# Data Dependences Critical Path Evaluation at C/C++ System Level Description

Anatoly Prihozhy, Marco Mattavelli, and Daniel Mlynek

Signal Processing Laboratory 3, Signal Processing Institute,
Swiss Federal Institute of Technology, Lausanne
LTS3/ELB-Ecublens, Lausanne, CH-1015, Switzerland
prihozhy@yahoo.com, marco.mattavelli@epfl.ch, dmlynek@txc.com
http://lsiwww.epfl.ch/

**Abstract.** This paper presents a model metrics, techniques and methodology for evaluating the critical path on the Data Flow Execution Graph (DFEG) of a system on chip specified as a C program. The paper describes an efficient dynamic critical path evaluation technique generating no execution graph explicitly. The technique includes two key stages: (1) the instrumentation of the C code and the mapping into a C++ code version, (2) the execution of the C++ code and actual evaluation of the critical path. The model metrics and techniques aim at the estimation of the bounding speed and parallelization potential of complex designs specified at system level.

## 1 Introduction

Nowadays, processing and compression algorithms, communication protocols and multimedia systems have reached an extremely high level of sophistication. Architectural implementation choices become extremely difficult tasks, leading to the need of more and more intensive specification and validation by means of C/C++ software implementations. At the highest possible algorithmic level, the evaluation of complexity and parallelization potential of the algorithm have to be derived from such software descriptions under real input conditions [2, 4, 8] in order to be able to take meaningful and efficient partitioning decisions. Depending on the specific goal of the complexity analysis, different approaches and tools have been developed in [4, 6, 12, 14]: (1) profilers, modifying the program to make it produces run-time data, (2) compilers, applying result-equivalent code replacements, (3) static methods, getting information from the source code, (4) descriptions by means of hardware description languages, (5) hardware specific tools, providing computational information to some extent. Nowadays, such approaches and tools are unsuited for implementation exploration at the C/C++ system-level description because either they apply at a stage where architectural decision is already been taken, either they provide results that depend too much on the simulation platform used or do not take into account the complexity and parallelization potential features that depend only on the algorithm itself and on the input data. Conversely, the SIT Software Instrumentation Tool [14, 13] is an automatic general instrumentation environment able to extract and measure data flow and complexity of algorithms/systems specified in C/C++. SIT measures the processing complexity in terms of executed operators, data types and bandwidths of the data exchanges, while still taking into account real input data conditions.

The problem of identifying one of the longest paths in a circuit is called a critical path problem [7]. The length of the critical path plays a key role in setting the clock cycle time and improving the chip performance. In message passing and shared-memory parallel programs [1], 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 critical path profiling is a metric explicitly developed for parallel programs [1] and proved to be useful. 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 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. In [7] the parallel algorithms for critical path problem at the gate level are studied, while [1] describes a runtime non-trace-based algorithm to compute the critical path profile of the execution of message passing and shared-memory parallel programs. The idea described in [16] is to insert parallelism analysis code into the sequential simulation program. When the modified sequential program is executed, the time complexity of the parallel simulation based on the Chandy-Misra protocol is computed. The critical path analysis also gives an effective basis for the scheduling of computations. The work presented in [3] proposes a task scheduling algorithm that allocates tasks with corrected critical path length. The technique described in [5] schedules task graphs, analyzing dynamically the schedule critical paths. The techniques based on the minimization of critical path length estimated as the maximal clique weight of the sequential and parallel operator graphs [9, 10] constitute an efficient approach to the generation of concurrent schedules.

Summarizing the previous results, it can be concluded that the majority of already developed tools aim at the critical path profiling for tuning existing parallel programs (Fig. 1a). In this paper, the objective is to propose a critical path model metric in order to be able to find out in which degree a given algorithm described in C satisfies the parallel implementation conditions (Fig.1b). The paper is organized as follows. Section 2 defines the critical path model metrics on data dependences. Section 3 describes techniques for evaluating the critical path length. Section 4 presents the mapping of the C-code into a C++-code version. Transformations for reducing the critical path length are discussed in Section 5. Experimental results on parallelization potential on two real world analysis cases are presented in Section 6.

