Részletes zárójelentés
K72611

A kutató munka tervezett ütemtervéhez képest változtatást igényelt a megfelelő dekompozíciós algoritmus kidolgozásának a vártnál hosszabb kutatási időigénye. A pályázat megfogalmazásakor ugyanis feltételeztük, hogy dekompozícióra, mint NP-teljes problémára könnyebben tudunk a céljainknak megfelelő megoldást kidolgozni gráfelméletből és a hardver/szoftver partícionálásból ismert algoritmusok módosítása révén. Az előre megadható szegmens-szám és kapacitás, a kommunikációs időigények, valamint a felhasználandó processzorok főbb tulajdonságainak figyelembe vehetősége az alkalmazott genetikus algoritmus célfüggvényének megfelelő kialakítására vonatkozó kutató és kísérleti munka révén megoldhatónak bizonyult. Ugyanakkor bebizonyosodott, hogy az eredetileg elképzelt módon nem lehet garantálni, hogy a keletkező szegmensek között ne alakulhassanak ki visszacsatolások, azaz formálisan hurkok, ami a teljes módszer hatékonyságát jelentősen csökkentette volna. A szegmenshurkok ugyanis előre nem látható mértékben eleve korlátozták volna a pipeline újraindítási idő előre megadhatóságát a magas szintű szintézis során. Ezáltal a kutatási terv egyik fő célkitűzése és a szakirodalmi eredményekhez képesti egyik legfontosabb újdonsága került volna veszélybe. Ennek a kutató munka során 2010 júniusában felmerült nehézségnek a leküzdése pótlólagos algoritmus-kutatást és kísérleti munkát igényelt. Az ebből fakadó időcsúszás mérséklése érdekében az eredetileg tervezetthez képest több párhuzamos hallgatói és doktoranduszi munkára volt szükség. Mindaddig, amíg nem találtunk megoldást, a részeredményeink nemzetközi konferenciákon történő publikálását nem tudtuk a kutatási tervben elképzelt módon megkezdeni. Fentiekből következően ugyancsak késedelmet szenvedett a teljes kutatási eredményt képező módszer és keretrendszer publikálása rangos nemzetközi folyóiratban. Miután megtaláltuk a megoldást, már célszerűbbnek találtuk, ha erre a folyóiratcikkre összpontosítunk. A megoldás lényege az, hogy az NP-teljes jellegű dekompozíciót megelőzően mindazokat a szegmensképző vágási lehetőségeket szisztematikusan meghatározzuk, amelyek nem okoznak szegmenshurkokat. Ezek közül a vágási lehetőségek közül is csak azokat kapja meg a genetikus algoritmus bemenetként, amelyek szegmensközi kommunikáció szempontjából a legkedvezőbbek. Így a genetikus algoritmus csak ezek közül a kedvező vágási lehetőségek közül tud választani az eredetileg kidolgozott költségfüggvény szerint. A megoldás részletes leírása az **1. Melléklet**. „***Mapping into a Cutting Matrix (CM)***” című fejezetében található. Ez a dokumentum képezte a közvetlen alapját a teljes kutató munkát összefoglaló azonos című publikációnknak, amelyet a szűkebb szakterület egyik legrangosabb folyóiratához (ACM Transactions on Design Automation of Electronic Systems) küldtünk meg 2012. szeptember 19-én. Jelenleg bírálati szakaszban van.

(Thank you for your recent manscript submission to ACM TODAES.
The Editorial Assistant will shortly transmit your paper to the Editor-in-Chief, who then assigns it to an Associate Editor. (If the paper is a revision of a previously submitted paper, it will automatically be assigned to the previous Associate Editor.)
The Associate Editor will be responsible for handling the review process for your paper -- and should be your primary point-of-contact for any questions you may have regarding your submission and the review process. Once the AE has been assigned, you will be able to locate their name and contact information on the website under "Status".
Please refer to your paper number in any future correspondence.)

Amint az a kutatási tervben és a részjelentésekben is hangsúlyozottan szerepel, a célul kitűzött módszer és keretrendszer meghatározó része a dekompozíciós eljárás, amelynek egyik újnak tekinthető eredménye a kialakuló szegmenseket megvalósító processzor egységek közötti kommunikáció előzetesen becsült időigényének a figyelembe vehetőségének módja a dekompozíció során. Ez különösen fontos olyan célrendszerekre történő alkalmazások esetén, amelyekben a processzorok közötti információcserének valamilyen szabványos sínrendszeren kell történnie. Ilyenkor nagy segítség a tervezés során, ha a különböző leggyakrabban alkalmazott sínrendszerekre egységes becslési módszer áll rendelkezésünkre. A becslési módszerrel kapcsolatos vizsgálatainkat és eredményeinket foglaltuk össze a **2. Mellékletben** leírt cikkben, amelyet 2012. október 12-én küldtünk a Periodica Polytechnica folyóirathoz, ahol jelenleg bírálati szakaszban van.

