) at the same constraints on resources and approaches the actual acceleration to the upper bound.

Table 1.4

Mapping of C control structures to C++ instrumented structures

Control structure in C	Instrumented structure in C++
if, ?:, switch, while, do, for	
if (TestExpr) ThenStat	*if* (PUSH_IF(TestExpr)) ThenStat POP1;
if (TestExpr) ThenStat *else* ElseStat	*if* (PUSH_IF(TestExpr)) ThenStat *else* ElseStat POP1;
Var = (TestExpr) ? TrueExpr : FalseExpr;	Var = POP_((PUSH_IF(TestExpr)) ? TrueExpr : FalseExpr);

switch (Select) {
case IntVal1: Stat1 *break*;
case IntVal2: Stat2 *break*;
…
default: Statn
}
	switch (PUSH_SWITCH(Select)) {
	case IntVal1: Stat1 *break*;
	case IntVal2: Stat2 *break*;
	…
	default: Statn
	} POP1;

while (TestExpr) Stat	*while* (PUSH_LOOP(TestExpr)) {Stat POP1;}
do Stat *while* (TestExpr)	PUSH(0) *do* Stat POP1; *while* (PUSH_LOOP(TestExpr))
for (Init; Cond; Step) Stat	*for* (Init; PUSH_LOOP(Cond); Step POP1) Stat
break, continue, return, goto	
for (Init; Cond; Step) {Stat1 *if* (TestExpr) {Stat2 *break*;} Stat3}	*for* (Init; PUSH_LOOP(Cond); Step POP1) {Stat1 *if* (PUSH_IF(TestExpr)) {Stat2 POP(2); *break*;} POP1; Stat3}
for (Init; Cond; Step) {Stat1 *if* (TestExpr) {Stat2 *continue*;} Stat3}	*for* (Init; PUSH_LOOP(Cond); Step POP1) {Stat1 *if* (PUSH_IF(TestExpr)) {Stat2 POP(2); *continue*;} POP1; Stat3}
TypeFun NameFun (Arg1,…, Argk) {Stat1 *if* (TestExpr) {Stat2 *return* Expr;} Stat3}	TypeFun NameFun(Arg1,…,Argk) {Stat1 *if* (PUSH_IF(TestExpr)) {Stat2 *return* POP_(1,Expr);} POP1; Stat3}
goto Label;	The unstructured program is transformed to an equivalent structured one

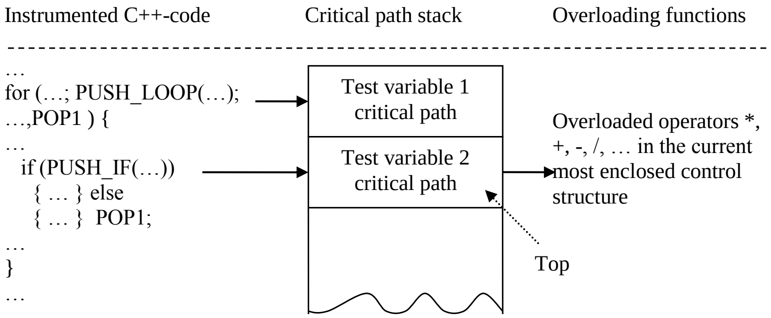

Instrumented C++-code
Critical path stack
Overloading functions
…
for (…; PUSH_LOOP(…);
…,POP1) {
…
if (PUSH_IF(…))
{ … } else
{ … } POP1;
…
}
…
Test variable 1 critical path
Test variable 2 critical path
Overloaded operators *, +, -, /, … in the current most enclosed control structure
Top

Figure 1.16. Generation of the conditional dependences using the critical path stack

Two types of transformation are investigated in the context of architectural synthesis. The transformations of the first type aim at the reduction of the critical path. The critical path evaluation tool helps to localize the transformations. The transformations of second type aim at breaking the true control structures in order to increase the effectiveness of behavioral synthesis and scheduling techniques. The transformation methodology allows the architectural implementation of parallelization potential by means of C-code transformation. The transformations promote the approach of DFG to DFEG.

The equivalent transformation of the source program is also a way of achieving the reduction of the critical path length and the increase of the parallelization potential of the C-code. No specific coding style is needed during creation of the source C-code, although the transformation itself may require specific code forms. The control and data flow transformation rules which are useful in the context of DFEG-based critical path reduction are as follows:

- Restructure, split, and transformation of statements
- Extraction of computations from control structures
- Algebraic transformation of arithmetic, logic and other type of expressions
- Merge of expressions and statements
- Unfolding loops and others.

Although most of the transformation rules have been previously considered in literature, these should be analyzed again in the context of dynamic global critical path definition and reduction on DFEG. For instance, the unfolding, retiming, and pipelining transformations aim at the reduction of iteration period on static DFG which cannot be less than the iteration bound [52]. Moreover the global critical path analysis helps to find places for the efficient application of the transformation rules.

It can be noted that the procedure of increase of the parallelization potential of a C/C++-code is an iterative process. Firstly, the source C-code is transformed and rebuild. Then it is instrumented and mapped to a C++-code version using an automatic tool. After the execution of the C++-code using real input data as stimuli, evaluation of the critical path, estimation of the possible acceleration, and localization of further transformations, the intermediate C-code can then be transformed again in order to perform the next iteration.

1.5.2 Preliminary transformation of loops

In order to be able to apply other transformation rules to the C/C++-code, the loop statements should be preliminary transformed by means of moving the iteration scheme into the loop body. The *for*-loop

```
for (T; D; S)  { B }
```

can be transformed to

```
for (T;  ;  )  { _C_=D;  if (_C_) { B  S }  else  break; }
```

The *while*-loop

```
while (D)  { B }
```

can be mapped to

```
while ( true )  { _C_=D;  if (_C_) { B }  else  break; }
```

The *while*-loop

do *B* ***while*** (*D*)*;*

can be transformed to

do *B* *_C_*=! *D;* ***if*** (*_C_*) ***break;*** ***while*** (***true***);

After these transformations, the extraction of computations from control structures, and other types of transformation are possible.

1.5.3 Extraction of computations from control structures

An efficient way of accelerating the computations is the extraction of operators from control structures and performing them in advance and in parallel. The extraction can follow the preliminary transformation of loops. Fig. 1.17 illustrates the extraction mechanism and transformation rules on the C-code presented in Fig. 1.3. The extraction implies the introduction of additional variables and computations. The critical path length for the first iteration of the loop is reduced from 28 to 16 while the complexity increases from 97 to 143 basic operations (Fig. 1.18). The maximum path length between the Z[i-1] and Z[i] value-nodes is equal to 9. The parallelization potential of the first iteration loop increases from 3.5 to 8.9.

```
#define L 10
void main () {
   float X[L] = {0.6F,0.1F,0.9F,0.3F,0.8F,0.5F,0.7F,0.2F,0.4F,0.7F};
   float Y[L]  ={0.3F,0.8F,0.4F,0.2F,0.1F,0.9F,0.5F,0.7F,0.1F,0.6F};
   float Z[L];     Z[0]=0.5F;
   for ( int i = 1;  ;  )    { int _C1_= i < L;   int i1 = i - 1;
       float& Z0=Z[i1];   float& X0=X[i1];   float& Y0=Y[i1];
       float& Z1=Z[i];     float& X1=X[i];     float& Y1=Y[i];
       if ( _C1_ )  {
           int _C2_= (Z0+X0*0.7F+Y0*0.3F+0.5F) < ((X1/X0)*0.1F + (Y1/Y0)*0.9F);
           float _Zi1_= X0*0.4F - Y0*0.3F + X1*0.2F - Y1*0.1F + Z0;
           float _Zi0_= X0*0.1F - Y0*0.2F + X1*0.3F - Y1*0.4F + Z0;
           if ( _C2_ )  Z1 =_Zi1_;  else  Z1 =_Zi0_;     i++;
       }  else   break;
   }
}
```

Figure 1.17. Transformation of the C/C++-code shown in Fig. 1.3 (transformation of the *for*-loop and extracting computations from the *if-then-else*-statement)

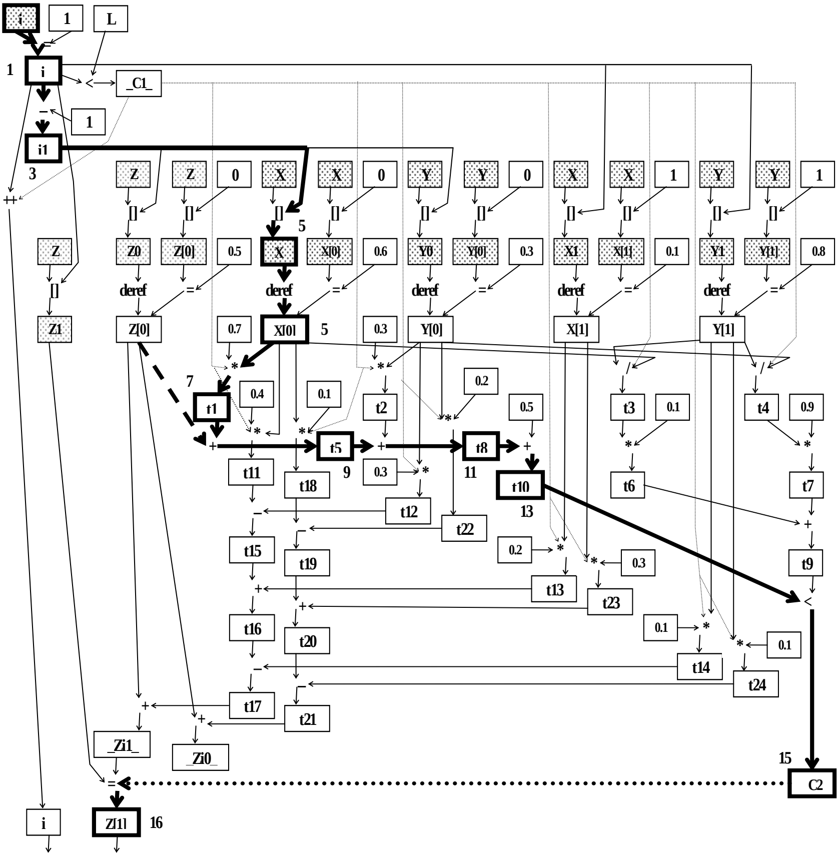


Figure 1.18. The DFEG fragment (first iteration of the loop) for the transformed C/C++-code shown in Fig. 1.17. The transformation is done by means of reconstruction of the loop-statement and extraction of computations from if-statements. The graph complexity implies the introduction of additional variables and computations. The critical path length is in bold.

Fig. 1.19 presents the instrumented C++-code that performs the same basic computations as the source C-code and additionally providing its parallelization potential as result of the program execution grown compared to the non-transformed graph. The critical path shown in bold is reduced. The longest path between Z[i-1] and Z[i] nodes is also in bold.

```
#define L 10
void main()     {
            CRITICAL_PATH_TURN_ON
      FLOAT X[L]={0.6F, 0.1F, 0.9F, 0.3F, 0.8F, 0.5F, 0.7F, 0.2F, 0.4F, 0.7F};
      FLOAT Y[L]={0.3F, 0.8F, 0.4F, 0.2F, 0.1F, 0.9F, 0.5F, 0.7F, 0.1F, 0.6F};
      PointerPrih<FLOAT> Xp=X;          PointerPrih<FLOAT> Yp=Y;
      FLOAT Z[L];         PointerPrih<FLOAT> Zp=Z;        Zp[0]=0.5F;
      for(INT i=1  ;   ; ) {        INT _C1_=i<L;        INT i1=i-1;
           FLOAT& Z0=Zp[i1];       FLOAT& X0=Xp[i1];      FLOAT& Y0=Yp[i1];
           FLOAT& Z1=Zp[i];        FLOAT& X1=Xp[i];       FLOAT& Y1=Yp[i];
           if(PUSH_LOOP(_C1_))   {
                 INT _C2_ = (Z0 + X0*0.7F + Y0*0.3F + 0.5F) < ((X1/X0)*0.1F + (Y1/Y0)*0.9F);
                FLOAT _Zi1_ = Z0 + X0*0.4F - Y0*0.3F + X1*0.2F - Y1*0.1F;
                FLOAT _Zi0_ = Z0 + X0*0.1F - Y0*0.2F + X1*0.3F - Y1*0.4F;
                if(PUSH_IF(_C2_))   Z1=_Zi1_;    else    Z1=_Zi0_;     POP1;
                i++;      POP1;
          } else break;
      }
             CRITICAL_PATH_TURN_OFF
}
```

Figure 1.19. Equivalent instrumented C++-code for the source code reported in Fig. 1.17 (Transf_1)

1.5.4 Transformation of expressions

The transformation of expressions is an efficient way of rebuilding the DFG and the DFEG of the C/C++-code. The objective of expression transformation is to rebuild the DFG in such a way as to reduce the number of operations on the critical path.

Fig. 1.20 presents a very simple transformation of expressions in the C/C++-code shown in Fig. 1.19. The transformation consists in changing the order of operation executions by means of using parenthesis. The transformed DFEG for the first iteration of the loop is presented in Fig. 1.21. The critical path length and the complexity of the loop’s first iteration is the same as for the DFEG presented in Fig. 1.18. In the meantime, the maximum path length between the Z[i-1] and Z[i] value-nodes decreases from 9 to 5 basic operations. This implies the reduction in the total critical path length for many iterations of the loop.

```
…
INT       _C2_ = ( Z0 + ( X0*0.7F + Y0*0.3F + 0.5F ) )  <  ( (X1/X0) *0.1F + (Y1/Y0) *0.9F );
FLOAT _Zi1_ =  Z0  +  ( X0*0.4F - Y0*0.3F + (X1*0.2F - Y1*0.1F) );
FLOAT _Zi0_ =  Z0  +  ( X0*0.1F - Y0*0.2F + (X1*0.3F - Y1*0.4F) );
…
```

Figure 1.20. Transformation and instrumentation of expressions (Fig. 1.18) in the C/C++-code (Transf_2)

1.5.5 Effectiveness of transformations

Table 1.5 provides a comparison of the parallelization potential of three different C-codes (and instrumented C++-codes) with the same functionality. The number of executed iterations of the loop is the same and equals 10.

It is easy to see that the extraction of computations from control structures and the transformation of expressions imply significant increase in the algorithm execution acceleration and in the parallelization potential. The Transf_1 performed by means of extraction of computations reduces the critical path length by 2.16 compared to the source code. The Transf_2 performed by means of reordering of operator executions in expressions additionally reduces the critical path length by 1.44. The overall reduction constitutes 3.11.

In the meantime, some transformations can imply the increase in the C-code complexity. Thus, the extraction of computations in Transf_1 implies the increase in C-code complexity by 1.3. The reasons are as follows:

- The reorganization of the C-code introduces additional variables and operators (operator executions)
- The extraction of computations from the if-statements implies the execution of operators in any case not depending on the value of test expressions; if the operators were under the control structures it would not be necessary to execute some of them.

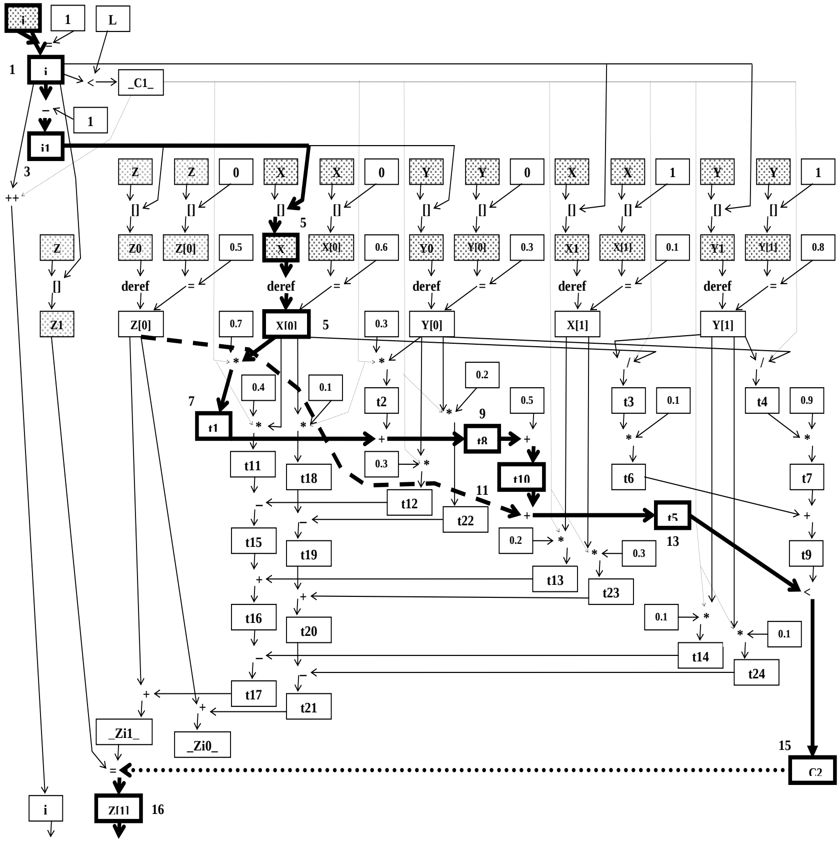


Figure 1.21. The DFEG illustrates reduction of the overall critical path length by means of transforming expressions. The reduction is obtained by reordering operators. The critical path is in bold. The distance between Z[i-1] and Z[i] nodes constitutes 5 operator-nodes instead of 9 operator-nodes in the previous DFEG

Table 1.5

Parameters of the source and transformed C/C++-code

Algorithm	Complexity	Critical path	Parallelization potential	Feasible acceleration
Source	802	171	4.7	1.00
Transf_1	1039	79	13.2	2.16
Transf_2	1039	55	18.9	3.11

1.6 Evaluation accuracy and limitations

There are some assumptions implemented in the current dynamic critical path evaluation tool version. One of them is that a value-node in the explicitly generated DFEG can have more than one incoming arcs with weights (intermediate critical path lengths) to which the max-operation is applied. All the weights could be computed simultaneously in a parallel implementation version of the tool. But the implemented tool version runs on a single-processor machine and executes the instrumented C++-code sequentially. Since only one additional critical path variable is associated with each main scalar variable, all the weights at the incoming arcs cannot be stored and processed simultaneously. The weights are processed sequentially, as the instrumented C++-code is being executed. As a result the execution of the max-operation is broken into several steps which can imply some slight inaccuracy in the critical path measure.

The second assumption is that the C-code should not contain a variable representing several other different variables whose lifetimes are not intersected. The critical path length for this single variable would differ from the critical path length for the several separate variables due to the use of the max-operation. This is a source of inaccuracy in the critical path evaluation.

There are few limitations on the evaluation technique. One of the most significant takes place for data that are interpreted in different way by means of different types. For instance, the following two declarations

```
long lvar [] = {1, 3, 5, 7, 9, 15};
char* cvar = (char*) lvar;
```

cannot be legally instrumented and processed as

```
LONG lvar [] = {1, 3, 5, 7, 9, 15};
PointerPrih<CHAR> cvar = (CHAR*) lvar;
```

However, the mentioned inaccuracies and limitations do not constitute a significant burden for most of the evaluations performed on complex multimedia algorithms. Some alternative implementation of the operators overloading capable of removing such limitations are under study. The critical path evaluation tool has been successfully used for large programs such as Wavelet algorithm implementations, the MPEG-4 Optimized Reference Software, and the Cryptographic toolkit [72, 84] and others, without requiring any code rewriting.

1.7 Conclusion

This chapter has presented a methodology for the measure of the parallelization potential of complex algorithms. The measure is based on the dynamic evaluation of the data flow execution graph and is performed by mapping a C-program into an instrumented C++ version, and then executing the equivalent C++ program under real input data. By combining critical path evaluations with code transformation techniques, an efficient methodology can be built for exploring parallel implementations of the algorithm, thus detecting efficient architectures the algorithm can be mapped to. The mapping from the C description to a C++ instrumented description that provides critical path measures can be done by an automatic software tool, avoiding resource consuming code rewriting.

Analyzing the obtained measures, for each methodological iteration, the most promising algorithms in terms of parallelization potential can be selected among many possible alternatives. Applying transformations to the algorithm and reducing the critical path length, thus further increasing the degree of parallelization, result very effective for the definition of efficient implementation architectures. The critical path length significantly influences the results of scheduling the implementations at several kinds of constraints on computational resources. The schedule cannot be faster than the critical path length. A systematic methodology for reduction of the critical path length guarantees more powerful scheduling results and implicitly provides improvements in the trade off “complexity–delay” that is common for software development, high-level synthesis and architecture design in various application fields.

2. PARALLELIZATION POTENTIAL OF MEANINGFUL HARDWARE / SOFTWARE APPLICATIONS

2.1. Parallelization potential of two-dimensional WAVELET codec

Impressive results on the parallelization potential have been obtained for the two-dimensional Wavelet codec implementations proposed in [83]. Tables 2.1, 2.2 and 2.3 report experimental results obtained on three versions of DFEG that are dynamically generated on different C-codes with the same functionality:

- DFEG of the original C-code as it was created (Case I)
- DFEG of Case I without nodes that describe control computations on the two dimensional array representing an image (Case II)
- The transformed C-code of Case II and its DFEG (Case III).

In Case I a portion of the C-code is responsible for global iterative traversal of the two dimensional array representing an image. In Case II the array is considered as a set of directly addressed and accessed separate scalar variables. The control computations associated with the iterative global traversal can be eliminated. An architecture which implements distributed on pixels computations can be generated. In Case III a C-code is obtained by means of transforming several expressions constituting the wavelet core.

The algorithm computational complexity increases as the image size grows. It constitutes from 79 to 493 million operations for Case I and from 30 to 187 million operations for Cases II and III. The average number of operations executed per pixel equals 257 in Case I, and equals 97 operations in Cases II and III. The data flow computations to be incorporated in the C-code implementation constitute 37.9%, and the control flow computations constitute 62.1%.

The increase in the image size implies the increase in the critical path length. The length varies in the range from 2.55 up to 6.17 thousand operations in Case I, in the range from 168 up to 196 operations for case II and in the range from 144 to 168 operations in Case III. After the equivalent transformation of WAVELET C-code and modifying its DFG (Case III), the critical path length has been reduced by 16.7% compared to Case II.

The parallelization potential depends on the image size and varies in the range from 30932 up to 79896 in Case I, varies in the range from 177938 up to 953426 in Case II and varies in the range from 207594 up to 1112331 in Case III.

Table 2.1

Experimental results for WAVELET (Case I)

N	Image		Algorithm parameters		
	Width	Height	Algorithm complexity	Critical path	Parallelization potential
1	640	480	78,876,895	2,550	30,932
2	800	600	123,272,013	3,146	39,184
3	1024	576	151,470,377	3,416	44,341
4	1152	864	255,635,992	4,470	57,189
5	1280	1024	336,716,669	5,172	65,104
6	1600	1200	493,277,958	6,174	79,896

Table 2.2

WAVELET without control computations (Case II)

N	Image		Algorithm parameters		
	Width	Height	Algorithm complexity	Critical path	Parallelization potential
1	640	480	29,893,590	168	177,938
2	800	600	46,711,523	168	278,045
3	1024	576	57,395,520	168	341,640
4	1152	864	96,855,102	168	576,519
5	1280	1024	127,568,960	196	650,862
6	1600	1200	186,871,523	196	953,426

Table 2.3

Transformed C-code of WAVELET (Case III)

N	Image		Algorithm parameters		
	Width	Height	Algorithm complexity	Critical path	Parallelization potential
1	640	480	29,893,590	144	207,594
2	800	600	46,711,523	144	324,386
3	1024	576	57,395,520	144	398,580
4	1152	864	96,855,102	144	672,605
5	1280	1024	127,568,960	168	759,339
6	1600	1200	186,871,523	168	1,112,331

Figures 2.1, 2.2 and 2.3 visualize the data reported in Tables 2.1, 2.2, and 2.3 and represent the trends in the Wavelet's complexity, critical path, and possible acceleration due to parallelization.

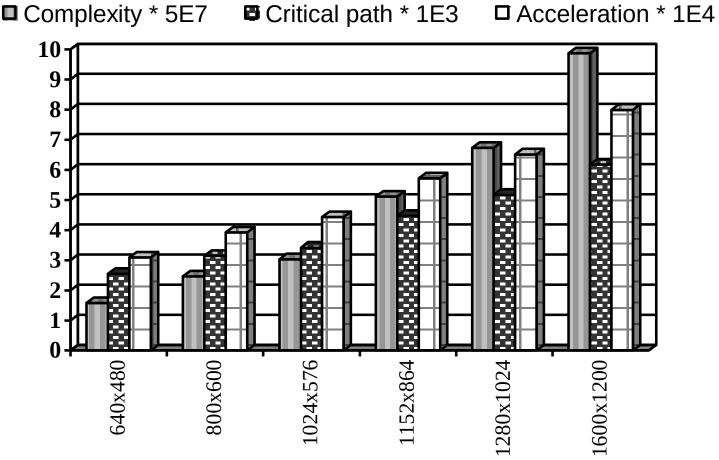


Figure 2.1. Algorithm complexity, critical path length, and acceleration due to parallelization versus image size for WAVELET with control computations (Case I)

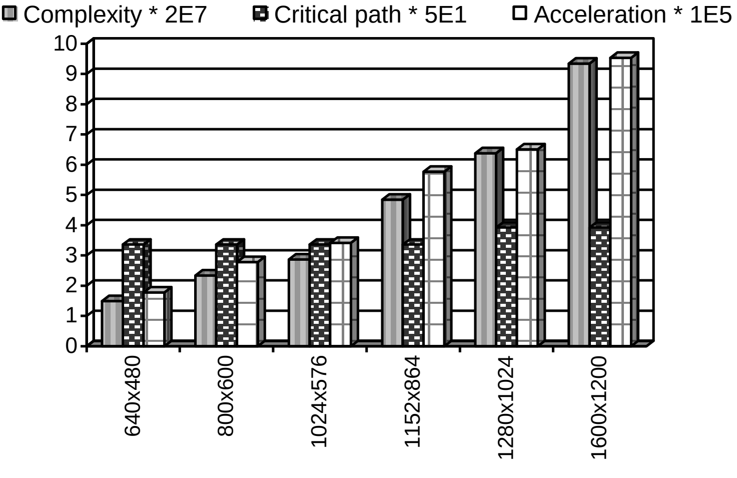


Figure 2.2. Algorithm complexity, critical path length, and possible acceleration versus image size for WAVELET without control computations (Case II)

As an image consists of a lot of pixels, the parallelization per pixel is a very important metrics characterizing the WAVELET algorithm. It is easy to see that the parallelization per pixel is equal to the complexity per pixel divided by the critical path length:

ParallelizationPerPixel = Parallelization_Potential / Image_Size =
*= Complexity / (Critical_Path_Length * Image_Size) =*
= (Complexity / Image_Size) / Critical_Path_Length.

If the WAVELET's parallel computations are assumed to be the two dimensional computations distributed on the pixels, we may ask the question, how many computations are common for neighbor pixels? If the critical path length were the same for each pixel and were equal to the complexity per pixel, we could conclude there are no common computations for neighbor pixels. Fig. 2.4 shows parallelization per pixel versus image size and proves that the common computations constitute more than 90% in Case I, constitute from 42% to 50% in Case II, and constitute from 33% to 42% in Case III. In Case III, the parallel computations are the most distributed.

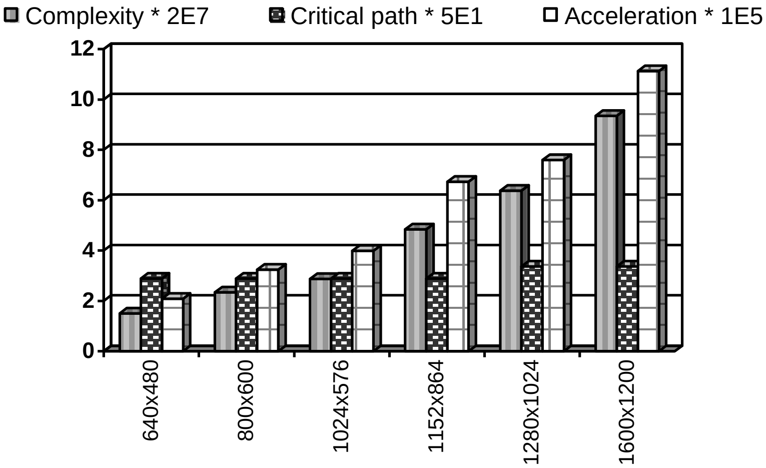


Figure 2.3. Algorithm complexity, critical path length, and possible acceleration versus image size for transformed code of WAVELET (Case III)

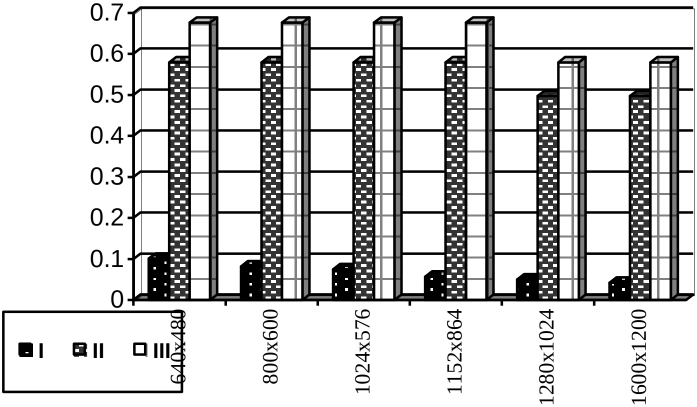


Figure 2.4. Parallelization per pixel versus image size (Cases I, II, and III)

2.2. Parallelization potential of RSAREF cryptographic toolkit

The RSAREF is a cryptographic toolkit [72, 84] designed to facilitate rapid development of Internet Privacy-Enhanced Mail (PEM) implementations. RSAREF supports the following PEM-specified algorithms: (1) RSA encryption and key generation, as defined by RSA Data Security's Public-Key Cryptography Standards (PKCS), (2) MD2 and MD5 message digests and (3) DES (Data Encryption Standard) in cipher-block chaining mode. The RSAREF is entirely written in C.

With RDEMO the cryptographic operations of signing, sealing, verifying, and opening files, as well as generating key pairs can be performed. Three series of experiments have been made: (1) Sign a file with private key, (2) Generate random DES key, encrypt content, and encrypt signature with DES key (seal a file) and (3) Generate RSA public/private key pair. Experimental results are presented in Tables 2.4 - 2.7. The possible acceleration due to parallelization potential of the C-code varies from 42.21 up to 136.93. Fig. 2.5 and Fig. 2.6 describe the algorithm complexity, critical path length and degree of parallelization versus the file and key sizes.

Table 2.4

Experimental results for RSAREF (sign a file)

Content size (Bytes)	Algorithm complexity	Critical path	Parallelization potential
281	21,804,816	502,000	43.44
621	21,826,260	509,660	42.83
971	21,846,076	517,582	42.21

Table 2.5

Experimental results for RSAREF (seal with sign)

Content size (Bytes)	Algorithm complexity	Critical path	Parallelization potential
281	26,781,944	502,685	53.28
621	29,553,488	510,345	57.91
971	32,455,414	518,267	62.62

Table 2.6

Experimental results for RSAREF (seal without sign)

Content size (Bytes)	Algorithm complexity	Critical path	Parallelization potential
281	4,978,890	77,491	64.25
621	7,728,936	77,491	99.74
971	10,611,022	77,491	136.93

Table 2.7

Experimental results for RSAREF (key-pair generation)

Key size (bits)	Algorithm complexity	Critical path	Parallelization potential
508	0.6E9	13.8E6	43.89
767	3.0E9	90.0E6	33.41
1024	21.9E9	806.4E6	27.12

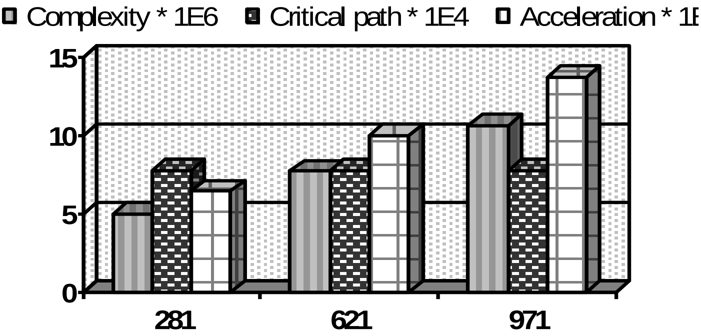


Figure 2.5. Algorithm complexity, critical path length, and possible acceleration versus file size for seal

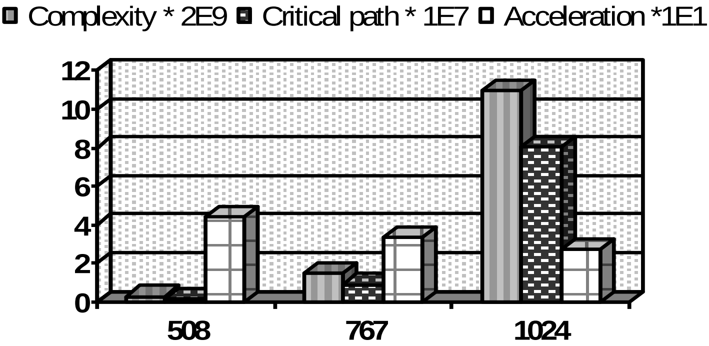


Figure 2.6. Algorithm complexity, critical path length, and possible acceleration versus key size for key pair generation

2.3. Parallelization potential of MPEG-4 video codec

2.3.1. MPEG-4 video codec

The computational complexity, critical path, and parallelization potential profiles measured on the algorithm partition tree or any other partitioning constitute an effective basis for timing and performance analysis of feasible parallel algorithm implementations. The results of metrics measuring enable to correctly select partitions that need accurate optimizations or that can produce considerable implementation speed-ups by means of parallel implementations. An example of results that enable correctly analyzing the critical functions of a complex video coding algorithm is reported. The results obtained on the measurement of the critical path and parallelization potential profiles of the MPEG-4 video codec and subsequent timing and performance analysis of the C/C++-code functions tree discover the ways of efficient code reconstruction and implementation definition.

The complex algorithm that is under analyses in this chapter is a software implementation of a part of MPEG-4 Video tools [38] as specified by the MPEG-4 Video standard (ISO/IEC 14496-2) reference software. This is an optimized enhanced compression codec (document M9632 in 65th meeting, July, 2003, Trondheim, Norway) based on the simple profile for representing visual data: video, still textures, synthetic images, etc. In this version, there are enhanced features: advanced error detection and correction services on top of H.263. H.263 is a standard video-conferencing codec optimized for low data rates and relatively low motion.

One new part that has been developed is a new video codec. This is joint work with the ITU who were defining an H.26L codec (follow on beyond H.261 and H.263). The work has been done by Joint Video Taskforce (JVT) working group and has become a new MPEG-4 video standard as part 10, i.e. ISO/IEC 14496-10 and is called Advanced Video Coding or *AVC* and is technically identical to the ITU-T H.264 standard.

Fig. 2.7 shows the typical structure of the Moving Picture Experts Group (MPEG) encoder. Motion estimation and compensation are key parts of video compression. They help remove temporal redundancies in images. The MPEG standard depends on two basic algorithms. Motion-compensated coding uses block-based motion vector estimation and compensation to remove temporal redundancies. Block discrete cosine transforms reduce spatial redundancy.

The MPEG standard uses three types of pictures that depend on the mode of motion prediction. The intra (I) picture serves as the reference picture for prediction. Block discrete cosine transforms code the intra pictures, and no motion estimation prevents long range error propagation. Coding the predicted (P) pictures uses forward prediction of motion. Each image is divided into macro blocks of size pixels and search blocks of the same size in the prior reference I frame or P frame. A second type of picture is the bidirectional interpolated (B) picture. Both forward and backward motion predictions are performed with respect to the prior or future reference I or P frames.

The two main types of motion estimation use pel-recursive algorithms or block matching algorithms. Pel-recursive algorithms predict the motion field at the decoder based on how neighboring pixels decoded in the current frame relate to pixels in the prior frame. Exhaustive search within a maximum displacement range leads to the absolute minimum for the energy of the prediction error and is optimal in this sense. Motion-compensated video coding relates the intensity of each pixel in the current frame to the intensity of some pixel in a prior frame. It links these pixels by predicting the motion of objects in the scene.

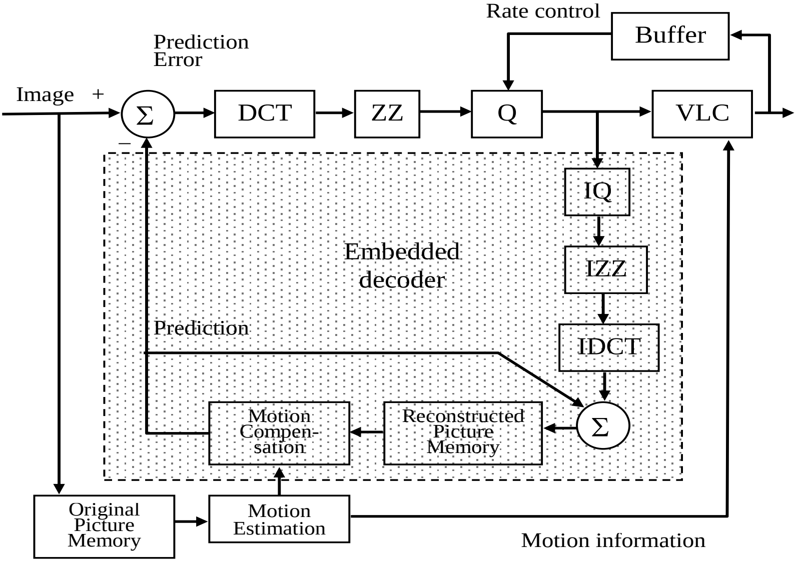


Figure 2.7. Block diagram of the MPEG encoder: DCT is discrete cosine transform, ZZ is zigzag scanning, Q is quantizer, VLC is variable length coding, IQ is inverse quantizer, IZZ is inverse zigzag scanning, and IDCT is inverse discrete cosine transform

The experimental results have been obtained for the following configuration of the EMPEG-4 video encoder software: the Microsoft Visual Studio 6.0 platform, the VM5+ rate control, the MVFAST in N4554 fast motion search, no error resilient, and the disabled post-filter.

The profiling is performed for the C/C++-code functions tree of the MPEG-4 video encoder shown in Fig. 2.8. The intra encoding is applied to the first frame and inter encoding is applied to the subsequent frames.

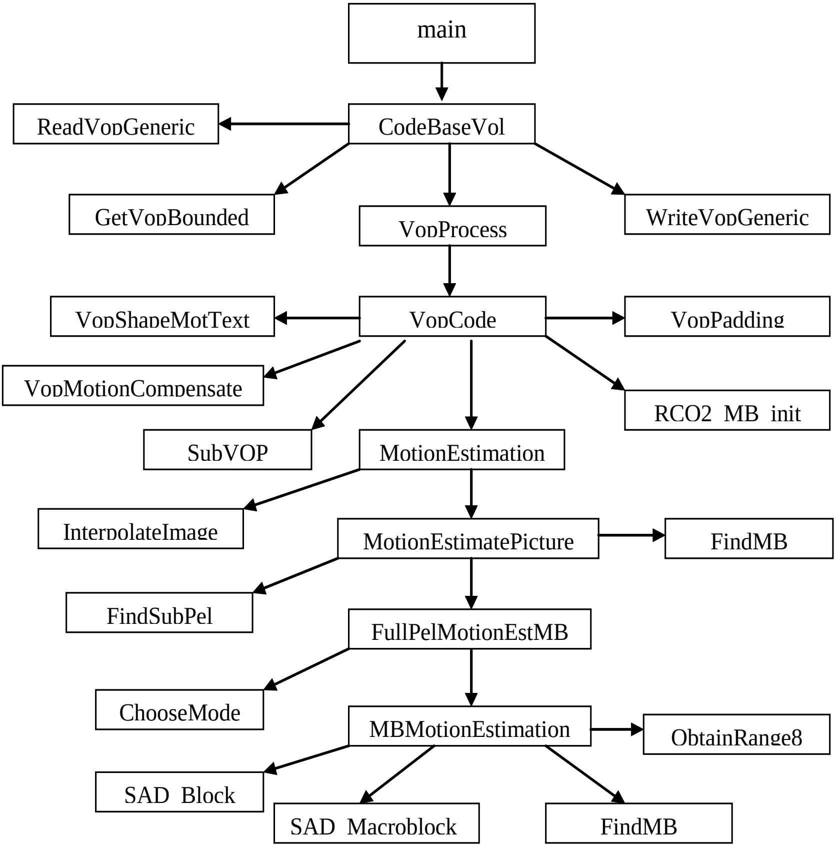


Figure 2.8. The tree of key functions of MPEG-4 encoder reference software (inter encoding)

2.3.2. Source video sequence

The video coder is performed on the following input sequence of frames: the Foreman source video sequences, 100 input/output frames, the CIF (352x288) and QCIF (176x144) picture sizes, the YUV (4:2:0) format, the I-PPP coding type, and 8 bits per pixel.

2.3.3. Profiling computational complexity

The video encoder computational complexity profile that is measured in the number of C/C++ operations is reported in Table 2.8 and Fig. 2.9. The number of calls is additionally reported for each function.

Table 2.8

Computational complexity and critical path of MPEG-4 encoder

Function	Own computational complexity	Number of function calls	Share (%) in overall complexity
main	9930795562	1	100.00
CodeBaseVol	9930773442	99	100.00
ReadVopGeneric	75274848	99	0.76
GetVopBounded	153268929	99	1.54
WriteVopGeneric	135777114	99	1.37
VopProcess	9566444136	99	96.33
VopCode	9491144637	99	95.57
VopShapeMotText	1873222888	99	18.86
VopMotionCompensate	195466324	99	2.37
SubVOP	125453097	99	1.97
RCQ2_MB_init	234913737	99	1.26
VopPadding	110013453	99	1.11
MotionEstimation	6940357262	99	69.89
InterpolateImage	692083854	99	6.97
MotionEstimatePicture	6243294170	99	62.87
FindSubPel	2897497781	182720	29.18
FindMB	69872128	36544	0.70
FullPelMotionEstMB	3273115389	39204	32.96
ChooseMode	259256052	39204	2.61
MBMotionEstimation	3013192869	39204	30.34
SAD_Block	1839728800	3728588	18.53
SAD_Macroblock	981290638	470111	9.88
FindMB	74958048	39204	0.75
ObtainRange8	9205936	156816	0.09

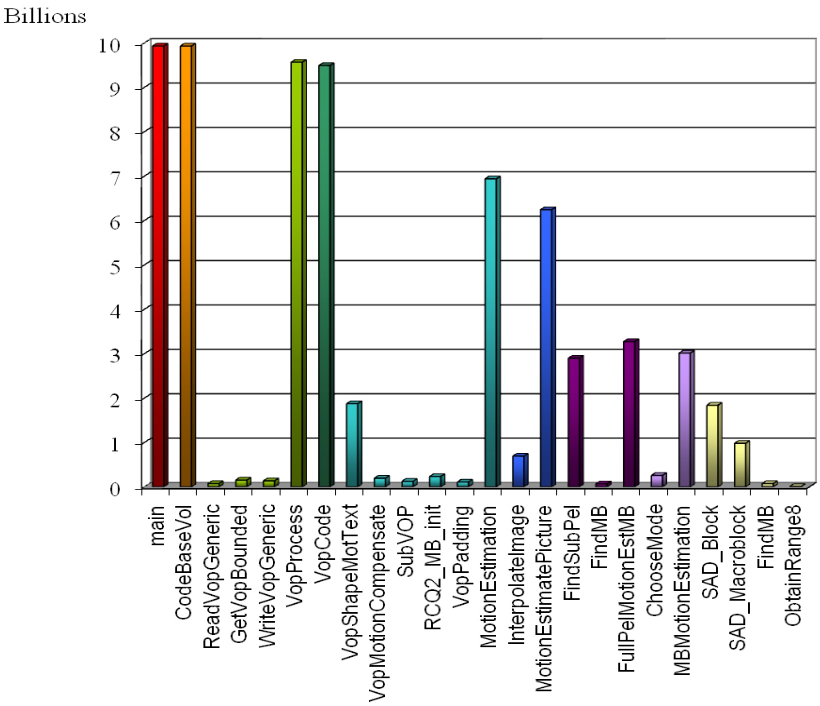


Figure 2.9. Own computational complexity of MPEG-4 encoder functions

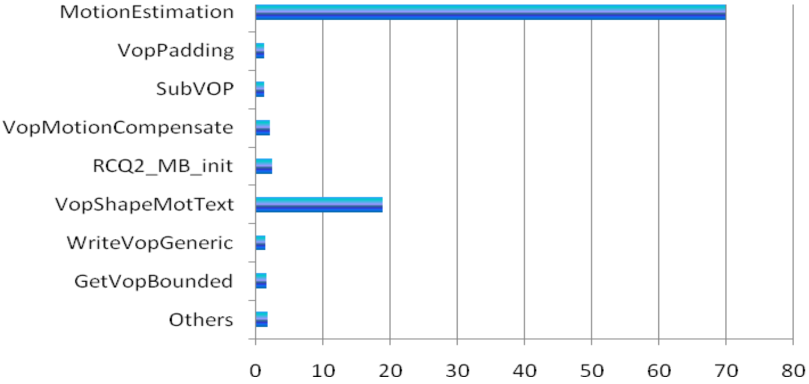


Figure 2.10. Computational complexity profiling of MPEG-4 video codec

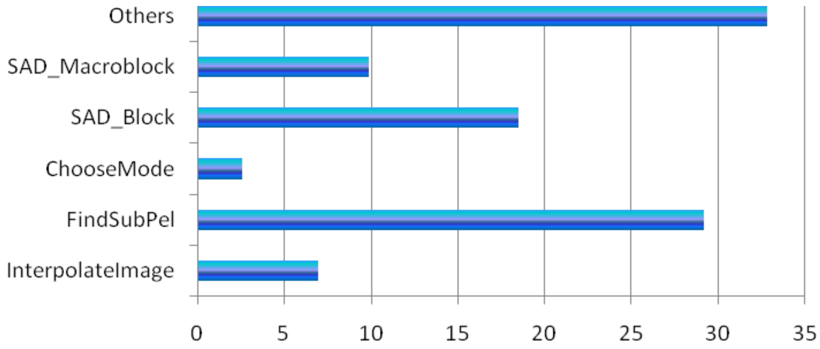


Figure 2.11. Computational complexity profiling of *MotionEstimation*

The encoder overall computational complexity equals 9'930'773'442 operations for 99 frames. *MotionEstimation* (69.89%) and *VopShapeMotText* (18.86%) are the most computational complexity consuming composite functions (Fig. 2.10).

The results of computational complexity profiling of *MotionEstimation* are shown in Fig. 2.11 The basic functions of *MotionEstimation* which mostly contribute to the computational complexity are: FindSubPel (29.18%), SAD_Block (18.53%), SAD_Macroblock (9.88%) and InterpolateImage (6.97%).