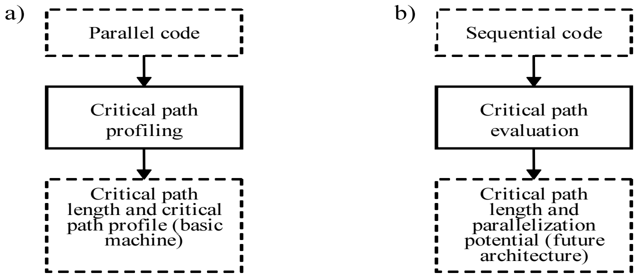
a)
Parallel code
Critical path profiling
Critical path length and critical path profile (basic machine)
b)
Sequential code
Critical path evaluation
Critical path length and parallelization potential (future architecture)

**Fig. 1.** Critical path profiling (a) of parallel code on event graphs versus critical path evaluation (b) of sequential code on data dependences graphs.

## 2 Definition of Critical Path on Data Dependences

The principles constituting the basis for the critical path model metric definition on a sequential C-code are the following:

- The critical path is defined on the C-code's execution data flow without taking into account the true control flow.
- The critical path length and the system parallelization potential are defined in terms of C language basic operations (including *read* and *write* operations) complexity. The parameters of the machine executing the instrumented C-code during evaluating the critical path are not taken into account.
- 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 Data Flow 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 ▨ 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 arcs: the data dependence arc denoted $\rightarrow$ and the conditional dependence arc denoted $\dashrightarrow$. 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 conditional value with an operator 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 then d*=2;* C-code is shown in Fig. 2a. It contains four value-nodes, one operator-node, three data and one conditional dependence arcs.

The DFEG is weighted with node complexities. All the complexities are accumulated at the operator-nodes and each C-operator is represented by a fragment in DFEG as shown in Fig. 2b. *Read* and *write* operators are associated with incoming and outgoing arcs of the operator-node respectively. 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 C-code complexity together with the critical path length describes its parallelization potential. The complexity of the fragment shown in Fig. 2b equals 4 basic operators. The internal critical path length on the graph fragment equals 3 because two *read* operations are executed in parallel.

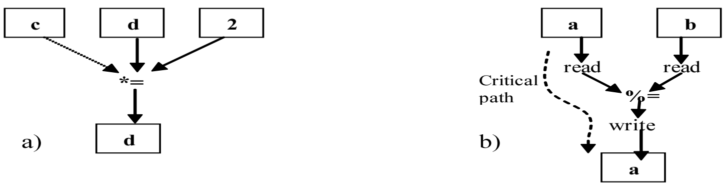

**Fig. 2.** (a) Example DFEG fragment for *if c then d*=2;* C-code. (b) Evaluation of the complexity and critical path of *a%=b;* C-code.

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

**Table 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 |
| Indexing | [] | 3 | 2 |
| Increment (decrement) | ++, -- | 3 | 3 |
| Unary minus and others | - | 2 | 2 |

```
void main() {
    unsigned long n=21414;
    int m[5], k=0;
    for(int i=0; i<5; i++) m[i]=0;
    while(n) {m[n%10]=1; n/=10;}
    for(int j=0; j<5; j++) if(m[j]) k++;
}
```