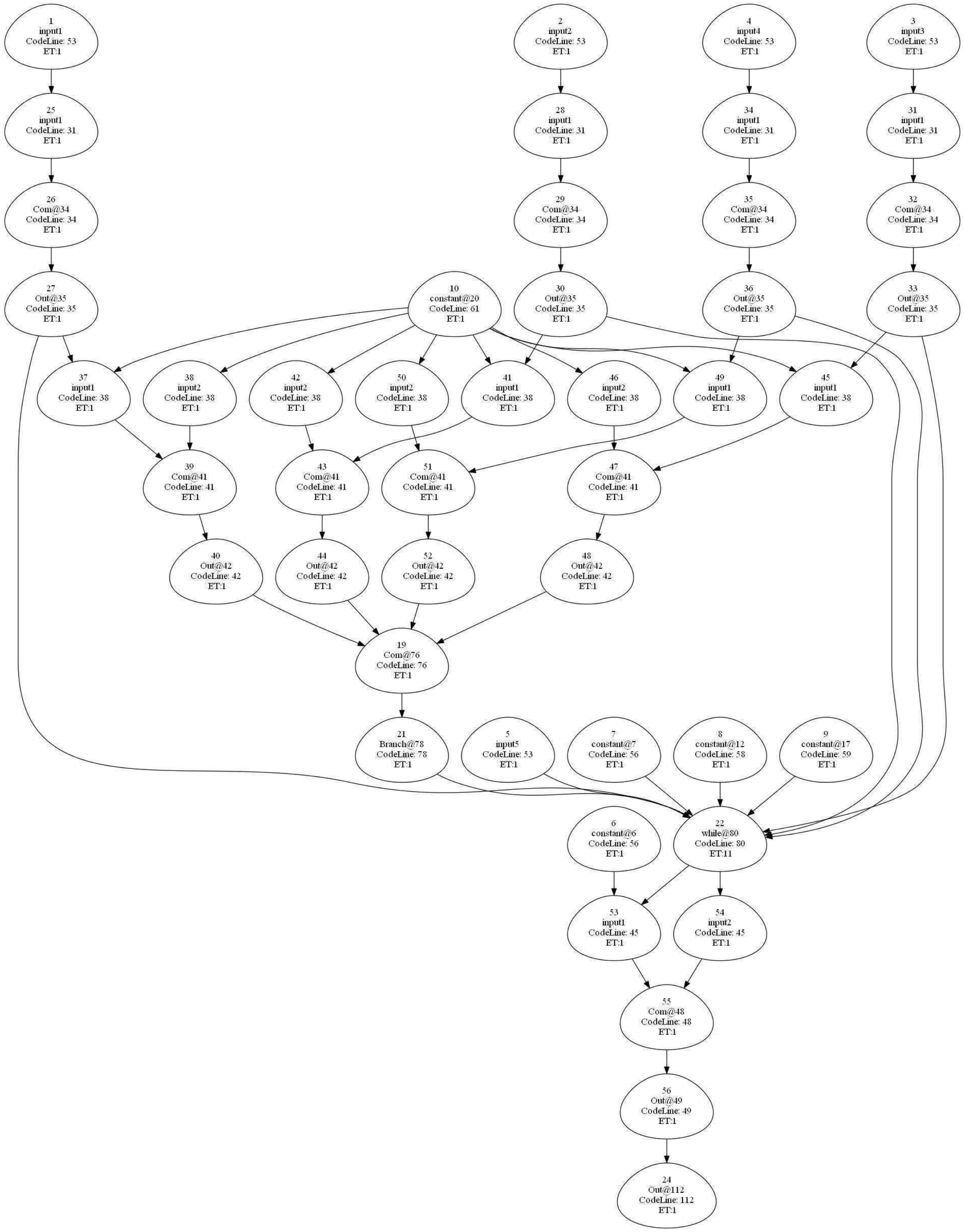
A kutatás eredményeként kifejlesztett tervező keretrendszer hatékonyságát jelentős mértékben befolyásolja a megoldandó feladatot leíró programban lévő ciklusok kezelésének módja a magas szintű szintézis fázisában. A tervező rendszerünk jelenlegi változatában a ciklus-kezelésnek egy egyszerú könnyen megvalósítható módját alkalmaztuk. Nyilvánvaló azonban, hogy a ciklusok lappangási idejének csökkentése révén csökkenthető a rendszer pipeline újraindítási idejét korlátozó hatás. A cikluskezelés algoritmusának kifejlesztése szerepel a jelen kutatás folytatási céljai között is. Ezzel kapcsolatos a **3. Melléklet**, amelynek alapján folyóiratcikket készítünk elő.