2.3.4. Profiling critical path

The video encoder critical path profile that is measured in the number of C/C++ operations is described by Table 2.9 and Fig. 2.12. We distinguish the own critical path of each function and the share of a function in the overall critical path of the encoder. The overall critical path of encoder is equal to 6'054'211 operations. *MotionEstimation* is the most contributing (83.42%) composite function in the overall path (Fig. 2.13). The basic functions which have significant share in the overall critical path are (Fig. 2.14): FindSubPel (37.21%), SAD_Macroblock (18.57%), VopShapeMotText (11.81%) and SAD_Block (4.60%).

Table 2.9

Own critical path and share in overall critical path of MPEG-4 encoder functions

Function	Own critical path	Share in overall critical path	Share in %
main	6054211	6054211	100.00
CodeBaseVol	6054211	6054211	100.00
ReadVopGeneric	101378	0	0.00
GetVopBounded	101732	101444	1.68
WriteVopGeneric	152065	0	0.00
VopProcess	6050363	5952722	98.32
VopCode	6050357	5952722	98.32
VopShapeMotText	2209971	714819	11.81
VopMotionCompensate	7417	8145	0.13
SubVOP	101577	442	0.01
RCQ2_MB_init	57979	30	0.00
VopPadding	15563	18	0.00
MotionEstimation	5819926	5050491	83.42
InterpolateImage	2233	95	0.00
MotionEstimatePicture	5819849	5050287	83.42
FindSubPel	2533334	2252627	37.21
FindMB	557468	0	0.00
FullPelMotionEstMB	3292285	2712966	44.81
ChooseMode	76274	68397	1.13
MBMotionEstimation	3188680	2617335	43.23
SAD_Block	313592	278560	4.60
SAD_Macroblock	1434231	1123982	18.57
FindMB	68	0	0.00
ObtainRange8	109719	97496	1.61

The comparison of the two profiles on the computational complexity and critical path proves that there are functions like *SAD_Block* whose share in the overall complexity (18.53%) is larger than their share (4.60%) in the overall critical path. It also proves that there are functions like *MotionEstimation* and *FindSubPel* whose share in the overall critical path (83.42% and 37.21% respectively) is larger than their share (69.89% and 29.18% respectively) in the overall complexity. If the goal is to reduce the critical path such functions should be considered and transformed, first of all, in order to increase their parallelization potential.

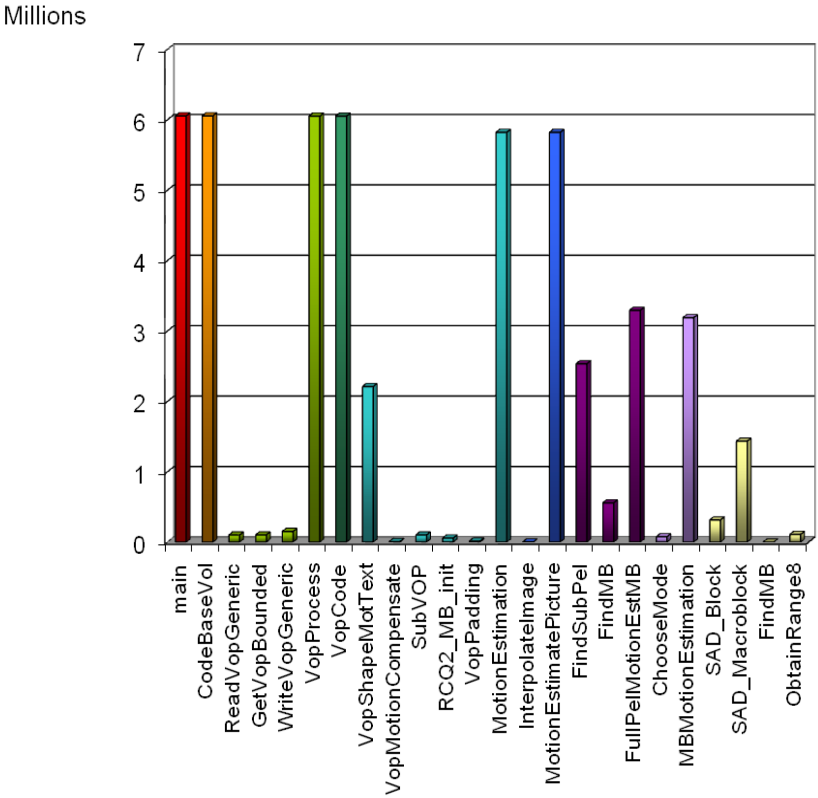


Figure 2.12. Own critical path of MPEG- encoder functions

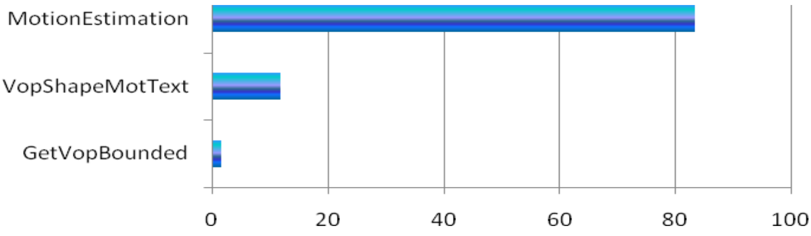


Figure 2.13. Critical path profiling of MPEG-4 video codec

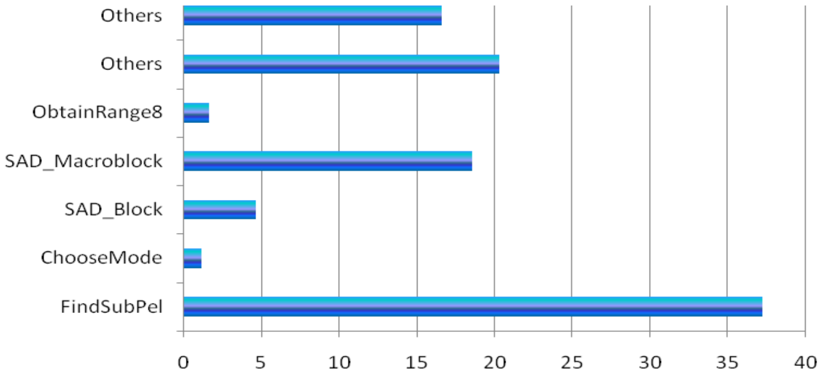


Figure 2.14. Critical path profiling of *MotionEstimation*

Some results of more detailed analyses of the critical path are reported in Table 2.10 and are shown in Fig. 2.15. A comparison of the own critical path, critical path profile, and critical path zeroing is given for video encoder functions. For some functions like *FindSubPel* the critical path share of 88.92% is almost the same as the own critical path (100%). For other functions like *VopShapeMotText* the critical path contribution of 32.35% is significantly less than the own critical path. It means the own path and the overall path are not significantly intersected.

Table 2.10

Own critical path share in overall critical path and critical path zeroing

Function	Own critical path, %	Share in overall critical path, %	Critical path zeroing, %
GetVopBounded	100	99.72	0.35
VopShapeMotText	100	32.35	7.56
MotionEstimation	100	86.78	49.47
FindSubPel	100	88.92	64.64
FullPelMotionEstMB	100	82.40	68.05
ChooseMode	100	89.67	89.61
SAD_Block	100	88.83	88.19
SAD_Macroblock	100	78.37	71.02
ObtainRange8	100	88.86	62.09

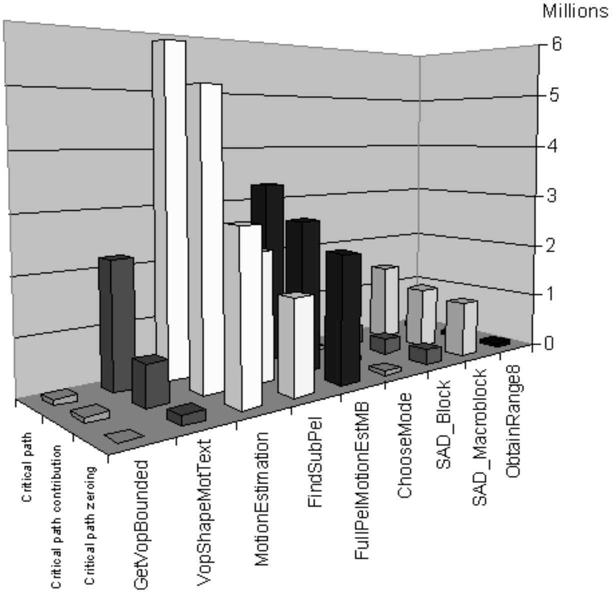


Figure 2.15. Own critical path, share in overall critical path, and critical path zeroing in MPEG-4 video codec functions

The critical path zeroing characterizes a possible reduction in the overall critical path due to reduction in the own critical path of a function. The amount of reduction varies in a wide range. For example, the critical path zeroing of function *SAD_Macroblock* is 71.02% at the share in the overall critical path of 78.37%. It means the most part of share can be potentially reduced due to the reconstruction of function *SAD_Macroblock*. Contrary, the critical path zeroing of function *VopShapeMotText* is 7.56% at the critical path share of 32.35%. Although the share is not too high against the own critical path, it is difficult to reduce it significantly.

2.3.5. Profiling parallelization potential

Table 2.11 presents the own parallelization potential of each function (column 2) and the parallelization potential of a function in the functions tree (column 3). They are estimated over the computational complexity (Table 2.8), critical path length and the share in the overall critical path (Table 2.9) of each function of the functions tree. The overall parallelization potential of the whole algorithm is 1640. The video encoder own parallelization potential profile with respect to the key functions of MPEG-4 video codec (Fig. 2.16), which consume the highest amount of computational complexity estimates the feasible acceleration of the future distributed architecture. The own parallelization potential defined as the computational complexity divided by the own critical path varies from 684 (*SAD_Macroblock*) to 309934 (*InterpolateImage*) and to 1102324 (FindMB).

Table 2.11

Parallelization potential of MPEG-4 video encoder functions

Function	Own parallelization potential	Parallelization potential in functions tree
main	1640.31	1640.31
CodeBaseVol	1640.31	1640.31
ReadVopGeneric	742.52	infinity
GetVopBounded	1506.60	1510.9
WriteVopGeneric	892.89	infinity
VopProcess	1581.14	1607.07
VopCode	1568.69	1594.42
VopShapeMotText	847.62	2620.6
VopMotionCompensate	26353.83	442231.5
SubVOP	1235.05	4181769.9
RCQ2_MB_init	4051.70	7830457.9
VopPadding	7068.91	6111858.5
MotionEstimation	1192.52	1374.2
InterpolateImage	309934.55	7285093.2
MotionEstimatePicture	1072.76	1236.22
FindSubPel	1143.75	1286.3
FindMB	125.34	infinity
FullPelMotionEstMB	994.18	1206.47
ChooseMode	3399.01	3790.5
MBMotionEstimation	944.97	1151.24
SAD_Block	5866.63	6604.4
SAD_Macroblock	684.19	873.0
FindMB	1102324.24	infinity
ObtainRange8	83.90	94.42

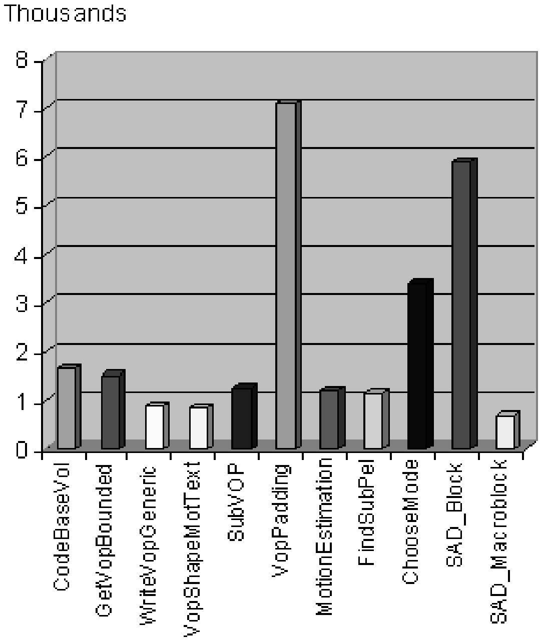


Figure 2.16. Parallelization potential of MPEG-4 video codec key functions

Another important characteristic of a function is the parallelization potential in the functions tree that is defined as the computational complexity divided by the share of the function in the overall critical path. As the critical path share is less than the own critical path, the parallelization potential in the functions tree is always not less than the own parallelization potential. Its value varies from 873 (*SAD_Macroblock*) till infinity (*ReadVopGeneric, WriteVopGeneric, FindMB*). The infinity value is obtained due to the zero share of the function in the overall critical path.

Growth of parallelization potential versus number of frames. Table 2.12 describes the growth of computational complexity, critical path and parallelization potential depending on the number of encoded video frames. Increasing the number of frames twice implies the growth of the complexity about twice (Fig. 2.17). The critical path length grows significantly slowly. The critical path length has grown by 38.4 times for 99 frames against 1 frame. It means, the parallelization among computations of consecutive frames of the same video sequence is possible.

Fig. 2.18 shows the degree of parallelization versus number of frames. The algorithm of encoding 10 frames can be 2.297 time parallelized against the algorithm of encoding 1 frame. On 20 frames the encoding algorithm can be 2.449 times parallelized over encoding 1 frame. For 99 frame the parallelization potential grows to 2.618 time. It is interesting that the parallelization potential of 2.626 for 50 frames is larger than that for 99 frames. It depends on the video sequence.

Table 2.12

Computational complexity, critical path and parallelization potential of MPEG-4 video encoder versus number of frames

Number of frames	Computational complexity	Critical path	Parallelization potential
1	98678728	157492	626.56
2	196200376	175382	1118.70
3	297762503	239334	1244.13
4	399369531	311858	1280.61
5	499918543	374486	1334.95
6	600758570	433265	1386.58
7	700916076	481945	1454.35
8	802059492	567969	1412.15
9	901871124	627270	1437.77
10	999191080	694366	1439.00
12	1200758747	819560	1465.13
14	1401196194	969017	1446.00
16	1601740755	1089697	1469.90
18	1799544062	1183045	1521.11
20	2001069072	1303941	1534.63
30	3006092583	1927676	1559.44
40	3993971189	2544380	1569.72
50	4981470788	3027194	1645.57
60	5974901481	3735201	1599.62
70	6917099291	4343810	1592.40
80	7947724259	5164862	1538.81
90	8996846808	5717979	1573.43
99	9930773442	6054211	1640.31

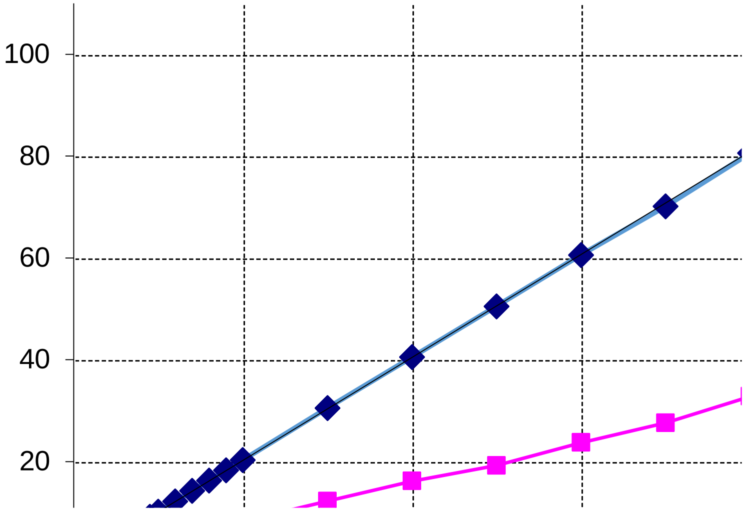


Figure 2.17. Growth of overall computational complexity and critical path length of MPEG-4 video encoder versus number of frames

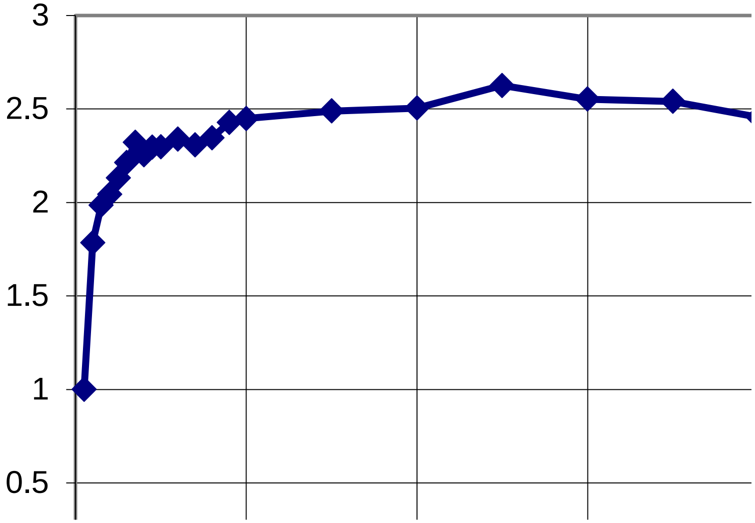


Figure 2.18. Growth of overall parallelization potential of MPEG-4 video encoder versus number of frames

2.4. Conclusion

The methodology of measuring the parallelization potential of complex algorithms that is presented in chapter 1, is applied to the reference software of several meaningful applications: two-dimensional WAVELET codec, RSAREF cryptographic toolkit, MPEG-4 video codec and others. This chapter has presented the results of measuring the computational complexity, critical path and level of parallelism which are hidden in the C-code of the applications.

3. TRANSFORMATION OF ALGORITHM TO BASIC SINGLE-BLOCK MODEL

3.1. Algorithm transformation flow

The idea of a step-by-step transformation of an algorithm [58, 63, 64, 66, 69, 70, 74, 76], which improves key parameters of complex sequential program code that is associated with an algorithm subsequent parallel implementation (Fig. 3.1) lies in the basis of the parallelism extraction method we develop in this chapter. Parallelism that is extracted from the sequential code is implemented further in a functionally equivalent parallel code or in a parallel hardware architecture. Extracting parallelism which improves parameters of subsequent implementation is a complex process that requires knowledge of key concepts about the static and dynamic properties of a program. The proposed method is based on the following key principles:

1. measurement of parameters of sequential code that predict parameters of subsequent parallel implementations;
2. step-by-step transformation of a serial code in order to improve parameters of an equivalent parallel code;
3. mapping sequential code into an efficient parallel implementation using a basic single-block flow model.

The measurement of parallelism hidden in sequential code is based on metrics and object-oriented instrumentation technology proposed in [61]. The main metrics characterizing parallelism, laid out in a sequential program, include the computational complexity of the program code, the critical path on the data flow graph of the program execution (DFEG - Data Flow Execution Graph), and the factor of maximum parallelization potential. We give brief definitions of these concepts, following this work. The computational complexity of a program code on a typical input data set is measured in the number of basic operations (operators or instructions) of the programming language, in which the code is written and performed during the execution of the program. Operations of access to data elements, including write operations to memory and read from memory, can be taken into account together with logical and arithmetic operations, comparison operations, etc.

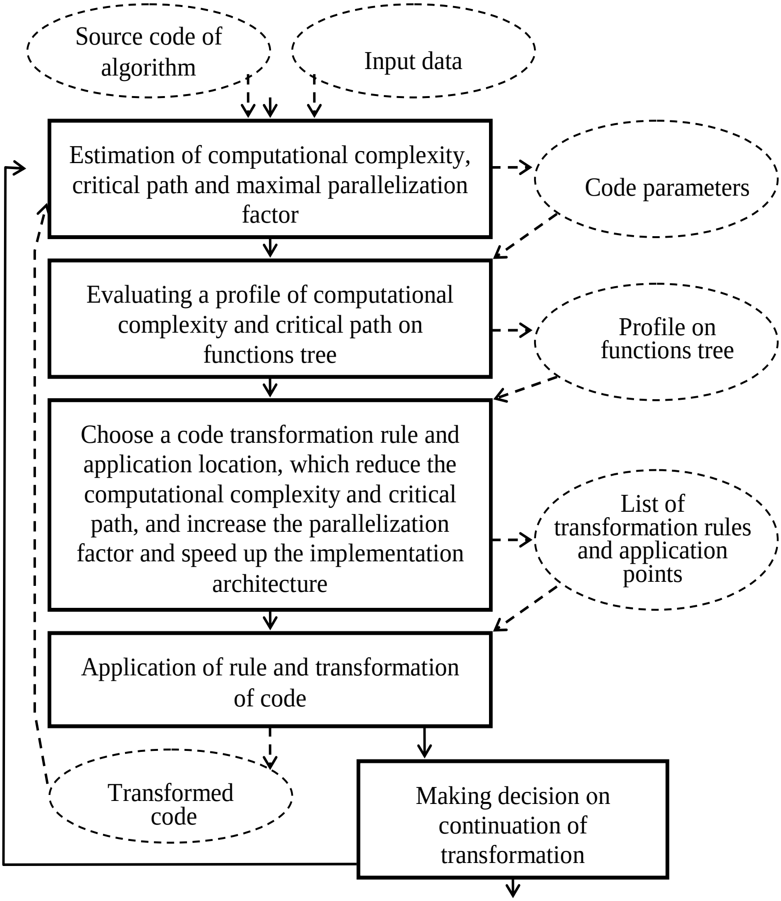

Source code of algorithm
Input data
Estimation of computational complexity, critical path and maximal parallelization factor
Code parameters
Evaluating a profile of computational complexity and critical path on functions tree
Profile on functions tree
Choose a code transformation rule and application location, which reduce the computational complexity and critical path, and increase the parallelization factor and speed up the implementation architecture
List of transformation rules and application points
Application of rule and transformation of code
Transformed code
Making decision on continuation of transformation

Figure 3.1. Algorithm parallelization flow by means of code transformation

Accounting for data exchange operations between parallel parts of the code is not possible at an early design stage, since the decomposition of the entire code into parallel parts will be carried out later.

Graph DFEG is an acyclic directed finite graph constructed on a set of vertices that are variables or operators, taken part in program execution. A set of arcs represents data dependencies between input variables and operators, and between operators and output variables, realized during execution of the code. The graph is generated for one set of input data. Changes in the input data entail, in general, changes in the size and structure of the graph. In fact one program code can produce tremendous number of different DFEGs. Implementation costs are associated with vertices-operators and vertices-variables of the graph. The number of vertices and their costs characterize the computational complexity of the code on the given input data. The longest weighted path on the graph is called the critical path. It characterizes the maximum execution time of the code in case of its parallel implementation. The ratio of the computational complexity to the critical path of the code is the parallelization potential factor. It characterizes the feasible maximum acceleration of the parallel implementation against the sequential implementation.

The serial code transformation method, which improves the parameters of the subsequent parallel implementation, is built on two components:

- transformation rules that can increase the parallelization factor of the code;
- the method of localization of code fragments, the transformation of which is most effective in reducing the computational complexity and reducing the critical path.

The following transformation rules are most promising, since they lead to the restructuring of control flow and data flow graphs that are beneficial for resolving parallelism and increasing the code parallelizability:

- splitting of control structures
- speculative computing of operators which are extracted from control structures;
- merging assignment operators and transforming expressions to reduce the maximum depth of expression trees
- unrolling of loops with static and dynamic iteration schemes and others

Localization of the points of rules application in the program code is carried out by building a computational complexity profile, a critical path profile, and parallelization potential profile. The computational complexity profile describes the contribution of each function in the total computational complexity of the entire code, expressed ultimately as a percentage. The critical path profile is a list of functions lying on the critical path and the contribution of each function to the total length of the overall critical path of the program code. The potential parallelization profile characterizes the level of feasible parallelization of each function in the functions tree. In order to improve the parameters of the entire code, functions and their components are revealed, which make the greatest contribution to the computational complexity and critical path, and these functions are purposefully transformed, increasing the parallelization factor while maintaining the algorithm optimization flow described in Fig. 3.1.

The method of mapping a sequential code to a parallel implementation provides a transition from the original sequential control flow to a parallel control flow while maintaining the original data flow. In this case, the system is divided into parts, taking into account the measured and detected parallelism in the sequential code, by decomposing both the operation part of the code and the data structures processed by the code. The original sequential control flow often prevents the system from being divided into parallel subsystems. Our method transforms the initial sequential control flow of the code and its parts to a basic single-block flow model [69-71, 76]. The equivalent parallel code is represented by means of basic primitives of the operating system, primitives of the MPI library, and by other facilities.

The advantages of the developed method are manifested in the following fundamental possibilities.

- The same behavioral description can be reduced to a form adequate to one or another parallel architecture;
- Limits of parallelization and limits of the system implementation can be extended;
- The transformed behavioral description may be more effective in terms of the applicability of the methods, strategies, and algorithms for solving optimization problems of automatic parallelization and scheduling.
- At the same constraints on a system parameter, it is possible to obtain more favorable values of other parameters.

- The method extends capabilities for design space exploration regarding software implementations and regarding hardware implementations.

3.2. Preliminary transformation of algorithm

One of the key concepts of a sequential program that has a decisive influence on the operation, properties and capabilities of models and methods for the extraction of parallelism is the concept of a linear block (basic block). A linear segment is a chain of operators sequentially executed one after another, which does not include transitions over branching instructions.

It is difficult to extract parallelism from looping / branching programs that are built using a combination or superposition of *while, do-while, if, switch* and other similar instructions, and perform processing of data of arbitrary types, in particular, pointers. Methods for the extraction of parallelism from programs in this category are little developed or not studied at all.

The most important step in transformation process is the elimination of multiple use of one variable by means of the introduction of new additional variables. Bellow we assume that single assignment requirement is met by all variables in the program code.

The rules for transforming the source code to a basic single-block model, that is introduced in this chapter, are constructed in such a way as to make the transition to using a limited subset of constructs, which are typical for an algorithm description language like C/C++. After that, they eliminate all complex control structures from the code, and separate the data flow from the control flow. As a result, the transformation of the source code is carried out in two stages, each of which uses its own set of transformation rules:

- Rules for transformation of sequential control flow for using a restricted subset of control instruction of the programming language;
- Rules for data flow extraction by means of stepwise elimination of original control structures and control flow.

The transition from a source code written in C language to a basic single-block model is performed by applying the transformation rules of the form

Left_part => Right part. (3.1)

In rule (3.1), *Left_part* and *Right part* represent program code fragments that are semantically (functionally) equivalent in sense of describing in different ways the same mapping of input data into output data.

The rules use the notation as follows: V, V_1,… are Boolean variables, C, C_1,… are Boolean expressions, Q, ,Q_1,… are instructions, S, S_1,…,R, R_1,… are sequences of instruction, and L, L_1,… are labels of loops.

The rules for preliminary transformation of the control flow of the original algorithm are shown in Fig. 3.2. They allow to transform such control structures as *if*, *switch*, *while*, *do-while*, *for*, *break*, *continue* and others. As a result it is possible to proceed to the use of a restricted subset of the C/C++ language.

Rule	Left part	Right part
R1	*while* (C) { S }	*while* (1) { *if* (C) { S } *else break*; }
R2	*do* { S } *while* (C);	*while* (1) { S *if* (!C) *break*; }
R3	*for*(S_1; C; S_2) { S_3 }	S_1; *for* (;;) { *if* (C) { S_3 S_2 } *else break*; }
R4	S_1 *if* (C) { *while* (1) { S_2 } }	V=1; *while* (1) {*if* (V) { S_1 V=0; } *if* (C) { S_2 } *else break*; }
R5	S_1 *if* (C) { *while* (1) { S_2 } }	V=1; *while* (1) {*if* (V) { S_1 } *if* (!V||C) { S_2 V=0; } *else break*; }
R6	*if* (C_1) { *break*; } *if* (C_2) { *break*; }	*if* (C_1||C_2) { *break*; }
R7	*if* (C) { *break*; } S	V=C; *if* (!V) { S } *if* (V) { *break*; }
R8	*for*(S_1 ; C_1; S_2) { S_3 *if* (C_2) *continue*; S_4 }	S_1 *for*(;;) { { *if* (!C_1) *break*; S_3 *if* (!C_2) { S_4 }; S_2 }
R9	*while* (1) { S_1 *if* (C) *continue*; S_2 }	*while* (1) { S_1 *if* (!C) {S_2} }
R10	*if* (C) { S_1 } *else* { S_0 }	V_1=C; V_2=!C; *if* (V_1) { S_1 } *if* V_2) { S_0 }
R11	*if* (V) { Q_1;…Q_k; }	*if* (V) {Q_1 } … *if* (V) { Q_k }
R12	*if* (V_1) { *if* (V_2) { S } }	V:=V_1 *&&* V_2; *if* (V) { S }
R13	*if* (V_1) { V_2=E*;* }	V_2 = (V_1 && E) || (!V_1 && V_2);
R14	*switch* (E) { *case* H_1: S_1 … *case* H_n : S_n *default*: S_{n+1} }	V=E; *if* (V==H_1) { S_1 } *else* … *if* (V==H_n) { S_n } *else* { S_{n+1} }

Figure 3.2. Rules for preliminary transformation of C code

Rule R1 converts a *while* loop to a loop with an infinite iteration scheme and a conditional *break* statement, which provides an exit from the loop when it is executed sequentially with the sequence *S* of operators. Rule R2 converts a loop *do-while* to a loop *while* with the infite iteration scheme and operator *break* under a conditional instruction, after executing the statements of the sequence S. Rule R3 converts the loop with an iteration scheme *for* to a loop with an infinite iteration scheme and a body, that includes an additional operator *break* under a conditional statement, body *S*3 of the original loop, and part *S*2 of the original iteration scheme.

Rule R4 allows to put an *if*-instruction covering a loop with an infinite iteration scheme and statements S_1 located before the *if*-instruction, inside the loop body in case the loop body statements *S*2 do not change the value of the expression *C*, introducing one additional Boolean variable *V*. Rule R5 is a generalization of the previous rule for the general case; it allows to insert an *if*-instruction covering a loop with an infinite iteration scheme and statements *S*2 that are located before the *if*-instruction, inside the loop body.

Rule R6 merges two operators *break*, covered by conditional statements, into one statement. Rule R7 allows the permutation of operator *break* that is under an *if*-instruction with condition *C*, with sequence S of statements located after *break*.

Rule R8 transforms operator *continue* covered by a conditional operator, which is a part of a loop body with iteration scheme *for*, to conditional execution of a part of the loop body. Rule R9 transforms operator *continue* covered by a conditional operator, to conditional execution of a part of the loop body.

Rule R10 splits one conditional *if*-instruction into two short conditional instructions; the passing condition of each of them is calculated by an additional assignment instruction. Rule R11 splits one short *if*-instruction with a sequence of statements, into an equivalent sequence of simpler short *if*-instructions with one nested statement; such a transformation is admissible if the value of variable *V* is not changed by operators $Q_1,...,Q_{k-1}$. Rule R12 reduces the system of two nested if-instructions to one conditional resultant instruction. Rule R13 allows to get rid of a short *if*-instruction, if it includes an assignment operator that works with

Boolean variables and expressions. Rule R14 transforms instruction *switch* to a system of nested conditional statements.

We demonstrate the application of control flow conversion rules using the RSA cryptographic standard [84], namely, using the C-code of function *NN_DigitDiv* that is shown in Fig. 3.3. The code can be classified as difficult to parallelize because its control flow is built from the superposition of *if* and *while* instructions. Applying the rules from Fig. 3.2 to the source code shown in Fig. 3.3, we obtain the preliminary transformed code shown in Fig. 3.4. All control structures are split and unified, the loops have an infinite iteration scheme in this code. Complex expressions are split, assignment instructions are associated with one logical, arithmetic, or other operator of the C language. Despite simplifications, the code contains 11 larger and 4 smaller linear basic blocks. It is difficult to extract parallelism from so many nested basic blocks.

3.3. Basic single-block flow model

This chapter presents a basic single-block flow model (BSBM) of a sequential algorithm in which the data flow is separated from the control flow [69, 70]. The goal of building BSBM is to efficiently extract data flow parallelism from difficult-to-parallel sequential looping-branching algorithms of general form by means of reducing the original control flow. The number of paths on the control flow graph of the algorithm, which determines the complexity of analyzing the source sequential code and synthesizing the parallel result code, grows exponentially depending on the size of the graph and is determined mainly by the number of basic blocks in the algorithm code. Reducing the number of basic blocks simplifies the control flow graph from the point of view of parallelization efficiency. The OBBM model includes only one basic block and provides real extraction of all types of parallelism from the original basic blocks. In fact, the model provides for merging the data flows of individual basic blocks into a single data flow of a single basic block while preserving potential parallelism and possible acceleration for execution on a multiprocessor system.

```
typedef unsigned long nnd;
typedef unsigned short nnfd;
#define nnhdb 16
#define mnnfd 0xffff
#define mnnd 0xffffffff
#define lhf(x) (nnfd)((x)&mnnfd)
#define hhf(x) (nnfd)(((x)>>nnhdb)&mnnfd)
#define tohf(x) (((nnd)(x))<<nnhdb)

void NN_DigitDiv (nnd *a, nnd b[2], nnd c) {
  nnd t[2], u, v;
  nnfd aHigh, aLow, cHigh, cLow;
  cHigh = hhf(c);  cLow = lhf(c);
  t[0] = b[0];  t[1] = b[1];
// Underestimate high half of quotient and subtract.
  if(cHigh == mnnfd)  aHigh = hhf(t[1]);  else
    aHigh = (nnfd)(t[1] / (cHigh + 1));
  u = (nnd)aHigh * (nnd)cLow;
  v = (nnd)aHigh * (nnd)cHigh;
  if((t[0] -= tohf(u)) > (mnnd - tohf(u)))  t[1]--;
  t[1] -= hhf(u);   t[1] -= v;
// Correct estimate.
  while((t[1]>cHigh)||((t[1]==cHigh)&&(t[0]>=tohf(cLow)))) {
    if((t[0]-=tohf(cLow))>mnnd-tohf(cLow)) t[1]--;
    t[1] -= cHigh;
    aHigh++;
  }
// Underestimate low half of quotient and subtract.
  if(cHigh==mnnfd)
    aLow = lhf(t[1]);
  else
    aLow = (nnfd)((nnd)(tohf(t[1])+hhf(t[0]))/(cHigh+1));
  u = (nnd)aLow * (nnd)cLow;
  v = (nnd)aLow * (nnd)cHigh;
  if((t[0]-=u) > (mnnd - u)) t[1]--;
  if((t[0]-=tohf(v))>(mnnd-tohf(v))) t[1]--;
  t[1] -= hhf(v);
// Correct estimate.
  while((t[1]>0)||((t[1]==0)&&t[0]>=c)) {
    if ((t[0]-=c)>(mnnd-c)) t[1]--;
    aLow++;
  }
  *a = tohf(aHigh) + aLow;
}
```

Figure 3.3. A fragment of C code for RSA

```
typedef unsigned long nnd;
typedef unsigned short nnfd;
#define nnhdb 16
#define mnnfd 0xffff
#define mnnd 0xffffffff
#define lhf(x) (nnfd)((x)&mnnfd)
#define hhf(x) \
  (nnfd)(((x)>>nnhdb)&mnnfd)
#define tohf(x) (((nnd)(x))<<nnhdb)

void NN_DigitDiv1(nnd *a,
     nnd b[2], nnd c) {
 nnd t[2],u,v;
 nnfd aHigh,aLow,cHigh,cLow;
 int c1,c2,c3,c4,c5,c6,c7,c8,c9;
 nnfd x1;
 nnd x2,x7,x8,x9,x10,x11,x16,x17;
 int x3,x4,x5,x6,x12,x13,x14,x15;

 cHigh = hhf(c);            // 1
 cLow = lhf(c);
 t[0] = b[0];   t[1] = b[1];
 c1 = cHigh==mnnfd;
 if(c1) {
     aHigh = hhf(t[1]);         // 2
 } else {
     x1 = cHigh+1;              // 3
     aHigh = (nnfd)(t[1]/x1);
 }
 u = (nnd)aHigh*(nnd)cLow;
 v = (nnd)aHigh*(nnd)cHigh;
 t[0]-= tohf(u);
 x2 = mnnd-tohf(u);
 c2 = t[0]>x2;
 if(c2)   t[1]--;
 t[1]-=hhf(u);
 t[1]-=v;
 while(1) {
     x3=t[1]>cHigh;           // 4
     x4=t[1]==cHigh;
     x5=t[0]>=tohf(cLow);
     x6= x4&&x5;
     c3= x3||x6;
     if(c3) {
          t[0]-=tohf(cLow);         // 5
          x7=mnnd-tohf(cLow);
           c4=t[0]>x7;
          if(c4) t[1]--;
          t[1]-=cHigh;
          aHigh++;
     } else {   break;  };            // 6
 }
 c5 = cHigh==mnnfd
 if(c5) {
     aLow=lhf(t[1]);               // 7
 } else {
     x8=tohf(t[1])+hhf(t[0]);    // 8
     x9=cHigh+1;
     aLow=(nnfd)((nnd)x8/x9);
 }
 u = (nnd)aLow*(nnd)cLow;
 v = (nnd)aLow*(nnd)cHigh;
 t[0] -= u;     x10 = mnnd-u;
 c6 = t[0]>x10;
 if(c6) t[1]--;
t[0]-=tohf(v);
x11=mnnd-tohf(v);
c7=t[0]>x11;
if(c7) t[1]--;
t[1]-=hhf(v);
while(1) {
     x12=t[1]>0;              // 9
     x13=t[1]==0;
     x14=t[0]>=c;
     x15=x13&&x14;
     c8=x12||x15;
     if(c8) {
          x16=t[0]-=c;       // 10
          x17=mnnd-c;
          c9=x16>x17;
          if(c9) t[1]--;
          aLow++;
     } else {
          break;              // 11
     }
 }
 *a=tohf(aHigh)+aLow;
}
```

Figure 3.4. Preliminary transformed fragment of C code for RSA

A schematic representation of OBBM in the C language is given in Fig. 3.5. Fig. 3.5a depicts the framework of the model at the level of a single function. Here *RType* is a type of the function's return value; *FName* is a name of the function; *FArgs* are descriptions of the formal arguments (parameters) of the function.

FDeclarations are local declarations within a function, including declarations of data flow variables and control variables, whose values are initialized. The entire operation part of the function is represented by a single *while* loop with an infinite iteration scheme. In case of simple algorithms, the loop may be absent altogether.

The body of the loop is unified. It is a sequence of single-type constructions which are truncated conditional instructions *if-then*, describing conditions $C_1, C_2, ..., C_k$ of executing statements $Q_1, Q_2, ..., Q_k$. The expressions $C_1, C_2, ..., C_k$ are represented by scalar Boolean variables, the values of which can be calculated by the preceding operators Q_i.

If a constant representing the truth value is used instead of C_i, the execution of Q_i is unconditional, and it can be released from the condition by removing the *if*-instruction. Possible variants of statement Q_i are presented in Fig. 3.5b - 3.5g. It can be an assignment statement with a unary or binary operator ⊗, a function call with actual parameters $e_1, ..., e_n$, *break*, *return* and others. The exit from the loop is performed using *break* statement, the exit from the function is performed using *return* statement, which returns the value of the function, defined by expression *expr*.

What is new in BSBM to extract concurrency? Since several basic blocks of the source code are executed within one iteration of the equivalent BSBM, while only one basic block is executed in the structured model, the total number of iterations in the BSBM loop is always less against the structured model.

Since all operators of the source code are incorporated in BSBM, it provides a complete analysis of data dependencies between the operators, followed by identifying pairs of parallelizable operators, pairs of orthogonal mutually exclusive operators, parallel branches in the algorithm, and a critical path that allows determining the performance of the parallelized code.

```
a)  <RType>  <FName>(<FArgs>) {
      <FDeclarations>
      while (1) {
         if($C_1$) { $Q_1$ }
         if($C_2$) { $Q_2$ }
         if($C_3$) { $Q_3$ }
         …
         if($C_k$) { $Q_k$ }
      }
    }
```

```
b)   v = ⊗ u;
c)   v = u ⊗ w;
d)   v ⊗= u;
e)   v = f($e_1$,…,$e_n$);
f)   break;
g)   return expr;
```

Figure 3.5. Basic single-block model of a function in C

When analyzing dependencies, variable C_i that represents the exacution condition is added to the input operands of Q_i. Due to the extraction of external parallelism among basic blocks, the potential parallelization factor of BSBM increases, the execution of the parallel code implementation is accelerated, and the algorithm can be effectively pipelined.

We illustrate BSBM with an example algorithm that finds the greatest common divisor (GCD). The representation of the model in C language is given in Fig. 3.6. It uses five variables *C*0, ... *C*4, which control the execution of single loop with an infinite iteration scheme. The body of the loop includes eight serialy executed statements. Half of them is executed unconditionally, implicitly using value *true* instead of conditional variables.

The variables and operators which calculate the values of these variables determine the conditions for terminating GCD with *return*, conditions for continuing calculations, and methods for recalculating the values of integer variables *X* and *Y*. Due to the absence of branching in the loop body and alignment of operators in one ruler, the procedure of analyzing dependencies among operators and parallelizing operators is an effective one. It is easy to see that the following pairs of statements can be executed in parallel: (2,3), (2,4), (2,5), (2,6), (2,7), (5,6), (5, 8), (6.7), (7.8).

3.4. Transformation of loops for basic single-block model

This chapter proposes a method [70, 71, 76] of the step-by-step transformation of an arbitrary sequential algorithm presented in C or in other algorithmic language to one loop of a basic single-block model. The method guarantees the generation of a model code in a finite number of steps for any source algorithm. The method is based on the application of the following key transformations of the algorithm:

```
int GCD (int X, int Y) {
        int C0=1, C1, C2, C3, C4;
        while (1) {
                C0 = X == Y;             // 1
                if (C0) return X;        // 2
                C2 = ! C0;               // 3
                if (C2) C1 = X < Y;      // 4
                C3 = C2 && C1;           // 5
                C4 = C2 && !C1;          // 6
                if (C3) Y = Y - X;       // 7
                if (C4) X = X - Y;       // 8
        }
}
```

Figure 3.6. Basic single-block model of GCD algorithm

1. insertion into the loop of operators that are located behind the loop;
2. insertion into the loop of operators that are located before the loop;
3. merging two adjacent nested loops into one;
4. merging breaking statements while merging loops.

A nontrivial program code typically contains an arbitrary structure of loops with various iteration schemes. In this loops structure, there are usually pairs of loops that are executed sequentially and pairs of loops nested one in other. We show that for any source system of loops, one can obtain an equivalent system of nested loops with infinite iteration schemes and break statements. Such a nested loops system can be further transformed into BSBM.

The basic rules of transforming an algorithm code to BSBM make extensive use of while (1) {S} and for (;;) {S} loops with infinite iteration schemes and a sequence S of statements. The transformation rules that are proposed in previous section allow the loops of an arbitrary structure to be converted into these iteration schemes.

The basic rules M1-M5 of transforming an algorithm to BSBM that is presented in C/C++ are given in Fig. 3.7. Rule M1 ensures that statements of sequence S_4 which is located behind the loop, are inserted into the loop body that is constructed of sequences S_1, S_2 and S_3 of statements and two *break* statements executed under conditions V_1 and V_2. Two *break* statements are merged into one, and sequence S4 before the single *break*. Additional conditional variable V_3 is introduced.

Sequence S_1 of statements that are located in front of the loop with the iteration scheme *while* (1) and body S_2, is inserted into the loop by Rule M2. Sequence S_1 is put at the beginning of the loop body under the *if*-instruction using a conditional variable V that is assigned value 1 before the loop and is assigned value 0 inside the loop. Due to such control structure, sequence S_1 executes exactly once at the beginning of the first iteration of the loop.

Rule M3 is an extension of rule M2. It inserts into the loop not only the sequence S_1 of statements that located in front of the loop, but also inserts the conditional *if*-instruction, which covers this loop. In this rule, an additional conditional variable V and an additional *break* statement are introduced.

Rule M4 is a further extension of rules M2 and M3. It inserts into the loop which is located in *else*-part of the conditional instruction, everything that is in front of and inside the conditional statement. The rule shows that all statements that are located before and behind the loop can be inserted into the body of the loop.

Rule M5 merges two adjacent nested loops into one. Due to the elimination of one loop, the depth of the nested system of loops is reduced by one. We assume that in the left part of the rule, variable V appears at the moment when statements S_1 that are located before the inner loop are inserted in the body of this inner loop. The initialization operator V = 1 that is put at the previous location of statements S_1 is the only obstacle for merging two loops into one loop.

№	Фрагмент до преобразования	Фрагмент после преобразования
M1	*while*(1) { S_1 *if*(V_1) *break*; S_2 *if*(V_2) *break*; S_3} S_4	*while*(1) { S_1 *if*(!V_1) { S_2 } V_3=$V_1\vert\vert V_2$; *if*(V_3) { S_4 *break*; } S_3 }
M2	S_1 *while*(1) { S_2 }	V=1; *while*(1) { *if*(V) { S_1 V=0; } S_2 }
M3	S_1 *if*(C) { *while*(1) { S_2 } }	V=1; *while*(1) { *if*(V) { S_1 V=0; } *if*(C) { S_2 } *else break*; }
M4	S_1 *if*(C_1) { S_2 } *else* { S_3 *while*(1) { S_4 *if*(C_2) *break*; S_5 } S_6 }	V_2=1; *while*(1) { *if*(V_2) { S_1 V_1=C_1; *if*(V_1) { S_2 } *else* { S_3 } V_2=0; } *if*(V_1) { S_4 *if*(C_2) { S_6 *break*; } S_5 } *else break*; }
M5	*while*(1) { V=1; *while*(1) { *if*(V) { S_1 V=0; } S_2 *if*(C) *break*; S_3 } }	V=1; *while*(1) { *if*(V) { S_1 V=0; } S_2 *if*(C) V=1; S_3 }

Figure 3.7. Transformation rules for loops to obtain BSBM

Since the initialization statement executes after exit from the inner loop due to the execution of *break* statement, we can replace *break* with the initialization statement, eliminate the inner loop and the initialization statement standing in front of the inner loop, and move forward the external loop. As a result, we obtain the right part of Rule M5.

Any algorithm in C language, that is preliminary transformed by means of rules R1-R14 to an intermediate form, can then be transformed by means of rules M1-M5 to BSBM with one loop. To explain our technique of such a transformation, let us transform a system of nested loops to one functionally equivalent loop, and transform a sequence of loops to one equivalent loop.