**Fig. 3.** C-code for counting 1, 2, 3, 4 and 5 digits in integer *n*.

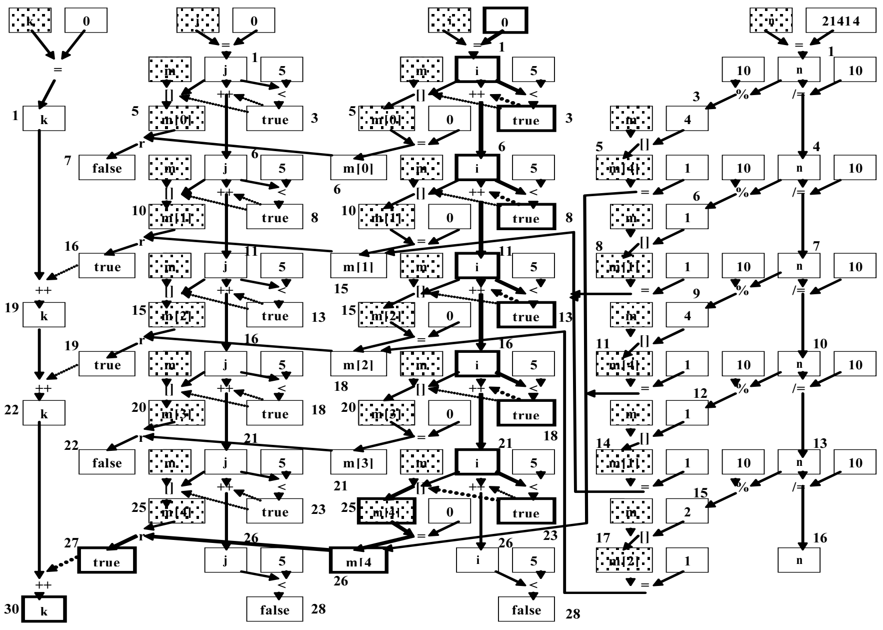

**Fig. 4.** The Data Flow Execution Graph (DFEG) for the C-code shown in Fig. 3.

**Table 2.** Evaluation of the complexity of graph shown in Fig. 4.

| Operator | Operator complexity | Number of operators | Total complexity |
|---|---|---|---|
| = | 1 | 14 | 14 |
| [] | 3 | 15 | 45 |
| < | 3 | 12 | 36 |
| ++ | 3 | 13 | 39 |
| /= | 4 | 5 | 20 |
| % | 3 | 5 | 15 |
| read | 1 | 5 | 5 |
| | | | $\sum = 174$ |

An example C-code for counting digits in an integer is presented in Fig. 3. The corresponding DFEG is shown in Fig. 4. The array components are treated as separate scalar elements. The results evaluating the algorithm complexity are reported in Table 2. The overall complexity is 174 basic operators.

## 3 Techniques for Critical Path Evaluation

The approach described in this paper to evaluate 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. 5a) under certain meaningful input data sets. 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. 6b), 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. 6a), 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.

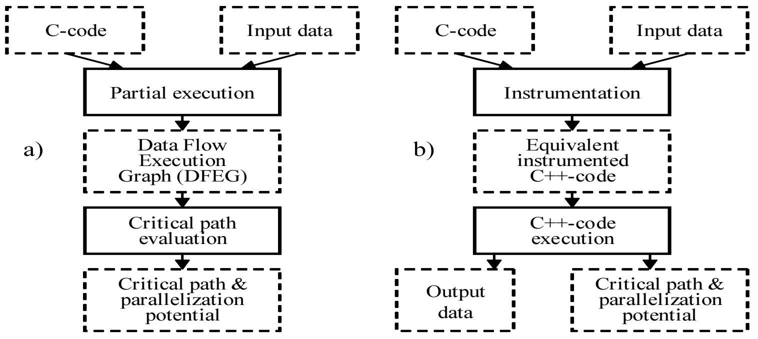
C-code
Input data
Partial execution
a)
Data Flow Execution Graph (DFEG)
Critical path evaluation
Critical path & parallelization potential
C-code
Input data
Instrumentation
b)
Equivalent instrumented C++-code
C++-code execution
Output data
Critical path & parallelization potential

**Fig. 5.** (a) Critical path evaluation by means of explicitly generation of DFEG; (b) Dynamic evaluation of the critical path by means of instrumenting and executing the C-code.

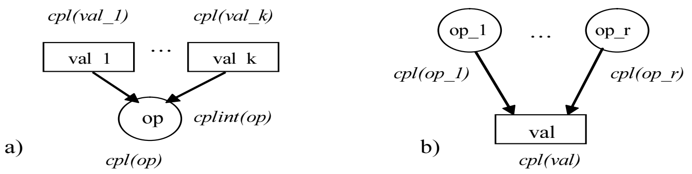

**Fig. 6.** (a) The graph fragment for evaluating the critical path for an operator; (b) The graph fragment for evaluating the critical path for a value (address).

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 no require building the graph. Such a technique is based on the flow shown in Fig. 5b. First, the C-code is instrumented and is transformed into an equivalent C++-code, in terms of the operators applied to the input data. Second, the C++-code is executed under the given input data, computing output data and evaluating the critical path and parallelization potential of the algorithm. In the C++-code, an additional variable *cpl* is bound with each actual scalar variable of the C-code (Fig. 7). The execution of a C-code operation also results in computing a new value of the associated additional variable in the C++-code.

In Fig. 4, address- and value-nodes are weighted with external critical path lengths. The critical path of the whole DFEG is shown in bold. The critical path length equals 30. The DFEG total complexity equals 174. The bounding possible acceleration describing the parallelization potential of the C-code is equal to 5.8.

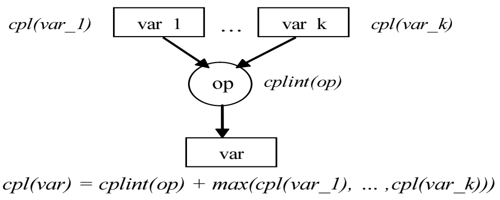

**Fig. 7.** The graph fragment for dynamic evaluation of the critical path.

## 4 Mapping the C-Code into a C++-Code Version

While mapping 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, and functions. To accomplish the evaluation, global and local classes and objects are used in the instrumented C++-code. Among them, there is a critical path stack implementing the mechanism of processing the conditional dependences associated with the nested control structures.

The basic types of the C language are mapped into classes in the C++ language. Each class contains a data element for a scalar variable from the C-code, a data element for the external critical path length and functions defining various operators on the data. Declarations of pointers to the basic types in the C-code are replaced with appropriate classes in the C++-code. An array of elements of a basic type in the C-code is mapped to an array of objects of the corresponding instrumented type. Other composite types of the C language are instrumented in the similar way. All the operators on addresses and values that will be performed during the C-code execution are instrumented during transition from the C-code to the C++-code. The operators on the C-types are overloaded by member functions of the instrumenting C++-classes. The C-functions are instrumented in such a way as to create transparency in transmission of the dependences from the external environment to the function body and from the function body to the external environment. The C-code complexity and critical path can be evaluated for any part of the C-code by means of control elements.

## 5 Critical Path Reduction by Means of C-Code Transformation

The true control structures of the C-code do not permit the direct implementation of the bounding acceleration [11]. The transformation methodology is a mechanism searching for appropriate architectural implementations. It allows the reduction of the execution time (control steps and clock cycles) at the same constraints on resources and approaches to the bonding acceleration. Two types of transformations are investigated in the context of architectural synthesis. The first type transformations aim at reducing the critical path. The transformations of the 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 implementation of the parallelization potential in architectures through C-code transformation. The transformations approach DFG to DFEG.

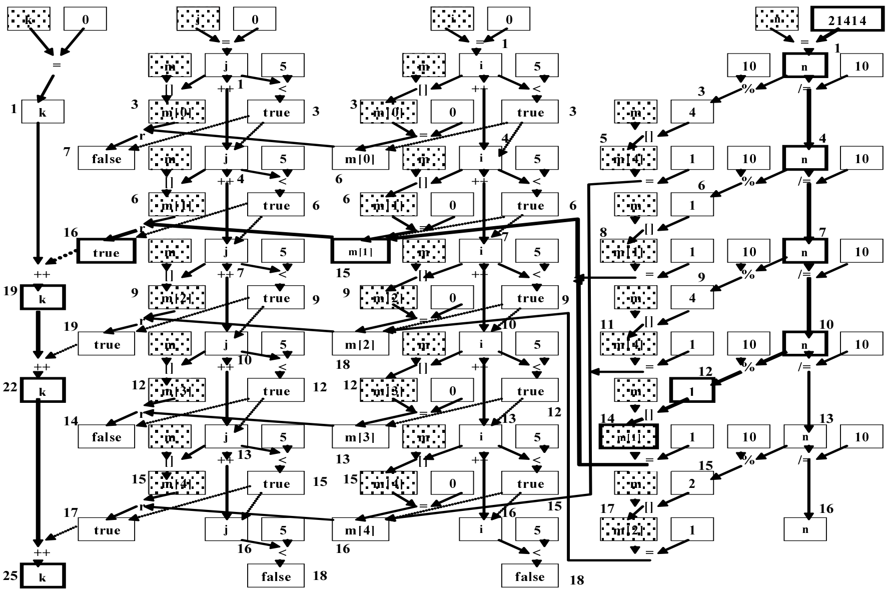

**Fig. 8.** The DFEG for the C-code shown in Fig. 3 and transformed by means of extracting computations from conditional statements.