Fig. 3.8 shows a technique of transforming two nested loops L1 and L2 to one loop. Inner loop L2 consists of iteration scheme *while*(1), a head sequence S_2 of statements, a *berak* statement under condition *if*(C_2), and a tail sequence R_2 of statements (Fig. 3.8a). The outer loop L1 consists of iteration scheme *while*(1), a head sequence S_1 of statements, a *berak* statement under condition *if*(C_1), a sequence R_1 of statements, the inner loop L2, and a tail sequence T_1 of statements. At the first step of transformation the technique inserts all statements of the body of loop L1, which are located before and behind L2, into loop L2 (Fig. 3.8b). Comments /*L1*/ and /*L2*/ indicate the loops associated with *break* statements.

```
L1: while(1) { S1 if(C1) break; R1
    L2: while(1) { S2 if(C2) break; R2 } T1
}
```

a)

```
L1: while(1) { V2=1;
  L2: while(1) { if(V2) { S1 if(C1) break;/*L1*/ R1 V2=0;}
    S2 if(C2) { T1 break;/*L2*/ } R2
}}
```

b)

```
V2=1; L1: while(1) {
  if(V2) { S1 if(C1) break; R1 V2=0;} S2 if(C2) { T1 V2=1;} else R2
}
```

c)

Figure 3.8. Transformation of two nested loops to one loop

At the second step of transformation the technique splits the initialization assignment "V_2=1;" into two copies, which are located before L1 and within L2. Then it merges loops L1 and L2 into one resulting loop (Fig. 3.8c).

The proposed technique of transforming two nested loops is generalized for arbitriraly number of nested loops using a method of mathematical induction. The induction step assumes that n-1 nested loops are merged in one loop, and proves that this resulting loop can be converted to a general form and then can be merged with the *n*th nested source loop.

Fig. 3.9 presents a technique of transforming two sequential loops L1 and L2 to one loop. Loop L1 consists of iteration scheme *while*(1) and a block, which includes a sequence S_1 of statements, a *berak* statement in *then*-part of conditional statement with condition C_1, and a tail sequence R_1 of statements in *else*-part (Fig. 3.9a).

The second loop L2 has the same structure. At the first step of transformation the technique inserts loop L2 into the body of loop L1. Now loop L2 is located in *then*-part of *if*-statement before *break* (Fig. 3.9b). At the second step (Fig. 3.9c), the technique separates loop L2 from the statements following it.

```
$L_1$: while(1) { $S_1$ if($C_1$) break; else {$R_1$} }
$L_2$: while(1) { $S_2$ if($C_2$) break; else {$R_2$} }
```

a)

```
$L_1$: while(1) { $S_1$ if($C_1$) {
  $L_2$: while(1) { $S_2$ if($C_2$) {break; /*L2*/} else {$R_2$} }
  break; /*L1*/ } else {$R_1$}
}
```

b)

```
$L_1$: while(1) { $S_1$ if($C_1$) {
  $L_2$: while(1) { $S_2$ if($C_2$) {break; /*L2*/} else {$R_2$} }
  } if($C_1$) {break; /*L1*/} else {$R_1$}
}
```

c)

```
$L_1$: while(1) { $V_2$=1;
  $L_2$: while(1) { if($V_2$) { $S_1$ $V_2$=0; }
        if($C_1$) { $S_2$ if($C_2$) {break; /*L2*/} else {$R_2$}} else break; /*L2*/
  }
  if($C_1$) {break; /*L1*/} else {$R_1$}
}
```

d)

```
$L_1$: while(1) { $V_2$=1;
  $L_2$: while(1) { if($V_2$) { $S_1$ $V_2$=0; }
        if($C_1$) { $S_2$ if($C_2$) {
              if($C_1$) {break; /*L1*/} else {$R_1$} break; /*L2*/ } else {$R_2$}
       } else { if($C_1$) {break; /*L1*/} else {$R_1$} break; /*L2*/}
}}
```

e)

```
$L_1$: while(1) { $V_2$=1;
  $L_2$: while(1) { if($V_2$) { $S_1$ $V_2$=0; }
     if($C_1$) { $S_2$ if($C_2$) {break; /*L1*/} else {$R_2$}} else {$R_1$ break; /*L2*/}
}}
```

f)

```
$V_2$=1;
while(1) {
    if($V_2$) { $S_1$ $V_2$=0; }
    if($C_1$) { $S_2$ if($C_2$) break; else {$R_2$}} else {$R_1$ $V_2$=1;}
}
```

g)

Figure 3.9. Transformation of two sequential loops to one loop

At the third step (Fig. 3.9d), the technique insertes sequence S1 of statements and conditional statement *if*(C_1) into the body of loop L2, using an additional variable V_2. It is assumed, statements S_2 and R_2 do not change the value of C_1.

At the fourth step (Fig. 3.9e), the *if*-statement located behind loop L2 is inserted into *then*- and *else* parts of *if*-statement that is in loop L2. At the fifth step (Fig. 3.9f), transformations are performed, which simplify the body of loop L2. Condition *if*(C_1) occors three times in loop L2, due to this two branches may be eliminated, and two subsequent *break* statements may be merged. At the sixth step (Fig. 3.9g), two nested loops can be directly merged.

The proposed technique of transforming two sequential loops is generalized for many sequential loops using a method of mathematical induction. The induction step assumes that *n*-1 sequential loops are merged in one loop, and proves that this resulting loop can be transformed to a general form and then can be merged with the *n*th sequentoal source loop.

We illustrate the proposed transformation techniques by performing the C-code transformation into BSBM of function *NN_DigitDiv* of the cryptographic RSA standard, with a preliminary converted control flow (Fig. 3.4). The partitioning of the body of function *NN_DigitDiv* into seven large basic blocks B1, ..., B7 gives the skeleton, shown in Fig. 3.10a. Fig. 3.10 presents eight steps of transformation of this skeleton fragment. The following transformation rules are implemented at these steps.

a) inserting basic block B_7 into the body of loop L_2;
b) inserting basic block B_4 and loop L_2 into the body of loop L_1, comments that represent loop labels are associated with *break* statements;
c) split of statement *if*(c_1) and separate of loop L_2 from basic blocks B_3, B_4;
d) split and insert of statement *if*(!c_1) inside of loop L_2;
e) insert of statements B_3 and *if*(c_1) *break*; inside of loop L_2;
f) insert of basic blocks B_2, B_3, B_4 inside of loop L_2;
g) elimination of loop L_2;
h) insert of basic block B_1 inside of loop L_1.

It is easy to see that the body of the resulting loop does not satisfy all the requirements of BSBM (Fig. 3.5), as the transformation process is not complete. Long conditional statements should be split into several short conditional statements. For instance, conditional statement "*if* (*u*1) {*B*1 *u*1 = 0;}" can be split into two simpler conditional statement: "*if* (*u*1) {*B*1}" and "*if* (*u*1) {*u*1 = 0;}". The nested statements "*if* (*u*2) {*B*2 *if* (*c*1) {*B*3} *else* {*B*4} *u*2 = 0;}" can be split into the statements chain "if (u2) {B2} u3 = u2 && c1; u4 = u2 &&! c1; if (u3) {B3} if (u4) {B4} if (u2) {u2 = 0;}". As a result, all basic blocks of the source code are finally located in the body of a single loop which contains one basic block. For its construction, only two additional Boolean variables *u*1 and *u*2 are used, which do not affect the internal and external parallelism of the basic blocks. The implicit dependences of basic blocks B2, B3, B4, B5, B6 and B7 on conditional variables c1 and c2 in the source code become explicit in BSBM, without reducing the amount of potential parallelism that is hidden in the original algorithm.

Fig. 3.11 shows the C-code which is obtained from the transformed skeleton fragment presented in Fig. 3.10h by means of substituting actual basic blocks in C instead of blocks symbols.

3.5. Transformation of nested branching code to basic single-block model

In BSBM the control flow is represented in a different way against the initial source code in C. Instead of nested general control structures it is a single loop with a set of assignments and *break* statements in the body, which are covered or uncovered with the short *if-then* statements. Fig. 3.12 shows an algorithm that is represented with recursive function *Split*, which is capable of splitting nested conditional instructions and generating a purely linear basic block. The split of control structures and the generation of BSBM preserve the original data flow in C-code, and convert the original control flow to additional data flow.

Recursive function *Split* has two formal parameters: block *p* and expression *c* describing the condition of executing the block. It assumes that the source C-code is a structured program which uses the objects as follows. The source code is represented as a *block* of statements.

a)

```
B1
L1: while (1) {
   B2
   if(c1) {B3} else break;
} B4
L2: while (1) { B5
   if(c2) {B6} else { B7 break;}
}
```

b)

```
B1
L1: while (1) {
 B2
 if(c1) {B3} else { B4
  L2: while (1) { B5
   if(c2) {B6} else {B7 break; /*L2*/ }
  } break; /*L1*/
}}
```

c)

```
B1
L1: while (1) { B2
 if(c1) {B3} else {B4}
 if(!c1) {
  L2: while (1) { B5
   if(c2) {B6} else {B7 break; /*L2*/ }
  }
  break; /*L1*/
 }
}
```

d)

```
B1
L1: while (1) { B2
 if(c1) {B3} else {B4}
 L2: while (1) {
  if(!c1) { B5
   if(c2) {B6} else {B7 break; /*L2*/ }
  } else break; /*L2*/
 }
 if(!c1) break; /*L1*/
}
```

e)

```
B1
L1: while (1) {
 B2
 if(c1) {B3} else {B4}
 L2: while (1) {
  if(!c1) {
   B5
   if(c2) {B6} else {B7 break; /*L1*/ }
  } else break; /*L2*/
 }
}
```

f)

```
B1
L1: while (1) { u2=1;
 L2: while (1) {
  if(u2) { B2
   if(c1) {B3} else {B4} u2=0;
  }
  if(!c1) { B5
   if(c2) {B6} else {B7 break; /*L1*/ }
  } else break; /*L2*/
 }
}
```

g)

```
B1 u2=1;
L1: while (1) {
 if(u2) {
  B2
  if(c1) {B3} else {B4}
  u2=0;
 }
 if(!c1) {
  B5
  if(c2) {B6} else {B7 break; }
 } else u2=1;
}
```

h)

```
u1=1; u2=1;
L1: while (1) {
 if(u1) {B1 u1=0;}
 if(u2) { B2
  if(c1) {B3} else {B4}
  u2=0;
 }
 if(!c1) {
  B5
  if(c2) {B6} else {B7 break; }
 } else u2=1;
}
```

Figure 3.10. Transformation of C code fragment for RSA to Basic single-block model

```
void NN_DigitDiv3(
      nnd *a, nnd b[2], nnd c) {
 nndt[2],u,v;
 nnfd aHigh,aLow,cHigh,cLow;
 int c1,c2,c3,c4,c5,c6,c7,c8,c9;
 int c10=1,c11=1;
 nnfd x1;
 nnd  x2,x7,x8,x9,x10,x11,x16,x17;
 int x3,x4,x5,x6,x12,x13,x14,x15;
 while(1) {
  if(c11) {
   cHigh=hhf(c);                    // 1
   cLow=lhf(c);                     // 2
   t[0]=b[0];                       // 3
   t[1]=b[1];                       // 4
   c1=cHigh==mnnfd;                 // 5
   if(c1) {
    aHigh=hhf(t[1]);                // 6
   } else {
    x1=cHigh+1;                     // 7
    aHigh=(nnfd)(t[1]/x1);          // 8
   }
   u=(nnd)aHigh*(nnd)cLow;          // 9
   v=(nnd)aHigh*(nnd)cHigh;         // 10
   t[0]-=tohf(u);                   // 11
   x2=mnnd-tohf(u);                 // 12
   c2=t[0]>x2;                      // 13
   if(c2) t[1]--;                   // 14
   t[1]-=hhf(u);                    // 15
   t[1]-=v;                         // 16
   c11=0;                           // 17
  }
  if(c10) {
   x3=t[1]>cHigh;                   // 18
   x4=t[1]==cHigh;                  // 19
   x5=t[0]>=tohf(cLow);             // 20
   x6=x4&&x5;                       // 21
   c3=x3||x6;                       // 22
   if(c3) {
    t[0]-=tohf(cLow);               // 23
    x7=mnnd-tohf(cLow);             // 24
    c4=t[0]>x7;                     // 25
    if(c4) t[1]--;                  // 26
    t[1]-=cHigh;                    // 27
    aHigh++;                        // 28
   } else {
    c5=cHigh==mnnfd;                // 29
    if(c5) {
     aLow=lhf(t[1]);                // 30
    } else {
     x8=tohf(t[1])+hhf(t[0]);       // 31
     x9=cHigh+1;                    // 32
     aLow=(nnfd)((nnd)x8/x9);       // 33
    }
    u=(nnd)aLow*(nnd)cLow;          // 34
    v=(nnd)aLow*(nnd)cHigh;         // 35
    t[0]-=u;                        // 36
    x10=mnnd-u;                     // 37
    c6=t[0]>x10;                    // 38
    if(c6) t[1]--;                  // 39
    t[0]-=tohf(v);                  // 40
    x11=mnnd-tohf(v);               // 41
    c7=t[0]>x11;                    // 42
    if(c7) t[1]--;                  // 43
    t[1]-=hhf(v);                   // 44
   }
   c10 = 0;                         // 45
  }
  if(!c3) {
   x12=t[1]>0;                      // 46
   x13=t[1]==0;                     // 47
   x14=t[0]>=c;                     // 48
   x15=x13&&x14;                    // 49
   c8=x12||x15;                     // 50
   if(c8) {
    x16=t[0]-=c;                    // 51
    x17=mnnd-c;                     // 52
    c9=x16>x17;                     // 53
    if(c9) t[1]--;                  // 54
    aLow++;                         // 55
   } else {
    *a=tohf(aHigh)+aLow;            // 56
    break;
   }
  } else
   c10 = 1;                         // 57
 }
}
```

Figure 3.11. C-code of basic single-block model for RSA

The *block* can be an empty block or a block of instructions. The instruction can be *if*-instruction, assignment, *break* or other instruction. The *if*-instruction consists of a condition, a *then*-part that is a *block*, and optionally an *else*-part that is also a *block*. It is assumed that the right part of assignment is an expression that is constructed of only one logical, arithmetic or other operator.

Function *Split* splits the nested and sequential conditional instructions into a single branched purely linear basic block that is constructed of the short *if-then* instructions with one operator in *then*-part and with the condition that is represented with a simple Boolean variable. It uses predicates as follows for analysis of the code:

- *is_if*(*s*) returns *true* if statement *s* is a conditional instruction, and returns *false* otherwise;
- *is_block*(*s*) returns *true* if statement *s* is a block (list of instructions), and returns *false* otherwise;

```
function Split(block p, condition c) {
  if (is_block(p)) {
    while (p != empty) {
      Split(head(p),c);  p := tail(p);
      Split(p,c);
    }
  } else if (is_if(p)) {
    ci := cond(p);  ct := generate();  s := build_and(c,ci,ct);
    add(Result, s);  split(then(p),ct);
    if (has_else(p)) {
      cn := generate();  sn := build_not(ci,cn);  add(Result, sn);
      ce := generate();  se := build_and(c,cn,ce);
      add(Result, se);  split(else(p),ce);
    }
  } else {
    othif := build_if(c,p);  add(Result, othif);
  }
}
```

Figure 3.12. Recursive algorithm of split of nested conditional instructions and generating the purely linear basic block

- *has_else*(*s*) returns *true* if conditional instruction *s* has an *else*-part, and returns *false* otherwise.

Function *Split* explores the following functions for analysis of blocks and *if*-statements:

- *head*(*p*) returns the first instruction of block *p*;
- *tail*(*p*) returns the rest instructions of block *p* or returns *empty*;
- *cond*(*s*) returns the conditional expression of *if*-instruction *s*.
- *then*(*s*) returns *then*-part of *if*-instruction *s*.
- *else*(*s*) returns *else*-part of *if*-instruction *s*.

The functions as follows are used in *Split* for synthesis of BSBM:

- *build_and*(*in1*,*in2*,*out*) returns *assign*-instruction that is built of Boolean operator *and*, Boolean input variables *in1* and *in2* and output variable *out*;
- *build_not*(*in*,*out*) returns *assign*-instruction that is built of operator *not*, input variable *in* and output variable *out*;
- *build_if*(*c*,*s*) returns statement *s* if condition *c* is *null*, otherwise it returns instruction *if-then* that is built of condition *c* and instruction *s* in *then*-part;
- *Result* is a global variable that represents a block of new instructions represented a a list that is generated by algorithm *Split* (initially the list is empty);
- *add*(*Result*, *s*) concatenates instruction *s* at the end of list *Result*;
- *generate*() returns a new Boolean variable.

A remarkable feature of BSBM is that the parallel-sequential entry of the statements of the original basic blocks into the body of the single loop of BSBM is a source of further parallelism extraction when using other methods, rules and facilities of transformation. Since the majority of statements of the BSBM's loop are under the short *if-then* instruction, rules for extracting operators from *if-then* are very attractive. The essence of the rules is as follows. Let the operators of basic block B_1 be in *then*-part of the conditional *if-then* statement, whose test variable *c*1 gets the value in basic block B_0, as shown in Fig. 3.13a. Obviously, in such a code, operators from B1 can be executed no earlier than operators from B0. Pairwise paralleling of operators from B0 and B1 is difficult.

Speculative execution is an optimization technique where a computer system performs some task that may not be needed. In order to apply such an execution to B1, we extract B1 from the *if-then* instruction, introducing some additional variables v'_1, …, v'_n, which are duplicates of resulting variables of B1, and add some reassignment statements in *then*-part (Fig. 3.13b). After such a step, more operators of B1 can execute in parallel with operators of B0. The critical path of the loop body becomes shorter.

a) **B**$_0$
if(c_1) { **B**$_1$ }

b) **B**$_0$
B$_{1'}$’
if(c_1) { v_1=v_1’;…v_n=v_n’; }

Figure 3.13. Extraction of basic block B1 from *if-then* for speculative execution

The application of the speculative execution rule (Fig. 3.13) to the RSA skeleton shown in Fig. 3.10h and to the corresponding C-code shown in Fig. 3.11 yields the BSBM code shown in Fig. 3.14. We move basic block B3 ahead of instruction *if* (c1) and move the basic block B6 ahead of instruction *if* (c2) according to this rule. The new code hereinafter referred to as TRANSF is faster than the original code of the RSA fragment, shown in Fig. 3.11.

3.6. Efficiency of basic single-block model

Let us perform a more thorough analysis of the static and dynamic parameters of the basic blocks of all versions of the parallelism extraction model, such as the original (SOURCE), structured (STRUCT), basic single-block (BSBM) and transformed basic single-block (TRANSF).

To estimate the static parameters of the bodies of all loops, as well as *then* and *else* parts of all conditional operators of the source code, we will consider them as independent basic blocks. To estimate the dynamic parameters, we will execute all the code models on the same input data.

```
void NN_DigitDiv3(
      nnd *a, nnd b[2], nnd c) {
 nndt[2],u,v,t0,t1;
 nnfd aHigh,aLow,cHigh,cLow;
 nnfd aL,aH,x1;
 int c1,c2,c3,c4,c5,c6,c7,c8,c9;
 int c10=1,c11=1;
 nnd x2,x7,x8,x9,x10,x11,x16,x17;
 int x3,x4,x5,x6,x12,x13,x14,x15;
 while(1) {
  if(c11) {
   cHigh=hhf(c);               // 1
   cLow=lhf(c);                // 2
   t[0]=b[0];   t[1]=b[1];     // 3,4
   c1=cHigh==mnnfd;            // 5
   if(c1) {
    aHigh=hhf(t[1]);           // 6
   } else {
    x1=cHigh+1;                // 7
    aHigh=(nnfd)(t[1]/x1);     // 8
   }
   u=(nnd)aHigh*(nnd)cLow; // 9
   v=(nnd)aHigh*(nnd)cHigh;  // 10
   t[0]-=tohf(u);              // 11
   x2=mnnd-tohf(u);            // 12
   c2=t[0]>x2;                 // 13
   if(c2) t[1]--;              // 14
   t[1]-=hhf(u);   t[1]-=v;    // 15,16
   c11=0;                      // 17
  }
  if(c10) {
   x3=t[1]>cHigh;              // 18
   x4=t[1]==cHigh;             // 19
   x5=t[0]>=tohf(cLow);        // 20
   x6=x4&&x5;                  // 21
   c3=x3||x6;                  // 22
   t0=t[0]-tohf(cLow);         // 23
   x7=mnnd-tohf(cLow);         // 24
   c4=t0>x7;                   // 25
   t1=t[1]-cHigh;              // 27
   if(c4)   t1--;              // 26
   aH=aHigh+1;                 // 28
   if(c3) {
    t[0]=t0;                   // 28a
    t[1]=t1; aHigh=aH;
   } else {
    c5=cHigh==mnnfd;           // 29
    if(c5) {
     aLow=lhf(t[1]);           // 30
    } else {
     x8=tohf(t[1])+hhf(t[0]);    // 31
     x9=cHigh+1;               // 32
     aLow=(nnfd)((nnd)x8/x9); // 33
    }
    u=(nnd)aLow*(nnd)cLow;// 34
    v=(nnd)aLow*(nnd)cHigh; // 35
    t[0]-=u;                   // 36
    x10=mnnd-u;                // 37
    c6=t[0]>x10;               // 38
    if(c6) t[1]--;             // 39
    t[0]-=tohf(v);             // 40
    x11=mnnd-tohf(v);          // 41
    c7=t[0]>x11;               // 42
    if(c7) t[1]--;             // 43
    t[1]-=hhf(v);              // 44
   }
   c10 = 0;                    // 45
  }
  if(!c3) {
   x12=t[1]>0;                 // 46
   x13=t[1]==0;                // 47
   x14=t[0]>=c;                // 48
   x15=x13&&x14;               // 49
   c8=x12||x15;                // 50
   x16=t[0]-c;                 // 51
   x17=mnnd-c;                 // 52
   c9=x16>x17;                 // 53
   if(c9)t1=t[1]-1;else t1=t[1];//54
   aL=aLow+1;                  // 55
   if(c8) {
    t[0]=x16;                  // 55a
    t[1]=t1; aLow=aL;
   } else {
     *a=tohf(aHigh)+aLow;      // 56
     break;
   }
  } else   c10 = 1;            // 57
 }}
```

Figure 3.14. Accelerated basic single-block model of C code fragment for RSA

Let us demonstrate the estimation of model parameters on the C-code of the *NN_DigitDiv* function (model SOURCE) that is a part of the RSA standard (the developers of this standard are Ronald Rivest, Adi Shamir and Leonard Adleman, 1977) [84], the skeleton of which, that is built on seven large basic blocks B1, ..., B7, is shown in Fig. 3.15a.

By transforming the fragment shown in Fig. 3.15a, we obtain a structured model STRUCT, presented in Fig. 3.15b. To do this, we have introduced a variable N, which takes the value of a basic block number, and have introduced a loop *for*, the iterations of which are repeated until the value of N falls outside the range of basic block numbers.

At each iteration of the loop, the *switch* statement switches to the corresponding basic block, followed by statements that determine the number of the next basic block which will be selected at the next iteration of the loop. The basic single-block model BSBM of function *NN_DigitDiv* is shown in Fig. 3.15c, and the result of its accelerating transformation (model TRANSF) is shown in Fig. 3.14.

a)

```
B1
L1: while (1) { B2
    if(c1) {B3} else break;
} B4
L2: while (1) { B5
    if(c2) {B6} else break;
} B7
```

b)

```
for(N=1; N<8; ) {
  switch(N) {
    case 1: B1  N=2; break;
    case 2: B2  if(c1) N=3; else N=4; break;
    case 3: B2  N=2; break;
    case 4: B4  N=5; break;
    case 5: B5  if(c2) N=6; else N=7; break;
    case 6: B6  N=5; break;
    case 7: B7  N=8; break;
  }
}
```

c)

```
u1 = 1;
u2 = 1;
while (1) {
    if(u1) { B1 }
    if(u1) { u1 = 0; }
    if(u2) { B2 }
    u3 = u2&&c1;
    u4 = u2&&!c1;
    if(u3) { B3 }
    if(u4) { B4 }
    if(u2) { u2 = 0; }
    u5 = !c1;
    if(u5) { B5 }
    u6 = u5&&c2;
    u7 = u5&&!c2;
    if(u6) { B6 }
    if(u7) { B7 break; }
    if(c1) { u2 = 1; }
}
```

Figure 3.15. Transform of C/C++ code fragment for RSA to structured model

These four models SOURCE, STRUCT, BSBM and TRANSF are executed on the following input data: the dividend is represented with b[0] = 717576735 nad b[1] = 2379867; the divisor is c=12345678. The obtained results are reported in Tables 3.1- 3.4.

Table 3.1

Dynamic parameters of the SOURCE model

Basic block	Statements	Exrecutions	Complexity	Critical path
1	5	1	5	5
2	1	0		
3	2	1	2	2
4	8	1	8	8
5	5	43	215	215
6	6	42	252	252
7	1	1	1	1
8	1	0		
9	3	1	3	3
10	11	1	11	11
11	5	71	355	355
12	5	70	350	350
13	1	1	1	1
Σ		233	1203	1203

Table 3.2

Dynamic parameters of the STRUCT model

Basic block	**State-ments**	**Execu-tions**	**Local critical path**	**Comple-xity**	**Total critical path**
for(;N<14;)	1	234	1	234	234
switch(N)	1	233	1	233	233
1	5	1	2	5	2
2	1	0	1		
3	2	1	2	2	2
4	8	1	6	8	6
5	5	43	3	215	129
6	6	42	4	252	168
7	1	1	1	1	1
8	1	0	1		
9	3	1	2	3	2
10	11	1	6	11	6
11	5	71	3	355	213
12	5	70	3	350	210
13	1	1	1	1	1
Σ		699		1670	1207

The dynamic parameters of the SOURCE model (Table 3.1) and STRUCT model (Table 3.2) are estimated in terms of basic blocks (Fig. 3.4), and the dynamic parameters of the BSBM (Table 3.3) and TRANSF (Table 3.4) models are described in more detail in terms of individual statements.

The parameters of a basic block are the number of its statements, the total number of their executions, the total computational complexity, measured as the total number of statements executions, and the total critical path length, while taking into account all the performances.

Table 3.3

Dynamic parameters of the BSBM model

Statement	Complexity	Critical path
1	1	1
2	1	
3	1	
4	1	
5	1	1
6		
7	1	1
8	1	1
9	1	1
10	1	1
11	1	
12	1	
13	1	
14	1	
15	1	
16	1	
17	1	
18	43	
19	43	
20	43	43
21	43	43
22	43	43
23	42	42
24	42	
25	42	42
26	42	42
27	42	42
28	42	
29	1	1
30		
31	1	1
32	1	
33	1	1
34	1	1
35	1	
36	1	1
37	1	
38	1	1
39	1	1
40	1	
41	1	
42	1	
43	1	1
44	1	1
45	43	
46	71	
47	71	71
48	71	
49	71	71
50	71	71
51	70	70
52	70	
53	70	70
54	70	70
55	70	
56	1	1
57	42	
Σ	1289	736

Table 3.4

Dynamic parameters of the TRANSF model

Statement	Complexity	Critical path
1	1	1
2	1	
3	1	
4	1	
5	1	1
6		
7	1	
8	1	1
9	1	1
10	1	1
11	1	
12	1	
13	1	
14	1	
15	1	
16	1	
17	1	
18	43	
19	43	
20	43	43
21	43	43
22	43	43
23	42	
24	42	
25	42	
26	42	
27	42	
28	42	
28a	42	42
29	1	1
30		
31	1	1
32	1	
33	1	1
34	1	1
35	1	
36	1	1
37	1	
38	1	1
39	1	1
40	1	
41	1	
42	1	
43	1	1
44	1	1
45	43	
46	71	
47	71	71
48	71	
49	71	71
50	71	71
51	70	
52	70	
53	70	
54	70	
54a	70	70
55	70	
56	1	1
57	42	
Σ	1401	469

For each basic block of the STRUCT model, the local critical path is also estimated. For each individual statement of the BSBM and TRANSF models, the contribution to the total computational complexity and the total critical path is estimated using the statements precedence graph shown in Fig. 3.16. The SOURCE code has 13 detailed basic blocks, the STRUCT code has 15 blocks, including two additional blocks which are *for* and *switch* statements. The OBBM model code and the transformed model TRANSF code have a single basic block. The total number of executions of basic blocks of the source code is 233, the structured code is three times more (699). The only basic block of BSBM and TRANSF models has 113 executions, which is equal to the number of complete iterations of the single loop.

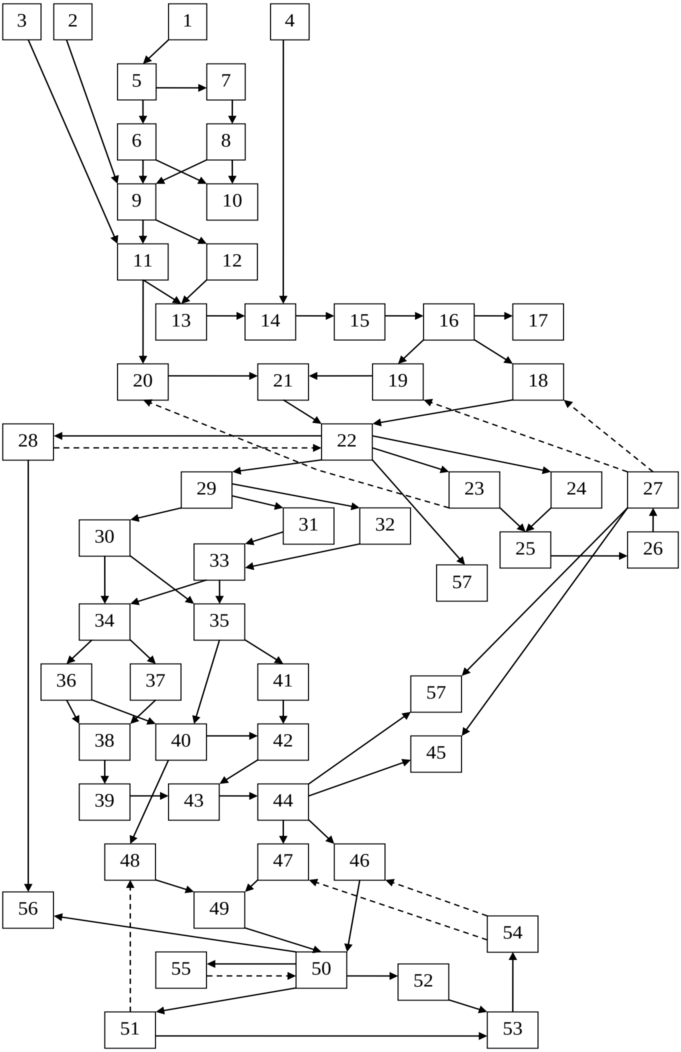


Figure 3.16. Statements precedence graph for basic single-block model

Static parameters of the models include the average number of statements in a basic block, which is equal to 4.15 and 3.73 for SOURCE and STRUCT respectively, and equal to 57 and 59 for BSBM and TRANSF respectively. Dynamic parameters include the average number of statements executions in a basic block. For models SOURCE and STRUCT it is equal to 5.15 and 2.39 respectively, and equal to 11.41 and 12.40 for BSBM and TRANSF respectively. The increase in the average number of statements executions against the static average number of statements in SOURCE is due to the frequent long runs.

The reduction of this parameter is approximately five times higher for BSBM and TRANSF, which is explained by the fact that the statements of the single block are executed only when certain conditions are met.

Static and dynamic estimates of the average number of statements on the critical path of the single basic block characterize the parallelization potential of the models. Taking into account the frequency of execution of basic blocks, these estimates give an evaluation of the total execution time of the entire code.

Since the total number of executions of all basic blocks of the BSBM is 113, we conclude that on average a 2.06 of basic blocks execute within on iteration of single loop.

At the same time, the number of iterations in the single loop of source code is 233, and the internal parallelism of the basic blocks of the source code provides a parallelization factor of 1.38.

Let's give a comparison of the parallelism extraction models, using the C-code of function *NN_DigitDiv*. Analyzing the skeleton of SOURCE model that is presented in Fig. 3.15a, it is easy to see that two *while* loops with labels L1, L2 separate the basic blocks B1, B4, B7 from each other. Basic blocks B2 and B3 of loop L1 and basic blocks B5 and B6 of loop L2 are separated by conditional instructions.

The basic blocks that are inside of a loop are also separated from the basic blocks outside the loop. Thus, the statements of different basic blocks of the source code are not mutually parallelizable.

It is easy to see that in STRUCT model shown in Fig. 3.15b, only one basic block of the source code executes at the current iteration of the loop. Consequently, the total number of iterations is equal to the sum of the numbers of block executions plus one. Along with noticeable overhead, this is a significant drawback of this model. It is impossible to perform and even more to parallelize two or more basic blocks within one iteration of the loop, although parallelization within basic block is possible. This is a serious obstacle for the subsequent use of other methods and tools for the extraction of parallelism that are based on the STRUCT model.

Both static and dynamic comparisons of parallelism extraction models are possible. The static comparison uses parameters of the control flow graphs and the data flow graphs of the models codes. The most important static parameters are the size of the graph, the length of the critical path on the graph, and others. The disadvantage of the static comparison is the inability to take into account the behavioral properties of the models in the process of solving typical problems.

As a consequence, the dynamic parameters of the models are preferable. The first dynamic parameter is the computational complexity *CCompl*(*M*) of the model *M,* which is measured in the number of executed operations of the programming language (or in the sum of the weights of the operations) on the sets of input data that are most typical at solving the problem. The second important dynamic parameter is the length of the critical path *CPath*(*M*) on the data flow graph, which is expanded during code execution. The third parameter can be calculated over the dividing the first parameter by the second one, it is a factor of potential parallelism *PFactor*(*M*) of model *M* that is calculated as

$$PFactor(M) = {}^{CCompl(M)}\!\big/_{CPath(M)} . \qquad (3.2)$$

The fourth parameter is the acceleration *Accel*(*M*) of model *M* against model SOURCE that is considered as a reference model. It does not depend on the computational complexity of both *M* and SOURCE. The acceleration can be calculated using the operation of dividing the critical path length of SOURCE by the critical path length of *M*:

$$Accel(M) = {}^{CPath(SOURCE)}\!\big/_{CPath(M)} . \qquad (3.3)$$

It is obviously, the smaller the length of the critical path, the larger the factor of parallelization potential of the model, and the higher the acceleration it yields.

Table 3.5 reports the parameters of models SOURCE, STRUCT, BSBM and TRASF which are measured on function *NN_DigitDiv*. The transition from SOURCE to STRUCT has increased the computational complexity from 1203 to 1670 statements executions, or by 40.5%. The transition from SOURCE to BSBM has implied lower growth of the computational complexity from 1203 to 1289. The transition from BSBM to TRASF has increased the computational complexity to 1401 or 8.7% higher. As for the critical path, the trend here is completely opposite to the computational complexity. The length of the critical path in the STRUCT model remains almost the same as in the SOURCE model (1207 against 1203). In the BSBM and TRASF models it is reduced against SOURCE from 1203 down to 736 and further down to 469, or by 38.8% and 60.0% respectively.

Table 3.5

Parameters of parallelism extraction models of RSA

Параметр	SOURCE	STRUCT	OBBM	TRASF
Вычислительная сложность	1203	1670	1289	1401
Критический путь	1203	1207	736	469
Коэффициент распараллеливания	1.00	1.38	1.75	2.99
Ускорение	1.00	1.00	1.64	2.57

The parallelization factor is an integrated indicator that takes into account changes both in computational complexity and in the critical path. The parallelization factor of STRUCT, BSBM and TRASF models increased by 1.38, 1.75 and 2.99 times compared to the original SOURCE model. It should be noted that in the STRUCT model, the growth is explained by increase in the computational complexity, the acceleration of computations has not occurred.

In contrast, in the BSBM and TRASF models, an acceleration of 1.64 and 2.57 times is achieved due to the extraction of parallelism.

Thus, the BSBM model with one basic block is organized in such a way, that the potential parallelism of operators is not reduced by the dominance of the sequential execution of control structures. The parallelism has effectively extracted by the split and eliminate of sequential control structures from the code.

3.7. Conclusion

A method of extraction of parallelism from a difficult to parallelize sequential algorithm is proposed. It uses a set of transformation rules and applies them step-by-step to the source code. The selection of the rules and choosing of the preferable code fragments, which have to be transformed, is performed in such a way as to obtain better parameters of the equivalent parallel code.

The transformation of the control flow of the source algorithm and its basic blocks ensure the extraction of the most important types of parallelism from hard-to-parallel loop / branching programs, which process arbitrary data types and are built using *while*, *do-while*, *if*, *switch* and other statements.

A basic single-block flow model (BSBM) of the algorithm, that is constructed of a single loop whose body includes a single basic-block, provides real extraction of parallelism of many types from the source code. The model provides efficient techniques for analyzing dependencies among statements, identifying pairs of parallelizable operators, pairs of orthogonal mutually exclusive operators, and parallel paths in the algorithm. It allows for efficient estimation of the computatioinal complexity, critical path length and parallelization factor of the code.

A step-by-step transformation method of an arbitrary sequential algorithm to the basic single-block flow model is proposed. It guarantees obtaining a model code in a finite number of steps for any source algorithm, based on universal transformation rules such as inserting operators before and after a loop, merging sequential and nested loops in one loop, eliminating loops' *continue* statements and merging *break* statements into one *break*, and others.

A technique of evaluating the static and dynamic parameters (including the level of hidden parallelism) of algorithm models has been developed. It is applied to the source, structured, single-block and transformed single-block models. The influence of the parameters of the basic blocks on the parameters of the whole models, including the degree of implicit potential parallelism of blocks and models is shown.

A detailed comparison of parameters of the four parallelism extraction models on the RSA standard is carried out. It proves the possibility of a significant reduction in the critical path length, of an increase of the parallelization factor, and an increase in the acceleration factor of the basic single-block flow model, and further modifications of this model against the known models.

4. ANALYSIS OF BASIC SINGLE-BLOCK MODEL

4.1. Goals of analysis

The main goal of analysis is the precise estimation of the computational complexity, critical path and parallelization potential of the basic single-block model. The estimation crucially depend on finding out data dependences among statements within one iteration of the loop and between consecutive loop iterations. In its turn, data dependences analysis cannot be performed without finding the statements which are mutually exclusive, and which of them are not. Mutually exclusive statements cannot be data dependent.

In this work, the algorithm (program code) transformation and analysis techniques are essentially base on single assignment model of a variable. It requires that each variable to be used once, although it may have several producers and several consumers. All producers must belong to mutually exclusive branches of nested conditional statements within one iteration of the loop and among several loop iterations.

4.2. Analysis of structured basic single-block model

The structured program improves the clarity, quality, and development time of an algorithm [11, 13]. It is constructed by use of the structured control flow constructs of selection (*if-then-else*), repetition (*while* and *for*), block structures, and procedures and functions. It explicitly defines all pairs of mutually exclusive operators over branches of nested if-then-else statements.

Example 4.1. Fig. 4.1 presents an example C/C++ code of structured program that is constructed of one loop and three branching statements. The loop contains data flow feedback. All pairs of mutually exclusive operators can be easily seen. All of them belong to one iteration of the loop. These cannot be seen for different loop iterations. Fig. 4.2 shows the equivalent basic single-block model that is a result of transformation of the source code. The iteration scheme of the loop is quite simple, but the number of short *if-then* statements equals 17 that is much larger than the number of *if-then-else* statements in the source code. It is difficultly to find out in the basic single-block model what pairs of operators are mutually exclusive and what of them are not.

```
void DataFlowFeedback(float *A, float *B, float *C, int n) {
        float a0, b0, c0, d1, d0 = 0.0f;
        for (int i = 0; i < n; ++i) {
                a0 = A[i];
                b0 = B[i];
                if (a0 == b0) {
                     c0 = 0.0f;
                } else {
                     d1 = a0 - b0;
                     if (b0 < 0)   c0 = d1 + 1.0f;
                     else {
                             if (d0 > 0)   c0 = d1 - 1.0f;
                             else   c0 = d0;
                     }
                     d0 = d1;
                }
                C[i] = c0;
        }
}
```

Figure 4.1. Example C/C++ structured looping/branching code with dataflow feedback

Given two *if-then* statements “*if* (t_i) S_i” and “*if* (t_j) S_j”, how to find out, wither S_i and S_j are mutually exclusive or not? It is clear that everything depends on the Boolean conditional variables t_i and t_j. In addition to value *false* (0), each of them can take value *true* (1). If both variables can take value 1 simultaneously, then S_i and S_j are not mutually exclusive. Therefore, S_i and S_j are mutually exclusive if the pair (t_i, t_j) can only take values 00, 10 and 01.

4.2.1. Evaluating conditional variables using Boolean expressions

Let T be a set of conditional Boolean variables occurred in *if-then* statements of the basic single-block model. The conditional variables determine the control flow of the single loop body. Let B be a set of other Boolean variables of the model which are used for evaluating the conditional variables. The set of primary Boolean variables is denoted as *P*. The basic single-block model evaluates these variables mostly using relational operators.

```
void DataFlowFeedback(float *A, float *B, float *C, int n) {
    float a0, b0, c0, d1, d0=0.0f;
    int i = 0;
    bool t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13;
    while (true) {
        t0 = i < n;                         // 1
        t1 = ! t0;                          // 2
        if(t1)   break;                     // 3
        if (t0)   a0 = A[i];                // 4
        if (t0)   b0 = B[i];                // 5
        if (t0)   t2 = a0 == b0;            // 6
        if (t0)   t3 = ! t2;                // 7
        t8 = t0 && t2;                      // 8
        if (t8)   c0 = 0.0f;                // 9
        t9 = t0 && t3;                      // 10
        if (t9)   d1 = a0 - b0;             // 11
        if (t9)   t4 = b0 < 0;              // 12
        if (t9)   t5 = ! t4;                // 13
        t10 = t9 && t4;                     // 14
        if (t10)   c0 = d1 + 1;             // 15
        t11 = t9 && t5;                     // 16
        if (t11)   t6 = d0 > 0;             // 17
        if (t11)   t7 = ! t6;               // 18
        t12 = t11 && t6;                    // 19
        if (t12)   c0 = d1 - 1;             // 20
        t13 = t11 && t7;                    // 21
        if (t13)   c0 = d0;                 // 22
        if (t9)   d0 = d1;                  // 23
        if (t0)   C[i] = c0;                // 24
        if (t0)   ++i;                      // 25
    }
}
```

Figure 4.2. Example structured basic single-block model with dataflow feedback

Intermediate Boolean variables of set $B \setminus P$ are evaluated over primary variables, and conditional variables are evaluated over primary and intermediate variables.

Example 4.2. There are 8 conditional Boolean variables in the basic single-block model shown in Fig. 4.2. They belong to set $T = \{t0, t1, t8, t9, t10, t11, t12, t13\}$. Set $B = \{t0, t2, t3, t4, t5, t6, t7\}$ of additional 7 Boolean variables helps to evaluate the conditional variables. It includes 4 primary variables, i.e. $P = \{t0, t2, t4, t6\}$. It also includes 3 intermediate variables, i.e. $B \setminus P = \{t3, t5, t7\}$.

We can evaluate all conditional variables with Boolean expressions, which can be extracted from the code. Thus conditional variable *t*0 is evaluated with expression "$i<n$" over relational operation, therefore it is simultaneously a primary variable. Conditional variable *t*1 is evaluated with expression "$\neg t0$" over Boolean negation. Conditional variable *t*11 is evaluated with expression "$t9 \wedge \mathrm{t5}$" over Boolean conjunction. In its turn, variable *t9* is evaluated with expression "$t0 \wedge \mathrm{t3}$", variable *t*5 is evaluated with expression "$\neg t4$", and variable *t*3 is evaluated with expression "$\neg t2$". After substitution, variable *t*11 can be evaluated with expression "$t0 \wedge \neg t2 \wedge \neg t4$" over three primary variables. Variable *t*11 takes values which depend on the combinations of values of variables *t*0, *t*2 and *t*4.

4.2.2. Relations among values of primary Boolean variables

Dependences among values of primary Boolean variables strongly influence dependences among values of conditional Boolean variables. These dependences can be described with relations on tuples of Boolean values.

When the values of conditional variable *t*11 are analyzed (Fig. 4.2), relations among values of primary variables *t*0, *t*2 and *t*4 must be considered. These relations are associated with the feasible bit-vector values of the primary variables which can appear during code execution. Expressions "$i < n$", "$a0 == b0$" and "$b0 < 0$" evaluate variables *t*0, *t*2 and *t*4 and, as can be easily seen, are independent. It means their vector value can take any tuple from 000 to 111.

Matrix *F* of feasible values of pairs of primary variables is represented with Equation (4.1).

$$F = \begin{array}{c|cccc|} t_1 & 9 & f_{12} & \cdots & f_{1k} \\ t_2 & f_{21} & 9 & \cdots & f_{2k} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ t_k & f_{k1} & f_{k2} & \cdots & 9 \end{array}. \qquad (4.1)$$

In matrix *F*, rows and columns correspond to primary variables t_i; feasibility characteristic function f_{ij} is a binary Boolean function $f(t_i,t_j)$ which determines a four-bit vector, which is encoded with decimal numbers 0…15, and whose elements show the feasibility (value 1) or infeasibility (value 0) of the corresponding two-bit values of primary variables t_i and t_j. Value 9 in the principal diagonal of matrix *F* represents the equivalence (≡) binary Boolean function (1001).

Matrix *F* allows computation of feasible bit-vector values of *n* variables over feasible values of variable pairs.

Example 4.3. For our example primary variables *t*0, *t*2 and *t*4, matrix *F* is as follows:

$$F = \begin{array}{c|ccc|} t_0 & 9 & 15 & 15 \\ t_2 & 15 & 9 & 15 \\ t_4 & 15 & 15 & 9 \end{array}.$$

In this matrix, value 15 (1111) represents binary Boolean function Constant 1 which means that all values 00, 01, 10 and 11 of pair (t_i,t_j) are feasible during execution of the code. Bellow we will see that f_{ij} can be other binary Boolean function.

4.2.3. Pairs of orthogonal conditional variables

We assume that two conditional Boolean variables t_i and t_j depend on the same set of primary variables. They are orthogonal if they never take value 1 simultaneously. In other words, if t_i takes 1 then t_j takes 0, and if t_j takes 1 then t_i takes 0.

In the structured basic single-block model, the orthogonal pairs of conditional variables can be found out using Boolean logic. Thus source conditional statement "*if* (t0) S_i *else* S_j" produces in the single-block model the code with additional Boolean variable t1, assignment statement and two short *if-then* statements: "t1=!t0; *if* (t0) S_i *if* (t1) S_j". It is easy to see that variables t0 and t1 are orthogonal, and two short *if-then* statements with S_i and S_j inside are mutually exclusive. Nested conditional statements "if (t0) *if* (t1) S_i *else* S_j" produce code "t2 = ! t1; t3 = t0 && t1; t4 = t0 && t2; *if* (t3) S_i; *if* (t4) S_j", which explores three additional Boolean variables. Variable pairs (t0, t3) and (t0, t4) are not orthogonal, while variables t3 and t4 are orthogonal as t1 and t2 are orthogonal.

In general form, the orthogonal condition can be represented with the following logical equation:

$$\forall p \ \big((t_i \to \neg t_j) \wedge (t_j \to \neg t_i)\big), \tag{4.2}$$

where $\neg$ is Boolean negation, $\wedge$ is Boolean conjunction, $\to$ is Boolean implication, p is a vector of primary variables which t_i and t_j depend on, and $\forall$ is a universal quantifier (for all p) which ties variables of p. After substitution of the evaluating expressions instead of variables t_i and t_j, Equation 4.2 can be expressed in terms of primary variables.