The equivalent transformation of the source C-code is a way of reducing the critical path length and increasing the parallelization potential. The control and data flow transformation rules allowing the critical path reduction are defined as follows: (1) restructuring, split and transformation of statements; (2) extraction of computations from control structures; (3) algebraic transformation of arithmetic, logic and other type of expressions; (4) merge of expressions and statements; (5) unfolding loops and others. In order to be able to apply other transformation rules to the source C-code, the loop statements should be preliminary transformed by moving the iteration scheme into the loop body. Although the extraction of operators from control structures and other transformations can imply introducing additional variables and computations, it is an efficient way of accelerating the computations and performing the computations in advance and in parallel.

The procedure of increasing the parallelization potential of C-code is an iterative process. First the C-code is transformed and rebuild. Then it is instrumented and mapped to a C++-code version. After the execution of the C++-code and estimation of the critical path and bound acceleration, the source C-code can be transformed again to perform the next iteration. Fig. 8 illustrates the usefulness of transformation rules on the C-code presented in Fig. 3. The critical path length is reduced from 30 to 25. The bounding possible acceleration potential increases from 5.8 to 6.96.

## 6 Results

Several experiments on the critical path and parallelization potential evaluation for two real benchmarks have been executed: Cryptographic toolkit [15] and Wavelet algorithm [13].

### 6.1 Cryptographic Toolkit

The RSAREF is a cryptographic toolkit [15] 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 written entirely 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 3 to 6. The possible bounding acceleration due to the possible parallelization of C-code varies from 42.21 to 136.93. Fig. 9 and Fig. 10 describe the algorithm complexity, critical path length and bound acceleration versus the file and key sizes.

**Table 3.** Experimental results for RSAREF (sign a file).

| Content size (Bytes) | Algorithm complexity | Critical path | Acceleration |
|---|---|---|---|
| 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 4.** Experimental results for RSAREF (seal with sign).

| Content size (Bytes) | Algorithm complexity | Critical path | Acceleration |
|---|---|---|---|
| 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 5.** Experimental results for RSAREF (seal without sign).

| Content size (Bytes) | Algorithm complexity | Critical path | Acceleration |
|---|---|---|---|
| 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 6.** Experimental results for RSAREF (keypair generation).

| Key size (Bits) | Algorithm complexity | Critical path | Acceleration |
|---|---|---|---|
| 508 | 0.6E9 | 13.8E6 | 43.89 |
| 767 | 3.0E9 | 90.0E6 | 33.41 |
| 1024 | 21.9E9 | 806.4E6 | 27.12 |

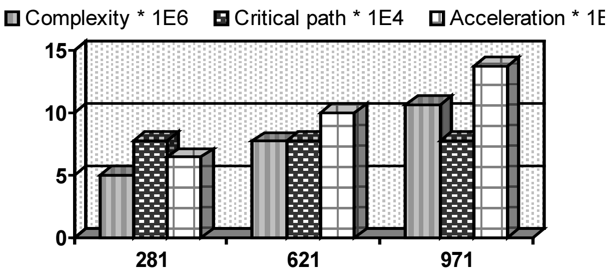

**Fig. 9.** Algorithm complexity, critical path length, and acceleration versus file size for seal.

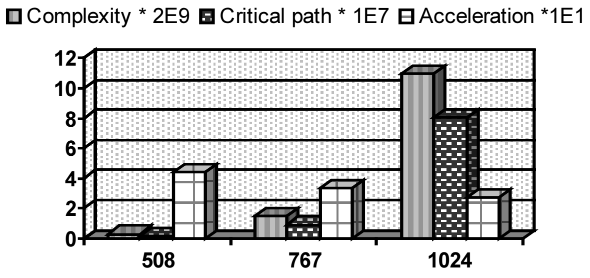

**Fig. 10.** Algorithm complexity, critical path length, and acceleration versus key size for key pair generation.

### 6.2 Wavelet Algorithm

More impressive results have been obtained for the two-dimensional Wavelet codec implementations proposed in [13]. The possible bounding acceleration varies from 30’932 to 1’112’331. The data flow computations to be incorporated in the C-code implementation constitute 37.9% and the control flow computations constitute 62.1%. After the equivalent transformation of WAVELET C-code and modifying its DFG, the critical path length has been reduced by 16.7%.