Example 4.4. Let us consider pair t9 and t13 of conditional variables. The expression of evaluating t9 is "$t0 \wedge \neg t2$" and the expression of evaluating t13 is "$t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6$". After substituting these expressions in Equation (4.2) we obtain:

$$((t0 \wedge \neg t2) \to \neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6)) \wedge$$
$$((t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6) \to \neg(t0 \wedge \neg t2)) =$$

$$(\neg(t0 \wedge \neg t2) \vee \neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6)) \wedge$$
$$(\neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6) \vee \neg(t0 \wedge \neg t2)) =$$

$$\neg t0 \vee t2 \vee t4 \vee t6.$$

The inferred disjunction of four literals is not Boolean function Constant 1, therefore variables t9 and t13 are not orthogonal.

Example 4.5. Now we consider variables $t10$ and $t13$. The expression of evaluating $t10$ is “$t0 \wedge \neg t2 \wedge \mathrm{t}4$” and the expression of evaluating $t13$ is “$t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6$”. After substituting these expressions in Equation (4.2) we obtain:

$$
\begin{aligned}
&((t0 \wedge \neg t2 \wedge \mathrm{t}4) \rightarrow \neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6)) \wedge \\
&((t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6) \rightarrow \neg(t0 \wedge \neg t2 \wedge \mathrm{t}4)) = \\
&(\neg(t0 \wedge \neg t2 \wedge \mathrm{t}4) \vee \neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6)) \wedge \\
&(\neg(t0 \wedge \neg t2 \wedge \neg t4 \wedge \neg t6) \vee \neg(t0 \wedge \neg t2 \wedge \mathrm{t}4)) =
\end{aligned}
$$

$$
\neg t0 \vee t2 \vee t4 \vee \neg t4 \vee t6 = 1.
$$

The inferred disjunction has literals $t4$ and $\neg t4$ as operands and is equivalent to Boolean function Constant 1, therefore variables $t10$ and $t13$ are orthogonal.

Matrix *Ort* (Equation 4.3) describes all pairs of orthogonal and all pairs of non-orthogonal conditional variables of the example basic single-block model. This matrix completely determines all pairs of mutually exclusive operators in the basic single-block mode.

$$
Ort = \begin{array}{r|cccccccc|}
t_0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
t_8 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
t_9 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
t_{10} & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
t_{11} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
t_{12} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
t_{13} & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}. \qquad (4.3)
$$

4.2.4. *Estimating parameters of basic single-block model*

Analysis of the basic single-block model yields computational complexity, critical path and parallelization potential of the block. These parameters depend on input data of the algorithm.

Various aspects of the model are important while estimating computational complexity: the cost of operators, the cost of control structures, and the cost of memory operations. The complexity of operators is derived directly from the behavioral description. The complexity of control structures and memory operations depends on the implementation method of algorithm and on the basic parallel architecture.

Example 4.6. The loop body shown in Fig. 4.2 includes 25 statements which contain only 4 arithmetic operators and 4 relational operators. It includes 10 low cost logic scalar operators and 17 *if-then* statements. Only 6 *if-then* statements may not be removed; one of them covers *break* operator, 4 of them select one value of 4 producers for variable *c*0, and another one select new value for state variable *d*0 which has two producers: operator 22 and the current state value. The model also includes 24 assignments and 3 array indexing operators.

The basic single-block model includes many *if-then* statements which allow flexible reordering of statements in the block and efficient partition of the model. Both pipelined and non-pipelined partitioning can be accomplished. Most of *if-then* statements need no implementation in the target architecture. In hardware architecture, only *if-then* statements are saved and implemented which describe more than one producers of a variable; other *if-then* statements may be removed from the model. In software architecture, *if-then* statements with identical conditional variable can be merged into one *if-then* statement, and one *if-then-else* statement can be generated of several *if-then* statements, thus reducing the amount of computations on one processor.

Now we transform the basic single-block model shown in Fig. 4.2 to a model for hardware implementation which is presented in Fig. 4.3. We remove 11 *if-then* statements which may be omitted without changing the algorithm behavior. We extract computations from 2 of 4 producers of variable *c*0 and merge these producers to one long conditional statement which can be implemented with multiplexor. Input and output variables of statements and matrix *Ort* of orthogonal conditional variables (Equation 4.3) which determine mutually exclusive operators are the basis for data dependences analyses. The main rule is as follows. If conditional variables t_i and t_j are orthogonal according to *Ort*, then statements "*if* (t_i) $y=x$;" and "*if* (t_j) $z=y$;" are independent, although variable *y* is output of the first statement and is input of the second statement. If t_i and t_j are not orthogonal, then the statements are data dependent.

Fig. 4.4 shows the data dependency graph for the transformed basic single-block model presented in Fig. 4.3. State variables *i* and *d*0 constitute dataflow feedback in the graph. Their values produced in one iteration of the loop are consumed in next loop iteration.

```
void DataFlowFeedback(float *A, float *B, float *C, int n) {
        float a0, b0, c0, d1, d0=0.0f;
        int i = 0;
        bool t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13;
        while (true) {
                t0 = i < n;                                         // 1
                t1 = ! t0;                                          // 2
                if(t1)  break;                                      // 3
                a0 = A[i];                                          // 4
                b0 = B[i];                                          // 5
                t2 = a0 == b0;                                      // 6
                t3 = ! t2;                                          // 7
                t8 = t0 && t2;                                      // 8
                t9 = t0 && t3;                                      // 9
                d1 = a0 - b0;                                       // 10
                t4 = b0 < 0;                                        // 11
                t5 = ! t4;                                          // 12
                t10 = t9 && t4;                                     // 13
                t11 = t9 && t5;                                     // 14
                t6 = d0 > 0;                                        // 15
                t7 = ! t6;                                          // 16
                t12 = t11 && t6;                                    // 17
                t13 = t11 && t7;                                    // 18
                c00 = d1 + 1;                                       // 19
                c01 = d1 - 1;                                       // 20
                if (t8) c0 = 0; else if (t10) c0 = c00; else
                        if (t12) c0 = c01; if (t13) c0 = d0;        // 21
                if (t9)  d0 = d1;                                   // 22
                C[i] = c0;                                          // 23
                ++i;                                                // 24
        }
}
```

Figure 4.3. Transformation of example model for hardware implementation

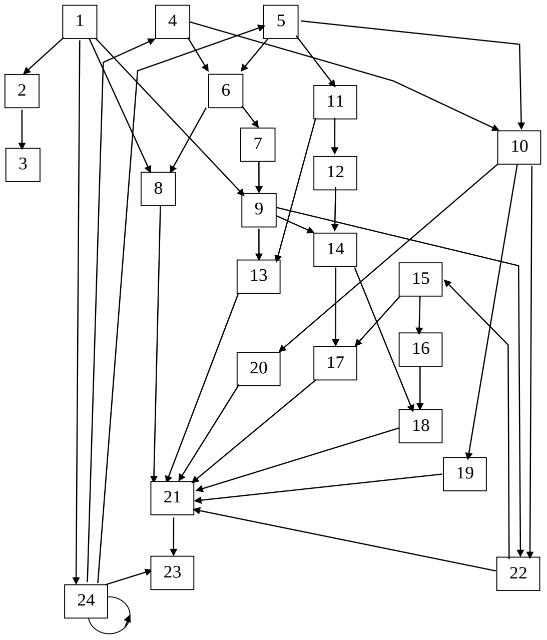


Figure 4.4. Data dependency graph of example basic single-block model in Fig. 4.3

The data dependencies among statements allow evaluation of the critical path and parallelization potential of the code. These parameters may vary depending on input data. In any case, the parallelization potential of code shown in Fig. 4.3 is higher than that one shown in Fig. 4.2.

4.3. Advanced analysis of basic single-block model

Very often programmers do not write purely structured code although this code is close to structured one. Fig. 4.5 shows an example of such code. This code indicates mutually exclusive branches rather with conditional expressions than with *then* and *else* alternatives. Such expressions intensively use relational operators.

The loop body contains at top level two *if-then* statements whose conditional expressions are constructed of variables *a*0 and *b*0, and are constructed of two relational operators “>” and “<=”. In its turn, the second *if-then* statement contains two additional *if-then* statements whose conditional expressions are constructed of two relational operators “<=” and “>”, and are constructed of variable *a*0 and literals 0 and 4.

Fig. 4.6 presents the basic single-block model that is functionally equivalent the source code. The logic part of the model differs from that of the previous model. Thus statement 18 describes a conditional expression for the fourth producer of variable *c*0. This producer must be mutually exclusive against three previous producers.

Analysis of such basic single-block model differs from the above considered structured basic single-block model.

```
void relationalOperators(float *A, float *B, float *C, int n) {
        float a0, b0, c0, d0 = 3;
        for (int i = 0; i < n; ++i) {
                a0 = A[i];
                b0 = B[i];
                if (a0 > b0) {
                        c0 = a0 - d0 * b0;
                }
                if (a0 <= b0) {
                        c0 = b0 - a0;
                        if (a0 <= 0) {
                                c0 = b0 + 1;
                        }
                        if (a0 > 4) {
                                c0 = a0 - 5;
                        }
                }
                C[i] = c0;
                d0 = c0;
        }
}
```

Figure 4.5. Example C/C++ non-structured code with dataflow feedback

4.3.1. Feasibility functions for pairs of primary Boolean variables

Section 4.3.2 proves that the feasibility function for two independent primary Boolean variables is Boolean constant 1. Various kinds of dependency between primary variables may exist in program code (Fig. 4.6). In this section, we analyze relational operators ==, !=, >, >=, < and <=from this point of view.

```
void relationalOperators_(float *A, float *B, float *C, int n) {
        float a0, b0, c0, c1, d0 = 3;
        int i = 0;
        bool t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11;
        while (true) {
                t0 = i < n;                              // 1
                t1 = ! t0;                               // 2
                if (t1) break;                           // 3
                if (t0)   a0 = A[i];                     // 4
                if (t0)   b0 = B[i];                     // 5
                if (t0)   t2 = a0 > b0;                  // 6
                t7 = t0 && t2;                           // 7
                if (t7)   c1 = d0 * b0;                  // 8
                if (t7)   c0 = a0 - c1;                  // 9
                if (t0)   t3 = a0 <= b0;                 // 10
                t8 = t0 && t3;                           // 11
                if (t8)   t4 = a0 <= 0;                  // 12
                t9 = t8 && t4;                           // 13
                if (t9)   c0 = b0 + 1;                   // 14
                if (t8)   t5 = a0 > 4;                   // 15
                t10 = t8 && t5;                          // 16
                if (t10)   c0 = a0 - 5;                  // 17
                if (t8)   t6 = ! (t4 || t5);             // 18
                t11 = t8 && t6;                          // 19
                if (t11)   c0 = b0 - a0;                 // 20
                if (t0)   C[i] = c0;                     // 21
                if (t0)   d0 = c0;                       // 22
                if (t0)   ++i;                           // 23
        }
}
```

Figure 4.6. Example non-structured basic single-block model with dataflow feedback

Let primary Boolean variables *t*i and *t*j be assigned values with two assignment statements, which contain binary relational operators in the right part whose input variables are identical (Equation (4.4)): *x* and *y* are numerical (may be other type of) variables, and R_i and R_j are relational operators. As many as 36 combinations of R_i and R_j are possible. All of them are represented with matrix *RV* shown in Fig. 4.7.

$$RV = \begin{array}{c|cccccc|} & = & != & < & <= & > & >= \\ = & 9 & 6 & 14 & 13 & 14 & 13 \\ != & 6 & 9 & 11 & 7 & 11 & 7 \\ < & 14 & 13 & 9 & 13 & 14 & 6 \\ <= & 11 & 7 & 11 & 9 & 6 & 7 \\ > & 14 & 13 & 14 & 6 & 9 & 13 \\ >= & 11 & 7 & 6 & 7 & 11 & 9 \end{array}$$

Figure 4.7. Feasibility functions for relational operators with identical input variables

Element $f_{Ri,Rj}$ of the matrix is a function that determines the feasibility of vector values of pair (t_i, t_j). Function 6 is Boolean exclusive or, function 7 is disjunction, functions 11 and 13 are implication, and function 14 is Sheffer stroke (negation of conjunction).

$$t_i = x\ R_i\ y; \qquad (4.4)$$
$$t_j = x\ R_j\ y;$$

Example 4.7. For instance, the feasibility function for operators >= and > and statements (4.4) has decimal code 11 that equivalent to binary code 1011. It means that pair (t_i, t_j) of Boolean variables can take values 00, 10 and 11, and cannot take value 01.

Indeed, if x >= y equals 0, then x > y cannot be equal to 1. Similarly, the feasibility function for operators <= and >= has decimal code 7 (binary code 0111). Value 00 is infeasible as x <= y equals 0 implies x >= y equals 1. The code in Fig. 4.6 contains statements "*t*2 = *a*0 > *b*0;" and "*t*3 = *a*0 <= *b*0;". The feasibility function for > and <= (Fig. 4.7) has code 6 (0110), as values 00 and 11 are infeasible, and values 01 and 10 are feasible.

Let variables t_i and t_j be assigned values by two assignment statements with relational operators, one identical variable and two different literals in the right part:

$$t_i = x\ R_i\ l_i; \qquad (4.5)$$

$$t_j = x\ R_j\ l_j;$$

where l_i and l_j are numerical literals that satisfy inequality $l_i < l_j$. Matrix *RL* (Fig. 4.8) describes feasibility functions for statements (4.5) and all 36 pairs of relational operators.

Example 4.8. For instance, the feasibility function (Fig. 4.8) for operators = and > has decimal code 14 (binary code 1110). It means that pair (t_i, t_j) of Boolean variables can take values 00, 01 and 10, and cannot take value 11. Indeed, at $l_i < l_j$, if x = l_i equals 0, then x > l_j equals 0 at x < l_i. Moreover, if x = l_i equals 1, then x > l_j cannot be equal to 1. The code in Fig. 4.6 contains statements "*t*4 = *a*0 <= 0;" and "*t*5 = *a*0 > 4;". The feasibility function for <=and > has code 14 (1110) in matrix *RL,* as values 00, 01 and 10 are feasible while value 11 is infeasible.

It should be noted that the feasibility functions that are located on principal diagonal of matrix *RL* may be different.

$$RL = \begin{array}{c|cccccc|} & = & != & < & <= & > & >= \\ = & 14 & 13 & 13 & 13 & 14 & 14 \\ != & 11 & 7 & 7 & 7 & 11 & 11 \\ < & 14 & 13 & 13 & 13 & 14 & 14 \\ <= & 14 & 13 & 13 & 13 & 14 & 14 \\ > & 11 & 7 & 7 & 7 & 11 & 11 \\ >= & 11 & 7 & 7 & 7 & 11 & 11 \end{array}$$

Figure 4.8. Feasibility functions for relational operators with identical input variable and different numerical literals

4.3.2. Feasibility functions and pairs of orthogonal variables

The orthogonal condition for two conditional variables t_i and t_j can be represented with the following logical equation:

$$\forall p\ \ \big(\lambda(p) \rightarrow \big((t_i \rightarrow \neg t_j) \wedge (t_j \rightarrow \neg t_i)\big)\big), \qquad (4.6)$$

where p is a vector of primary Boolean variables, and $\lambda(p)$ is a conjunction of feasibility functions for all pairs of conditional variables. Equation (4.6) means that in case $\lambda(p)$ is true we need a proof that $(t_i \rightarrow \neg t_j) \wedge (t_j \rightarrow \neg t_i)$ is true, and in case $\lambda(p)$ is false we need no any such proof.

There are five primary Boolean variables $t0$, $t2$, $t3$, $t4$ and $t5$ in the code shown in Fig. 4.6. Equation (4.7) describes feasibility functions for all pairs of these variables.

$$F = \begin{array}{c|ccccc|} t_0 & 9 & 15 & 15 & 15 & 15 \\ t_2 & 15 & 9 & 6 & 15 & 15 \\ t_3 & 15 & 6 & 9 & 15 & 15 \\ t_4 & 15 & 15 & 15 & 9 & 14 \\ t_5 & 15 & 15 & 15 & 14 & 9 \end{array} \,. \qquad (4.7)$$

We can represent functions 9 and 15 with Boolean constant 1. The matrix is a basis for construction of the conjunction of feasibility functions:

$\lambda(p) = (t2 \oplus t3) \wedge \neg(t4 \wedge t5)$.

Example 4.9. Let us check, if conditional variables $t7$ and $t8$ are orthogonal. The code shown in Fig. 4.6 allows the derivation of evaluating expressions for $t7$ and $t8$:

$t7 = t0 \wedge t2$,
$t8 = t0 \wedge t3$.

Substitution of these expressions in (4.2) yields the logical equation as follows:

$(t2 \oplus t3) \wedge\neg(t4 \wedge t5) \rightarrow$
$((t0 \wedge t2 \rightarrow \neg(t0 \wedge t3)) \wedge (t0 \wedge t3 \rightarrow \neg\, (t0 \wedge t2))) =$
$(t2 \wedge t3) \vee \neg t2 \vee \neg t3 \vee (t4 \wedge t5) \vee t0 = 1$.

As it can be seen, this equation is equivalent to Boolean function Constant 1, therefore variables $t7$ and $t8$ are orthogonal.

Example 4.10. Let us now consider conditional variables $t9$ and $t11$:

$t9 = t8 \wedge t4 = t0 \wedge t3 \wedge t4$
$t11 = \mathrm{t}8 \wedge \mathrm{t}6 = t0 \wedge t3 \wedge \neg t4 \wedge \neg t5.$

Equation (4.2) for these variables is as follows:

$(t2 \oplus t3) \wedge \neg(t4 \wedge t5) \rightarrow$
$((t0 \wedge t3 \wedge t4 \rightarrow \neg(t0 \wedge t3 \wedge \neg t4 \wedge \neg t5)) \wedge$
$(t0 \wedge t3 \wedge \neg t4 \wedge \neg t5 \rightarrow \neg\, (t0 \wedge t3 \wedge t4))) =$

$\neg(t2 \oplus t3) \vee (t4 \wedge t5) \vee \neg t0 \vee \neg t3 \vee \neg t4 \vee t4 \vee t5 = 1.$

The inferred disjunction contains literals $\neg t4$ and $t4$ and is equivalent to Boolean function Constant 1, therefore variables $t9$ and $t11$ are orthogonal.

Matrix *Ort* (4.8) describes all pairs of orthogonal conditional variables in the code shown in Fig. 4.6.

$$Ort = \begin{array}{r|ccccccc|}
t_0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
t_1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
t_7 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
t_8 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
t_9 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
t_{10} & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
t_{11} & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}. \qquad (4.8)$$

Matrix *Ort* determines all pairs of mutually exclusive operators which are covered by 16 *if-then* statements (Fig. 4.6).

There are other cases when a set of primary variables may not take arbitrary vector value. All the cases and corresponding rules for computing the feasibility functions are accumulated in a data base and are used during analysis of the basic single-block model.

4.4. Formal method of basic single-block model analysis

Now we formulate in general form the equations and tasks that are used for analysis of the basic single-block model. Let $t=t_1,\dots t_n$ be a vector of conditional Boolean variables, $p=p_1,\dots,p_k$ be a vector of primary Boolean variables, $g=g_1(p),\dots,g_n(p)$ be a vector of Boolean functions that evaluate conditional variables over primary variables, and $F = \{f(p_i, p_j) \mid i, j = 1,\dots,k, i<j\}$ is a set of feasible functions for values of pairs of primary variables.

4.4.1. Tautology problem for a pair of conditional variables

For two conditional variables t_i and t_j, whose evaluating functions are $g_i(p)$ and $g_j(p)$, the orthogonal condition can be represented with Boolean Equation (4.9), if all pairs of primary variables are mutually independent:

$$\forall p \left((g_i(p) \to \neg g_j(p)) \wedge (g_j(p) \to \neg g_i(p))\right). \tag{4.9}$$

This equation can be transformed to the equation as follows:

$$\forall p \ \left[\mu(p)\right], \tag{4.10}$$

where $\mu(p) = \neg g_i(p) \vee \neg g_j(p)$. Equation (4.10) represents a logical tautology. It must be solved in terms of primary variables which are independent. Any vector value of primary variables must satisfy the equation. Solving this equation may have high computational complexity and consume huge CPU time, if the number of primary variables grows significantly.

4.4.2. Partial tautology problem for orthogonal variables

If at least one pair of primary variables is dependent, the conjunction $\lambda(p)$ of Boolean feasibility functions is not equivalent to Boolean constant 1. In this case, the characteristic function λ(p) describes the set of feasible vector values of primary variables p:

$$\lambda(p) = \underset{\substack{i,j\in\{1\ldots k\}\\ i<j}}{AND} f(p_i, p_j). \tag{4.11}$$

A relaxation of Equation (4.10) is possible, as only a part of primary variables values is feasible. The orthogonal condition for two conditional variables t_i and t_j with evaluating functions $g_i(p)$ and $g_j(p)$ can be represented with Boolean equation (4.12):

$$\forall p \ \left(\lambda(p) \to \mu(p)\right). \tag{4.12}$$

In fact, Equation (4.12) is a partial tautology as we do not need a proof of μ(p) ≡ 1 if λ(p) = 0. At the same time, the procedure of traversal all vector values of p has very high computational complexity. We can avoid this procedure by reformulating tautology (4.12) to a satisfiability (SAT) problem.

4.4.3. Contradiction procedure and SAT problem for orthogonal variables

Very often it is easier to solve the orthogonal variables problem by means of transition to an inverse problem. Applying the first De Morgan's law to expression (4.12), we obtain:

$$\neg\,\exists p \ \neg\left(\lambda(p) \to \mu(p)\right), \tag{4.13}$$

where ∃ is an existential quantifier (there exists p) which ties variables of p. After substituting the evaluating functions instead of μ(p), replacing implication with disjunction and applying the second De Morgan’s law we have:

$$\neg\,\exists p \ \neg\left(\neg\lambda(p) \vee \mu(p)\right) =$$

$$\neg\exists p\ \left(\lambda(p)\wedge\neg\mu(p)\right) =$$

$$\neg\exists p\ \left(\lambda(p)\ \wedge g_i(p)\wedge g_j(p)\right). \qquad (4.14)$$

Such approach to solving the problem is called a contradiction procedure, and Equation (4.14) formulates a satisfiability problem. To perform objection of (4.14), it is sufficient to find a vector value of *p* which satisfies function $\lambda(p)\wedge g_{\mathrm{i}}(p) \wedge g_{\mathrm{j}}(p)$. In case such value does not exist, this function is equivalent to Boolean constant 0, and problems (4.14) and (4.12) are solved.

Example 4.11. For example, we apply the problem (4.14) to proving that conditional variables *t*7 and *t*8 are orthogonal in the code shown in Fig. 4.6. Substitution of expression $(t2\oplus t3)\wedge\neg(t4\wedge t5)$ instead of $\lambda(p)$, expression $t0\wedge t2$ instead of $g_{\mathrm{i}}(p)=t7$, and expression $t0\wedge t3$ instead of $g_{\mathrm{j}}(p)=t8$ in (4.14) leads to transformations as follows:

$$\neg\ [((t2 \oplus t3) \wedge\neg(t4 \wedge t5) \wedge t0 \wedge t2 \wedge t0 \wedge t3)]\ =$$
$$\neg\ [(t2 \wedge \neg t3 \wedge\neg(t4 \wedge t5) \wedge t0 \wedge t2 \wedge t3) \vee$$
$$(\neg t2 \wedge t3 \wedge\neg(t4 \wedge t5) \wedge t0 \wedge t2 \wedge t3)]\ =$$

$$\neg\ [0 \vee 0]\ = \neg\ [0]\ =\ 1.$$

The source expression is equivalent to disjunction of two conjunctions. The first conjunction contains opposite literals $\neg t3$ and *t*3 and is equal to 0. The second conjunction contains opposite literals $\neg t2$ and *t*2 and is also equal to 0. As a result, expression (4.14) is equal to 1, and variables *t*7 and *t*8 are orthogonal

4.4.4. Problem solving over minimization of partial functions

Using a pair of completely specified functions μ(p) and λ(*p*) we can construct a partial (incompletely specified) Boolean function

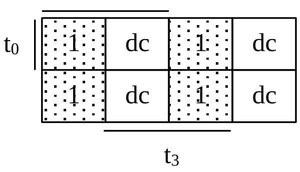
$$\varphi(p)=(\mu(p),\ \lambda(p)). \qquad (4.15)$$

This function depends on Boolean arguments p and can take three values: 0, 1 and *dc* (don’t care value). Boolean function $\neg\mu(p)\wedge\lambda(p)$ describes off-set $\varphi^{\text{off}}(p)$ of partial function $\varphi(p)$. Boolean function $\mu(p)\wedge\lambda(p)$ describes on-set $\varphi^{\text{on}}(p)$ of function $\varphi(p)$. Boolean function $\neg\lambda(p)$ describes don’t-care-set $\varphi^{\text{dc}}(p)$ of function $\varphi(p)$.

Solving the orthogonal variables problem can be performed by minimization of function $\mu(p)$ through appropriate changing its value with new value 0 or 1 on those values a of variable p for which $\lambda(a)$ is false. If new function $\mu'(p)$ is Boolean constant 1, then conditional variables t_{i} and t_{j} are orthogonal, otherwise they are not orthogonal. The minimization of function can be performed with Karnaugh map.

For example, for conditional variables $t7$ and $t8$ function $\mu(p)$ is described with expression $\neg(t0\wedge t2)\vee\neg(t0\wedge t3)$, and function $\lambda(p)$ is described with expression $(t2\oplus t3)\wedge\neg(t4\wedge t5)$. Function $\mu(p)$ depends on three variables $t0$, $t2$ and $t3$, meanwhile function $\lambda(p)$ depends on four variables $t2$, $t3$, $t4$ and $t5$. Function $\mu(p)$ does not depend on $t4$ and $t5$, therefore we omit them in new function $\lambda'(p) = t2\oplus t3$. The on-set of $\lambda'(p)$ is larger than that one of $\lambda(p)$. This is a guarantee for the correct minimization of $\varphi(p)$.

Example 4.12. Fig. 4.9 shows the Karnaugh map of partial function $\varphi'(p) = (\mu(p), \lambda'(p))$. It is easy to see, function $\mu(p)$ can be replaced with Boolean constant 1 in $\varphi(p)$, and therefore conditional variables $t7$ and $t8$ are proved to be orthogonal.



1	dc	1	dc
1	dc	1	dc

Figure 4.9. Karnaugh map of partial function $\varphi'(p)$

4.4.5. Orthogonal subsets of the set of conditional variables

Let $C = \{c_1 \ldots c_m\}$ be a subset of all set of the Boolean conditional variables that are represented with vector *t*. Variables of *C* are orthogonal if only one of them can take value 1, while the others take value 0 at any state of the code execution. This is formalized with expressions as follows.

$$\forall p \underset{i=1,\ldots,m}{AND}\left(\left(\lambda(p) \rightarrow \eta_i(p)\right)\right), \tag{4.16}$$

where

$$\begin{aligned} &\eta_i(p) = c_i(p) \rightarrow \\ &\quad \left(\neg c_1(p) \wedge \ldots \wedge \neg c_{i-1}(p) \wedge \neg c_{i+1}(p) \wedge \ldots \wedge \neg c_m(p)\right). \end{aligned} \tag{4.17}$$

Solving the orthogonal problem for *m* conditional variables is equivalent to solving the orthogonal problem of all non-ordered pairs of these variables. All subsets of orthogonal conditional variables can be determined from matrix *Ort*, for example, from matrix (4.8). Thus subset *C* = {*t*1, *t*7, *t*9, *t*10, *t*11} is maximal one for matrix (4.8). The set of all such subsets can be considered as a set of cliques of a non-directed graph that is represented with matrix *Ort*.

4.5. Analysis of basic single-block model with control flow feedback

The conditional variables and *if-then* statements define the control flow within one iteration of the single loop of the basic single-block model. The dataflow feedback can influence the control flow implicitly over recalculating conditional variables which depend on dataflow variables at each iteration of the loop. Very often, algorithms obtain the property of control flow feedback. In this case, conditional variables are global with respect to the loop, and their values are recalculated within the loop body.

Example 4.13. Fig. 4.10 shows an example C/C++ looping/ branching code with control flow feedback. Boolean variables *s*0 and *s*1, which are initialized in their declaration, represent control flow state in the loop body. The body recalculates the values of the variables in one iteration. Fig. 4.11 presents a basic single-block model that is derived from the example code by means of equivalent code transformation.

```
void ControlFlowFeedback(float *A, float *B, float *C, int n) {
    float a0, b0, c0, d1;
    bool s0 = true;
    bool s1 = false;
    for (int i = 0; i < N; ++i) {
        a0 = A[i];
        b0 = B[i];
        d1 = a0 - b0;
        if (d1 > 0) {
            if (s0) c0 = d1 - 2;
            if (s1) {
                c0 = d1;
                s0 = true;   s1 = false;
            }
        } else {
            if (s0) {
                c0 = d1;
                s0 = false;   s1 = true;
            }
            if (s1) c0 = d1 + 2;
        }
        C[i] = c0;
    }
}
```

Figure 4.10. Example C/C++ looping/branching code with control flow feedback

A problem is how to recognize, what pairs of conditional variables are orthogonal and what pairs are not in the model. Our focus is on control flow state variables, as the analysis technique for other type of pairs of conditional variables we have already considered and developed. Regarding the example basic single-block model, our focus is on state variables *s*0 and *s*1. It should be noted that although these variables are conditional in the source C/C++ code (Fig. 4.10), they are rather intermediate than conditional in the basic single-block model (Fig. 4.11). The orthogonal problem remains in any case.

```
void ControlFlowFeedback_(float *A, float *B, float *C, int n) {
        float a0, b0, c0, d1;
        bool s0 = true;
        bool s1 = false;
        int i = 0;
        bool t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11;
        while (true) {
                t0 = i < n;
                t1 = ! t0;
                if (t1)   break;
                if (t0)   a0 = A[i];
                if (t0)   b0 = B[i];
                if (t0)   d1 = a0 - b0;
                if (t0)   t2 = d1 > 0;
                if (t0)   t3 = !t2;
                if (t0)   t4 = t2 && s0;
                if (t0)   t5 = t2 && s1;
                t8 = t0 && t4;
                if (t8)   c0 = d1 - 2;
                t9 = t0 && t5;
                if (t9)   c0 = d1;
                if (t9)   s0 = true;
                if (t9)   s1 = false;
                if (t0)   t6 = t3 && s0;
                t10 = t0 && t6;
                if (t10)   c0 = d1;
                if (t10)   s0 = false;
                if (t10)   s1 = true;
                if (t0)   t7 = t3 && s1;
                t11 = t0 && t7;
                if (t11)   c0 = d1 + 2;
                if (t0)   C[i] = c0;
                if (t0)   ++i;
        }
}
```

Figure 4.11. Example basic single-block model with control flow feedback

Below we develop a formal method that is based on mathematical induction technique. This technique requires two cases to be proved. The base case proves that the orthogonal variables property holds for the initial computational state before entering the loop. The induction step proves that, if the property holds for one loop iteration *l*, then it holds for the next iteration l+1. The base case uses the initial values of the state variables. The induction step uses the statements of the loop body which update the values of state variables.

Let s_i and s_j be two Boolean control flow state variables whose orthogonal property we are going to prove.

Base case. If one of two variables is initialized to 1 and other variable is initialized to 0, then the orthogonal property holds.

Induction step. Represent the values of state variables s_i and s_j in next iteration of the loop with s'_i and s'_j. These variables can be evaluated over primary variables and current-iteration variables s_i and s_j, using evaluating Boolean functions $g_i(q)$ and $g_j(q)$. Vector q represents both primary variables and state variables within current iteration.

If the orthogonal variables property holds for variables s_i and s_j at the current iteration of loop, then our goal is to prove that it holds at the next iteration for variables s'_i and s'_j. This can be represented with implication

$$\begin{aligned} &\forall p \left((\neg s_i \vee \neg s_j) \rightarrow (\neg s'_i \vee \neg s'_j)\right) \; = \\ &\forall p \left((\neg s_i \vee \neg s_j) \rightarrow (\neg g_i(q) \vee \neg g_j(q))\right) \; . \end{aligned} \tag{4.18}$$

If expression (4.18) is a tautology, i.e. is satisfied for any value of p, then state variables s_i and s_j are orthogonal.

The C/C++ code, and in particular statements "*if* (t_b) s_i= e_b;" which assign a new value to state variable s_i, are a source for construction of evaluating function $g_i(q)$:

$$g_i(q) = \left[\bigvee_{b=1}^{B} (t_b \wedge e_b)\right] \vee \left[s_i \wedge \neg \bigvee_{b=1}^{B} t_b\right], \tag{4.19}$$

where B is the number *if-then* statements for s_i. The first term of disjunction in (4.19) represents new value of s_i, and the second term represents current value.

In case, the primary variables are dependent, and the feasibility function $\lambda(p)$ is not equivalent to Boolean constant 1, expression (4.18) for the orthogonal state variables property should be modified in order to take into account two situations: when the values of state variables are updated in the loop body, and when they save previous values:

$$\forall p \begin{pmatrix} (\neg s_i \vee \neg s_j) \rightarrow \\ (\neg(\lambda(p) \wedge g_i(q) \vee \neg\lambda(p) \wedge s_i) \vee \\ \neg(\lambda(p) \wedge g_j(q) \vee \neg\lambda(p) \wedge s_j)) \end{pmatrix}. \qquad (4.20)$$

Expression (4.20) describes that variables s_i and s_j can update their values when λ(p) is true, and these variables do not change their values when λ(p) is false.

Example 4.14. Let us prove the orthogonal property for state variables $s0$ and $s1$ of the code shown in Fig. 4.11.

It is easy to see, that in *base case* the initial values 1 (*true*) and 0 (*false*) of the variables are orthogonal.

In *induction step*, we assume that $s0$ and $s1$ are orthogonal in current iteration, and the Boolean function $\neg s0 \vee \neg s1$ is true.

Function g_{s0} that evaluates new value of $s0$ is

$$g_{\mathrm{s0}} = t0 \wedge (t9{\wedge}1 \vee t10{\wedge}0 \vee \neg(t9{\vee}t10) \wedge s0) \vee \neg t0{\wedge}s0 =$$
$$= t0{\wedge}t2{\wedge}s1 \vee t2{\wedge}s0{\wedge}\neg s1 \vee \neg t0{\wedge}s0.$$

Function g_{s1} that evaluates new value of $s1$ is

$$g_{\mathrm{s1}} = t0 \wedge (t9{\wedge}0 \vee t10{\wedge}1 \vee \neg(t9{\vee}t10) \wedge s1) \vee \neg t0{\wedge}s1 =$$
$$= t0{\wedge}\neg t2{\wedge}s0 \vee \neg t2{\wedge}\neg s0{\wedge}s1 \vee \neg t0{\wedge}s1.$$

In the C/C++ code, all primary variables are independent, therefore λ(p) = 1 and the orthogonal property for $s0$ and $s1$ holds when (4.18) is tautology. Performing transformations for a sub-expression of (4.18) we obtain:

$$\neg g_{\mathrm{s0}} \vee \neg g_{\mathrm{s1}} = \neg\, (t0{\wedge}t2{\wedge}s1 \vee t2{\wedge}s0{\wedge}\neg s1 \vee \neg t0{\wedge}s0) \vee$$
$$\neg\, (t0{\wedge}\neg t2{\wedge}s0 \vee \neg t2{\wedge}\neg s0{\wedge}s1 \vee \neg t0{\wedge}s1) =$$
$$= t0 \vee \neg s0 \vee \neg s1.$$

The whole expression (4.18) for $s0$ and $s1$ can be written and transformed as:

$$(\neg s0 \vee \neg s1) \rightarrow (t0 \vee \neg s0 \vee \neg s1) =$$
$$s0 \wedge s1 \vee t0 \vee \neg s0 \vee \neg s1 = 1.$$

This expression is equivalent to Boolean constant 1, therefore state variables $s0$ and $s1$ are orthogonal in induction step.

Finally we can conclude that state variables $s0$ and $s1$ are orthogonal in the code shown in Fig. 4.11.

It should be noted, the proposed method can be generalized for more than two state variables. For many variables, the orthogonal property can be represented using expressions (4.16) and (4.17).

4.6. Conclusion

The evaluation of the computational complexity, critical path and parallelization potential of an algorithm that is represented with the basic single-block model is more complicated against the algorithm structural model where mutually exclusive operators are described explicitly. This is due to implicit dependences among statements in the basic single-block model.

To find the mutually exclusive short *if-then* statements, a logic analysis technique has been proposed that is capable of determining pairs of orthogonal conditional variables. For structured BSBM, it uses a mechanism of the conditional variable evaluation with Boolean expressions and functions, relations among values of primary Boolean variables, and a definition of the orthogonal condition with a tautological logical equation.

To perform the advanced analysis of basic single-block model, a concept of feasibility functions for pairs of primary Boolean variables has been introduced, which allows for determining values combinations the primary variables can take during algorithm execution. The formulation of the orthogonal condition for conditional variables is extended taking into account the feasibility functions.

In the case of basic single-block model with control flow feedback, a mathematical induction technique is proposed for determining orthogonal pairs of conditional variables.

5. SYNTHESIS AND OPTIMIZATION OF COMPUTATIONAL PIPELINES

This chapter introduces a new methodology for pipeline synthesis with applications to data flow high level system design. The pipeline synthesis is applied to BSBM whose operators are translated into graphs and dependencies relations that are then processed for the pipeline architecture optimization. For each pipeline-stage time, firstly a minimal number of pipeline stages is determined and then an optimal assignment of operators to stages is generated with the objective of minimizing the total pipeline register size. The obtained optimal pipeline schedule is automatically transformed into a pipeline structure that then can be synthesized to efficient hardware implementations. Two new pipeline scheduling techniques, i.e., a least cost search branch and bound technique, and a heuristic technique have been developed. The first technique yields global optimum solutions for middle size designs, whereas the second one generates close-to-optimal solutions for large designs. Experimental results on FPGA designs show that the total pipeline registers size gain in a range up to 4.68x can be achieved. The new algorithms overcome the known ASAP and ALAP techniques concerning the amount of pipeline registers size by up to 100% on average.

5.1 Computational pipelines

Pipelining is a well-known, efficient and effective way of increasing the operating frequency and the associated throughput of data intensive digital systems in various application fields. A pipelined system is usually described by an appropriate concurrent design language.

Pipelining can be seen as the transformation of a source behavioral specification into a functionally equivalent description, which partitions all operators into pipeline-stage-fragments that are executed in time-sliced fashion. Complex digital systems are typically characterized by irregular structures, thus it is impossible to perform a straightforward mapping of the behavioral specification into a pipeline implementation.

In this chapter, an approach for the transformation of an irregular complex digital system, which is described in a system description language, into pipelined implementations achieving increased operating frequency after hardware synthesis is developed. Therefore, this chapter develops an efficient pipelining model for large digital system designs implying several "low cost" chained operators in one basic processing block, which takes into account key parameters of behavioral elements including the variable sizes, the operators delay, the relations on the set of variables and operators, and finally the mutually exclusive operators handling. The approach does not explore the resource sharing for both functional units and registers as such option would result in the serialization of the operator execution and would cause a slowdown of the overall dataflow implementation.

Two optimization problems are crucial for this approach: the component selection and the optimization of the pipeline registers. The first problem is thoroughly developed in [3]. The second problem is attacked in [77-82]. It is the main subject of this chapter. We explicitly define the whole pipeline solutions space and propose an efficient heuristic optimization algorithm which is capable of pipelining large designs with the objective of minimizing the overall registers size and to increase the operating frequency after register transfer level (RTL) synthesis.

This chapter is organized as follows: firstly it provides an overview of related works on pipeline synthesis and optimizations. Secondly, it describes the methodology based on dataflow pipeline synthesis. Thirdly, it presents the relations and associated graphs for the pipeline modeling and its optimization. Fourthly, it describes the time constrained optimization for pipelines. Then it presents a new heuristic algorithm which speeds up the optimization process for very large designs. And finally, experimental results are reported for several video processing applications, which are followed by conclusions.

5.2 Pipelining of algorithms

In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one [3], [17]. The elements of a pipeline are executed in a time-sliced fashion; in such case, pipeline registers are inserted in between pipeline stages. The pipeline stage time has to be larger than the longest delay between pipeline stages. A pipelined system requires more resources than one that executes one batch at a time, because its stages cannot reuse the resources of a previous stage. Numerous languages and intermediate representations have been created for describing pipelines, among them can be mentioned the programming language C [89], [51], [12], data flow graphs (DFG) [9], signal flow graphs [28], [88], transactional specifications [50] and other notations [39], [29], [61]. Pipelines can also be synthesized directly from binaries [51]. CAL is a formal dataflow language that was recently developed and standardized to address the goal of high-level system specification and design, particularly addressing the wide field of streaming applications [16], [49]. The concurrent algorithmic language, CAL is capable of representing pipelined networks of actors.

A pipeline system is characterized by several parameters such as the clock cycle time, the stage cycle time, the number of pipeline stages, the latency, the data initiation interval, the turnaround time and the throughput. The pipeline synthesis problem can be constrained either by resources or time, or by a combination of both.

An important concept in the pipelining circuit is retiming, which exploits the ability to move registers in the circuit so as to decrease the length of the longest path and preserve its functional behavior [40], [44], [85]. The concept is based on the assumption that the pipeline structure has already been fixed and considers only the problem of adding pipeline buffers with the objective of improving the performance.

The work of Sehwa [53] can be considered as the first pipeline synthesis program. It minimizes the time delay using a modified list scheduling algorithm with a resource allocation table. The force directed scheduling that has been proposed in [54] and modified in [88], [25] performs a time-constrained functional pipelining. ATOMICS [20] performs loop optima-zation starting by estimating a latency and an inter-iteration precedence. The pipelined DSP data-path synthesis system called SODAS [28], receives a signal flow graph as input and generates a trade-off for the pipeline designs by changing the synthesis parameters of the data initiation interval, the clock cycle time and the number of pipeline stages. In [86] an adaptation of the ASAP list scheduling and the iterative modulo scheduling are used for the design space exploration based on slow, but area efficient modules, and fast, but area consuming modules. Speculative loop pipelining from binaries, proposed in [51], speculatively generates a pipeline netlist at compile time and modifies it according to the result of the runtime analysis. The automatic pipelining proposed in [50] requires user-specified pipeline-stage boundaries and synthesizes a pipeline which allows the concurrent execution of multiple overlapped transactions in different stages. Integer linear programming formulations of the pipeline optimization problem, as an efficient approach for the design space exploration, are also presented in [10], [26], [23].

Pipelining is an effective method for optimizing the execution of loops. The loop winding method is proposed in Elf [18]. The percolation based scheduling [56] deals with the loop winding by starting with an optimal schedule [2] that is obtained without considering resource constraints. The PLS pipelining [24] is another effective method to optimize loops for DSP. The rotation scheduling of loop pipelining by means of the retiming the processing nodes is introduced in [9]. The pipeline vectorization method, based on pipelining the inner most loops in a loop nest by removing the vector dependences, is proposed in [89].

The problem of pipeline scheduling of DFGs for a variable number of pipeline stages under throughput constraints is addressed in [4], [35], [3]. The macro pipelining based scheduling technique [4] aims at pipelining heterogeneous multiprocessor systems. The number of pipeline stages is identified during the scheduling and the pipeline cycle delay is minimized in two steps. The first step finds a global coarse solution by using the ratio cut partitioning, and the second step improves the result by repartitioning the solution. The ratio balances the load on processors with the communication traffic in the interconnection network. This model cannot be directly applied to high-level synthesis with the objective of register size minimization as it is based on pure timing model.

A novel pipeline decomposition tree (PDT) based scheduling framework at system level is presented in [35]. It groups the tasks into clusters and groups the clusters into partitions which are assigned to pipeline stages. Partitions at different depth levels of the PDT can be flexibly configured to generate various stage-length pipelines. The equations that are used for decomposing the cluster dependency graph into two subgraphs equalize the partitions with regard to execution times and inter cluster dependences within one stage. The cluster-partition concept does not aim at minimizing the data transfer between adjacent pipeline stages and cannot be directly used for pipeline register minimization, as our method presented in this chapter can do.

The cost-optimized algorithm for selecting the components and pipelining a DFG, given a library of multiple implementations of operators and latency constraint, is presented in [3]. The algorithm starts by mapping each operator to the fastest component and then slows down operators by mapping them to progressively slower components in order to balance the use of slow and fast components and minimize the total cost. At each slowdown the algorithm partitions the DFG into a minimal number of stages to meet the stage delay constraint. Then it traverses the graph in downward and upward directions and accumulates delays in order to associate pipeline registers with edges. In comparison with [3] our methodology does not consider the component selection, but exhaustively minimize the register size over all the pipeline stages for the selected component implementations. The ASAP and ALAP algorithms constructed on the operator conflict graph in this book are similar to the downward and upward direction traversal algorithms.

Several previous works, including [19], [15] have discussed the relationship between design scheduling and register size. Most of them are devoted to the non-pipelined designs and all of them exploit resource sharing. The interdependent heuristic code scheduling technique and the DAG-driven register allocator are proposed in [19]. Such method reduces stage delays of a given pipeline for the given number of general purpose registers. Contrary to our work it explores intensively the resource sharing for registers and pipelined functional units keeping a constant size for the variables.

Modulo scheduling followed by stage scheduling is another efficient technique for exploiting instruction level parallelism in loops [15]. The stage scheduling performs exhaustive and heuristic searches of minimum register requirements for one modulo schedule by shifting operations by multiples of the initiation interval cycles. The resource sharing is used twice, for functional units and also for registers. The stage scheduling processes only a restricted part of the whole solution space as modifications of only one modulo schedule are considered. Differently from [15], our work searches for the fastest pipeline schedule at a minimum pipeline register cost over all the solution space for large dataflow hardware designs at various stage counts without any resource sharing.

Since deep-submicron silicon technology provides large amounts of available resources, faster pipelines without (or with minimal) sharing of resources can be synthesized with advantages in performance, without incurring in too much penalties in terms of additional silicon surface. The pipeline optimization model proposed in [34] is based on precise mathematical formulation of the optimization problem which uses the coloring of vertices of an operator conflict directed graph and an explicit stack mechanism for optimal solution search.

5.3 Pipelining data flow programs

For the pipelining methodology introduced here, it is also important that, as described in [69], a method of transforming a mixed control-data flow high-level behavioral description to a purely dataflow description consisting of the single basic block by means of elimination of control structures is available. Therefore, here the emphasis is again on efficient and affective techniques of pipeline synthesis and optimization that are based on the single basic block model. Resource sharing approaches are not employed, but pipeline scheduling for chained operators is exploited intensively. These two basics are well associated with FPGA based synthesis of pipelines from DFGs with many low cost operators describing random logic. In this book the focus is on pipelines with only one clock cycle for each stage (Fig. 5.1).

5.4 Modeling pipelines with relations and graphs

5.4.1 Relations and graphs on sets of operators, variables and pipeline stages

The dataflow program under pipelining is transformed to an acyclic DFG ([48], [7]). After that the DFG is analyzed. The analysis of DFG results in a number of relations and other graphs which constitute a basis for the creation of pipeline optimization methodologies.

Let $N=\{1,\ldots,n\}$ be a set of algorithm operators, $M=\{1,\ldots,m\}$ be a set of algorithm variables including input and output tokens and $S=\{1,\ldots,k\}$ be a set of pipeline stages. A set of input variables of operator $i=1,\ldots,n$ is denoted as $in(i)$ and a set of its output variables is denoted as $out(p)$.

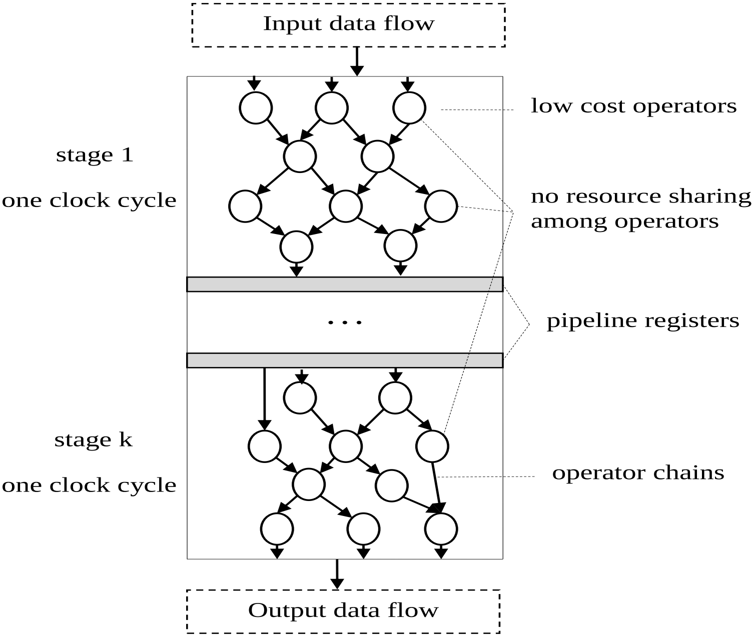


Figure 5.1. Pipeline scheduling with chaining and without resource sharing

From the sets, a set $cons(j)\subseteq N$ of consumers and a set $prod(j)\subseteq N$ of producers is being computed for each variable $j\in M$.

The *operator precedence relation P* describes a partial order on the set of operators that is derived from the analysis of data dependences between operators in DFG. The *operator direct precedence relation* P_{direct} is computed as minimal anti-transitive relation of precedence relation P. This relation also represents the *direct precedence graph*, G_P. In this book only acyclic graphs G_P are considered.

The *pipeline stage time* T_{stage} is defined as a worst case delay of all operator chains within one stage. In pipeline optimization, the time can be treated as pipeline constraint. The constraint essentially influences the pipeline frequency, throughput and load of equipment.

The longest path delays between operators constitute a basis for defining pipeline constraints and are called the *lengths of longest paths on the operator direct precedence graph*. A matrix $L=\{l_{i,j} \mid l_{i,j}\in\Re, i,j\in N\}$ of dimension $n\times n$ describes the delays. As graph G_p is DAG matrix L can be computed in a polynomial time. For DFG shown in Fig. 5.2 and for its elements that are described in Table 5.1, matrix L is given in Fig. 5.3.

In Table 5.1, for the operand size of 8 the relative delays of "+", "–", *bitand* and *bitxor* operators are taken as 1.0, 1.1, 0.1 and 0.1 respectively. For other operand sizes the operator relative delays are recalculated using a linear timing model. Constants are not listed in this table. The additive timing model has been used, although we consider more complex timing models of operators and paths.

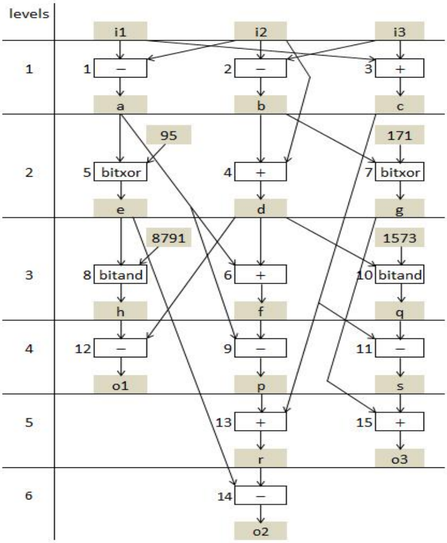


Figure 5.2. Example data flow graph consisting of 15 operators and 18 variables. Variables i1, i2 and i3 are inputs and variables o1, o2 and o3 are outputs

$$L = \begin{vmatrix}
2.20 & 0.00 & 0.00 & 0.00 & 2.30 & 4.20 & 0.00 & 2.40 & 6.40 & 0.00 & 0.00 & 4.88 & 8.02 & 10.50 & 0.00 \\
 & 1.65 & 0.00 & 3.40 & 0.00 & 5.40 & 1.75 & 0.00 & 7.60 & 3.50 & 5.43 & 5.88 & 9.23 & 11.70 & 6.68 \\
 & & 2.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 3.93 & 0.00 & 3.63 & 6.10 & 5.18 \\
 & & & 1.75 & 0.00 & 3.75 & 0.00 & 0.00 & 5.95 & 1.85 & 3.78 & 4.23 & 7.57 & 10.05 & 5.03 \\
 & & & & 0.10 & 0.00 & 0.00 & 0.20 & 0.00 & 0.00 & 0.00 & 2.68 & 0.00 & 2.58 & 0.00 \\
 & & & & & 2.00 & 0.00 & 0.00 & 4.20 & 0.00 & 0.00 & 0.00 & 5.82 & 8.30 & 0.00 \\
 & & & & & & 0.10 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.35 \\
 & & & & & & & 0.10 & 0.00 & 0.00 & 0.00 & 2.58 & 0.00 & 0.00 & 0.00 \\
 & & & & & & & & 2.20 & 0.00 & 0.00 & 0.00 & 3.83 & 6.30 & 0.00 \\
 & & & & & & & & & 0.10 & 2.03 & 0.00 & 0.00 & 0.00 & 3.28 \\
 & & & & & & & & & & 1.93 & 0.00 & 0.00 & 0.00 & 3.18 \\
 & & & & & & & & & & & 2.48 & 0.00 & 0.00 & 0.00 \\
 & & & & & & & & & & & & 1.63 & 4.10 & 0.00 \\
 & & & & & & & & & & & & & 2.48 & 0.00 \\
 & & & & & & & & & & & & & & 1.25
\end{vmatrix}$$

Figure 5.3. Longest paths matrix *L* for the dataflow graph in Fig. 5.2

Table 5.1

Elements of example data flow graph

Operators			Variables			
N	Type	Relative delay	N	Name	Mode	Size
1	–	2.20	1	i1	in	16
2	–	1.65	2	i2	in	12
3	+	2.00	3	i3	in	12
4	+	1.75	4	a	loc	16
5	bitxor	0.10	5	b	loc	10
6	+	2.00	6	c	loc	13
7	bitxor	0.10	7	d	loc	14
8	bitand	0.10	8	e	loc	18
9	–	2.20	9	f	loc	16
10	bitand	0.10	10	g	loc	6
11	–	1.93	11	h	loc	18
12	–	2.48	12	p	loc	13
13	+	1.62	13	q	loc	14
14	–	2.48	14	r	loc	13
15	+	1.25	15	s	loc	10
			16	o1	out	17
			17	o2	out	14
			18	o3	out	10

Operator conflict relation. If in matrix L the value of $l_{i\,j}$ is larger than T_{stage} we say that there is a pipeline stage conflict between operators i and j. In order to avoid the conflict, such operators must be scheduled to different pipeline stages. The operator conflict relation is a set $C = \{(i,j) \mid i,j{\in}N, l_{\text{ij}}{>}T_{\text{stage}}\}$. Inclusion $C{\subseteq}P$ holds for this relation. In pipeline scheduling, C may be replaced with its minimal anti-transitive version C^a.

Operator nonconflict relation. It is defined as $Cn{=}P\backslash C$. Inclusion $\varnothing{\subseteq}Cn{\subseteq}P$ holds for the relation. In pipeline scheduling Cn may be replaced with its minimal anti-transitive version Cn^a. For matrix L and $T_{\text{stage}}{=}3.825$ the conflict C and nonconflict Cn relations are presented in Fig. 5.4.

Operator conflict graph. The conflict relation, C^a describes a minimal ant-transitive operator conflict graph, $G_{\text{C}}{}^{\text{a}}$. A set of direct predecessors of operator p in the graph will be denoted as *cdpred*(p) and a set of direct successors will be denoted as *cdsucc*(p).

An operator minimal ant-transitive nonconflict graph, G_{Cn} is created in a similar way. A set of direct predecessors of operator p in the graph will be denoted as *ncdpred*(p) and a set of direct successors will be denoted as *ncdsucc*(p). To speed up the optimization process, we consider only minimal anti-transitive operator conflict and nonconflict graphs.

$$C = \left|\begin{array}{ccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & 0 & 1 & 0 \\
 & & & & & & & & & & & & & 0 & 0 \\
 & & & & & & & & & & & & & & 0
\end{array}\right|
\qquad
Cn = \left|\begin{array}{ccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & & & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 & & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & & & & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & & & & & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & & & & & & & & & 0 & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & & 0 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & & & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & 0 & 0 & 0 \\
 & & & & & & & & & & & & & 0 & 0 \\
 & & & & & & & & & & & & & & 0
\end{array}\right|$$

Figure 5.4. Example matrices C and C_{n} for the dataflow graph in Fig. 5.2

Mapping of operators onto pipeline stages is defined as *stage*: $N \rightarrow S$. According to the mapping, $s=stage(p)$ is a stage $s \in S$ assigned to operator $p \in N$.

5.4.2 Number of pipeline stages versus stage time

The number l of pipeline stages is defined by the length of a longest path in G_C. For l-stage pipeline a minimum stage time is denoted as $T_{stage}(l)$. The stage time for l is larger than the stage time for $l+1$. Therefore all pipelines that are generated for a stage time T_{stage} from the range $F_{stage}(l+1) \leq T_{stage} < T_{stage}(l)$ have the same number l of stages as shown in Fig. 5.5. In order to generate all possible pipelines all values of elements l_{ij} that occur in matrix L have to be used as stage time. We denote the least stage time that is equal to a largest operator delay as T_{min} and denote the largest stage time that is equal to the length of a longest path in matrix L as T_{max}. At $T_{stage}=T_{min}$ the number of stages is a maximum and is equal to S_{max}. At $T_{stage}=T_{max}$ the number of stages is a minimum and is equal to 1.

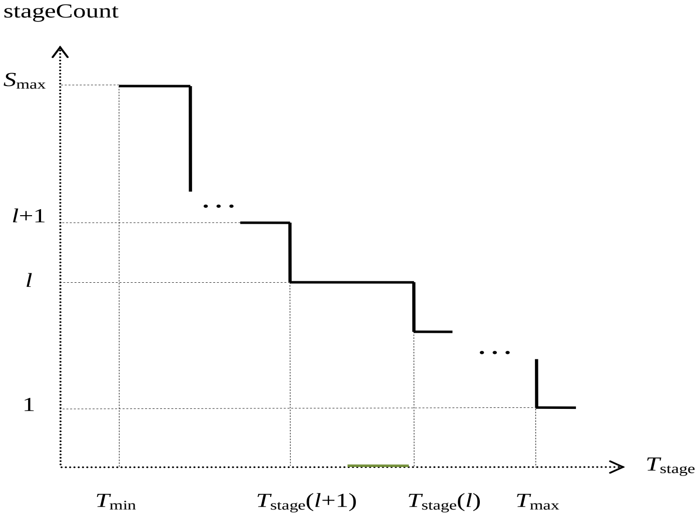


Figure 5.5. Number of pipeline stages versus stage delay

5.4.3 As soon as possible (ASAP) and as late as possible (ALAP) pipeline schedules

Classical ASAP may not be applied to scheduling of pipelines. We propose a modified version of ASAP. The main modification is that the operator precedence relation that is used as input data is replaced with the operator conflict relation. Due to this the operators are distributed not on a set of control steps but on a set of pipeline stages. Besides that the technique gives the fastest pipeline schedule without sharing resources. The number of stages in the schedule is equal to the length of a longest path in the operator conflict graph.

Similar observations concern ALAP. The mobility of operator *p* in pipeline under optimization is defined as its ability to be scheduled to various pipeline stages. The earliest stage operator *p* may be assigned to is *asap*(*p*) and the latest stage is *alap*(*p*). Hence the operator *p* mobility can be estimated as *mobility*(*p*)=*alap*(*p*)-*asap*(*p*)+1. Fig. 5.6 shows ASAP and Fig. 5.7 shows ALAP pipeline schedule for the dataflow graph example in Fig. 5.2. Operators 1, 2, 6, 9, 13 and 14 have mobility of 1, operator 4 has mobility of 2, operators 3, 11, 12 and 15 have mobility of 3 and operators 5, 7, 8 and 10 have mobility of 4.

5.5 Time constrained optimization of pipelines

5.5.1 A set of pipelines with the same stage time

In previous chapter pipelines with a minimal number of stages for a given stage time were generated. Among them are ASAP and ALAP pipelines. It appears that a huge set of pipelines with the same stage count can be generated from the same operator conflict and non-conflict graphs. The number of pipelines growth exponentially depending on the number of operators. Pipeline parameters are different. As the time parameters are already taken into account in the stage time constraint, areas cost parameters have to be analyzed. First of all, variations in assignment of operators to pipeline stages may influence the size of pipeline registers. It appears that a large reduction in pipeline register size is feasible.

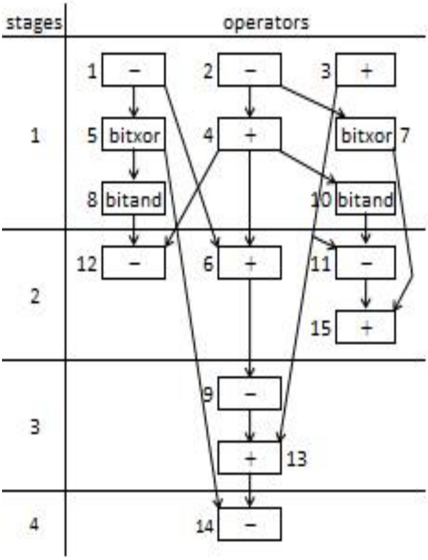


Figure 5.6. Asap pipeline schedule for T$_{stage}$=3.825

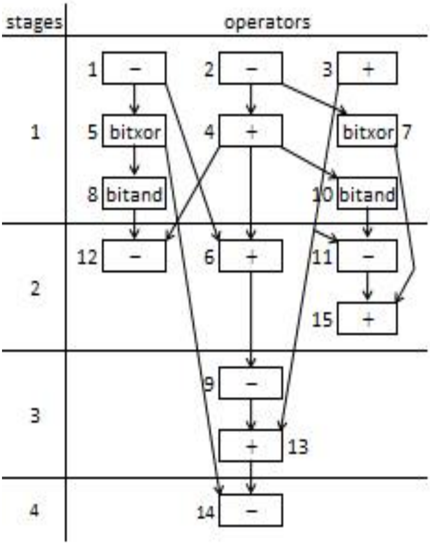


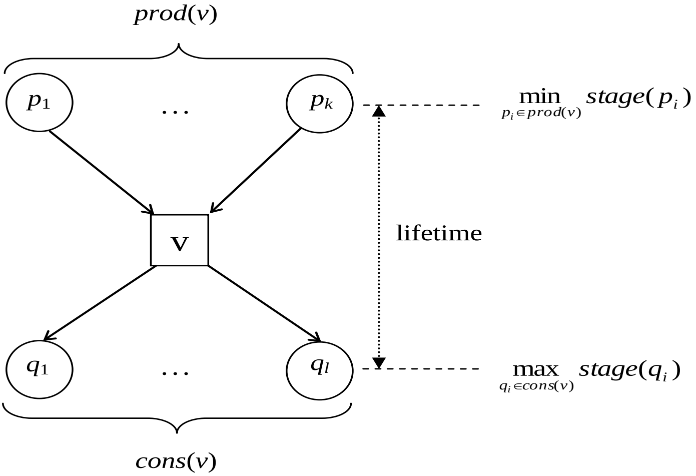
Figure 5.7. Alap pipeline schedules for T$_{stage}$=3.825

5.5.2 Evaluation of overall pipeline registers size

The operator and variable clustering within one pipeline stage is a way to achieve pipeline optimization. All variables within one stage are represented as wires. If a variable is produced in one stage and consumed in the next neighbor stage then it is replaced with a register inserted in between the stages. Several registers should be inserted instead of a variable that transmits data over several stages.

The number of pipeline registers that are introduced for variable *v* depends on the lifetime of *v*. For one variable *v* this dependence is illustrated in Fig. 5.8. Variable lifetime is determined by the earliest stage of producers and the latest stage of consumers. Two and more producers v:=e_1; … v:=e_k; have to be under conditional instructions with orthogonal test variables $c_1 \ldots c_k$:

if c_1 *then* v:=e_1; *endif*
…
if c_k *then* v:=e_k; *endif*



$prod(v)$
p_1
...
p_k
$\min_{p_i \in prod(v)} stage(p_i)$
v
lifetime
q_1
...
q_l
$\max_{q_i \in cons(v)} stage(q_i)$
$cons(v)$

Figure 5.8. Pipeline stage range (lifetime) of variable *v*

An overall registers size *RS* takes into account registers for all variables of *M*. For the single assignment model of the source algorithm description the total pipeline register size is estimated as follows:

$$RS(stage) = \sum_{v \in V} size(v) \times \begin{pmatrix} \max\limits_{q \in cons(v)} stage(q) - \\ \min\limits_{p \in prod(v)} stage(p) \end{pmatrix}. \tag{5.1}$$

The size *RS* is a sum of register sizes that are introduced for each variable *v* depending on the latest stage of consumers *cons*(*v*) and the earliest stage of producers *prod*(*v*). The variable size and its lifetime may dominate each other.

5.5.3 Optimization task: objective function and constraints

Pipelines with different number of stages can be generated by means of varying the operator conflict matrix. Different pipelines with the same number of stages can be generated by means of varying the stage time T_{stage} in the range from $T_{\text{stage}}(l)$ to $T_{\text{stage}}(l+1)$ where *l* is a stage count. Different pipelines are also possible due to varying the mapping *stage* at the given T_{stage} and conflict graph. Let Ω be a set of possible valid mappings of operators onto pipeline stages. The objective function as follows minimizes the total pipeline registers size over all mappings of Ω:

$$\min_{stage \in \Omega} RS(stage)\,. \tag{5.2}$$

Every valid mapping *stage*(*p*) must meet operator, time and precedence constraints as follows:

- for each $p \in N$ inequality $asap(p) \le stage(p) \le alap(p)$ must hold;
- if for two operators *p* and *q* inequality $l_{p,q} > T_{stage}$ holds, then the operators may not be included in the same stage (inequality

stage(*p*)≠*stage*(*q*) must hold), otherwise the operators may be included in the same stage;
- If a pair (*p*, *q*) of operators belongs to *C* then inequality *stage*(*p*)<*stage*(*q*) must hold;
- If a pair (*p*, *q*) belongs to C_n then inequality *stage*(*p*)≤*stage*(*q*) must hold.

The constraints define the structure of solution space. It should be noticed that the optimization problem (5.2) is nonlinear and retiming [40] may not be used for solving it, as every operator employs only one variable for transferring its output value to consumers in pipeline.

5.6 Least cost search branch and bound technique for pipeline optimization

Three strategies, i.e. breadth first search (BFS), depth search first (DFS) and least cost search (LCS) are available to find a minimum cost solution of an optimization problem. A search with bounding functions is branch and bound (BB) search. In [77] a DFSBB technique for optimization of pipeline schedules was proposed. In paper [79] a LCSBB technique, which overcomes DFSBB with respect to pipeline quality and optimization tool throughput, is proposed.

5.6.1 Pipeline schedule search tree

The search tree structure is shown in Fig. 5.9. The nonterminal nodes of the tree are associated with assignments of operators to pipeline stages.

A level *p* of the tree corresponds to operator *p*. Level's nodes describe various assignments of operator *p* to pipeline stages from $early_i(p)$ to $late_i(p)$. Index *i* indicates a path in the tree from root to the node. Various paths show various incomplete or complete assignments of operators to stages.

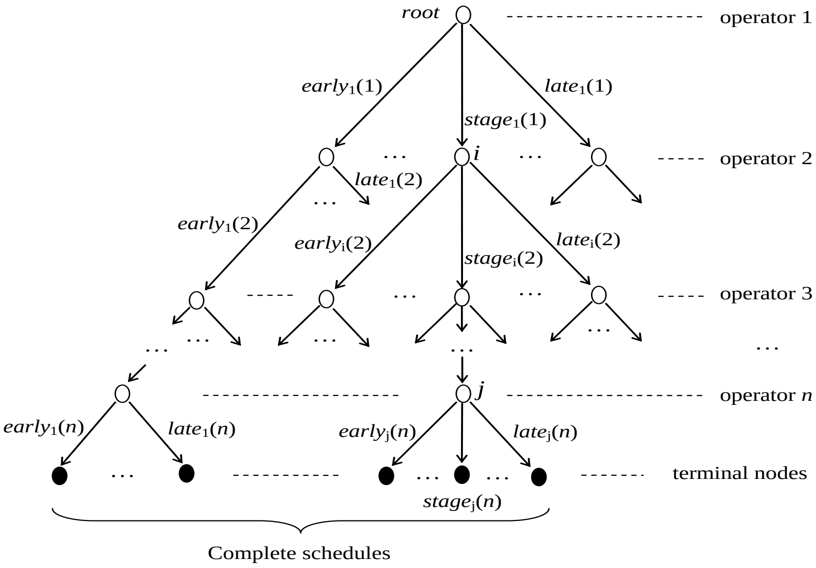


Figure 5.9. Search tree for pipeline optimization. The tree size is 2^{100} at μ=2 and n=100, and the size is 3^{1000} at μ=3 and n=1000

All terminal nodes at level *n*+1 of the tree describe candidate complete solutions. The search tree size can be estimated as μ^n where μ is an average mobility of operators in the number of pipeline stages. The size grows rapidly depending on operator count and stage count. The search tree is generated dynamically by means of expanding non-terminal nodes. Its size depends on the operator expanding order. Reordering of operators is a mechanism of increasing efficiency of the branch and bound optimization technique.

LCSBB estimates a lower bound of total register size (*LBRS*) for each expanded non-terminal node. Initial lower bound $LBRS_0$ is estimated for root. At step *t* of operator *p* scheduling, the lower bound $LBRS_t$=$LBRS_{t-1}$+ΔS_p is estimated for each *stage*(*p*) from $early_i(p)$ to $late_i(p)$ where ΔS_p is the increase of the lower bound after assignment of *p* to *stage*(*p*). The stage with minimum of ΔS_p is preferably selected for passing to the next nonterminal node in the search tree. After *n* steps, $LBRS_n$ is equal to the actual *RS*(*stage*) .

5.6.2 Incomplete mapping of operators onto pipeline stages

For operator *p* a Boolean variable *assign*(*p*) is introduced. It takes value *true* when the operator has been already assigned to a pipeline *stage*(*p*). If the operator has not been assigned to a stage then the value of *assign*(*p*) is equal to *false* and the value of *stage*(*p*) is *undefined*.

A procedure of mapping the operators to pipeline stages as a step by step process of updating the variables *assign*(*p*) and *stage*(*p*) is defined. Let $assign_t(p)$ and $stage_t(p)$ be state variables at step *t*. For some operator *p* whose *stage*(*p*) is determined, $assign_t(p)$ may have value *true* and for other operator whose *stage*(*p*) is undetermined it may have value *false*. Then $LBRS_t$ can be estimated as:

$$LBRS_t = \sum_{v \in V} size(v) \times pos\left(\begin{array}{c} \max\limits_{p \in cons(v)} stas(p) - \\ \min\limits_{p \in prod(v)} stal(p) \end{array} \right). \qquad (5.3)$$

where *pos*(*x*)=*x* if *x*>0, and *pos*(*x*)=0 otherwise; *stas*(*p*)=*asap*(*p*) if *assign*(*p*)=*false*, and *stas*(*p*)=*stage*(*p*) if *assign*(*p*)=*true*.

It is easy to see that the inequality $LBRS_t \leq LBRS_{t+1}$ holds for all steps *t* as according to (5.3) the assignment of next operator to a pipeline stage can only increase the lower bound of register size. Equation (5.3) has high computational complexity in order to execute at expanding of each node of the search tree. Computing ΔS_p between two neighbor nodes in the search tree seems to have significantly less computational complexity.

5.6.3 Updating overall registers size lower bound

Let *p* be an operator which is assigned at step *t* to a pipeline stage *stage*(*p*) implying the value of *assign*(*p*) to be changed from *false* to *true*.

The operator may produce more than one variable of *out*(*p*) while may consume more than one variable of *in*(*p*). *LBRS* may change for each of these variables.

Now a procedure of computing ΔS_{p} for all variables that are associated with operator *p* is considered. Each output variable $u \in out(p)$ may influence the increase of register size. For all variables of *out*(*p*) the increase of register size lower bound can be estimated as:

$$\Delta S_p^{'} = \sum_{u \in out(p)} size(u) \times \left[\begin{array}{l} pos\left(\begin{array}{l} STAS(u) - \\ \min\left(stage(p), STAL_p(u)\right) \end{array} \right) - \\ pos\left(\begin{array}{l} STAS(u) - \\ \min\left(alap(p), STAL_p(u)\right) \end{array} \right) \end{array} \right], \qquad (5.4)$$

where

$$STAL_p(u) = \min_{\substack{q \in prod(u), \\ q \neq p}} stal(q)$$

and

$$STAS(u) = \max_{q \in cons(u)} stas(q).$$

For all variables of *in*(*p*) the increase of register size lower bound is

$$\Delta S_p^{''} = \sum_{v \in in(p)} size(v) \times \left[\begin{array}{l} pos\left(\begin{array}{l} \max\left(stage(p), STAS_p(v)\right) - \\ STAL(v) \end{array} \right) - \\ pos\left(\begin{array}{l} \max\left(asap(p), STAS_p(v)\right) - \\ STAL(v) \end{array} \right) \end{array} \right], \qquad (5.5)$$

where

$$STAS_p(v) = \max_{\substack{q \in cons(v), \\ q \neq p}} stas(q)$$

and

$$STAL(v) = \min_{q \in prod(v)} stal(q).$$

Taking into account (5.4) and (5.5) the increase of *LBRS* for operator *p* assignment can be estimated as:

$$\Delta S_p = \Delta S'_p + \Delta S''_p. \tag{5.6}$$

As *stage*(*p*) may vary for most of operators depending on their mobility, it can significantly influence the value of Δ*S*$_p$. The increase of *stage*(*p*) can cause decrease of Δ*S'*$_p$ and increase of Δ*S''*$_p$. As a result Δ*S*$_p$ has a local minimum.

5.6.4 Computing earliest and latest pipeline stages of operator

The dynamic earliest *early*(*p*) and latest *late*(*p*) pipeline stages of operator *p* are bounds of a range of varying *stage*(*p*). First of all, the value of *early*(*p*) depends essentially on the array variables *assign* and *stage* and scalar variable *asap*(*p*). Secondly, it depends on the set *cdpred*(*p*) of direct predecessors of operator *p* in the operator conflict graph. Thirdly, the value of *early*(*p*) depends on the set *ncdpred*(*p*) of direct predecessors of operator *p* in the operator nonconflict graph. The value of *early*(*p*) can be estimated as follows:

$$early(p) = \max\begin{bmatrix} asap(p), \\ \left(\max\limits_{\substack{q\in cdpred(p),\\ assign(q)=true}} stage(q)\right)+1, \\ \max\limits_{\substack{q\in ncdpred(p),\\ assign(q)=true}} stage(q) \end{bmatrix}. \tag{5.7}$$

When no direct predecessor of *cdpred*(*p*) and *ncdpred*(*p*) has been assigned to a pipeline stage yet, the second and third operands in (5.7) are equal to 1 and 0 respectively.

First of all, the value of *late*(*p*) depends essentially on the array variables *assign* and *stage* and scalar variable *alap*(*p*). Secondly, it depends on the set *cdsucc*(*p*) of successors of operator *p*in the conflict graph. Thirdly, it depends on the set *ncdsucc*(*p*) of direct successors of operator *p*in the nonconflict graph. The value of *late*(*p*) can be estimated as follows:

$$late(p) = \min\left[\begin{array}{l} alap(p), \\ \left(\min\limits_{\substack{q \in cdsucc(p), \\ assign(q)=true}} stage(q)\right) - 1, \\ \min\limits_{\substack{q \in ncdsucc(p), \\ assign(q)=true}} stage(q) \end{array}\right]. \tag{5.8}$$

When no direct successor of *cdsucc*(*p*) and *ncdsucc*(*p*) has been assigned to a pipeline stage yet, the second and third operands in (5.8) are equal to ∞.

5.6.5 Operator assignment conflicts

The *early*(*p*) and *late*(*p*) stages of operator *p* estimated with (5.7) and (5.8) are not fully accurate, but are fast estimations of the stage range bounds. The estimated range may be wider than the actual range. This may imply operator-to-stage assignment conflicts when the early stage is larger than the late stage: *early*(*p*)>*late*(*p*). A reason for such result is the influence of predecessors and successors that are already assigned to pipeline stages on the assignment of *p*. No stage may be assigned to the operator *p* in this case. The conflict problem is being solved by the reassignment of operator predecessors and/or successors to other more suitable pipeline stages. This may imply additional backtrackings during the traversal of the search tree.

5.6.6 Least cost search branch and bound minimization of overall pipeline registers size

The least cost search branch and bound technique (LCSBB) is described in Fig. 5.10. The technique is represented as a recursive function *LCSBBScheduling* with one input, *top*. It uses global variables as follows:

```
LCSBBScheduling(top) begin
  if top<0 then return; end if;
  if top≥n then
    pipelineCount:=pipelineCount+1;
    OptimalSchedule:=Extract(Stack);
    Best:=Stack(top-1).bound;
    return;
  end if
  p:=order$^{-1}$(top);
  b:=Stack(top).late–Stack(top).early+1;
  for s in Stack(top).early to Stack(top).late do
    $\delta w(s)$:=RegSizeIncrease(p, s);
  end for;
  To sort b stages on increase of $\delta w(s)$ and compute Stack(top).rank(i), i=1…b;
  for i in 1 to b do
    s:= Stack(top).rank(i);
    Stack(top).stage :=s;
    lowerBound:=Stack(top).bound+$\delta w(s)$;
    if lowerBound≥Best then pruneCount:=pruneCount+1; return; end if;
    if top<n-1 then
        q:=order$^{-1}$(top+1);
        Stack(top+1).bound:=lowerBound;
        Stack(top+1).early:=EarlyStage(q);
        Stack(top+1).late:=LateStage(q);
        if Stack(top+1).early>Stack(top+1).late then continue; end if
    end if;
    LCSBBScheduling (top+1);
  end for;
end.
```

Figure 5.10. Least cost search branch and bound technique (LCSBB)
for pipeline optimization

- *Stack* is an array of records that includes such elements as a pipeline *stage*, lower *bound* of total register width, *early* and *late* stages of operator; rank of stages where, *rank*(*i*) is a stage at position *i* and *rp* is current position of the rank; the current record of the stack is indexed with *top*;
- *OptimalSchedule* is the current best assignment of operators to pipeline stages;
- *Best* is the current best *LBRS*;

- *order*$^{-1}$ is the mapping of stack records onto operators;
- *pipelineCount* is the number of generated complete assignments;
- *pruneCount* is the number of pruned nodes of the tree.

The slave function *Extract*(*Stack*) generates a next complete assignment of operators to pipeline stages. Function *RegSizeIncrease*(*p,s*) computes $\Delta S_p(s)$ using (5.4)-(5.6). Function *EarlyStage*(*q*) computes *early*(*q*) for operator *q* using (5.7), and function *LateStage*(*q*) computes *late*(*q*) for operator *q* using (5.8).The optimal pipeline schedule for the example DFG is shown in Fig. 5.11.

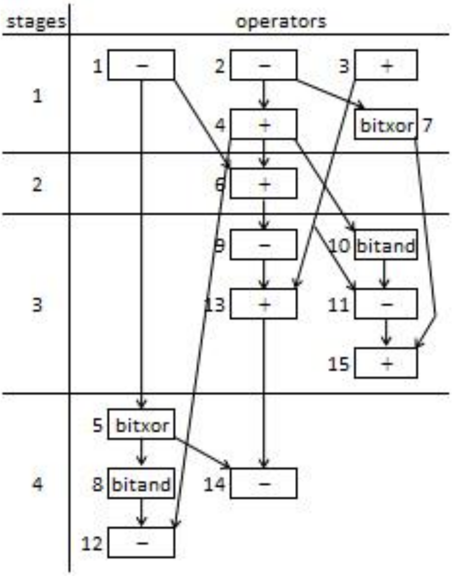


Figure 5.11. Optimal pipeline schedule for T_{stage}=3.825. The schedule includes 13 pipeline registers consisting of 167 bits. ASAP schedule includes 17 registers consisting of 247 bits and ALAP schedule includes 16 registers consisting of 216 bits

5.7 Heuristic technique for optimization of pipelines

It was assumed in LCSBB that *early*(*q*) and *late*(*q*) of a nonscheduled operator *q* can be estimated using *asap*(*p*), *alap*(*p*) and *stage*(*p*) of all other operator *p*. This is not a completely accurate estimation, although LCSBB is capable of finding a global optimum. In fact *early*(*q*) and *late*(*q*) of nonscheduled operator *q* may directly or indirectly depend on *early*(*r*) and *late*(*r*) of other nonscheduled operator *r*.

The accurate recalculation of *early*(*q*) and *late*(*q*) is a time consuming procedure and it may significantly slow down LCSBB. Moreover LCSBB takes some assumptions concerning register size lower bound estimation. The main assumption is that *asap*(*q*) and *alap*(*q*) are used in (5.7) and (5.8) although tighter bounds *early*(*q*) and *late*(*q*) of stages which are available for operator *q* can be computed. The heuristic pipeline optimization technique proposed in paper [79] finds only one feasible complete assignment of operators to pipeline stages that gives possibly minimal total registers size. In order to find a best path from root to a leaf of the search tree the technique needs efficient heuristics.

5.7.1 Dynamic evaluation of earliest and latest stages of operators

Assume that at step *t*-1 the values of $assign_{t-1}(p)$, $early_{t-1}(p)$ and $late_{t-1}(p)$ are determined for $p \in N$. The sets *cdpred*(*p*), *ncdpred*(*p*), *cdsucc*(*p*), *ncdsucc*(*p*) of predecessors and successors of each operator *p* in the conflict and non-conflict graphs have been computed, they stay the same for all scheduling steps.

Let at step *t* nonscheduled operator *r* has been mapped onto *stage*(*r*) and assignment $assign_t(r)$:=*true* has been already performed. Then, the early pipeline stage of a nonscheduled operator $p \in N$ can be evaluated recurrently as

$$early_t(p) = \max\begin{bmatrix} early_{t-1}(p), \\ \left(\max\limits_{q \in cdpred(p)} searly_t(q)\right) + 1, \\ \max\limits_{q \in ncdpred(p)} searly_t(q) \end{bmatrix}, \qquad (5.9)$$

where $searly_t(q)=early_t(q)$ if $assign_t(q)=false$, and $searly_t(q)=stage_t(q)$ if $assign_t(q)=true$. It is easy to observe that the inequality $early_t(p)\geq early_{t-1}(p)$ holds for all $t=1\ldots n$.

At step t the late pipeline stage of a nonscheduled operator $p\in N$ can be evaluated recurrently as

$$late_t(p)=\min\begin{bmatrix} late_{t-1}(p), \\ \left(\min\limits_{q\in cdsucc(p)} slate_t(q)\right)-1, \\ \min\limits_{q\in ncdsucc(p)} slate_t(q) \end{bmatrix}, \tag{5.10}$$

where $slate_t(q)=late_t(q)$ if $assign_t(q)=false$, and $slate_t(q)= stage(q)$ if $assign_t(q)=true$. It is easy to observe that the inequality $late_t(p)\leq late_{t-1}(p)$ holds for all $t=1\ldots n$. Equations (5.9) and (5.10) demand an appropriate order of nonscheduled operators to properly compute $early_t(p)$ and $late_t(p)$. For $early_t(p)$ this order is $p=1\ldots n$. For $p=1\ldots r$-1, $early_t(p)=early_{t-1}(p)$ and for $p=r+1\ldots n$, $early_t(p)$ is estimated with (5.9). For $late_t(p)$ this order is $p=n\ldots 1$. For $p=n\ldots r+1$, $late_t(p)=late_{t-1}(p)$ and for $p=r$-1…1, $late_t(p)$ is estimated with (5.10).

5.7.2 Dynamic estimation of overall registers size lower bound

At step t the values of $assign_t(p)$, $p\in N$ describe the current incomplete assignment of operators to pipeline stages. If $assign_t(p)=true$ then $stage(p)$ has been already determined. If $assign_t(p)=false$ then $early_t(p)$ and $late_t(p)$ have been already computed for p. Then the total register size lower bound at scheduling step t can be estimated as:

$$LBRS_t=\sum_{v\in V} size(v)\times pos\begin{pmatrix} \max\limits_{p\in cons(v)} searly_t(p)- \\ \min\limits_{p\in prod(v)} slate_t(p) \end{pmatrix}.. \tag{5.11}$$

LBRS$_t$ computed with (5.11) is a more precise estimation over those computed with (5.3). Obviously *LBRS*$_t$ can only increase with increasing of *t*. It means that the inequality $lowerBound_{t-1} \le lowerBound_t$ holds for all scheduling steps $t=1\ldots n$.

5.7.3 Dynamic ordering of operators

Reordering of operators makes pipeline scheduling more efficient versus optimization time and pipeline quality. The dynamic heuristic pipeline scheduling uses a heuristic weight χ_p of nonscheduled operator $p \in N_{non}$:

$$\chi_p = \sum_{i=1}^{\kappa} \omega_i \cdot \rho_i(p), \qquad (5.12)$$

where *k* is the number of heuristic parameters; ω_i, $i=1\ldots$k are heuristic factors satisfying the equality $\sum_{i=1\ldots k} \omega_i = 1$.The heuristic parameters $\rho_i(p), i=1\ldots$k describe features of nonscheduled operator *p* in pipeline under optimization.

The parameters are defined to satisfy two key requirements. Firstly, they have to vary in the range from 0 to 1. Secondly, the higher value of the parameter is expected to imply better pipeline parameters. Then the value of χ_p varies in the range from 0 to 1. Operator $p \in N_{non}$ with the maximum value of χ_p is selected as the next scheduled operator.

Four heuristic parameters have been used in the example shown in Fig. 5.12 for the DFG of Fig. 5.2 and Table 5.1. The mobility of operators 1, 2, 6, 9, 13 and 14 determined by ASAP and ALAP schedules (Fig. 5.6 and Fig. 5.7) is equal to 1, therefore these operators are assigned to stages 1, 1, 2, 3, 3 and 4 respectively. The early-late stages of the rest operators 3, 4, 5, 7, 8, 10, 11, 12 and 15 are 1-3, 1-2, 1-4, 1-4, 1-4, 1-4, 2-4, 2-4 and 2-4 respectively.

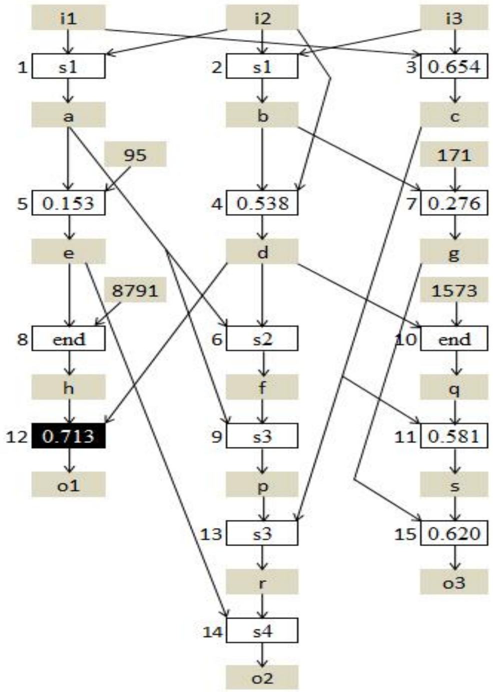


Figure. 5.12. Selection of next operator for assignment to a stage. Operators 1, 2, 6, 9, 13 and 14 are already assigned to stages s1-s4 implying the lower bound register size, *LBRS*=61; operators 8 and 10 will be assigned at the end of the scheduling process; for each other operator the heuristic weight, χ_P is computed. Operator 12 has the maximal weight of 0.713 and is selected as the next scheduled operator

The overall *LBRS* that is estimated with Equation (11) is equal to 61 as only variables *a*, *f* and *r* have a nonzero register size lower bound of 32, 16 and 13 respectively. Thus *LBRS* of variable *a* is 16×(max(1,2,3)–min(1)) = 16×2 = 32 as its consumers are operators 5, 6 and 9 and its producer is operator 1. Operators 6, 9 and 1 are already assigned to stages 2, 3 and 1 and the early stage of operator 5 is 1.

Operators 8 and 10 have the equal size of input and output therefore the movement of operators over stages does not change the overall pipeline register size. Those operators will be assigned at the end of scheduling process. The heuristic weight, χ_p is computed for each other operator with Equation (12) at the vector of heuristic factors, ω=(0.210, 0.301, 0.087, 0.401), which gives the global optimum solution. Operator 12 has the maximum weight of 0.713 and is selected as the next scheduled operator. Its heuristic parameters, $\rho(12) = (0.5, 0.933, 0.125, 0.789)$ are evaluated according to the following procedure.

The first parameter is a complement-on-one of the relative dynamic operator mobility over pipeline stages, $\rho_1(12)$=1-(4-3)/(4-2)=0.5 where 3 is the dynamic mobility of operator 12. The maximal mobility among nonscheduled operators (operators 5 and 7 have the mobility of 4) is 4 and 2 is the minimal mobility among nonscheduled operators (operator 4 has the mobility of 2). Low values of the mobility imply high values of ρ_1.

Operator 12 can be assigned to stages 2, 3 and 4. For each possible assignment a new *LBRS* is computed. For instance, if stage 2 is chosen then the early-late stages of operators 4, 5 and 8 are modified with (9) and (10) to 1-1, 1-2 and 1-2 respectively. As a result the *LBRS* of variables *d*, *e* and *o*1 is changed with (11) from 0 to 14, 36 and 34 respectively. The overall *LBRS* increases from 61 to 145. If stages 3 and 4 are chosen for operator 12 then the overall *LBRS* grows form 61 to 110 and 89 respectively.

The second parameter is a relative *LBRS* difference over available stages of operator 12 among all nonscheduled operator, ρ_2(12)=(145-89)/60=0.933 where 145 is *LBRS* computed with (11) after assignment of operator 12 to stage 2; 89 is *LBRS* after assignment of operator 12 to stage 4; 60 is the maximal *LBRS* difference among available stages over all nonscheduled operator (operator 15 has the maximal difference of 60). Parameter ρ_2 shows the relative difference between the best and worst cases of operator assignment to available stages.

The third parameter is a relative minimal *LBRS* increase over all nonscheduled operator, $\rho_3(12) = 1-(89-61)/(93-61) = 0.125$ where 89 is the minimal *LBRS* increase of operator 12 over stages 2-4; 61 is the minimal *LBRS* increase of operator 15; 93 is the minimal *LBRS* increase (maximum among the operators) of operator 5. Parameter ρ_3 shows the relative increase of *LBRS* after assignment of the operator to the best available stage.

The fourth parameter is a relative difference between inputs and outputs sizes of the operator, $\rho_4(12)$ = (18+14–17)/ (16+16-13)=0.789 where 18, 14 and 17 are the sizes of variables *h*, *d* and *o*1 (operator 12) and 16, 16 and 13 are the sizes of variables *a*, *f* and *p* (operator 9 whose size difference is the largest). Parameter ρ_4 shows the importance of moving the operator over stages: moving should be done to the earliest available stage if the outputs size is larger than the inputs size and vice versa.

Stage 4 has a minimum *LBRS* of 89 among stages 2, 3 and 4 and is selected for assignment of operator 12. Applying the heuristics to other operators yields the operator sequence, 11, 3, 4, 15, 7, 5, 8 and 10, the corresponding *LBRS* sequence, 102, 115, 129, 139, 151, 167, 167 and 167, and the corresponding stage sequence, 3, 1, 1, 3, 1, 4, 4 and 3. The resulting pipeline schedule is the global optimum solution shown in Fig. 5.11.

5.7.4 Heuristic technique for pipeline optimization

The heuristic technique (HT) is represented in Fig. 5.13 as recursive function *HeuristicScheduling* with one input *top*. It uses global variables as follows:

- *Stack* is an array of *n*+1 records that include such elements as *operator*, *rank* of avalable pipeline stages for the *operator*, current position *rp* in the rank, current *stage* assigned to *operator*, *mobility* of *operator*, current lower *bound* of total register size, set *nschop* of nonscheduled operators, arrays of *early* and *late* stages for the nonscheduled operators, array *vbnd* of register sizes for all variable in the scheduled algorithm;
- *pipelineSchedule* is a mapping of operators onto pipeline stages generated by the heuristic algorithm;
- *registerTotalSize* is the overall pipeline registers size.

The slave function *ExtractPipelineSchedule* generates the resulting *pipelineSchedule*. Function *ChooseOperator* computes heuristic parameters ρ for nonscheduled operators from the set *nschop* and chooses an operator with the maximal heuristic weight (5.12).

```
HeuristicScheduling(top) begin
```

```
    if $top \geq n$ then
      pipelineSchedule := ExtractPipelineSchedule(Stack);
      registerTotalSize := Stack(top-1).bound;
      return;
    end if;
    p:=ChooseOperator(Stack(top).nschop);
    Stack(top).operator:=p;
    Stack(top).rank:=GenerateRank(p);
    mobility:=|Stack(top).rank|;
    for rp in 1 to mobility do
      stage:=Stack(top).rank(rp);
      Stack(top).stage:=stage;
      Stack(top).bound:=RegisterLowerBound(p, stage, Stack(top).vbnd);
      Stack(top+1).nschop:=Stack(top).nschop\{p};
      Stack(top+1).early := EarlyStages(Stack(top).early);
      Stack(top+1).late:=LateStages(Stack(top).late);
      if AssignmentConflict(Stack(top+1).early, Stack(top+1).late) then
          continue;
      end if;
      Stack(top+1).vbnd := VariableRegisterBounds(Stack(top).vbnd);
      HeuristicScheduling(top+1);
    end for;
  end.
```

Figure 5.13. Heuristic technique HT for pipeline optimization

Function *GenerateRank* computes a rank of pipeline stages which are available for the selected operator *p*. Function *RegisterLowerBound* estimates *LBRS* for *p* using (5.11). Function *EarlyStage*s computes using (5.9) the early stage of each nonscheduled operator after assignment of *p* to an avalable *stage*. Function *LateStage*s computes using (5.10) the late stage of each nonscheduled operator. Function *AssignmentConflict* returns *true* if a nonscheduled operator *q* has been found for which *early*(*q*)>*late*(*q*), otherwise it returns *false*. Function *VariableRegisterBounds* recalculates using (5.11) the lower bound register size for each of *vbnd*.