## 7 Conclusion

The parallelization potential evaluation tool together with the transformation techniques complements the behavioral synthesis tools. Analyzing the results obtained by the tool, the most promising algorithm can be selected among many alternatives. Moreover, the equivalent transformation of algorithm such as for the bounding of possible acceleration of the future parallel architecture can be performed. The critical path length influences significantly the scheduling results at any constraints on resources. The schedule cannot be faster than the critical path length. The reduction of the critical path guarantees more powerful scheduling results and implies the improvement in the trade off “complexity–delay” in behavioral synthesis [11].

**Acknowledgements.** The authors are grateful to Massimo Ravasi and Clerc Christophe for very useful discussions on the topic under consideration and their help in creating the tool.

## References

1. Hollingsworth J., Critical Path Profiling of Message Passing and Shared-Memory Programs, IEEE Trans. on Parallel and Distributed Systems, vol. 9, n. 10, 1998, pp. 1029–1040.
2. Juarez E., Mattavelli M. and Mlynek D., A System-on-a-chip for MPEG-4 Multimedia Stream Processing and Communication, IEEE International Symposium on Circuits and Systems, May 28–31 2000, Geneva, Switzerland.
3. Kobayashi S. and Sagi S., Task Scheduling Algorithm with Corrected Critical Path Length, ISS, Vol. J81-D-I, No.2, pp. 187–194.
4. Kuhn P., Instrumentation tools and methods for MPEG-4 VM: Review and a new proposal, Tech. Rep. M0838, ISO/IEC, Mar. 1996.
5. Kwong Y.-K., Ahmad I., Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors, IEEE Trans. Parallel and Distributed Systems, vol. 7, n. 5, 1996, pp. 506–521.
6. Li Y.S. and Malik S., Performance analysis of embedded software using implicit path enumeration, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1997, Vol. 16, pp. 1477–1487,.
7. Liu L., Du D. and Chen H.-C., An Efficient Parallel Critical Path Algorithm, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 13, n. 7, 1994, pp. 909–919.
8. Mattavelli M. and Brunetton S., Implementing real-time video decoding on multimedia processors by complexity prediction techniques, IEEE Transactions on Consumer Electronics, vol. 44, pp. 760–767, Aug. 1998.

9. Prihozhy A., Mlynek D., Solomennik M. and Mattavelli M., Techniques for Optimization of Net Algorithms, PARELEC 2002 – Parallel Computing in Electrical Engineering, IEEE CS Press, 2002, pp. 211–216.
10. Prihozhy A., Net Scheduling in High-Level Synthesis, IEEE Design & Test of Computers, 1996 spring, pp. 24–33.
11. Prihozhy A., High-Level Synthesis through Transforming VHDL Models, in Book "System-on-Chip Methodologies and Design Languages", Kluwer Academic Publishers, 2001, pp.135–146.
12. Pushner P. and Koza C., Calculating the maximum execution time of real-time programs, Journal of Real- Time Systems, vol. 1, pp. 160–176, Sept. 1989.
13. Ravasi M., Mattavelli M. High-level algorithmic complexity evaluation for system design, to appear on the International Journal on System Architectures, 2003.
14. Ravasi M., Mattavelli M., Schumacher P., Turney R., High-Level Algorithmic Complexity Analysis for the Implementation of a Motion-JPEG2000 Encoder, submitted to PATMOS'2003
15. RSA Data Security, Inc. PKCS #1: RSA Encryption Standard. Version 1.4, June 1991.
16. Wong Y.-C., Hwang S.-Y. and Lin Y., A Parallelism Analyzer for Conservative Parallel Simulation, IEEE Trans. on Parallel and Distributed Systems, vol. 6, n. 6, 1995, pp. 628–638.