5.7.5 Tuning heuristic factors

The heuristic weight χ_p is a criterion for choosing a next scheduled operator. The operator is assigned to a pipeline stage which gives a minimum of *LBRS*. The result of the operator choice significantly depends not only on the heuristic parameters of ρ, but also on the heuristic factors of ω. The factors determine the weight of each parameter in the criterion. Important parameters should have larger factor value. The optimization problem in the solution space described by the vector ω is to determine the importance of each parameter during pipeline optimization. Conducted experiments show that the problem has many local optima. As the function *HeuristicScheduling* is fast enough and it is possible to generate many vectors ω and to compute *RS* for each of them, a random search and a genetic algorithm have been used to solve this problem. The random search is capable of finding an optimal solution for *RS*, but very often it yields a suboptimal solution.

5.8. Conclusion

A new pipeline synthesis and optimization methodology that starting from partitions of a large dataflow design increases the data throughput of whole design by selecting design partitions and by generating the pipelined implementations has been presented. The methodology is capable of determining the most appropriate pipeline stage time and the number of pipeline stages for each partition of the dataflow design.

Two pipeline optimization techniques that minimizes the total pipeline register size for each stage time and the stage count have also been developed. The first methodology is called" least cost search branch and bound" and the second is referred to as a "heuristic pipeline optimization". The branch and bound algorithm is capable of finding the global optimum pipeline schedule for low size designs, whereas the heuristic algorithm is capable of finding close-to-optimal solutions also in the case of large designs.

6. OPTIMIZATION OF PIPELINES FOR MEANINGFUL APPLICATIONS

The pipeline optimization algorithms LCSBB and HT that are proposed in the previous chapter and the downward and upward direction traversal algorithms that are proposed in [3] and represented as ASAP and ALAP are compared in this chapter on several meaningful test benches.

6.1. Bayer filter based on improved linear interpolation

The Bayer filter test bench (Fig. 6.1) was considered in detail in [77]. It contains 13 input ports, 5 output ports and 63 local variables, totally 81 variables. The variable size varies in the range from 8 to 23, and the average size is 20.41. The Bayer filter also contains 68 operators including 32 additions, 19 subtractions, 3 multiplications and 13 bitand operators.

A relative time delay is assigned to each operator as follows: 1.0 for addition, 1.1 for subtraction, 3.0 for multiplication and 0.02 for bitand. The total delay of all operators is 62.2. The design critical path length is 15.62 or 25.1% over the total operator delay.

As reported in [79], pipelines with 2 to 7 stages were optimized by DFSBB and synthesized to FPGAs. The global optimum was obtained for 2 and 3 stage pipelines and suboptimal solutions were generated for 4 up to 7 stage pipelines.

Table 6.1 reports pipeline scheduling results obtained by four scheduling techniques: LCSBB, HT, ASAP and ALAP. One pipeline stage count is represented with a range of stage time. LCSBB has generated a global optimum for each stage time. HT has given total register size that is very close to global optimum, 2.0% more on average. LCSBB and HT show superior results compared to both ASAP and ALAP, 48.8% and 82.6% on average respectively.

It should be noted that for 7-stage pipeline the number of pipeline registers, 60 is comparable with the number of operators, 68. This proves the importance of register minimization problem.

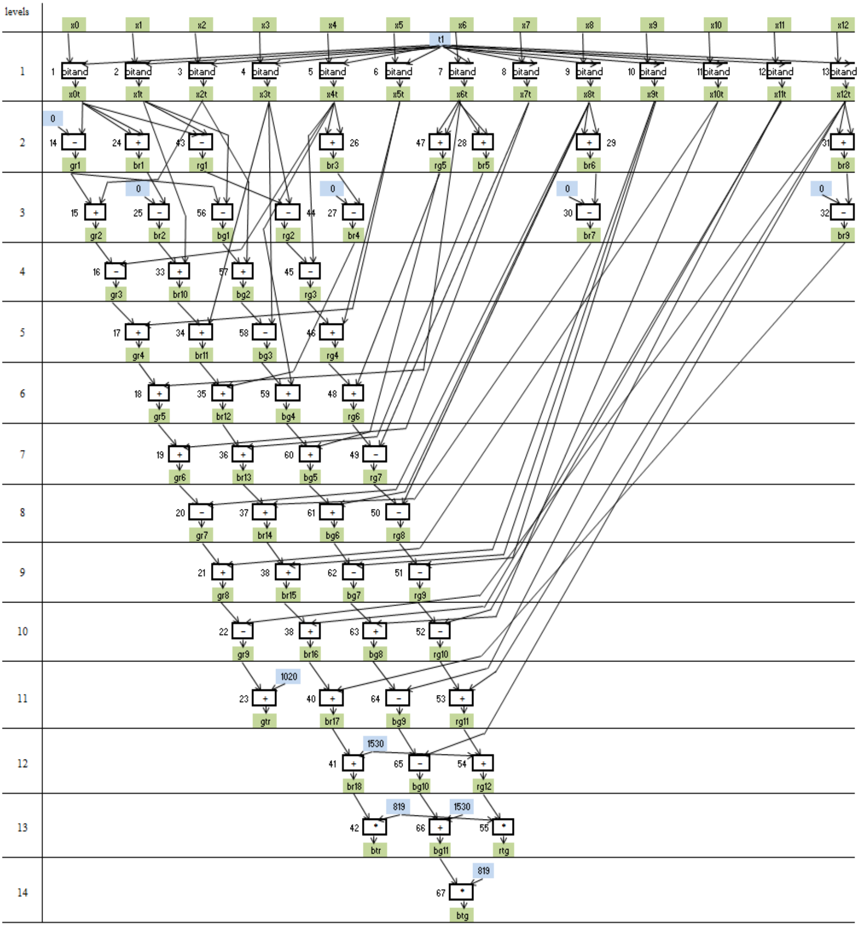


Figure 6.1. Data flow of Bayer filter

Table 6.2 reports parameters of LCSBB that are obtained on Bayer filter. The CPU time is less than 1 sec for 2 pipeline stages and is equal to 959 sec for 7 stages. The number of pruned branches grows rapidly up to 1818112224 and the number of updated optimal schedules grows form 1 to 4 with increasing the number of stages from 2 to 7. The number of conflicts also grows rapidly.

Table 6.1

Results for Bayer filter obtained by LCSBB, HT, ASAP and ALAP

Stages	Stage time	LCSBB	HT		ASAP		ALAP	
		Register size	Register size	%	Register size	%	Register size	%
1	15.62	0	0		0		0	
2	15.60	100	100	0.0	100	0.0	142	42.0
	11.62	100	100	0.0	100	0.0	248	148.0
	10.52	108	123	13.9	123	13.9	286	164.8
	10.12	116	116	0.0	146	25.9	263	126.7
	9.42	116	116	0.0	169	45.7	301	159.5
	8.52	124	124	0.0	192	54.8	270	117.7
	8.30	147	147	0.0	215	46.3	285	93.9
3	7.42	232	232	0.0	315	35.8	381	64.2
	7.12	240	240	0.0	338	40.8	464	93.3
	6.32	240	240	0.0	361	50.4	479	99.6
	6.30	271	286	5.5	407	50.2	494	82.3
	6.12	294	294	0.0	430	46.3	532	81.0
	5.32	340	355	4.4	453	33.2	501	47.4
4	5.22	387	396	2.3	599	54.8	681	76.0
	5.12	403	403	0.0	622	54.3	658	63.3
	5.10	403	403	0.0	668	65.8	673	67.0
	4.32	426	441	3.5	691	62.2	672	57.8
	4.30	472	472	0.0	760	61.0	702	48.7
5	4.10	573	588	2.6	883	54.1	836	45.9
	4.00	596	612	2.7	929	55.9	920	54.4
	3.32	650	657	1.1	975	50.0	926	42.5
	3.22	650	665	2.3	998	53.5	926	42.5
6	3.20	752	759	0.9	1167	55.2	1128	50.0
	3.12	759	774	2.0	1213	59.8	1235	62.7
	3.10	842	881	4.6	1259	49.5	1250	48.5
7	3.00	960	990	3.1	1451	51.2	1383	44.1
On average:				2.0		48.8		82.6

Table 6.2

Parameters of LCSBB on Bayer filter

Stages	Stage time	LCS branch and bound			
		CPU time	Pruned branches	Schedules	Conflicts
2	8.30	<1	22	1	2
3	5.32	<1	3422	2	0
4	4.30	1	62453	1	64
5	3.22	<1	96725	2	119
6	3.10	14	24103979	3	294947
7	3.00	959	1818112224	4	151966

Table 6.3

Comparison of DFSBB against LCSBB (times) on Bayer filter

Stages	CPU time	Pruned branches	Schedules	Conflicts
2	1.00	14.09	6.00	1
3	1.00	6.86	4.50	1
4	1.00	6.14	27.00	5.48
5	1.00	10.76	15.50	53.47
6	8.36	21.45	12.00	0.04
7	1.54	3.03	11.75	216.71
	2.32	10.39	12.79	46.28

For DFSBB the CPU time, the number of pruned branches, the number of feasible complete schedules and the number of operator assignment conflicts are by 2.32x, 10.39x, 12.79x and 46.28x larger on average respectively over LCSBB (Table 6.3). LCSBB has given less overall registers size of 15.4% and 14.1% over DFSBB for the stage time of 3.1 (6 stages) and 3.0 (7 stages) respectively [79]. The CPU time for the heuristic algorithm has not exceeded 1 sec for all stage time and all count of pipeline stages.

The heuristic factors have been tuned for each stage time using the random search technique. Each factor has been varied in a wide range of values. The average factors over all the stage times and the stage counts are $\omega_{average}$=(0.292, 0.299, 0.213, 0.196). Therefore, each factor is a significant heuristic. Due to the variations of factors, total register size variations in the range from 0% up to 36.4% have been observed, 10.4% on average. Therefore the tuning of the heuristic factors is an effective mechanism of pipeline optimization.

It should be noted that the solution space grows very rapidly in the case of the Bayer filter design depending on the number of pipeline stages. It means that a huge number of pipeline schedules exist which are very close to each other with respect to the total register size. In general, the capability of LCSBB depends on the number of operators, on the length of the critical path, on the mobility of operators and on the number of pipeline stages. LCSBB is capable of synthesizing optimal pipelines with a low number of pipeline stages for large DFGs (>1000 operators) which have long critical paths and low mobility of operators. For large designs the fast HT algorithm becomes a preferable option.

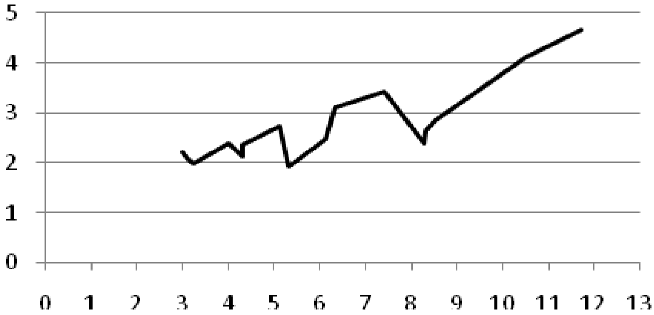


Figure 6.2. The ratio “maximum/ minimum” of total register size vs. pipeline stage time for the Bayer filter design case. The ratio varies in the range from 1.94x to 4.68x

In order to measure the distribution of the total register size variation, LCSBB has been modified in such a way to maximize the register size instead of minimizing it. The resource gain has been estimated with the ratio “maximum / minimum” of the total register size that is shown in Fig. 6.2 as a function of pipeline stage time and can reach the significant value of 4.68x. Each local minimum corresponds to the minimum stage time for each number of pipeline stages.

6.2. Forward 8×8 discrete cosine transform

The forward discrete cosine transform FDCT64 has been implemented in CAL as an actor consisting of one action and then automatically transformed into a single basic block model by means of applying various transformation including unrolling loops. The FDCT64 is a relatively large design with 64 input ports, 64 output ports and 2304 local variables, for a total of 2432 variables. The word size varies in the range from 1 up to 32 bit and the average size is equal to about 23. FDCT64 core processing module includes 2368 operators of which 336 are additions, 496 are subtractions, 64 are multiplications, 576 are right shifts, 64 are left shifts and 832 are static assignments. A relative time delay has been assigned to each operator as follows: 1.0 for addition, 1.1 for subtraction, 3.0 for multiplication and 0.1 for shift. The static assignment has no hardware implementation correspondence, therefore its relative delay has been set to 0.0. The total delay for all operators is 1137.6. The design critical path length is 19.6 or 1.72% over total operator delay. LCSBB cannot yield global optimum results for a large design such as FDCT64, therefore HT algorithm has been used in this case.

Table 6.4 reports the parameters of pipeline schedules generated for FDCT64. For each stage count in column 1 the minimal stage time is given in column 2. Column 3 reports the register size depending on the stage time for HP. The size has increased from 688 for 2 stages to 24896 for 9 stages. The results for ASAP that are given in columns 4 and 5 are much worse (100.3% on average) over HT. ALAP has produced better results on average (columns 6 and 7) over ASAP, and worse results (50.0% on average) over HT. Within 2 and 3 pipeline stages the total register size grows slowly from 688 to 4096 and starting from 4 stages the size grows rapidly.

Table 6.4

Results for FDCT64 obtained by HT, ASAP and ALAP

Stages	Stage time	HT	ASAP		ALAP	
		Register size (bit)	Register size (bit)	%	Register size (bit)	%
2	19.59	688	2088	203.5	688	0.0
	18.00	2048	2864	39.8	3072	50.0
	17.00	2048	3776	84.4	4096	100.0
	16.00	2048	3488	70.3	4608	125.0
	15.00	2048	3608	76.2	4096	100.0
	14.00	2048	4160	103.1	3072	50.0
	13.00	2304	4736	105.6	2304	0.0
	11.00	2560	7112	177.8	2560	0.0
	10.00	3584	8672	142.0	4096	14.3
	9.71	4096	7536	84.0	4608	12.5
3	9.00	4096	10272	150.8	7168	75.0
	8.00	4096	9656	135.7	7168	75.0
	6.70	4352	8752	101.1	7168	64.7
4	5.40	9440	17200	82.2	11264	19.3
5	4.30	12608	22104	75.3	13568	7.6
6	4.10	15872	27168	71.2	27392	72.6
7	3.30	20128	33232	65.1	32256	60.3
8	3.20	22784	37352	63.9	36528	60.3
9	3.00	24896	43344	74.1	40704	63.5
On average, %:				100.3		50.0

It is interesting to notice that the CPU time used by the HT algorithm has resulted to stay within 2 sec for all pipeline stages. For large design such as the FDCT64 only static heuristic parameters have been exploited by the HT algorithm for operator ordering.

In case of FDCT64 design, HT yields much better results than LCSBB in case of the Bayer filter, when comparing both to results of ASAP and ALAP. Therefore, it can be concluded that HT is capable of generating large pipelines that are close to optimal solutions.

6.3. Experimental results for random middle size designs

A program for the random generation of data flow graphs has been developed to test the performances of a design with different statistical properties of their operators. The generic parameters are the number of operators, variables, input and output ports, the operator types, operator delays and variable sizes. For conducting the experiments, the operator types and associated probabilities have been chosen as follows: addition (0.3), subtraction (0.25), multiplication (0.1), shift (0.1) and bitand (0.25). The operator delays have been assigned to the same values used in the previous experiments. The variable lifetime in terms of operator interval is also a control parameter. By varying the parameter value it is possible to generate DFGs with different critical path length.

In order to measure parameters of pipeline optimization techniques, two random design series have been generated. The first one includes five middle size designs constructed of 100 up to 300 operators. In Table 6.5, the total operator delay and critical path length is indicated for each design. The critical path length varies in the range from 20% down to 16% of the total operator delay. For each variable its size was randomly generated in the range from 4 to 28 and the average size is indicated for each design in Table 6.5.

Each pipeline schedule constructed of 2, 3, 4 and 5 stages was optimized with respect to the total register size for each design by LCSBB, HT, ASAP and ALAP. In Table 6.5 the register size is given in bits for LCSBB. The symbol * indicates suboptimal solutions for 5-stage pipelines, therefore the comparison of HT, ASAP and ALAP over LCSBB in % is given for 2-, 3- and 4-stage pipelines. The average register size produced by HT is only 2.28% larger than the minimum size produced by LCSBB. It means that HT can be considered as a "close-to-optimal" optimization technique. Again, ASAP and ALAP results end to be much worse (53.8% and 48.3% respectively) than LCSBB.

Table 6.5

Results for random middle size designs

Parameters		Design				
Number of operators		100	150	200	250	300
Total operator delay		111	152	175	231	280
Critical path		23.3	31.9	33.0	39.2	43.4
Variable average size		16.25	15.94	15.53	15.21	15.36
Pipeline registers size given by LCSBB (bit)						
Stages	2	334	209	175	244	182
	3	457	413	291	348	454
	4	686	595	478	697	601
	5	901*	805*	701*	732*	826*
Register size HT (%)		3.6	0.4	2.4	3.5	1.5
Register size ASAP (%)		49.3	50.8	64.1	31.1	73.9
Register size ALAP (%)		43.0	49.6	50.1	37.7	61.0
Variables average lifetime given by LCSBB (stages)						
Stages	2	0.189	0.086	0.057	0.073	0.057
	3	0.302	0.173	0.109	0.107	0.117
	4	0.434	0.265	0.175	0.195	0.165
	5	0.594	0.333	0.245	0.221	0.215
Var lifetime HT (%)		4.5	0.8	0.0	-4.2	0.9
Var lifetime ASAP (%)		35.5	30.3	27.8	11.4	30.7
Var lifetime ALAP (%)		24.8	21.3	35.2	21.3	18.8
Decrease in pipeline registers average size over variables average size (%)						
Register size LCSBB %		8.4	9.3	16.2	18.2	34.1
Register size HT (%)		9.0	9.6	14.2	11.8	33.5
Register size ASAP (%)		-2.7	-5.0	-7.4	0.3	2.5
Register size ALAP (%)		-6.5	-10.4	4.5	2.7	0.3
Average CPU time for LCSBB (sec)						
CPU time LCSBB (sec)		444	376	276	508	678

.

Two key factors influence the minimization of the total pipeline register size: the reduction of the variables average lifetime in terms of pipeline stages and the decrease in the pipeline registers average size over the variables average size. In Table 6.5 the variable lifetimes grow with the increase of the stages number. The lifetime given by HT is very close to the lifetime given by LCSBB. The variables average lifetime for ASAP is 27.1% and for ALAP is 24.3% larger than for LCSBB. LCSBB and HT have decreased significantly the pipeline registers average size over the variables average size (17.2% and 15.6% on average respectively). It is interesting to notice that HT (9.0) succeed in reducing the word width better than the LCBB (8.4) for designs with about 100 operators. At the same time LCBB reduces the lifetime of the registers in comparison to the HT algorithm by a factor of 4.5%. As a result the LCBB algorithm outperforms HT with respect to the register size by a factor of 3.6%. ASAP and ALAP are not capable of assigning pipeline registers to small size variables. Due to such limitation, the registers average size results to increase of a factor 2.5% for ASAP and of 1.9% for ALAP over the variable average size. LCSBB consumes more CPU time, from 276 to 678 sec.

6.4. Experimental results for random large size designs

The second series includes five large designs constructed of 1000 up to 5000 operators (Table 6.6). The critical path length is about 10% of the total operator delay for all design. The results reported in Table 6.6 show that HT yields superior results compared to both ASAP and ALAP, 32.7% and 40.6% on average respectively concerning the total register size and 16.9% and 25.5% with respect to the variables average lifetime. In contrast to ASAP and ALAP, it also results into smaller size variables that are mapped onto the pipeline registers. It is also important to notice that HT requires very limited CPU time for large design, only from 4 to 112 sec, thus, could be successfully used in commercial pipeline optimization tools.

6.5. Conclusion

Based on the mathematical models, design formulations and selected algorithms, a program that automatically transforms a non-pipelined algorithm into a pipelined design within a range of 1-2 min of CPU time has been developed.

The experiments performed on the design test benches of a Bayer filter, 8×8FDCT, middle size and large random designs, have proven that the proposed LCSBB and HT algorithms yields much better results against ASAP and ALAP. Results characterized by a low pipeline registers size has been achieved by means of reducing the variable average lifetime in terms of pipeline stage interval and choosing small size variables that are mapped onto pipeline registers inserted in between stages.

Table 6.6

Results for random large designs

Parameters		Design				
Number of operators		1000	2000	3000	4000	5000
Operator total delay		885	1721	2640	3573	4469
Critical path		88	173	266	351	447
Variable average size		15.83	15.55	15.52	15.51	15.47
Pipeline registers size given by HT (bit)						
Stages	2	484	568	676	976	1090
	3	940	952	1455	1748	2388
	4	1469	1712	2192	3067	3318
	5	1995	2093	2895	3773	4430
Register size ASAP (%)		32.51	32.89	33.77	33.22	31.11
Register size ALAP (%)		66.59	43.94	32.34	27.38	32.64
Variables average lifetime given by HT (stages)						
Stages	2	0.032	0.020	0.017	0.017	0.016
	3	0.069	0.036	0.036	0.033	0.033
	4	0.102	0.059	0.052	0.052	0.049
	5	0.141	0.075	0.070	0.069	0.065
Var lifetime ASAP (%)		19.19	12.34	19.96	18.14	14.63
Var lifetime ALAP (%)		51.73	29.42	15.43	16.92	14.20
Decrease in pipeline registers average size over variables average size (%)						
Register size HT (%)		12.03	13.10	14.14	10.76	13.33
Register size ASAP (%)		0.52	-4.38	2.59	-1.79	-0.84
Register size ALAP (%)		1.79	1.97	-0.42	1.73	-2.42
Average CPU time for HT (sec)						
CPU time for HT (sec)		4	7	26	55	112

7. GENETIC ALGORITHM FOR TUNING OPTIMIZATION HEURISTICS

7.1. Heuristics for solving optimization problems

A heuristic technique or simply a heuristic, is any approach to problem solving that employs a practical method, not guaranteed to be optimal, perfect, but instead sufficient for reaching an immediate goal [55]. Where finding an optimal solution is impossible or impractical, heuristic methods can speed up the process of finding a satisfactory solution.

Heuristics can be mental shortcuts that ease the cognitive load of making a decision. A rule of thumb, a guesstimate, an educated guess, an intuitive judgment, a common sense and profiling are examples that employ heuristics.

Heuristic is the strategy derived from previous experiences with similar problems. This strategy relies on using readily accessible information to control problem solving in human beings and computers. The analysis of heuristic search procedures includes a classification of graph search strategies that put into perspective the approaches found in typical presentations of search procedures.

Weight $\chi(p)$ represents an integrated heuristic of selecting operator p in the partially generated pipeline, which is estimated as:

$$\chi(p) = \sum_{i=1}^{\kappa} \omega_i \cdot \rho_i(p), \tag{7.1}$$

where $\rho_i(p)$ is a heuristic parameter of operator p; ω_i is a factor at the heuristic parameter; k is a number of parameters. The heuristic factors ω must satisfy the equality as follows:

$$\sum_{i=1}^{\kappa} \omega_i = 1. \tag{7.2}$$

In the heuristic algorithm, the heuristic parameters are dynamically modified as they depend on the current optimization state, which is updated on passing from one loop iteration to another. The heuristic parameter $\rho_i(p)$ must meet the following two requirements. Its value has to vary in the range from 0 to 1. The higher value of the parameter, the better pipeline properties are expected. The operator $p \in Q$ with the maximum value of $\chi(p)$ is selected as the next candidate for scheduling.

Operator p^* whose weight $\chi(p^*)$ is a maximal one among all nonscheduled operators of set Q is the most preferable candidate for scheduling at the next step:

$$\chi(p^*) = \max_{p \in Q} \chi(p). \tag{7.3}$$

Let us consider in detail four heuristic parameters ρ_1–ρ_4, the heuristic technique HT uses and dynamically recalculates (see Section 5.7.3 of this book) at each step of the pipeline optimization process. Fig. 7.1. gives a strict definition of these parameters.

First parameter $\rho_1(p)$ is a complement-on-one of the relative dynamic mobility of operator $p \in Q$ over pipeline stages that are available for p. It is estimated on absolute *mobility*(p), minimal *mobility*$^{\min}$ and maximal *mobility*$^{\max}$ over all operator of Q. Low value of *mobility*(p) implies high value of ρ_1. This parameter will also be referred as *mob*.

Second parameter $\rho_2(p)$ is a relative *lbrs* difference over available for p pipeline stages, among all operator of Q. It is estimated over minimal *rslb*$^{\min}$(p) and maximal *rslb*$^{\max}$(p) on available stages, and maximal *drslb*$^{\max}$ on all operator of Q. The higher *lbrs* difference for p the higher value of ρ_2. This parameter will also be referred as *drslb*.

Third parameter $\rho_3(p)$ is a complement-on-one of the relative increase of minimal over stages *lbrs* for p against minimal *lbrs* over nonscheduled operator of Q. The parameter shows the increase of minimal *lbrs* after assignment of p to the best available stage against all operator of Q. It is estimated over minimal *rslb*$^{\min}$(p) of p on available stages, minimal *oprslb*$^{\min}$ and maximal *oprslb*$^{\max}$ on the set Q of nonscheduled operator. This parameter will also be referred as *mrslb*.

$\rho_1(p) = 1 - \frac{mobility(p) - mobility^{\min}}{mobility^{\max} - mobility^{\min}},$ $mobility(p) = late(p) - early(p) + 1,$ $mobility^{\min} = \min_{q \in Q} mobility(q),$ $mobility^{\max} = \max_{q \in Q} mobility(q)$	$\rho_2(p) = \frac{rslb^{\max}(p) - rslb^{\min}(p)}{drslb^{\max}},$ $rslb^{\min}(p) = \min_{early(p) \le s \le late(p)} rslb(p, s),$ $rslb^{\max}(p) = \max_{early(p) \le s \le late(p)} rslb(p, s),$ $drslb^{\max} = \max_{q \in Q}\left(rslb^{\max}(q) - rslb^{\min}(q)\right)$
$\rho_3(p) = 1 - \frac{rslb^{\min}(p) - oprslb^{\min}}{oprslb^{\max} - oprslb^{\min}},$ $oprslb^{\min} = \min_{q \in Q} rslb^{\min}(q),$ $oprslb^{\max} = \max_{q \in Q} rslb^{\max}(q)$	$\rho_4(p) = \frac{\lvert insize(p) - outsize(p) \rvert}{dsize^{\max}},$ $insize(p) = \sum_{v \in input(p)} size(v),$ $outsize(p) = \sum_{v \in output(p)} size(v),$ $dsize^{\max} = \max_{q \in Q} \lvert insize(q) - outsize(q) \rvert$

Figure 7.1. Heuristics ρ1–*mob*, ρ2–*drslb*, ρ3–*rslb* and ρ4–*dios* for dynamic ordering of operators at each step of pipeline optimization

Fourth parameter ρ4(*p*) is a relative difference between overall input variables size and overall output variables size of operator *p*. The parameter shows the importance of moving the operator over stages: moving should be done to the earliest stage if the output variables size is larger than the input variables size and vice versa. It is estimated over overall inputs size *insize*(*p*), outputs size *outsize*(*p*) and the maximal size difference *dsize*$^{\max}$ over all operator of *Q*. This parameter will also be referred as *dios*.

7.2. Motivation of tuning heuristics

The significance of the heuristic parameter ρi(*p*), *i* = 1 ... *k* in the integrated heuristic $\chi(p)$ is determined by the value of factor ω_i. The higher is the value, the more important is the parameter. Best optimization results are usually correlated with the use of most important heuristics. Searching for the best values of the factors in vector ω is a complicated optimization problem with many local optima.

Thus, in the pipeline optimization heuristic algorithm HT, the choice of the next scheduled operator essentially depends not only on four heuristic parameters ρ_1 - ρ_4, but also on the heuristic factors ω_1 - ω_4. Fig. 7.2 shows that for a 3-stage pipeline TB1000 the overall pipeline registers size varies between 871 and 1163 bits at $\omega_1=0$, $\omega_3=0,\dots,1$, $\omega_4=0,\dots,1$ and $\omega_2=1-\omega_1-\omega_3-\omega_4$. A high value of ω_i is associated with the high importance of the corresponding parameter ρ_i in the weight $\chi(p)$. A low value of ω_i is taken when the corresponding parameter ρ_i poorly recognizes the best solutions.

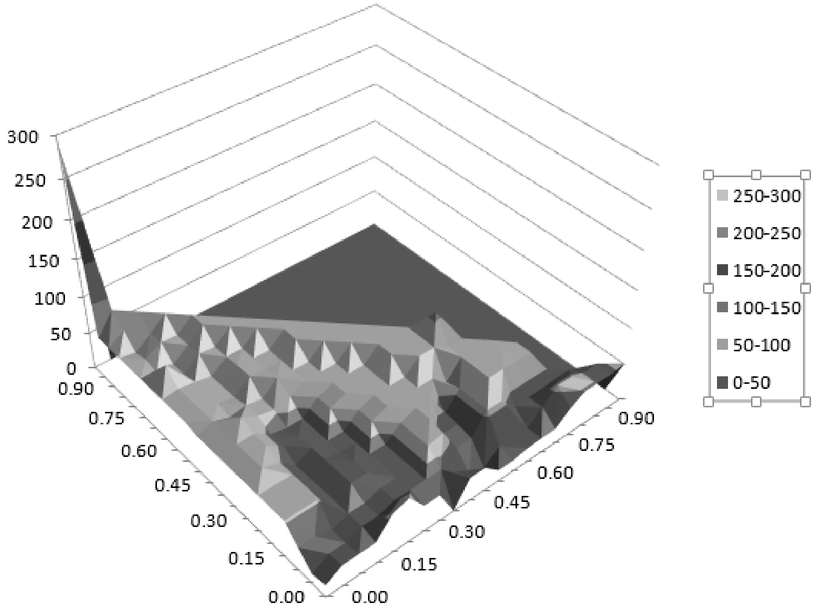


Figure 7.2. Overall pipeline registers size variations from 871 to 1163 in solution space projection 3-stage pipeline TB1000: heuristic factor ω_4 (*dios*) is horizontal axis, factor ω_3 (*mrslb*) is vertical axis, factor $\omega_1=0$ (*mob*), and factor $\omega_2= 1-\omega_1-\omega_3-\omega_4$ (*drslb*)

It is easy to see that finding an optimal value of vector ω is a complicated problem as the registers size has many local minima in the multidimensional space. This Chapter presents a genetic algorithm that is capable of efficiently solving heuristics tuning problem.

7.3. Genetic algorithm of tuning heuristics

7.3.1. Basics

The vector of heuristic factors $\omega=(\omega_1,\ldots,\omega_k)$ is an *individual*. The heuristic factor ω_i is a g*en*. The *population* is a set of individuals which exist during the genetic algorithm operation. The *generation* is a set of individuals which exist during one iteration of the genetic algorithm. A *fitness function* $F(\omega)$ of individual ω represents quality of the corresponding solution. In the pipeline optimization problem, it is determined over the objective function that is a minimum of the overall pipeline registers size $RS(\omega)$ obtained by HA. The fitness function $F(\omega)$ is a difference between the maximum of $RS(\omega^{\text{worst}})$ of the worst individual in the population and $RS(\omega)$ of individual ω.

7.3.2. Genetic operations

The selection operation aims at choosing parents to perform a crossover or mutation operation and produce a next generation of individuals.

The fitness proportionate selection (*FPS*) evaluates the fitness function $F(\omega)$ for each individual ω and normalizes each fitness value with dividing it by the sum of all fitness values. The sum of normalized values equals 1 and the values can be considered as probabilities. The population is sorted on descending of fitness values. Accumulated normalized fitness values are computed, a random number *r* between 0 and 1 is chosen and the selected individual is the first one whose accumulated normalized value is greater than *r*.

The *worst parent selection* (*WPS*) chooses a parent with the worst fitness value and replaces it in the next generation with the best offspring in case the fitness value of the offspring is larger than fitness value of the parent.

The worst individual selection (*WIS*) chooses the individual with the worst fitness value in the current population and replaces it with the best offspring in case the fitness value of the offspring is larger than fitness value of the individual.

The half uniform crossover (*HUX*) chooses randomly half of gen indices that are represented with a subset K^1 of the set $K=\{1,\ldots,k\}$. *HUX* is a partially matched crossover. The simple recombination of parent's gens is not sufficient for obtaining a correct offspring as for the new individual the sum of heuristic factors may appear unequal to 1. Two following cases for two parents are differentiated.

Case 1. The fitness values of ω^1 and ω^2 are approximately equal: $F(\omega^1)\approx(F(\omega^2)$. In this case *HUX* tries to save the genotype of parent ω^1 in the first offspring and the genotype of parent ω^2 in the second offspring. The first offspring ω^3 is constructed of original gens of parent ω^1 which are indexed with $i\in K^1$ and of normalized gens of parent ω^2 which are indexed with $i\in K\backslash K^1$. The second offspring ω^4 is constructed of original gens of parent ω^2 which are indexed with $i\in K\backslash K^1$ and of normalized gens of parent ω^1 which are indexed with $i\in K^1$. The gen normalization is performed with the ratios as follows:

$$a=\sum_{i\in K^1}\omega_i^1\ , \tag{7.4}$$

$$b=\sum_{i\in K^1}\omega_i^2\ , \tag{7.5}$$

$$\gamma^1=(1-a)/(1-b), \tag{7.6}$$

$$\gamma^2=b/a\,. \tag{7.7}$$

Ratio γ^1 aims at the normalization of gens of ω^2 in offspring ω^3: $\omega^3{}_\mathrm{i}=\gamma^1\times\omega^2{}_\mathrm{i}$ for $i\in K\backslash K^1$. Ratio γ^2 aims at the normalization of gens of ω^1 in offspring ω^4: $\omega^4{}_\mathrm{i}=\gamma^2\times\omega^1{}_\mathrm{i}$ for $i\in K^1$.

Case 2. The fitness value of ω^1 significantly exceeds the fitness value of ω^2: $F(\omega^1)>(F(\omega^2)$. In this case *HUX* tries to save the genotype of parent ω^1 in both offspring. The first offspring is the same as ω^3. The second offspring ω^5 is constructed of original gens of parent ω^1 which are indexed with $i \in K \backslash K^1$ and of the normalized gens of ω^2 which are indexed with $i \in K^1$. The gen normalization is performed as $\omega^5{}_{\mathrm{i}}=\omega^2{}_{\mathrm{i}}/\gamma^2$ for $i \in K^1$.

An illustration of two cases of crossover *HUX* is given in Fig. 7.3. In case 1 the fitness values of 175 and 173 of two parents are close, and each offspring ω^3 and ω^4 tries to save the genotype of both parents. In case 2 the fitness value of 175 of first parent is significantly higher than the fitness value of 92 of second parent and each offspring, ω^3 and ω^5 tries to save the genotype of the first parent.

$\omega^1=(0.31, 0.47, 0.09, 0.13)$, $\omega^2=(0.25, 0.14, 0.53, 0.08)$
$K^1=\{1, 3\}$, $K \backslash K^1=\{2, 4\}$
$\mathrm{a}=0.31+0.09=0.4$, $\mathrm{b}=0.25+0.53=0.78$
$\gamma^1=(1-0.4)/(1-0.78)=2.73$, $\gamma^2=0.78/0.4=1.95$
$\omega^3=(0.31, 0.38, 0.09, 0.22)$

Case 1:	Case 2:
$F(\omega^1)=175$, $F(\omega^2)=173$	$F(\omega^1)=175$, $F(\omega^2)=92$
$\omega^4=(0.60, 0.14, 0.18, 0.08)$	$\omega^5=(0.13, 0.47, 0.27, 0.13)$

Figure 7.3. Illustration of crossover *HUX*

The single offspring crossover (*SOX*) takes two parents, ω^1 and ω^2 and produces one individual. Firstly, the heuristic factor weights α and β are computed as:

$$\alpha = F(\omega^1) / (F(\omega^1) + F(\omega^2)), \tag{7.8}$$

$$\beta = 1 - \alpha. \tag{7.9}$$

Secondly, the single offspring ω is calculated as a vector of weighted sum of parent gens:

$$\omega_{\mathrm{i}} = \alpha \times \omega^1{}_{\mathrm{i}} + \beta \times \omega^2{}_{\mathrm{i}} \quad \text{for } i=1\ldots k. \tag{7.10}$$

SOX:
ω^1=(0.31, 0.47, 0.09, 0.13)
ω^2=(0.25, 0.14, 0.53, 0.08)
$F(\omega^1)$=175
$F(\omega^2)$=92
α=175/(175+92)=0.655
β=1–0.655=0.345
ω=(0.29, 0.36, 0.24, 0.11)

TGM:
ε=0.3
ω=(0.31, 0.47, 0.09, 0.13)
i=2, j=4
δ=0.47*0.3=0.14
ω_2=0.47–0.14=0.33
ω_4=0.13+0.14=0.27
ω=(0.31, 0.33, 0.09, 0.27)

Figure 7.4. Illustration of crossover *SOX* and mutation *TGM*

The offspring meets all the requirements to the individual. Its gens are closer to the first parent if $F(\omega^1)>F(\omega^2)$, and are closer to the second parent otherwise. This crossover tries to scan the region of the search space that is closer to the point with best fitness function. An illustration of crossover *SOX* is given in Fig. 7.4, left. The fitness values of 175 and 92 of two parents are used to calculate factors α and β. Value 0.655 of α is higher than value 0.345 of β. Therefore, offspring ω is closer to parent ω^1 over parent ω^2.

The two gene mutation (*TGM*) alters two heuristic factor values in one parent ω^1 from its initial state. The heuristic factors ω^1_i and ω^1_j are selected randomly. The corresponding factor values in the single offspring ω are calculates with a *mutation factor* ε whose value satisfies inequality $0<\varepsilon<1$:

$$\delta = \varepsilon \times \omega^1_{i},, \tag{7.11}$$

$$\omega_i = \omega^1_i - \delta, \tag{7.12}$$

$$\omega_j = \omega^1_j + \delta. \tag{7.13}$$

TGM is capable of correctly changing the value of any two heuristic factors in opposite direction. An illustration of *TGM* is given in Fig. 7.4, right. The value of randomly chosen heuristic factor ω_2 is decreased by 0.14 at the mutation factor ε=0.3 and the value of ω_4 is increased by 0.14. To determine what will be performed next, crossover or mutation, two probabilities are used: p_{cross} and p_{mut}.

7.3.3. Genetic algorithm

Fig. 7.5 summarizes the genetic algorithm (GA). GA consists of an initialization stage and a loop that iteratively updates the population by means of genetic operators in such a way as to find a schedule with the minimal overall pipeline registers size. For a small design or a large design with few pipeline stages, the exit condition is defined over the maximum number of iterations, which give no improvement of the best individual. For a large design and/or a large number of pipeline stages, it is defined over a CPU time constraint.

GA is a random strategy at all steps of its operation. It randomly chooses the genetic operation, randomly chooses parents for performing crossover and mutation operations, randomly performs these operations, and randomly updates the population of individuals.

```
1. Produce initial population by repeatedly generating k-1 random
      numbers $\mu_i$, $i$=1…$k$-1 between 0 and 1, ordering the numbers
      on ascending, computing next individual as $\omega=(\mu_1,$
      $\mu_2-\mu_1,\ldots,\mu_{k-1}-\mu_{k-2}, 1-\mu_{k-1})$, and adding it to the population.
2. Perform the heuristic algorithm for each individual $\omega$ that is
      interpreted as a vector of heuristic factors, find the worst individual,
      compute the fitness function $F(\omega)$ for all individuals
      and reorder the individuals on descending of $F(\omega)$.
3. while (not Exit condition) do
      4. Randomly choose genetic operation, crossover or mutation
            with probabilities $p_{\mathrm{cross}}$ and $p_{\mathrm{mut}}$ respectively.
      5. Randomly choose parents using selection operation FPS.
      6. Perform crossover HUCX or SOCX and obtain two or one
            offspring.
      7. Perform the heuristic algorithm for each offspring to obtain
            $F(\omega)$ for each offspring $\omega$.
      8. Perform selection operation WPS or WIS to update population.
end while
9. Return the best individual.
```

Figure 7.5. Genetic algorithm for optimization of heuristic factors

7.4. Two modes of exploiting the genetic algorithm

GA can be exploited in two modes: (1) while actually solving the optimization problem in real time, and (2) during accumulation of knowledge on the best heuristic factors. In the first mode, GA searches for the best heuristic factors for one set of input data of the heuristic algorithm. In the second mode, GA accompanies the heuristic algorithm regarding heuristic factors, which executes many times on various input data. It results in generating cumulative distribution functions for all heuristic factors.

7.4.1. Solving optimization problem

The first mode of actual solving the optimization task over tuning the heuristic factors requires GA to be capable of generating at least 50-100 individuals in population in acceptable CPU time. In this case the runtime of the heuristic algorithm (in particular the runtime of HT of pipeline optimization) should not exceed 1-2 sec.

For Intel i3 CPU it is feasible for 1-7 stage pipelines for the design size of 1000 operators. For larger designs, GA can find a high quality solution for only 2-3 stage pipelines. Of course GA works perfectly for designs of <1000 operators.

7.4.2. Evaluation of cumulative distribution functions

The second mode aims at preliminary extracting and accumulating knowledge on the heuristics and on the best heuristic factors $\omega_1 - \omega_k$, which describe importance of these heuristics in the integrated optimization criterion. In this case, the GA runtime constraint may be taken of tens and hundreds of minute.

Vector ω^{best} of the best heuristic factors can be treated as a random continuous variable. Its probability distribution can be estimated based on the multiple execution of GA on various design and various number of pipeline stages. Let U be the number of GA runs and ω^u is the vector of best heuristic factor values for the u^{th} run, u=1…U. Each projection ω_i^u, i=1…k of the vector can be represented with a histogram $h_i(j)$,

j=1…w that divides (Fig. 7.6) the interval [0,1] of ω_i^u values into w parts with $step$=1/w. The j^{th} subinterval, j=1…w includes the factor values of the range from (j–1)×$step$ to j×$step$. The value of $h_i(j)$ is the number of vector factors whose value belong to the j^{th} subinterval in projection ω_i^u.

Fig. 7.6 gives an example of the histogram for factor ω_2 at the *drslb* heuristic. The histogram aims at estimating the probability density function $f_i(j)=h_i(j)/U$, and the cumulative probability distribution function, $Q_i(j)=\sum_{v=1}^{j} f_i(v)$. Fig. 7.7 provides an example of the cumulative probability function of factor ω_2.

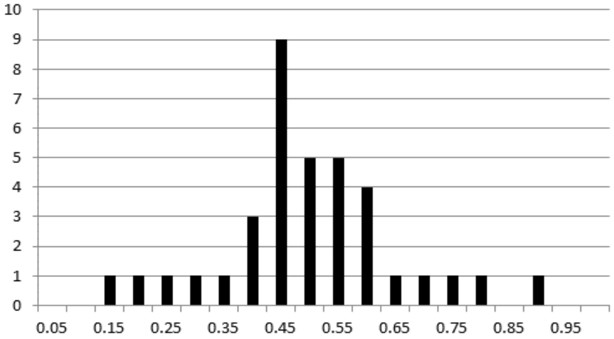


Figure 7.6. Histogram of the best value of factor ω_2 (*drslb*) in HT

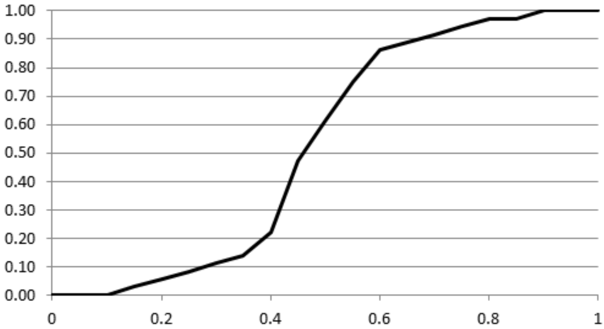


Figure 7.7. Estimation of cumulative probability distribution function (CDF) for the best value of factor ω_2 (*drslb*)

7.4.3. Random algorithm of searching for heuristic factors

In their turn, the cumulative function is a basis for efficient random search for an optimal solution using the heuristic algorithm. This search can produce the initial generation of individuals and initialize the population in GA. Fig. 7.8 presents a random search algorithm (RA) for producing promising heuristic factors. For each heuristic, it firstly generates a uniform random number, determines the lowest subinterval whose cumulative function value is not less than the random number, and calculates a preliminary heuristic factor value. Secondly, it normalizes the vector of preliminary factor values by means of computing the values sum and dividing each factor value by this sum. And finally, it calls the heuristic algorithm with the randomly obtained factors, which returns the value of fitness function. This procedure repeats until the CPU time constraint is not met.

```
51. for each $i$ in $\{1\dots k\}$ do
    2. Generate random number $\mu_i$ with uniform probability distribution.
    3. Find lowest value $j$ between 1 and $w$ such that $Q_i(j) \geq \mu_i$.
    4. Assign $\omega_i = j \times step$.
  end for
5. Calculate sum $s$ of $\omega_i$, $i=1\dots k$.
6. For $i$ in $\{1\dots k\}$ assign $\omega_i := \omega_i/s$ (normalization) .
7. Compute fitness function $F(\omega)$ by call the heuristic algorithm with
   heuristic factors $\omega=(\omega_1,\dots,\omega_k)$ as actual parameters.
```

Figure 7.8. Random search algorithm (RA) for generating heuristic factors

7.5. Experimental results

7.5.1. Test benches for pipeline optimization

The random test benches (TB1000-TB5000) that are described in [79] and consist of 1000-5000 operators are exploited intensively in this book in order to study properties of the heuristics and algorithms that have been proposed. TB1000-TB5000 are constructed of such operators as addition, subtraction, multiplication and logical operators. The probability of appearing the operators is 0.3, 0.25, 0.1 and 0.35 and the operator relative delay is 1.0, 1.1, 3.0 and 0.1 respectively.

The variable size varies between 4 and 28 and equals to 15.83 on average. The design critical path length is about 10% of the total operators delay for all test benches. All experiments were performed on the Intel® Core™ i3 CPU 550 @ 3.20 GHz 3.19 GHz, 4 GB.

7.5.2. Optimization of heuristics

We have written a computer program that measures the effectiveness of each of four heuristics for pipeline optimization with respect to the registers size minimization. The program divides each axis of the search space, which is associated with a heuristic factor, into intervals by means of 21 points from 0.0 to 1.0 with the step of 0.05. As only three of four heuristics are mutually independent, it represents the search space with 1771 points corresponding to various combinations of the factor values. Fig. 7.9 shows that the number of combinations equals 1 if one of the factors has the value of 1.0 and the number equals 231 if one of the factors has the value of 0.0. The program computes the overall pipeline registers size using HT for each point of the search space.

Two of four heuristics can be used in the static mode. These are ω_1 (*mob*) and $\omega_4=1-\omega_1$ (*dios*). In this case, the optimization space includes only 21 points (Fig. 7.10). The statically heuristic algorithm orders operators before optimizing the pipeline. The pipeline optimization results can significantly depend on what point has been chosen. The value of 0.25 of the static heuristic factor ω_1 decreases the registers size *RS* by 12.1% over the value of 0.60.

Fig. 7.11 presents a minimum of *RS* for TB1000 3-stage pipeline among all factor value combinations. The average minimum *RS* is equal to 881.1 for ω_2, is equal to 887.19 for ω_4, is equal to 887.95 for ω_1 and is equal to 923.38 for ω_3. Fig. 7.12 also proves the effectiveness of the heuristic ω_2. The average *RS* decreases with increasing the value of ω_2. This is a sign of high importance of *drslb* in the weighted criterion (7.1). The importance of other heuristics decreases in the order as follows: ω_4, ω_1 and ω_3. It can be seen, the lower value of ω_3 implies the lower *RS*.

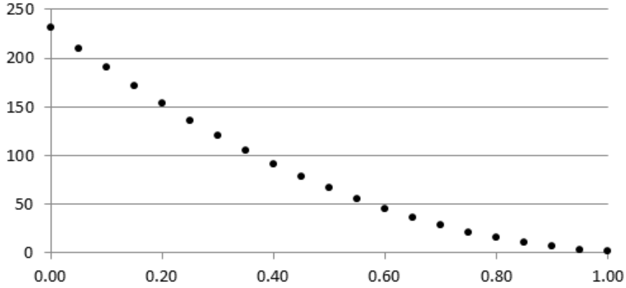


Figure 7.9. Number of combinations of three heuristic factors values vs. the factor value of selected heuristic (21 values for one factor)

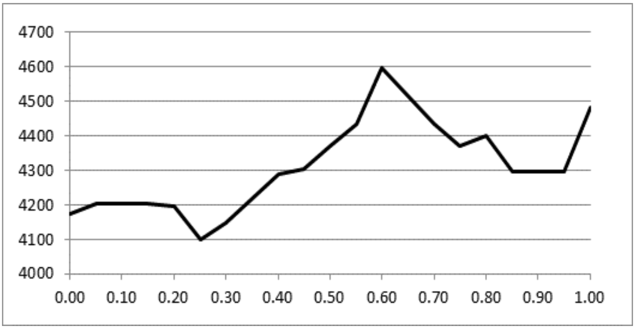


Figure 7.10. Overall pipeline registers size *RS* (bits) for 10-stage pipeline TB1000 obtained by heuristic algorithm that uses static heuristics vs. heuristic factor ω_1 (*mob*)

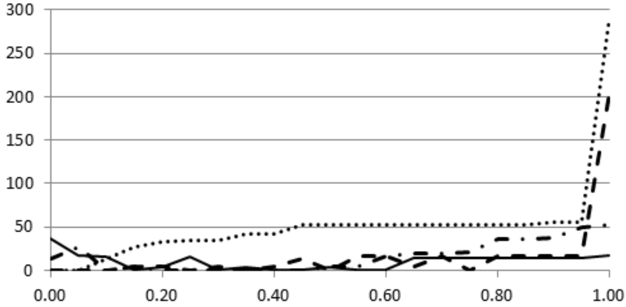


Figure 7.11. Minimum of overall registers size subtracted by 871 vs. heuristic factor value: ω_1–*mob* (dash), ω_2–*drslb* (solid), ω_3–*mrslb* (round dot), ω_4–*dios* (dash dot)

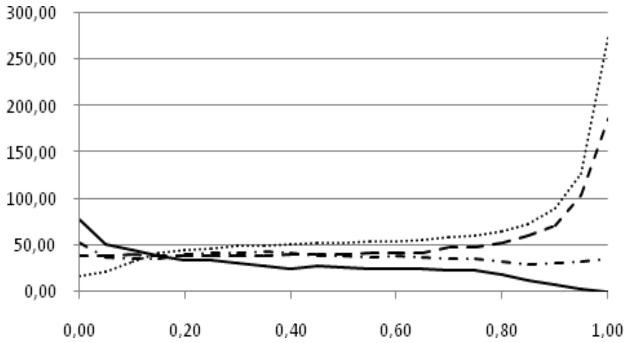


Figure 7.12. Average overall registers size subtracted by 888 vs. heuristic factor value: ω_1–*mob* (dash), ω_2–*drslb* (solid), ω_3–*mrslb* (round dot), ω_4–*dios* (dash dot)

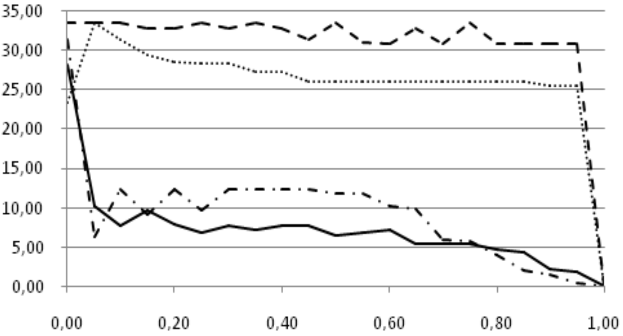


Figure 7.13. Overall registers size range (%) vs. heuristic factor value: ω_1–*mob* (dash), ω_2–*drslb* (solid), ω_3–*mrslb* (round dot), ω_4–*dios* (dash dot)

It is important for the optimization, what heuristic factor is capable of changing *RS* and in what direction. Fig. 7.13 reports that choosing a particular value of ω_1 or ω_3 allows large variations in *RS* due to varying the value of other factors. At the same time an appropriate selection of the value of ω_2 or ω_4 reduces the variations and may lead to rapidly finding a minimum of *RS*. It should be noted that there is a slight correlation between *drslb* and *dios* and between *drslb* and *mob*.

7.5.3. Cumulative distribution probability functions of heuristic factors

GA is capable of obtaining the best heuristic factor values for various deigns and various number of pipeline stages. The cumulative distribution probability functions (CDF) that are shown in Fig. 7.14 are generated on the best values of heuristic factors that result from numerous optimization runs for the designs TB1000-TB5000.

The average values of the best factors at the *mob*, *drslb*, *mrslb* and *dios* heuristics are as follows: ω_2=0.466, ω_1=0.292, ω_4=0.186 and ω_3=0.056. Each best factor takes values in a restricted interval. Thus, ω_2 should be between 0.15 and 0.9, ω_1 should be between 0.0 and 0.6, ω_4 should be between 0.0 and 0.55, and ω_3 should be between 0.0 and 0.25.

CDFs that are presented in Fig. 7.14 are an effective facility for generating the initial population in GA using the random algorithm RA (Fig. 7.8). These functions can be also used as a fast solution search tool in the case when only few HT runs can be done in an acceptable runtime.

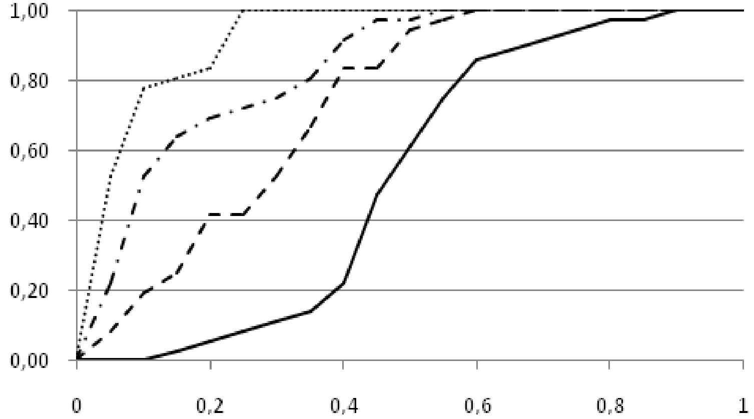


Figure 7.14. Cumulative probability distribution functions (CDF) for best heuristic factors: ω_1–*mob* (dash), ω_2–*drslb* (solid), ω_3–*mrslb* (round dot), ω_4–*dios* (dash dot)

7.5.4. Tuning genetic algorithm

In order to properly choose in each design case the most efficient genetic operations among those proposed in section 7.3.2, several experiments have been done on large designs. Two of three curves that are shown in Fig. 7.15 compare two crossovers *HUX* and *SOX* in case when the mutation operation is not used. The first 50 individuals are generated randomly with a uniform probability distribution. Starting conditions for *SOX* (884.8) have appeared to be preferable over starting conditions for HUX (885.8).

But very quickly (after generating the 80th individual) *HUX* started to give the registers size *RS* much lower than *SOX* and this difference increases with evolution of the population. The conclusion is as follows: *SOX* is preferable on a restricted population size and *HUX* is preferable when more individuals can be generated. Both crossovers can be used in the same genetic algorithm. The choice of one of them can be performed randomly at each iteration of the genetic algorithm.

Third curve *CDF-HUX* in Fig. 7.15 shows that the replacement of the uniform probability distribution with the cumulative probability distribution functions for the heuristics factors that are shown in Fig. 7.14 speeds up the reduction of *RS* for a low-size population but can give a worse result for a large-size population. The random search algorithm RA generates individuals (heuristic factors) in the initial population and can be used for implementing the mutation operation.

Several experiments have been done in order to formulate the rules of choice between the operations of worst parent selection (*WPS*) and worst individual in population selection (*WIS*). Both operations give close results very frequently for the pipeline optimization task. At the same time, *WPS* may appear prefarable over *WIS* as it can preserve the diverse genofond of the population. On its turn, *WIS* can produce a population that includes a lot of very close individuals. For designs that can be optimized with a large population, probability p_{cross} of crossover may be close to 1. For designs that can be represented with a small population, probability p_{mut} of mutation and mutation factor ε should be increased as in this case the search space can be scanned more thoroughly.

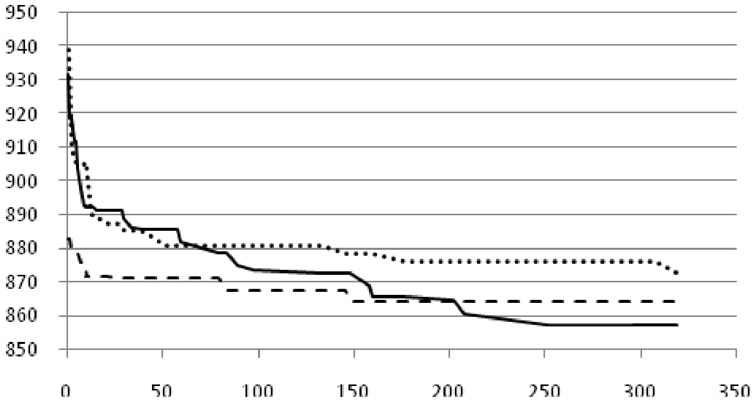


Figure 7.15. Overall registers size *RS* (bit) in 3-stage pipeline optimized by crossover HUX (solid), SOX (round dot) and CDF-HUX (dash) vs. population size (average on 5 runs of TB1000 for each crossover)

7.5.5. Effectiveness of genetic algorithm

Fig. 7.16 shows the dependency of the overall pipeline registers size *RS* on the design size, which is obtained for three optimization algorithms: ASAP, ALAP and GA. The size varies from 1000 to 5000 operators, and the registers size varies from 1288 to 4237 bits. GA overcomes ASAP by 50.31%, 50.66%, 49.98%, 43.34% and 35.02% for the design size of 1000, 2000, 3000, 4000 and 5000 operators respectively. GA has 105.43% of gain over ALAP for the design size of 1000 operators. Then the gain reduces to 43.95%, 36.68%, 31.95% and 35.63% for the size of 2000, 3000, 4000 and 5000 operators respectively. The gain is obtained due to the own properties of the heuristic algorithm (about 70%), and due to the optimization of heuristic factors by the genetic algorithm GA (about 30%).

The design size significantly influences the runtime of the heuristic algorithm in particular, and influences the runtime of GA in general. Within 100 sec of CPU, the genetic algorithm GA has generated 874, 294, 95, 65 and 79 individuals of the population for the design size of 1000, 2000, 3000, 4000 and 5000 operators respectively.

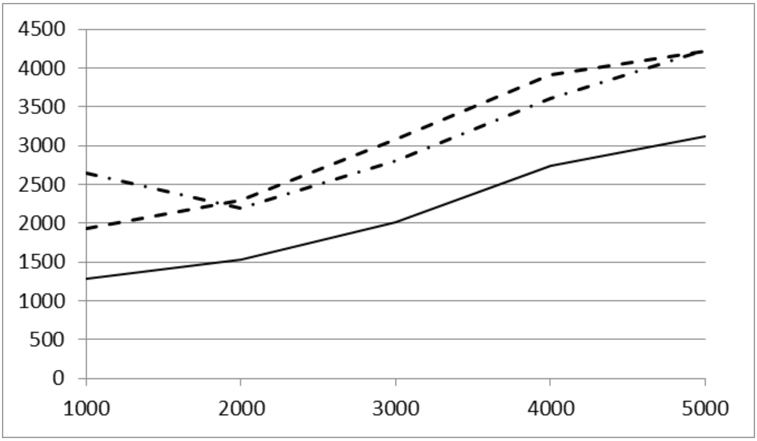


Figure 7.16. Overall registers size *RS* (bit) in 4-stage pipeline optimized by GA (sold), ASAP (dash) и ALAP (dash dot) vs. design size

7.6. Conclusion

Exact optimization techniques yield a global optimum solution for small-size problems. Heuristic optimization techniques are capable of handling large-size problems but are not able to guarantee finding an exact solution. They can find a close to optimum solution, which depends on heuristics that are exploited.

Usually several heuristics can be incorporated in a heuristic algorithm. It is difficult to determine, which of them are more important, and which are less important. In this chapter, we have presented a genetic algorithm, which can search for an optimal heuristic factor for each heuristic that is exploited. The factor determines the importance of the given heuristic in an integrated heuristic, which recognizes preferable solutions during solving the optimization problem.

8. NET ALGORITHMS

A schedule for a sequential finite state machine defines a distribution of statements on control steps taking into account constraints on time and resources. A net schedule defines both a partial precedence and concurrent execution of the statements under the same constraints.

This chapter introduces a net scheduling and allocation model, a method, and techniques that allow to generate net schedules which minimize either the execution time or resources. The net schedule is a source to synthesize a sequential schedule with chaining, multi-cycling, and pipelining. Moreover the net schedule can be directly mapped to a computing architecture or a parallel program. Experimental results show that the net schedule execution time is more than 20% less than the sequential schedule execution time in the case of variable execution time of operators, statements and program code fragments.

The theoretical models and methods of this chapter can first of all be applied to the design and optimization of digital systems. Thus they are implemented in the Ahiles VHDL-based high-level synthesis system which is described below. Additionally these models and methods can be used for the generation and optimization of parallel programs.

8.1. Sequential scheduling of algorithms

Scheduling is the first task in the synthesis process. Its results are most important for the final parameters of the design. It should be noted, the scheduling task is a NP-hard problem.

The known scheduling techniques such as ASAP, ALAP, list, freedom-based, force-directed, path-based scheduling, and integer linear programming formulation use the precedence relation between operators/statements which is extracted from the data and control flow graphs as input data for sequential scheduling.

Usually two optimization criteria are considered during the scheduling: to minimize the execution time and to minimize the resources. In the first case statements are parallelized to execute on the same control step. In the second case, statements are distributed on different control steps to execute on a same functional unit.

Two basic scheduling techniques synthesize the sequential schedule with the shortest execution time and maximum resources. These are “as soon as possible” (ASAP) and “as late as possible” (ALAP) [53]. ASAP schedules statements on the control steps from the first to the last. A statement is scheduled immediately if its predecessors have been scheduled. ALAP schedules statements on the control steps from the last to the first. A statement is scheduled immediately if its successors have been scheduled.

```
entity DIFFEQ is
  port(DXP,AP,XP,UP: in BIT_VECTOR(7 downto 0);
        YP : inout BIT_VECTOR(7 downto 0);
        CLOCK,START: in BIT;
        READY : out BIT);
end DIFFEQ;
architecture BEHAVIOR of DIFFEQ is
begin
  process
      variable DX,A,X,Y,U : BIT_VECTOR(7 downto 0);
      variable B,C,D,E,H,G : BIT_VECTOR(7 downto 0);
      variable R:BOOLEAN;
  begin
      wait until CLOCK'EVENT and CLOCK='1' and START='1';
      READY<='0'; DX:=DXP; A:=AP; X:=XP; Y:=YP; U:=UP;
      loop
          R:=X<A;                                          --1
          exit when not R;                                 --2
          C:=X+(2*X);                                      --3
          B:=U*DX;                                         --4
          D:=B*C;                                          --5
          G:=U-D;                                          --6
          E:=Y*DX;                                         --7
          H:=E+(2*E);                                      --8
          U:=G-H;                                          --9
          X:=X+DX;                                         --10
          Y:=Y+B;                                          --11
      end loop;
      wait until CLOCK'EVENT and CLOCK='1';
      READY<='1'; YP<=Y;
  end process;
end BEHAVIOR;
```

Figure 8.1. Differential equation integrating algorithm (*DiffEq*) in VHDL

The differential equation integrating algorithm (*DiffEq*) [20] shown in Fig. 8.1 and represented in VHDL [45] is used in this paper to illustrate scheduling techniques. ASAP scheduling is given in Fig. 8.2 and ALAP scheduling is given in Fig. 8.3.

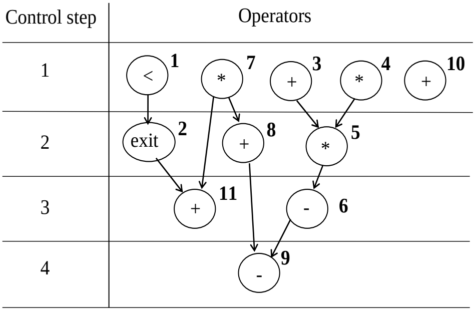


Figure 8.2. How ASAP scheduling handles *DiffEq*

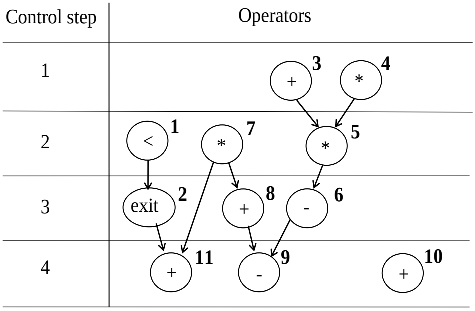


Figure 8.3. How ALAP scheduling handles *DiffEq*

List scheduling is a resources scheduling technique [53]. It assumes a number of function units of each type to be given. The technique schedules statements consecutively from the first to the last control steps, taking into account the constraints on resources. List scheduling for *DiffEq* with two ALUs and one multiplier is show in Fig. 8.4.

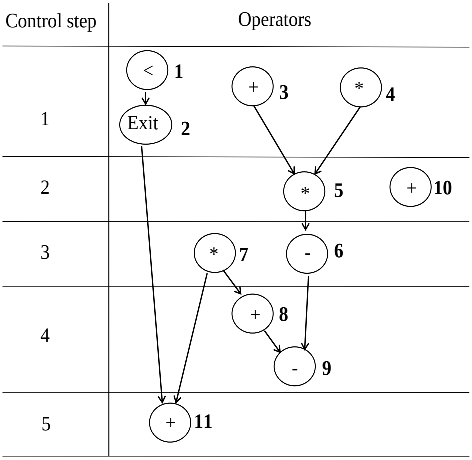


Figure 8.4. How list scheduling handles *DiffEq*

Freedom-based scheduling maps statements onto control steps taking into account the range of the steps which could be paired with the statement. The statements on the critical path can be assigned to the tightest range of steps and have to be scheduled at the beginning of the scheduling process.

Force-directed scheduling is a time-constrained scheduling technique. The technique schedules statements step by step in accordance with "force" values. Each scheduling step follows by the re-evaluation of the "force" values.

Integer linear programming formulation can be resource, time, and feasible constrained scheduling technique. It can find optimal solutions for practical problems. The described scheduling techniques can generate schedules with chaining, multicycling, pipelining. They assume the operator execution time to be constant.

The goal of allocation is to minimize computational resources. Effective allocation algorithms are based on interference and preference graph coloring. The goal of allocation is to minimize the resources.

The algorithm operations are mapped onto functional units, variables are mapped onto registers, and data dependences are mapped onto multiplexers, buses and partitioned buses. It should be noted, the allocation task is a NP-hard problem.

After the data path is synthesized, control signals (select signals for multi-functional units and multiplexers, load-enable signals for registers) are introduced and a finite state machine is generated based on the scheduled behavioral description. The data path and finite state machine are described in a hardware description language and used as input for logic synthesis.

Effective high-level synthesis methodologies, algorithms and systems (AMICAL, Cathedral, CMUDA, DAA, ELLA, HAL, HIS, Yorktown Silicon Compiler, PASS, PSAL2, Sehwa, and others) for digital circuits have been developed [20, 53, 54]. Using a source behavioral description in a hardware description language (for example VHDL [62]) they design a register transfer level (RTL) structure consisting of two parts: the data path (DP) and the control unit (CU). To synthesize the data path, the following tasks are solved: compiling a behavioral description into an internal form, control and data flow graphs (CFG and DFG) generation, analyzing these graphs, scheduling, and allocation of statements. Known scheduling techniques synthesizing a sequential schedule introduce control steps and finite state machine (FSM) states into the behavior distributing the statements on the control steps.

8.2. Net scheduling of algorithms

8.2.1. Net schedule

The statements, control steps, and FSM states are considered in the sequential schedule. The sequential schedule describes a distribution of the statements onto the control steps and FSM states. Net scheduling does not introduce control steps and states; it defines only precedence and concurrency between statements, which conserves both time and resources [60].

Let $N=\{1,...n\}$ be a set of the statement numbers. Directed graph $G_H=(N,H)$ can describe the net schedule, where H is the statement's direct precedence relation. If statements $i_1,...,i_k$ are direct predecessors of statement *j* in the net schedule, then *j* may execute when all of its predecessors have finished executing.

In a binary matrix, an element of the matrix can take one of two values. Elements of a triple matrix can take one of three elements.

Binary matrix *Q* describes data dependences between the statements, in which element $q_{i,j}$ equals 1 if *i* is a predecessor of *j*, and equals 0 otherwise. In triple matrix *W*, element $w_{i,j}$ equals

- 0 if the statements *i* and *j* may not execute on the same functional unit
- 1 if the statements may execute on the same functional unit sequentially
- 2 if the statements may execute on the same functional unit concurrently

The last case applies when the statements are orthogonal [57]; that is the statements are “*if c1 then P1; end if;*“ and “*if c2 then P2;* end if;” and conditional signals or variables *c*1 and *c*2 are orthogonal (their conjunction equals false). We can equivalently transform any VHDL behavioral description, without changing mapping functions, to the form consisting of if-then statements and loop-statements without an iteration scheme [69, 70]. Note that an orthogonal statement cannot precede another orthogonal statement. For the *DiffEq* in Fig. 8.1, Ahiles gives us the matrices *Q* and *W* in Fig. 8.5, assuming that "<", "+", "-" operators execute on the same ALU. Statements *i* and *j* are sequential if a path exists between *i* and j on the graph G_H, otherwise the statements are concurrent.

$$Q = \begin{bmatrix} 0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&1&1&0&0&1&0&0 \\ 0&0&0&0&1&1&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&1&0&1 \\ 0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix} \qquad W = \begin{bmatrix} 0&0&1&0&0&1&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&1&0&1&1&1&1 \\ 0&0&0&0&1&0&1&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0&0 \\ 1&0&1&0&0&0&0&1&1&1&1 \\ 0&0&0&1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&1&0&0&1&1&1 \\ 1&0&1&0&0&1&0&1&0&1&1 \\ 1&0&1&0&0&1&0&1&1&0&1 \\ 1&0&1&0&0&1&0&1&1&1&0 \end{bmatrix}$$

Figure 8.5. Matrices for *DiffEq*

Two statements are mutually exclusive in the net schedule if they never execute simultaneously. For this to be true, the statements must be orthogonal or sequential. If mutually exclusive statements may execute on the same functional unit, they are compatible and can share same resources.

The zero elements of the top part of the *Q* matrix define maximum set $D_M = \{(i,j) \mid q_{i,j}=0,\ i<j\}$ of the concurrent statements pairs. Among the pairs of set D_M are the pairs of set D_O, which are orthogonal statements. Set $D=D_M$ defines the net schedule of maximum concurrency. The nodes and arcs in Fig. 8.2 represent the most concurrent schedule for *DiffEq*.

Let t_{j} and s_{j} be functional unit *j's* execution time and cost respectively. Time t_{j} can be constant or variable. If *fu*(*i*) denotes the type of functional unit executing statement *i*, net schedule execution time is

$$T = \max_{u \in U_{\overline{D}}} \sum_{i \in u} t_{fu(i)}\ , \tag{8.1}$$

where $U_{\overline{D}}$ is the set of cliques of graph $G_{\overline{D}} = \left(N, \overline{D}\right)$ constructed on set *N* of the nodes that represent statements and on set $\overline{D}$ of the edges that represent sequential statements pairs. The graph $G_{\overline{D}}$ clique that gives the maximum sum of the statements' execution time defines the execution time. Schedule cost is

$$S = \sum_{j=1}^{N_{FU}} s_j \times \left(\max_{v \in V_D} m_{jv} \right), \tag{8.2}$$

where N_{FU} is the number of functional unit types. V_D is the set of the cliques of graph $G_{\mathrm{D}} = (N, D)$ constructed on set *N* of the nodes and on set *D* of the edges that are the pairs of concurrent statements. The number of the functional units of type *j* needed to execute clique *v* statements concurrently is m_{jv}.

The sum of costs of the various functional unit types defines the total cost. Clique set V_D provides the number of functional units of any type. Sets $U_{\overline{D}}$ and V_D describe the maximum paths and sections on graph G_H. Hence, the longest path defines the execution time, and the widest section defines the cost.

8.2.2. Optimizing a net schedule

To optimize a schedule, net scheduling must meet one of two goals:

- minimizing the execution time with given constraints on the resources
- minimizing the resources with given constraints on the execution time

While set D_M determines the most concurrent (and thus fastest) net schedule, subset D of D_M determines a net schedule of less concurrency, yet lower system cost. Set D also defines execution time T and cost S.

Finding pairs of concurrent operators

Ahiles can find up to 2^r different net schedules, where r is the cardinality of set D_M.

Because of the concurrent execution of any pair of set D_O of orthogonal statements which does not require additional execution time and resources, we can always include D_O into D. For instance, *DiffEq* can be potentially a source for generating 2^{39} net schedules.

Synthesizing a net schedule involves solving one of two optimization tasks, depending on the optimization criteria selected:

$$\min_{D \subseteq D_M} \{T_D \mid S_D \le S_0\} \tag{8.3}$$

or

$$\min_{D \subseteq D_M} \{S_D \mid T_D \le T_0\}, \tag{8.4}$$

where T_O and S_O are the constraints on execution time and cost. To account for execution time (Equation 8.1) and cost (Equation 8.2) estimates, we reformulate Equation 8.3 and 8.4 as the tasks

$$\min_{D\subseteq D_M} \left\{ \max_{u\in U_{\bar{D}}} \sum_{i\in u} t_{fu(i)} \;\middle|\; \sum_{j=1}^{N_{FU}} s_j \times \left(\max_{v\in V_D} m_{jv} \right) \le S_0 \right\} \qquad (8.5)$$

and

$$\min_{D\subseteq D_M} \left\{ \sum_{j=1}^{N_{FU}} s_j \times \left(\max_{v\in V_D} m_{jv} \right) \;\middle|\; \max_{u\in U_{\bar{D}}} \sum_{i\in u} t_{fu(i)} \le S_0 \right\}. \qquad (8.6)$$

Two techniques let us generate *D* while solving Equation 8.5 and 8.6 consecutively adding pairs to *D* and consecutively moving pairs out of *D*. The first technique solves Equation 8.5 and starts with set $D=D_O$. The second technique solves Equation 8.6, starts with set D_M, and never moves orthogonal pairs D_O. (Because of the concurrency of orthogonal statements, the pairs of D_O do not require additional execution time and resources. Hence we can always include D_O in *D*.)

Both techniques select a pair for including or removing by analyzing the maximum-weight cliques of sets $U_{\bar{D}}$ and V_D; the techniques select pairs that decrease the execution time and not increase the cost. The addition of pairs to set *D* is complete when any pair together with *D* produces cost *S* greater than bounding cost S_O or produces a number of functional units greater than the bounding number. Removing pairs from set *D* is complete when each pair to be removed implies time *T* greater than bounding time T_O. Adding or removing pairs in different order yields different contents for *D*.

Fig. 8.6 shows the influence *D* has on the net schedule execution time *T* and cost *S*. When *D* is empty, the process yields the net schedule of maximum execution time T_{max} and minimum cost S_{min}. When *D* equals D_M, the net schedule uses minimum execution time T_{min} and maximum cost S_{max}. Including a pair in *D* can decrease the execution time, while removing a pair from *D* can decrease the cost.

Recalculating clique sets

Adding or removing a pair from D changes the clique set according to four rules. Two rules transform $U_{\overline{D}}$ into $U_{\overline{D''}}$ when we add pair $d = (i, j) \in D_M$ into set D creating new set $D''=D\cup\{d\}$. The first rule splits a clique containing statements i and j into two new cliques of less cardinality; the second rule allows the removal of cliques from the new set D'':

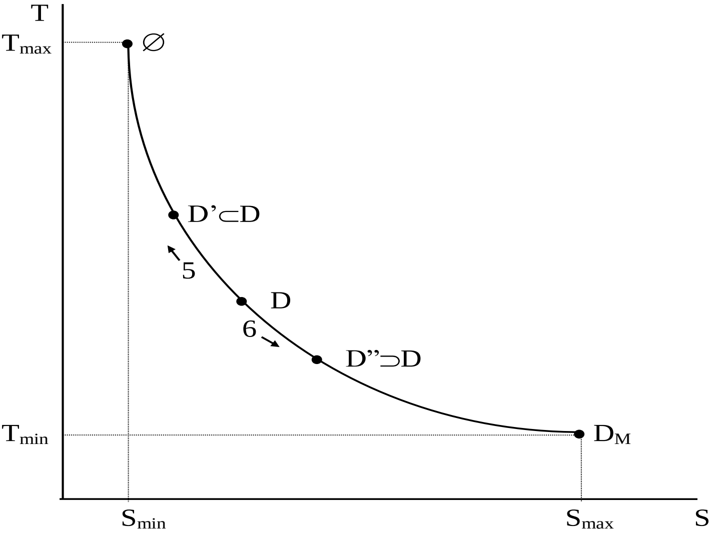


Figure 8.6. Set *D*'s effect on concurrency space

- Rule 1 (splitting) – If element $u \in U_{\overline{D}}$ satisfies the condition that $\{i, j\} \subseteq u$, then the elements $u \setminus \{i\}$ and $u \setminus \{j\}$ are added to set $U_{\overline{D''}}$; otherwise element u is;
- Rule 2 (absorbing) – If in set $U_{\overline{D''}}$ two elements u' and u'' exist for which $u' \supseteq u''$, then element u'' is removed from the set.

Two additional rules recalculate set V_D as new set $V_{D''}$. The third rule combines two cliques containing both statements *i* and *j* into a new clique that is included into set $V_{D''}$. The fourth rule removes the absorbed cliques from the set:

- Rule 3 (merging) – If $v' \cup v'' \supseteq \{i, j\}$ is true for $v', v'' \in V_D$ then element $v = (v' \cap v'') \cup \{i, j\}$ is added to $V_{D''}$. All elements of V_D are also included in $V_{D''}$.
- Rule 4 (absorbing) – If in set $V_{D''}$ two elements *v*' and *v*'' exist for which $v' \supseteq v''$ then element *v*'' is removed from the set.

If we remove pair *d* from set *D* and $D' = D \setminus \{d\}$ is the new set, then rules 1 and 2 transform set V_D into the set $V_{D''}$ and rules 3 and 4 transform set $U_{\overline{D}}$ into the set $U_{\overline{D''}}$.

Solving Equation 8.5 to minimize the execution time for *DiffEq* with one multiplier and two ALUs (t_{MUL} = 100 ns, t_{ALU} = 40 ns, s_{MUL} = 5 and s_{ALU} = 1) produces set , which contains 31 pairs, as described by the zero elements of the top right part of matrix Q_D^x (Fig. 8.7). The markings along the column heads indicate the exit statement, *e*, and VHDL operators <, +, =, and –.

No pair can be added to *D* without increasing the number of functional units and exceeding the constraints. For each clique of set $U_{\overline{D}}$, the execution time is the sum of the clique statements' execution time. Overall execution time is 340 ns, and the cost is 7 (Fig. 8.8).

If we add pair (*i*, *j*) to set *D* statements *i* and *j* are concurrent; if (*i*, *j*) is not included in set D_M, statement *i* precedes statement *j*. For pair (*i*, *j*) of set D_M not included in set *D*, we know that statements *i* and *j* are not concurrent, but do not know whether *i* should precede *j* or *j* should precede *i*.

Introducing Boolean variable x_{ij} into matrix Q_D^x for pair (*i*, *j*) and its negotiation $\overline{x}_{ij}$ for pair (*j*, *i*) solve this problem. If x_{ij} equals 1, statement *i* precedes statement *j*. If the value equals 0, *j* precedes *i*. Thus, while many net schedules possible for a given *D*, for some sets *D* no net schedule exists.

$$Q_D^x = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & x & 0 & x & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & x \\ 0 & 0 & 0 & 0 & 1 & 1 & x & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & x & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & x \\ 0 & 0 & 0 & \bar{x} & \bar{x} & 0 & 0 & 1 & 1 & 0 & 1 \\ \bar{x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{x} & 0 & 0 & 0 & 0 & 0 & 0 & \bar{x} & 0 & 0 & x \\ 0 & 0 & \bar{x} & 0 & 0 & \bar{x} & 0 & 0 & 0 & \bar{x} & 0 \end{bmatrix}$$

Figure 8.7. Matrix of set *D* for *DiffEq* with one multiplier and two *ALUs*

$U_{\bar{D}}$ =

	<	e	+	*	*	-	*	+	-	+	+	Time
1	1	0	0	0	0	0	0	1	0	1	0	120
2	0	0	1	0	0	1	0	0	0	0	1	120
3	1	0	0	0	0	0	0	0	0	1	1	120
4	0	0	0	0	0	0	1	1	1	0	0	180
5	0	0	0	1	1	1	0	0	1	0	0	280
6	0	0	0	1	1	0	1	0	1	0	0	340
7	0	0	1	0	1	1	0	0	1	0	0	220
8	1	1	0	0	0	0	0	0	0	0	1	280
9	0	0	0	0	0	0	0	0	0	0	1	140
												T=340

V_D=

	<	e	+	*	*	-	*	+	-	+	+	ALU	Mul	Exit	Cost
1	0	1	0	0	0	0	0	0	1	1	0	2	0	1	6
2	0	1	0	0	0	1	0	1	0	0	0	2	0	1	6
3	1	0	0	0	0	0	0	0	1	0	0	2	0	0	6
4	0	1	1	0	0	0	1	0	0	1	0	2	1	1	7
5	0	1	1	1	0	0	0	0	0	1	0	2	1	1	7
6	0	1	1	1	0	0	0	1	0	0	0	2	1	1	7
7	1	0	0	0	0	1	1	0	0	0	0	2	1	0	7
8	0	0	0	0	0	0	0	0	1	0	1	2	0	0	2
9	0	1	0	0	0	1	1	0	0	1	0	2	1	1	7
10	1	0	1	1	0	0	0	0	0	0	0	2	1	0	7
11	1	0	1	0	0	0	1	0	0	0	0	2	1	0	7
12	0	0	0	1	0	0	0	1	0	0	1	2	1	0	7
13	0	0	0	0	1	0	0	1	0	0	1	2	1	0	7
14	0	1	0	0	1	0	0	0	0	1	0	1	1	1	6
15	0	1	0	0	1	0	0	1	0	0	0	1	1	1	6
16	1	0	0	0	1	0	0	0	0	0	0	1	1	0	6
												2	1	1	S=7

Figure 8.8. Clique sets for matrix $Q^x{}_D$

8.2.3 Tackling the existence problem

Formulation of existence problem

Given set *D*, that net schedule G_H exists in which pairs of concurrent statements constitute *D*. Clique set $U_{\overline{D}}$ must be the maximum-paths set of graph G_H, and clique set V_D must be the maximum-sections set of the graph; otherwise this proposition is not true.

For set *D* and the given values of variables x_{ij}, a net schedule exists if the matrix derived from the matrix Q_D^x by substituting the variable values describes a transitive relation. This transitivity condition expresses the requirement that the net schedule must have exactly that level of concurrency defined by set *D*.

The relation is transitive if the following combined logical equation has at least one solution for x_{ij}.

$$L_1 = \bigvee_{\substack{i,j,k \subseteq N \\ (i,j)\in D \\ (i,k)\notin D \\ (k,j)\notin D}} \left\lfloor (z_{ik} \wedge z_{kj}) \vee (z_{jk} \wedge z_{ki}) \right\rfloor = 0\,,$$

$$L_2 = \bigvee_{\substack{i,j,k \subseteq N \\ (i,j)\notin D \\ (i,k)\notin D \\ (k,j)\notin D}} \left\lfloor (z_{ij} \wedge z_{ik} \wedge z_{kj}) \vee (z_{ji} \wedge z_{jk} \wedge z_{ki}) \right\rfloor = 0\,, \tag{8.7}$$

$$z_{ij} = \begin{cases} 1, & \text{\textit{if}}\ (i,\, j) \notin D_M \ \textit{and}\ i < j, \\ 0, & \text{\textit{if}}\ (i,\, j) \notin D_M \ \textit{and}\ \ j < i, \\ x_{ij}\,, & \text{\textit{if}}\ (i,\, j) \in D_M \setminus D, \textit{and}\ i < j, \\ \overline{x}_{ij}\,, & \text{\textit{if}}\ (j, i) \in D_M \setminus D, \textit{and}\ \ j < i. \end{cases}$$

In combined Equation (8.7) variables z_{ij} are intermediate. Equation L1 describes the transitivity conditions for the elements of set *D* and equation L2 describes the transitivity conditions for the elements not belonging to this set.

One algorithm effectively solves the combined Equation (8.7) by constructing graph G_D^x and searching for its non-conflicting labeling (Fig. 8.9). The graph nodes are variables x_{ij} that correspond to non-concurrent statements pairs. The algorithm introduces edge (x_{ij}, x_{ik}) if statements *j* and *k* are concurrent and pair (*j*, *k*) belongs to set *D*. It labels the graph nodes 0 and 1. The initial label 1 is assigned to the nodes belonging to set $\{x_{\mathrm{ij}} \mid (i, j) \notin \mathrm{D_M}, i, j = 1, ..., n, i < j\}$ of the Boolean variables that correspond to the nonconcurrent statements pairs not introduced into set D_{M}. If an edge connects two variables x_{ij}, x_{jk} that satisfy constraint $i<j<k$, it is labeled +, otherwise it's labeled –.

Labeling conflicts

Labeling two variables and the edge connecting them creates one type of conflict if the variable labels are the same and the edge label is +, or the variable labels are different and the edge label is –. If the graph has at least one of this first type of conflict, the equation for L_1 has no solution.

For variables x_{ij}, x_{ik}, and x_{kj} where $i<k<j$, there is a second type conflict if variable x_{ij}'s value equals 0 (1) and the values of x_{ik} and x_{kj} equal 1 (0). If the graph has at least one of the second type of conflict the equation for L_2 has no solution. To generate a net schedule, the algorithm must label the nodes in such a way as to avoid the conflicts of both types.

Fig. 8.9 shows graph G_D^x for the matrix in Fig. 8.7. Node $x_{5,6}$ has the label 1 and connects with node $x_{6,11}$ via edge ($x_{5,6}$, $x_{6,11}$), labeled +; hence, node $x_{6,11}$ must be labeled 0. Nodes $x_{6,11}$ and $x_{1,11}$ have different labels and are connect via the edge labeled –; this is a conflict. Equation (8.7) has no solution. Therefore, the number of net schedules of the different concurrency levels is less than the number of subsets of set D_M.

Solving conflicts

If L_1 and/or L_2 have no solution, the algorithm searches for subset *D'* of set *D* to solve Equation (8.5) and set *D''* that includes set *D* to solve Equation (8.6) (see Fig. 8.6). Sets *D*' and *D*″ must satisfy Equation (8.7). Set *D*' gives a less expensive net schedule, while set *D*″ gives a faster one.

How do we find appropriate sets *D*′ and *D*"? A program like Ahiles could use various procedures to solve this problem, but the main idea is to reduce or extend set *D* to avoid the conflicts. It is better to minimize the number of the concurrent statements pairs removed from or added to set *D*, but the algorithm should examine the influence of each pair on the execution time and cost as well.

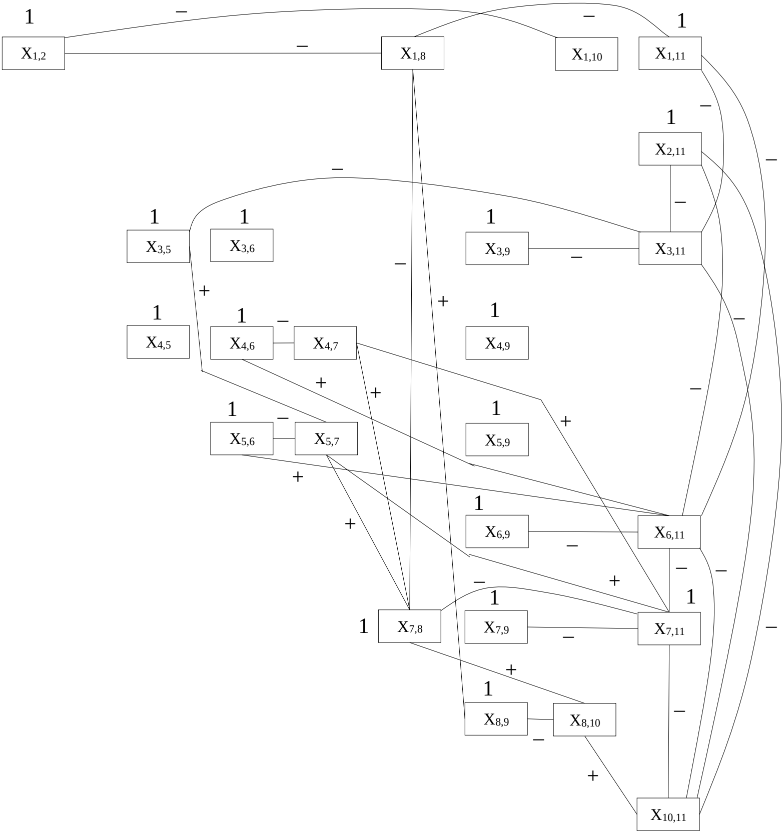


Figure 8.9. Graph G_D^x of logical equation L1, showing a labeling conflict of the first type

The gradient method is the basis for one possible procedure. The procedure firstly uses a branch and bound technique to find labeling that avoids conflicts of the second type and minimizes the number of the conflicts of the first type. Then for each pair that we can remove from or add to *D*, it estimates the number of conflicts remaining and selects the pair producing the minimum number of remaining conflicts. The procedure repeats until no labeling conflicts remain. This technique's efficiency depends on the estimating method.

Now consider Equation (8.5). There are 10 conflicts for set *D* described by the matrix in Fig. 8.7. Matrix *Cnf* in Fig. 8.10 describes for each pair the number of conflicts that will remain if we remove the pair.

$$
\begin{array}{c|ccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
1 & - & - & 11 & 12 & 10 & 9 & 11 & - & 9 & - & - \\
2 & - & - & 10 & 14 & 12 & 10 & 14 & 12 & 10 & 10 & - \\
3 & - & - & - & 11 & - & - & 10 & 11 & - & 10 & - \\
4 & - & - & - & - & - & - & - & 10 & - & 13 & 9 \\
5 & - & - & - & - & - & - & - & 12 & - & 15 & 13 \\
6 & - & - & - & - & - & - & - & 10 & - & 15 & - \\
7 & - & - & - & - & - & - & - & - & - & 10 & - \\
8 & - & - & - & - & - & - & - & - & - & - & 10 \\
9 & - & - & - & - & - & - & - & - & - & 8 & 11 \\
10 & - & - & - & - & - & - & - & - & - & - & - \\
11 & - & - & - & - & - & - & - & - & - & - & -
\end{array}
$$

Figure 8.10. Matrix *Cnf* records the number of conflict remaining for each pair if we remove it from *D*

As the matrix shows, removing a pair can sometimes create more conflicts; for example, removing pair (5, 10) would result in 15 conflicts, five more than we started with. Pair (6, 7) has the minimum number-seven-of remaining conflicts, and the procedure selects it for removal. Several steps of removing pairs and replacing 0 elements with *x* elements transforms the matrix in Fig. 8.7 into matrix Q_D^x (Fig. 8.11) describing set *D* of cardinality 22 that satisfies Equation (8.7).

This procedure also transforms the clique sets (see Fig. 8.12). While the execution time increases from 340 ns to 380 ns because nine pairs of concurrent statements became non-concurrent, the cost remains unchanged.

$$Q_D^x = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & x & 0 & x & x & x & 1 \\ 0 & 0 & x & 0 & 0 & x & 0 & 0 & x & 0 & 1 \\ 0 & \bar{x} & 0 & 0 & 1 & 1 & x & 0 & 1 & 0 & x \\ 0 & 0 & 0 & 0 & 1 & 1 & x & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & x & 0 & 1 & 0 & 0 \\ \bar{x} & \bar{x} & 0 & 0 & 0 & 0 & x & 0 & 1 & x & x \\ 0 & 0 & 0 & \bar{x} & \bar{x} & \bar{x} & 0 & 1 & 1 & 0 & 1 \\ \bar{x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x & 0 \\ \bar{x} & \bar{x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ \bar{x} & 0 & 0 & 0 & 0 & \bar{x} & 0 & \bar{x} & \bar{x} & 0 & x \\ 0 & 0 & \bar{x} & 0 & 0 & \bar{x} & 0 & 0 & \bar{x} & \bar{x} & 0 \end{bmatrix}$$

Figure 8.11. Matrix Q_D^x from Fig. 8.7, with the labeling conflicts resolved

$U_{\bar{D}}$ =

	<	e	+	*	*	-	*	+	-	+	+	Time
1	0	0	0	0	0	1	1	0	1	0	1	220
2	1	0	0	0	0	1	0	0	1	1	1	200
3	1	1	0	0	0	1	0	0	1	0	1	160
4	0	1	1	0	0	1	0	0	1	0	1	160
5	1	0	0	0	0	0	0	1	1	1	0	160
6	0	0	0	1	1	1	1	0	1	0	0	380
7	0	0	0	0	0	0	1	1	1	0	0	180
8	0	0	1	0	1	1	0	0	1	0	0	220
												T=380

V_D=

	<	e	+	*	*	-	*	+	-	+	+	ALU	Mul	Exit	Cost
1	0	1	0	0	0	0	0	0	1	1	0	1	1	1	6
2	0	1	0	0	0	1	0	1	0	0	0	1	1	1	6
3	1	0	0	0	0	0	0	0	1	0	0	1	1	1	6
4	0	1	1	0	0	0	1	0	0	1	0	2	0	0	2
5	0	1	1	1	0	0	0	0	0	1	0	2	1	0	7
6	0	1	1	1	0	0	0	1	0	0	0	2	1	0	7
7	1	0	0	0	0	1	1	0	0	0	0	2	1	0	7
8	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1
9	0	1	0	0	0	1	1	0	0	1	0	2	1	0	7
10	1	0	1	1	0	0	0	0	0	0	0	2	1	0	7
11	1	0	1	0	0	0	1	0	0	0	0	2	1	0	7
12	0	0	0	1	0	0	0	1	0	0	1	2	1	0	7
13	0	0	0	0	1	0	0	1	0	0	1	1	1	1	6
14	0	1	0	0	1	0	0	0	0	1	0	1	1	1	6
15	0	1	0	0	1	0	0	1	0	0	0	1	1	0	6
												2	1	1	S=7

Figure 8.12. New clique sets for matrix $Q^x{}_D$, with the labeling conflicts resolved

8.3 Generating a net schedule

Now graph G^x_D has no labeling conflicts. Ahiles can generate an appropriate net schedule after considering the labels of the graph nodes as Boolean variable values and substituting these values into the matrix Q^x_D,

Ahiles generates matrix Q_D. (Fig. 8.13). Since this matrix defines a transitive relation and no row i and column j exists for which $q_{ij}=q_{ji}=1$, Ahiles can reorder the matrix rows and columns to obtain a matrix with zeros below the principal diagonal (Fig. 8.14). This reordered matrix defines the statements' precedence relation in a net schedule. To determine the net schedule Ahiles uses the following procedure to calculate matrix H_D that defines the statements' direct precedence:

```
$H := Q_D$;
**for** $i \in \{0,...,N\text{-}1\}$ **do**
**for** $j \in \{0,...,i\}$ **do**
**if** $(h_{j+1,N-i+j} = 1) \wedge \left( \bigvee_{k=1}^{N} (h_{j+1,k} \wedge h_{k,N-i+j} = 1) \right)$ **then**
$h_{j+1,N-i+j}$ := 0;
**end** j;
**end** i;
```

Starting from the right top corner of H_D, the procedure replaces 1 values with 0 values for elements h_{ij} for which the Boolean multiplication of the row i and column j gives the value 1. For matrix Q_D, the procedure gives the matrix H_D shown in Fig. 8.15 and the net schedule shown in Fig. 8.16.

Table 8.1 gives some experimental results for net schedules that Ahiles synthesized for DiffEq. I measured the time spent to generate the net schedules on a 486, 50-MHz PC. Theory predicts a probabilistic growth of the number of the graph cliques depending on the size of the graph. However, sets $U_{\overline{D}}$ and V_D include few cliques for the net schedules, thus avoiding labeling conflicts, the most complex problem in net schedule synthesis.

$$\begin{array}{r|ccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
4 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{array}$$

Figure 8.13. Matrix Q_{D} before reordering

$$\begin{array}{r|ccccccccccc|}
 & 1 & 3 & 2 & 7 & 4 & 5 & 10 & 8 & 11 & 6 & 9 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
3 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
7 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 8.14. Matrix Q_{D} after reordering to obtain a zero bottom part

$$\begin{array}{r|ccccccccccc|}
 & 1 & 3 & 2 & 7 & 4 & 5 & 10 & 8 & 11 & 6 & 9 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 8.15. Matrix H_{D} calculated form matrix Q_{D}

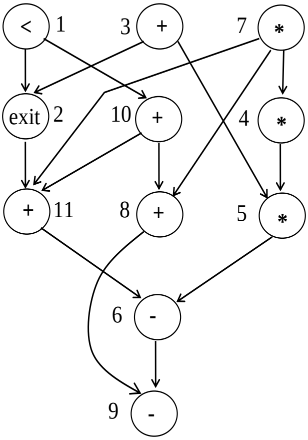


Figure 8.16. Net schedule for H_D, which uses two *ALUs* and one multiplier

Table 8.1

Ahiles net schedule synthesis results for DiffEq

N	Functional units		Cardinality		No of conflicts	Removed pairs	No of cliques		Time (sec)
	No of ALUs	No of Multipliers	D_M	Initial D			U_D	V_D	
1	2	1	39	31	6	5	6	22	0.11
2	1	1	39	21	7	4	5	14	0.11
3	2	1p	61	51	15	12	11	26	0.44

8.4 Transition from net schedule to sequential schedule

Ordinary, multicycling and chaining algorithms produce sequential schedules from net schedules. These algorithms do not care about computational resources as the resource constraints have been already taken into account while generating the net schedule.

The well-known ASAP and ALAP scheduling algorithms assume the statements' execution time is less or equal to the clock cycle period. To generate schedule these algorithms use the net schedule graph for G_H as input data. In this case the number of statements in the longest path of the net schedule graph determines the minimal number of control steps in the sequential ordinary schedule. In the case, multicycling is applied to long-delay operators and statements, ASAP and ALAP account for the execution time of each statement that can be larger than the clock cycle period. One statement can be scheduled to several consecutive control steps and several corresponding clock cycles.

In case, chaining is applied to short-delay operators and statements, ASAP and ALAP assign chains of operators and statements to one control step and one corresponding clock cycle.

Given constraints on resources, the list scheduling technique can generate a sequential schedule from the net schedule while minimizing the number of control steps. In this case, the list scheduling technique is capable of optimizing ordinary, multicycling and chaining sequential schedules, which follows the synthesis and optimization of a resource-constrained net schedule.

8.5 Graph language and tool for creation and simulation of sequential and net algorithms

This section presents a graph language and a tool for visual interactive development and simulation of net algorithms. The language is developed on the basis of C language, but the principles laid down in it are applicable to other languages as well. The graph language is based on the following construction principles:

1. The graph vertices are associated with primitives of the C language: data types, variables, constants, logical and arithmetic operators, statements, control structures, etc.;

2. The graph arcs connect the vertices and describe the control flow, which can be either sequential or parallel;

3. Various labels are assigned to vertices and arcs, the interpretation of which establishes the semantics of the language;

4 The. graph execution is dynamically visualized;

5. Source and termination vertices are fixed in the graph; only one active vertex is executed in the graph that describes a sequential algorithm, and several vertices are active and execute in the graph that describes a concurrent algorithm;

6. The control flow of is described by tokens that mark the arcs and move when the algorithm execute;

7. If tokens appear at all input arcs of a certain vertex, the vertex fires, and the corresponding statement or variable assignment is performed followed by moving the tokens from the input to the output arcs of the vertex;

8. The token at exactly one output arc characterizes the sequential execution of the graph; tokens at several output arcs characterizes its concurrent execution;

9. Firing of a vertex can be unconditional or conditional; in the second case a test Boolean variable is associated with the vertex; if the variable value is true when the vertex fires, the corresponding statement executes; otherwise it does not execute; in any case, the tokens move from the input to the output arcs;

10. The graph interacts in the process of operation with a storage of variables and with a storage of statements;

11. In the process of interpreting and executing the visualized graph, variable values are updated in the storage.

Fig. 8.17 shows the environment of visual simulation of a graph that describes sequential behavior. The top part of the interface includes a menu bar and a toolbar. The window of visualizing the graph includes images of vertices and arcs.

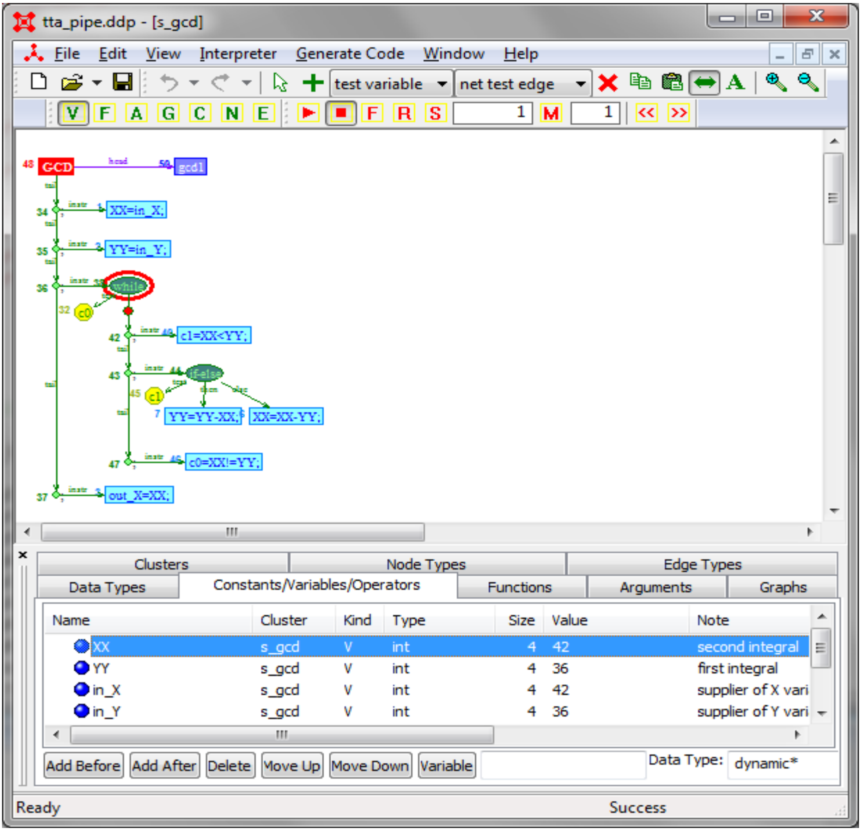
Two upper rectangular vertices describe the cluster name and the graph header. Seven lower rectangular vertices represent assignment statements, three of which contain one operator in the right part.

The upper oval vertex describes the *while* loop, and the lower oval vertex describes the *if-then-else* statement. Small round vertices with identifiers c0 and c1 inside describe test variables. Even smaller round vertices describe the operation of sequential execution.

All arcs are labeled. The oval vertex *while* has a stroke indicating that the vertex is firing. The token moves along an arc directed into the loop body. The lower part of the interface describes the storage of variables. For each variable, the cluster to which it belongs, the kind, type, size in bytes, current value and comment are indicated.

Fig. 8.18 illustrates a graph and its execution, which model and simulate a concurrent algorithm. The graph representation is a result of transforming a branching and looping behavioral description that is performed in a high level language into the basic single-block flow model, which allows for a maximum of asynchronous parallelization and a minimum of the critical path in the graph. Such a graph can represent a parallel asynchronous behavior of an embedded system.

The graph vertices are assignment statements and variables. The assignment statement depicted by blue rectangles reads the values of the input variables from the storage (bottom in this figure), perform logical, arithmetic, or other operations and write the values of the output variables into the storage.

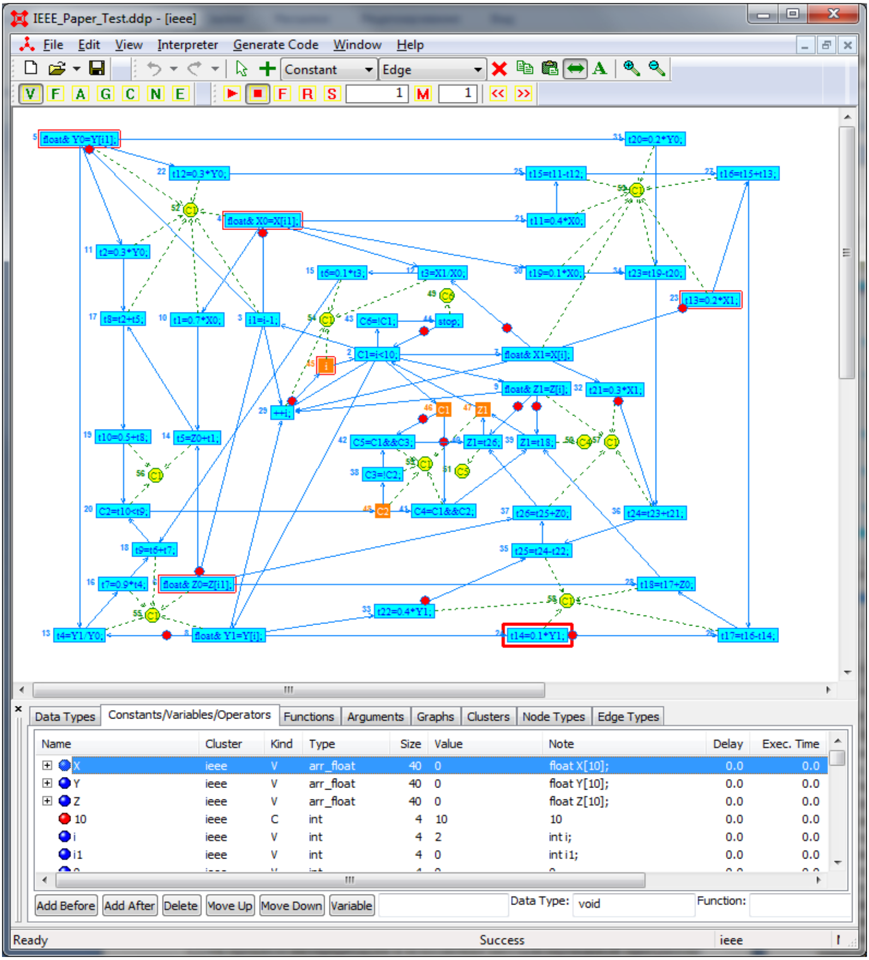


Name	Cluster	Kind	Type	Size	Value	Note
● XX	s_gcd	V	int	4	42	second integral
● YY	s_gcd	V	int	4	36	first integral
● in_X	s_gcd	V	int	4	42	supplier of X vari
● in_Y	s_gcd	V	int	4	36	supplier of Y vari

Data Type: dynamic*

Ready Success

Figure 8.17. Interface of graph-based environment for execution of sequential algorithm



Name	Cluster	Kind	Type	Size	Value	Note	Delay	Exec. Time
X	ieee	V	arr_float	40	0	float X[10];	0.0	0.0
Y	ieee	V	arr_float	40	0	float Y[10];	0.0	0.0
Z	ieee	V	arr_float	40	0	float Z[10];	0.0	0.0
10	ieee	C	int	4	10	10	0.0	0.0
i	ieee	V	int	4	2	int i;	0.0	0.0
i1	ieee	V	int	4	0	int i1;	0.0	0.0

Figure 8.18. Illustration of executing a concurrent algorithm

Vertices-variables are divided into two classes: conditional (control) variables (depicted in yellow circles) and value-assigned variables (depicted in orange rectangles). The value of a conditional variables affects the execution of statements (value *true* enables execution, and value *false* disables execution).

The vertices of the graph are connected with arcs. The arcs that are represented with solid lines connect vertices-statements to vertices-variables, and vice versa. The arcs represented by dashed lines connect the control variables with the statements, the performance of which they influence. Solid arcs can be marked with markers that indicate the flow of control or data in the basic single-block model. When tokens appear at all input arcs of a certain vertex, this vertex fires, and the tokens move to the output arcs. The statement that is associated with the vertex executes if the control variable connected to the vertex by the dashed arc takes value *true*.

The firing vertex is indicated by a bypass line. Several vertices of the graph can simultaneously fire, and the more such vertices, the higher the level of concurrency of the behavior. The correct transformation of a behavior into the basic single-block model ensures that the tokens do not crawl on each other and ensures that two or more tokens on the same arc cannot appear at the same time. The dynamics of the graph can be controlled through a number of tools provided by our simulation system.

The proposed transformational techniques and tools support the creation of net algorithms that are parallel in space and process one data set. They also support the creation of asynchronous pipelines that are parallel in time and process a flow of data sets [76]. The method provides for the creation of both regular asynchronous pipelines as well as irregular asynchronous pipelines, whose stages differ from each other. When designing a pipeline, the original control flow is eliminated, the pure data flow is extracted, and the network is divided into pipeline stages.

Each pipeline stage is represented with a subnet that implements the required functionality, performs certain operations and interact with neighbor stages-subnets by data exchange and handshaking mechanism. Synchronization of the subnets is localized and performed by means of a request / acknowledgment mechanism, which is implemented by moving the tokens along arcs, which connect the subnets of neighbor pipeline stages. The synthesis of the asynchronous pipeline is performed in a regular way by the method, which explores the basic single-block model and is described in Chapter 5 of this book.

Fig. 8.19 shows a net modular algorithm graph that asynchronously implements the TTA true audio codec [71, 78]. The blue rectangular vertices of the graph represent whole modules (functions) instead of simple statements. The input of this algorithm is a data stream of audio frames. Our method transforms this net algorithm into a two-stage asynchronous pipeline that is shown in Fig. 8.20.

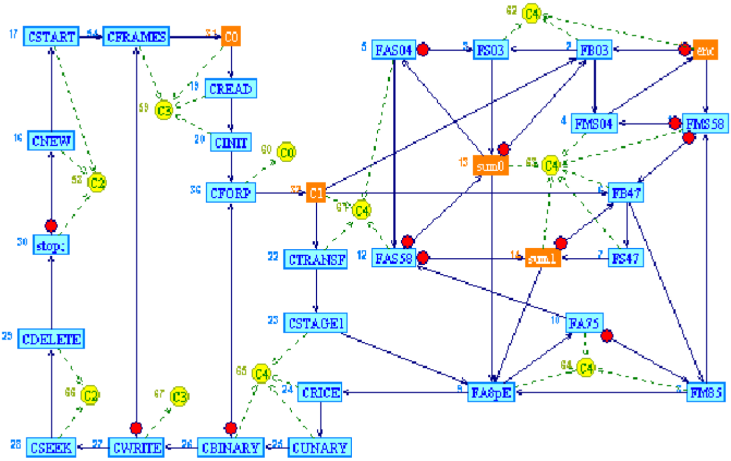


Figure 8.19. Graph of asynchronous net algorithm of audio coder TTA

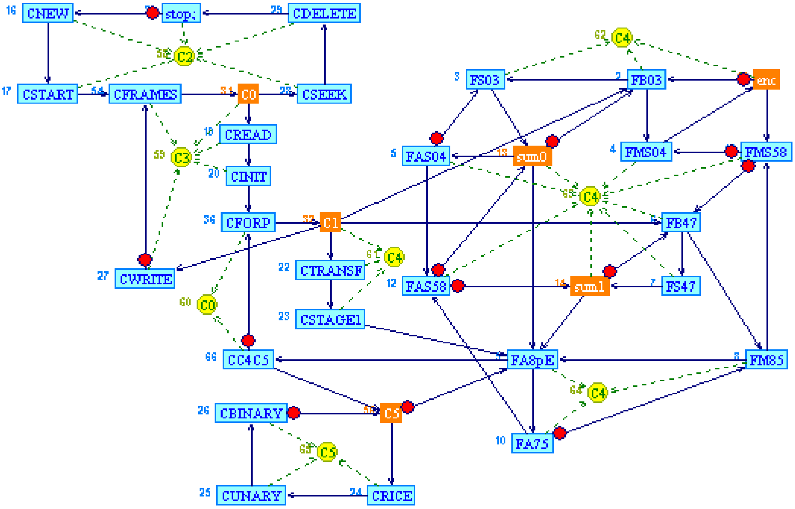


Figure 8.20. Graph of asynchronous 2-stage pipeline for audio coder TTA

The first stage of the pipeline runs in parallel with the second stage that is a hybrid filter. Table 8.2 reports parameters of the pipelined asynchronous audio encoder TTA, which are measured when encoding an excerpt from a melody by Italian baroque composer Tomaso Giovanni Albinoni. The two-stage net pipeline yields the acceleration factor of 3.4 against a non-parallel implementation. It should be noted that non-pipelined net algorithm and graph shown in Fig. 8.19 yields a smaller acceleration factor of 2.2.

Table 8.2

Parameters of asynchronous pipeline implementing the TTA audio compressor

Parameter	Value
Number of variables of all types	40
Number of vertices in the graph of net algorithm	41
Number of edges in the graph of net algorithm	83
Total number firings of graph vertices	5053
Total number of tokens at the edges of graph	7806
Вычислительная сложность алгоритма	47615
Critical path on data flow graph of net algorithm	13829
Parallelization factor	3.4

8.6 Experimental results

The model, method and techniques described here are used to develop a VHDL-based, high-level synthesis system called Ahiles [61]. The system inputs are a behavioral VHDL description, transition probabilities for branch statements of the description, an optimization task, and functional unit descriptions. The outputs are a register transfer level (RTL) structure composed of the structure parameters and the data path and the finite state machine (FSM).

Firstly, AHILES compiles the design specification and transforms the behavioral description to a special behavioral model. The system uses an internal format to speed up the design process and to allow the generation of high-quality designs. Still in the internal form, the description is diagnosed and analyzed, then presented in the control and data flow graphs. Ahiles then solves the scheduling, allocation, and binding tasks and generates the data path and finite state machine in the internal form. Reverse translation maps the register transfer level structure into VHDL text.

We can link the VHDL design to several VHDL libraries and units. Ahiles was written mainly in C and runs on an IBM PC platform. I obtained the results described below on a PC 486/50.

VHDL compiler parameters appear in Table 8.3. For large designs, the compiler throughput is more than 200 lines per second. The average size of the design internal form is only 1.4 times greater than the size of the source VHDL text.

Table 8.4 shows DiffEq's parameters for the net-based and list scheduling. In each case, the net-based scheduling technique (in this case, Ahiles) introduced the smaller number of control steps.

Table 8.3

Parameters for the Ahiles VHDL compiler

Parameter	Benchmark				
	Bubble	Gcd	Gcdf	Kalman	Pid
VHDL text (lines)	119	50	60	220	724
VHDL text (bytes)	3009	2089	2844	7966	23138
Internal form (bytes)	5570	2573	2925	12393	30280
Compilation time (s)	0.71	0.49	0.77	1.45	2.53
Throughput (lines/s)	168	102	78	149	286
Throughput (lines/s)	4238	4263	3694	5382	9145

Table 8.4

Synthesis results for DiffEq using various techniques within Ahiles

Parameter	Technique			
	Ordinary	Chaining	Multicycling	Pipelining
Clock cycle (ns)	120	120	60	60
Number of ALUs	2	2	1	2
Number of multipliers	1	1	1	1(p)
Control steps (list)	5	5	9	6
Control steps (net based)	5	4	8	6
Number of registers	8	8	8	7
Number of multiplexors	10	9	8	11
Number of multiplexor inputs	25	23	23	27

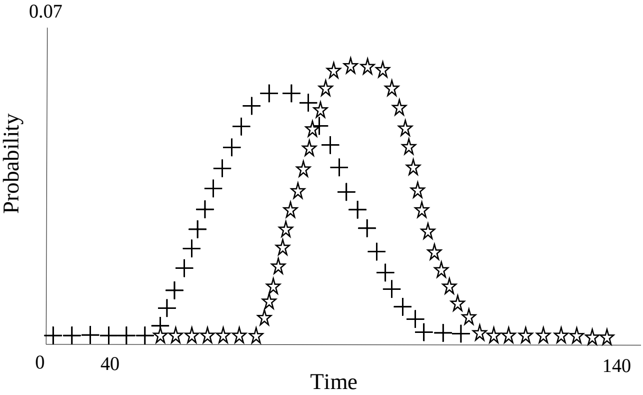


Figure 8.19. Frequency function for the net (+) and sequential (*) schedule execution time

Fig. 8.19 displays graphically the advantage net scheduling provides over sequential scheduling in the case of variable execution time of functional unites: Its frequency function peaks earlier and at a smaller probability level.

Table 8.5 reports the results that Ahiles obtained for five benchmarks. Bernard Courtois and Polen Kission of the Techniques of Informatics and Micro-electronics for Computer Architecture (TIMA) Laboratory provided the Bubble, Gcdf, and Pid benchmarks. I borrowed the Gcd and Kalman benchmarks from the works of Bergamaschi, and Morison and Newton. Due to equivalent transformation of the source behavioral descriptions and new scheduling techniques, Ahiles minimized the number of the finite state machine states. The system either maps behavioral description operators to the data path functional units or introduces them into the finite state machine.

Fig. 8.20 shows an example of a pipelined net schedule Ahiles has generated. Functional units are pipelined [53], therefore we split multiplication operators into parts, one for each stage of the pipeline. Each part precedes its successor. For DIFFEQ the net schedule of maximum concurrency with two-stage pipelined multiplication is presented in Fig. 8.20. A net schedule of less concurrency is synthesized by the net scheduling technique with constraints on the cost and number of the functional units as well as on the number of pipeline stages.

Table 8.5

Synthesis results for Ahiles

Parameter	Benchmark				
	Bubble	Gcd	Gcdf	Kalman	Pid
Number of states	20	2	5	16	23
Number of ALUs	0	1	1	1	1
Registers/bits	7/104	2/32	2/64	18/138	13/389
RAMs	1	0	0	3	0
ROMs	0	0	0	3	1
Multiplexor/input	4/13	4/8	4/8	14/36	8/33
Collectors	0	0	0	5	9
Structure (lines)	416	164	184	1000	723
Structure (bytes)	12383	4647	5241	31550	22938
Time (s)	6.03	4.79	4.84	12.04	7.15

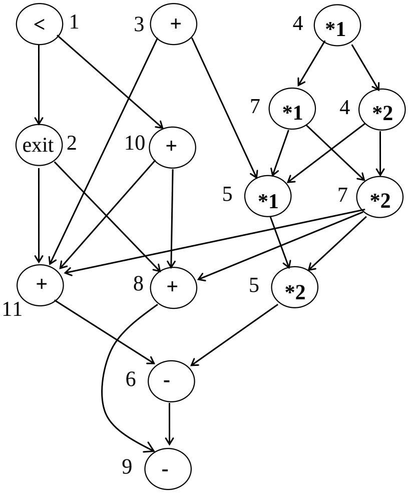


Figure 8.20. Net schedule Ahiles generated for a system with two ALUs and one pipelined multiplier. The designators *1 and *2 indicate the first and second stage of the pipelined multiplier

8.7 Conclusion

A new model of solving the scheduling and allocation tasks in high-level synthesis systems has been proposed. The net scheduling techniques that are developed in this chapter can synthesize the optimal net schedules on two criteria: the minimum execution time, and the minimum cost. The net schedule existence problem is formulated as a combined logical equation solving problem. An efficient technique for solving the logical equations of certain type has been proposed. The net schedule can be either directly used for digital system synthesis, or can be a source for generating sequential schedules with chaining, multicycling, and pipelining, which use pipelined and non-pipelined functional units.

As the obtained results show, net-based scheduling systems like Ahiles can produce synchronous designs more efficiently than the known sequential-scheduling techniques. At the same time, net-scheduling mainly targets the design and optimization of asynchronous systems, both hardware and software. The synchronization mechanisms may vary in a wide range. Net scheduling is extremely useful for modeling, synthesis and optimization of software for computer networks.

REFERENCES

1. AHG report on editorial convergence of MPEG-4 reference software, ISO/IEC JTC1/SC29/WG11 MPEG2003/9632, July 2003.
2. Aiken, A. Optimal loop parallelization / A.Aiken and A.Nicolau // in Proc. of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1988.
3. Bakshi, S. and Gajski, D. D. Component selection for high-performance Pipelines / S. Bakshi and D. D. Gajski // IEEE Trans. VLSI Syst., vol. 4, no. 2, pp. 181-194, 1996.
4. Banerjee, S. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems / S.Banerjee, T. Hamada, P. Chau and R. Fellman // IEEE Trans. Signal Processing, vol. 43, no. 6, pp. 1468-1484, June 1995.
5. Barford, P. Critical Path Analysis of TCP Transactions / P. Barford, M. Crovella // IEEE/ACM Transactions on Networking, vol. 9, 2001, No. 3, pp. 238-248.
6. Bezati, E. Synthesis and optimization of high-level stream programs / E. Bezati, S.Casale-Brunet, M. Mattavelli, and J. Janneck // in Proc. of the 2013 Electronic System Level Synthesis Conference, 2013, pp. 1–6.
7. Brunet, S. Profiling of dataflow programs using post mortem causation traces / S. Brunet, , M. Mattavelli and J. Janneck // in Signal Processing Systems (SiPS), 2012 IEEE Workshop on, Oct 2012, pp. 220–225.
8. Chang, P. A Decomposition Approach for Balancing Large-Scale Acyclic Data Flow Graphs / P. Chang, C.S. Lee // IEEE Transactions on Computers, vol. 39, No. 1, 1990, pp. 34-46.
9. Chao, L.-F.A. Rotation scheduling: a loop pipelining algorithm / L.-F.A. Chao, LaPaugh, and E.-M. Sha // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 16, no. 3, pp. 229–239, Mar 1997.

10. Cong, J. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation / J. Cong and Z. Zhang // *Design Automation Conference (DAC)*, Jul. 2006.

11. Corrado, B. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules / B. Corrado, G. Jacopini // Communications of the ACM. – 1966, **9** (5): 366–371.

12. Demicheli, G. Hardware synthesis from C/C++ models / G. Demicheli // in Design, Automation and Test in Europe, Conference and Exhibition 1999, pp. 382–383.

13. Dijkstra, E.W. Letters to the editor: go to statement considered harmful / E.W. Dijkstra // Communications of the ACM. – 1968, **11** (3): 147–148.

14. Drechsler, R. A genetic algorithm for variable ordering / R. Drechsler, B. Becker, N. Gockel // IEE Proceedings, 143(6), 1996, pp. 364-368.

15. Eichenberger, A.E. Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule / A.E. Eichenberger and E.S. Davidson // in Proc. 28th Int. Symp. on Microarchitecture, 1995, pp. 338-349.

16. Eker, J. CAL Language Report: Specification of the CAL Actor Language / J. Eker and J. Janneck, // University of California-Berkeley, December 2003.

17. Gao, L. A software pipelining algorithm in high-level synthesis for fpga architectures / L. Gao, D. Zaretsky, G. Mittal, D. Schonfeld, and P. Banerjee // in Proceedings of the 10th International Symposium on Quality Electronic Design, ISQED 2009, 2009, pp. 297–302.

18. Girczyc, E.M. Loop winding–a data flow approach to functional pipelining / E.M. Girczyc // in Proc. of the IEEE ISCAS, May 1987, pp. 382–385.

19. Goodman, J.R. and Hsu, W.C. Code Scheduling and Register Allocation in Large Basic Blocks / J.R. Goodman and W.C. Hsu // in Proc. Int. Conf. on Supercomputing, 1988, pp. 442-452.

20. Goossens, G. An efficient micro-code compiler for applications specific DSP processors / G. Goossens, J.Rabaey, J. Vandewalle and H.D. Man // IEEE Trans. Computer-Aided Design, vol. 9, pp. 925–937, June 1990.

21. Henkel, J. An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity that is Driven by High-Level Estimation Techniques / J. Henkel, R. Ernst // IEEE Transactions on VLSI Systems, vol. 9, No. 2, 2001, pp. 273-289.

22. Hollingsworth, J. Critical Path Profiling of Message Passing and Shared-Memory Programs / J. Hollingsworth // IEEE Transactions on Parallel and Distributed Systems, vol. 9, No. 10, 1998, pp. 1029-1040.

23. Hu, X. Minimizing the number of delay buffers in the synchronization of pipelined systems / X. Hu, R.G. Harber and S.C. Bass // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 13, no. 12, pp. 1441-1449, Dec1994.

24. Hwang, C.-T. Pls: A scheduler for pipeline synthesis / C.-T. Hwang, Y.-C. Hsu and Y.-L. Lin // Trans. Comp.-Aided Des. Integr. Cir. Sys., vol. 12, no. 9, pp. 1279–1286, September 1993.

25. Hwang, K.S. Scheduling and hardware sharing in pipelined data paths / K. S. Hwang, A. E. Casavant, C.-T. Chang, M. A. d’Abreu // Proc. ICCAD-89, November 1989, pp. 24–27.

26. Javaid, H. Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems / H. Javaid, A. Ignjatovic and S. Parameswaran // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 29, no. 11, pp. 1777–1789, Nov 2010.

27. Juarez, E. A System-on-a-chip for MPEG-4 Multimedia Stream Processing and Communication / E. Juarez, M. Mattavelli and D. Mlynek // IEEE International Symposium on Circuits and Systems, May 28-31 2000, Geneva, Switzerland.

28. Jun, H.-S. Design of a pipelined datapath synthesis system for digital signal processing / H.-S. Jun and S.-Y. Hwang // Trans. Comp.-

Aided Des. Integ. Cir. Sys., vol. 12, no. 3, pp. 292–303, September 1994.

29. Kahn, G. The semantics of a simple language for parallel programming / G. Kahn // Information Processing, pp. 471–475, 1974.

30. Kahn, D. B. M. G. Coroutines and networks of parallel processes / D. B. M. G. Kahn // Information Processing, 1977, pp. 993–998.

31. Karp, R.M. Turing Award Lecture: Combinatorics, complexity and randomness, Communications of the ACM, vol. 29, no. 2, pp. 98-109, Feb. 1986.

32. Knuth, D.E. The art of computer programming: Fundamental algorithms / D.E. Knuth // Addison Wesley, 1968, Vol. 1. – 735 p.

33. Knuth, D.E. The art of computer programming: Seminumerical algorithms / D.E. Knuth // Addison Wesley, 1969, Vol. 2. – 724 p.

34. Knuth, D.E. The art of computer programming: Sorting and searching / D.E. Knuth // Addison Wesley, 1973, Vol. 3. – 844 p.

35. Ko, D.-I. The pipeline decomposition tree: an analysis tool for multiprocessor implementation of image processing applications / D.-I. Ko and S.S. Bhattacharyya // in Proc. CODES+ISSS '06: 4th Int. Conf. on Hardware/software codesign and system synthesis, 2006, pp. 52-57.

36. Kobayashi, S. Task Scheduling Algorithm with Corrected Critical Path Length / S. Kobayashi and S. Sagi // ISS, vol. J81-D-I, No.2, pp. 187-194.

37. Kuhn, P. Instrumentation Tools and Methods for MPEG-4 VM: Review and a New Proposal / P. Kuhn // Tech. Rep. M0838, ISO/IEC, Mar. 1996.

38. Kwok, Y.-K. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors / Y.-K. Kwok, I. Ahmad // IEEE Transactions on Parallel and Distributed Systems, vol. 7, No. 5, 1996, pp. 506-521.

39. Lee, E.A. Synchronous data flow / E.A. Lee and D.G. Messerschmitt // Proceedings of the IEEE, vol. 75, pp. 1235–1245, 1987.

40. Leiserson, C.E. Optimizing synchronous systems / C.E. Leiserson and J.B. Saxe // Journal of VLSI Computer Systems, vol. 1, no. 1, pp. 41–67, 1983.

41. Liu, L. An Efficient Parallel Critical Path Algorithm / L. Liu, D. Du and H.-C. Chen // IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 13, No. 7, 1994, pp. 909-919.

42. Lucarz, C. Dataflow/actor-oriented language for the design of complex signal processing systems / C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. Janneck, I. Miller, D. Parlour // Proc. Conf. on Design and Architectures for Signal and Image processing, November 2008, pp.1-8.

43. Lucke, L. Data-Flow Transformations for Critical Path Time Reduction in High-Level DSP Synthesis / L. Lucke, K. Parhi // IEEE Transactions on Computer Aided Design, vol. 12, No. 7, 1993, pp. 1063-1068.

44. Malik, S. Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis / S. Malik, K.J. Singh, R.K. Brayton and A. Sangiovanni-Vincentelli // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 12, no. 5, pp. 568–578, May 1993.

45. Mermet, J.P. ed., Fundamentals and Standards in Hardware Description Languages. Boston: Kluwer Academic Publishers, 1993.

46. Mittal, S. A Survey of Value Prediction Techniques for Leveraging Value Locality / S. Mittal // Concurrency and Computation, 2017.

47. Mattavelli, M. Implementing Real-Time Video Decoding on Multimedia Processors by Complexity Prediction Techniques / M. Mattavelli and S. Brunetton // IEEE Transactions on Consumer Electronics, vol. 44, 1998, pp. 760-767.

48. Mattavelli, M. Methods to explore design space for MPEG RVC codec specifications / M. Mattavelli, S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier and J. Janneck // Signal processing Image Communication, Elsevier, 2013.

49. Mattavelli, M. The Reconfigurable Video Coding Standard, [Standards in a Nutshell] / M. Mattavelli, I. Amer, M. Raulet // Signal Processing Magazine, IEEE 27 (3) (2010) 159 –167.

50. Nurvitadhi, E. Automatic pipelining from transactional datapath specifications / E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 30, no. 3, pp. 441–454, March 2011.

51. Oh, S. Speculative loop pipelining in binary translation for hardware acceleration / S. Oh, T.G. Kim, J. Cho and E.Bozorgzadeh // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 27, no. 3, pp. 409–422, March 2008.

52. Parhi, K. VLSI Digital Signal Processing Systems: Design and Implementation / K. Parhi // Wiley Interscience, 1999.

53. Park, N. Sehwa: A software package for synthesis of pipelines from behavioral specifications / N. Park and A.C. Parker // IEEE Trans. Computer-Aided Design, vol. 7, pp. 358–370, March 1988.

54. Paulin, P.G. Force-directed scheduling for the behavioral synthesis of ASIC’s / P. G. Paulin, and J. P. Knight // IEEE Trans. Computer-Aided Design, vol. 8, pp. 661–679, June 1989.

55. Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving / J. Pearl // New York, Addison-Wesley, 1983. – 382 p.

56. Potasman, R., Lis, J., Aiken, A. and Nicolau, A. Percolation based synthesis / R. Potasman, J. Lis, A. Aiken and A.Nicolau // in Proc. 27th Design Automation Conf., 1990, pp. 444–449.

57. Prihozhy, A. Theory of equivalent transformation of algorithms in VLSI CAD / V. Mischenko, A. Prihozhy // Minsk: Navuka i technika, 1991. – 260 p.

58. Prihozhy A. Methods of equivalent transformation of logical algorithms in VLSI CAD / A. Prihozhy // Vesti Academy of Sciences of Belarus, ser. f.-m.s., 1992, №2, c.86-92.

59. Prihozhy, A.A. High-Level Synthesis Methodology / A. Prihozhy // Minsk, Inst. Eng. Cybernetics, National Academy of Sciences, Belarus, 1993. – 47 p.

60. Prihozhy, A.A. AHILES: Performance Driven High-Level Synthesis from VHDL Description / A.A. Prihozhy, A.N. Smolsky // Int. Conf. "Design Methodologies for Microelectronics", Slovakia, Bratislava, 1995, pp.45-52.

61. Prihozhy, A. Net scheduling in high-level synthesis / A. Prihozhy // IEEE Design & Test of Computers, 1996 spring, pp. 24-33.

62. Prihozhy, A.A. Use of VHDL-Based Design Methodology and the AHILES System for Education in Belarus / A. Prihozhy // Chapter in book "Microelectronics Education", World Scientific, 1996, pp.217-220.

63. Prihozhy, A.A. If-Diagrams: Theory and Application / A.A. Prihozhy // Proc. of the European Conference PATMOS'97. – UCL, Belgium, 1997. – P. 369 – 378.

64. Prihozhy, A.A. Parallel Computing with If-Decision-Diagrams / A.A. Prihozhy, P.U. Brancevich // Proc. of the Int. Conference PARELEC'98. – Poland, Technical University of Bialystok, 1998. – P. 179 – 184.

65. Prihozhy, A. Asynchronous Scheduling and Allocation, in Proceedings of the Conference DATE 98 / A.A. Prihozhy // IEEE Computer Society, Paris, France, 1998, pp.934-935.

66. Prihozhy A. Mathematical methods and software for high-level structural-parametric synthesis of digital systems / A. Prihozhy // Thesis Doctor Habilitation: 05.13.11. – Minsk, 1998. – 496 c.

67. Prihozhy, A.A. Digital System High-Level Synthesis Technology / A. Prihozhy, R.M. Merdjani, S.V. Zemlyanik // Proc. Int. Conf. "Information Technologies for Education, Science and Business", Minsk, Belarus,1999, pp.145-149.

68. Prihozhy, A.A. Automatic Parallelization of Net Algorithms / A.A. Prihozhy, R. Merdjani, F. Iskandar // Proc. Int. Conf. on Parallel

Computing in Electrical Engineering – PARELEC'2000. Canada, 2000, IEEE Computer Society Press, pp. 24-28.

69. Prihozhy, A. High-Level Synthesis through Transforming VHDL Models / A.A. Prihozhy // in Book "System-on-Chip Methodologies and Design Languages", Kluwer Academic Publishers, 2001, pp.135-146.

70. Prihozhy, A. High-level Synthesis through Transforming VHDL Models / A. Prihozhy // System-on-Chip Methodologies and Design Languages, Kluwer Academic Publishers, 2001, pp.135-146.

71. Prihozhy, A., Techniques for Optimization of Net Algorithms / A. Prihozhy, D. Mlynek, M. Solomennik and M. Mattavelli // PARELEC 2002 – Parallel Computing in Electrical Engineering, IEEE CS Press, 2002, pp. 211-216.

72. Prihozhy, A. Data Dependences Critical Path Evaluation at C/C++ System Level Description / Prihozhy, A., Mattavelli, M. and Mlynek, D. // Chapter in Book "Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation", LNCS 2799, Springer, 2003, pp.569-579.

73. Prihozhy, A.A. Evaluation of Parallelization Potential for Efficient Multimedia Implementations: Dynamic Evaluation of Algorithm Critical Path / A. Prihozhy, M. Mattavelli, D. Mlynek // IEEE Trans. on Circuits and Systems for Video Technology, Vol. 15, No. 5, May 2005, pp.593-608.

74. Prihozhy, A.A. If-Decision Diagram Based Modeling and Synthesis of Incompletely Specified Digital Systems / A.A. Prihozhy, B. Becker // Electronics and communications, Special Issue on Electronics Design. – Kyiv, 2005, pp. 103 – 108.

75. Prihozhy, A. Design of Parallel Implementations by Means of Abstract Dynamic Critical Path Based Profiling of Complex Sequential Algorithms / A. Prihozhy and D. Mlynek // in Book "Integrated Circuit and System Design", LNCS 4148, Springer, 2006, pp.1-11.

76. Prihozhy A. Methodology for transformation of program code to improve parameters of parallel implementations of audio codecs /

A. Prihozhy, M. Solomennik, D. Mlynek // – Electronics and Communication, Vol. 31, Kyiv, 2006.

77. Prihozhy, A. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL / A.-H. Ab Rahman, A. Prihozhy and M. Mattavelli, // EURASIP Journal on Image and Video Processing, vol. 2011:19, pp. 1–28, 2011. [Online]. Available: http://dx.doi.org/10.1186/16875281-2011-19.

78. Prihozhy A. Optimization Methodology for Complex FPGA-based Signal Processing Systems with CAL / A. Ab-Rahman, H. Amer, A. Prihozhy, C. Lucarz, M. Mattavelli // Int. Conf. on Design & Architectures for Signal and Image Processing - DASIP'2011, Tampere, Finland, Nov. 2-4, 2011. – France, ECSI, IEEE.

79. Prihozhy, A. Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs / A. Prihozhy, E. Bezati, H. Rahman, M. Mattavelli. // IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 34, No. 10, 2015, pp. 1613-1626.

80. Prihozhy A. Heuristic genetic algorithm for optimizing computational pipelines / A. Prihozhy, A. Zhdanouski, A. Karasik, M. Mattavelli // Doklady BGUIR, 2017, № 1, c. 34-41.

81. Prihozhy, A.A. Semantic model for high-level synthesis of dataflow pipelines / A.A. Prihozhy, O.N. Karasik, O.M. Frolov // Open Semantic Technologies for Intelligent Systems, Proceedings of International Conference, February 2017. – Minsk, BSUIR, 2017, pp. 415-418.

82. Prihozhy, A. Efficient Dynamic Optimization Heuristics for Dataflow Pipelines / A. Prihozhy, S. Casale-Brunet, E. Bezati, M. Mattavelli. // 2018 IEEE International Workshop on Signal Processing Systems (SiPS), 21-24 Oct. 2018, IEEE, pp. 337-342.

83. Ravasi, M. High-level Algorithmic Complexity Evaluation for System Design / M. Ravasi, M. Mattavelli // International Journal on System Architectures, vol. 48/13-15, Elsevier Publisher, 2003, pp. 403-427.

84. RSA Data Security, Inc. PKCS #1: RSA Encryption Standard. Version 1.4, June 1991.

85. Shenoy, N. Retiming: Theory and practice / N. Shenoy // VLSI Journal Integr., vol. 22, no. 1-2, pp. 1–21, 1997.

86. Sun, W. FPGA pipeline synthesis design exploration using module selection and resource sharing / W. Sun, M. Wirthlin, and S. Neuendorffer // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 26, no. 2, 2007, pp. 254–265.

87. True Audio Codec Software [Electronic resource]. - 2018. – Mode of access: http://www.true-audio.com. - Date of access: 11.11.2018.

88. Verhaegh, W.F.J. Improved force-directed scheduling in high-throughput digital signal processing / W.F.J. Verhaegh, P.E.R. Lippens, E. H.L. Aarts, J.H.M. Korst, J. Van Meerbergen and A. van der Werf // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 14, no. 8, pp. 945–960, Aug 1995.

89. Weinhardt, M. Pipeline vectorization / M. Weinhardt and W. Luk // Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 20, no. 2, pp. 234–248, Feb. 2001.

90. Wipliez, M. Software code generation for the RVC-CAL language / M. Wipliez, G. Roquier, and J. Nezan // Journal of Signal Processing Systems, pp. 1–11, 2009.

91. Wong, Y.-C. A Parallelism Analyzer for Conservative Parallel Simulation / Y.-C. Wong, S.-Y. Hwang and Y. Lin // IEEE Transactions on Parallel and Distributed Systems, vol. 6, No. 6, 1995, pp. 628-638.

Научное издание

ПРИХОЖИЙ Анатолий Алексеевич

АНАЛИЗ, ПРЕОБРАЗОВАНИЕ И ОПТИМИЗАЦИЯ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Текст на английском языке

Подписано в печать 19.03.2019. Формат 60×84 $^{1}/_{16}$. Бумага офсетная. Ризография.
Усл. печ. л. 13,37. Уч.-изд. л. 10,45. Тираж 100. Заказ 91.

Издатель и полиграфическое исполнение: Белорусский национальный технический университет. Свидетельство о государственной регистрации издателя, изготовителя, распространителя печатных изданий № 1/173 от 12.02.2014. Пр. Независимости, 65. 220013, г. Минск.