A kísérleti példákon és benchmark-feladatokon kiértékelt rész-algoritmusokon kívül a kutatási tervben célul tűztük ki, hogy az elkészült tervező keretrendszert meglévő, lényegében intuitív módon kialakított többprocesszoros rendszerek újratervezése révén is összehasonlító értekelésnek vetjük alá. Ilyen meglévő rendszernek a

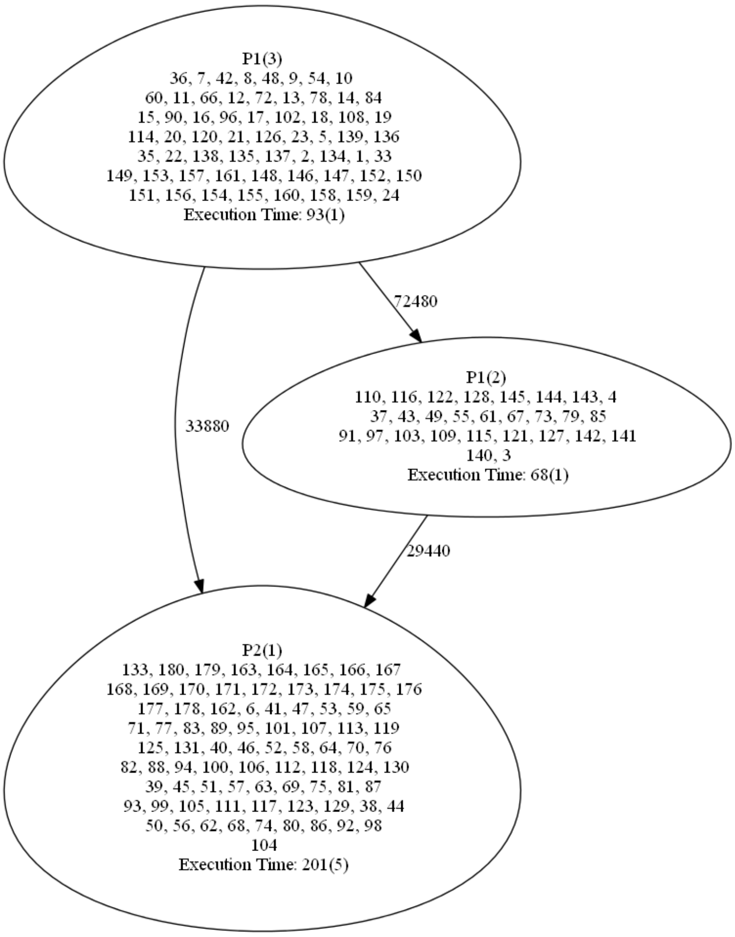
[1] GORACZKO, Michel ; LIU, Jie ; LYMBEROPOULOS, Dimitrios ; MATIC, Slobodan ; PRIYANTHA, Bodhi ; ZHAO, Feng ; FIX, Limor (Bearb.): Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems.. In: *DAC* : ACM, 2008. - ISBN 978-1-60558-115-6, p-p. 191-196

[2] M. Brandstein and H. Silverman. A robust method for speech signal time-delay estimation in reverberant rooms. In ICASSP, page 375. IEEE Computer Society, 1997.

publikációkban leírt, hangforrás lokalizációt megoldó rendszert választottuk. A feladat C nyelvű leírásából az alábbi adatfolyam gráfot (SHSDG) generáltunk.


1 input1 CodeLine: 53 ET:1
2 input2 CodeLine: 53 ET:1
4 input4 CodeLine: 53 ET:1
3 input3 CodeLine: 53 ET:1
25 input1 CodeLine: 31 ET:1
28 input1 CodeLine: 31 ET:1
34 input1 CodeLine: 31 ET:1
31 input1 CodeLine: 31 ET:1
26 Com@34 CodeLine: 34 ET:1
29 Com@34 CodeLine: 34 ET:1
35 Com@34 CodeLine: 34 ET:1
32 Com@34 CodeLine: 34 ET:1
27 Out@35 CodeLine: 35 ET:1
10 constant@20 CodeLine: 61 ET:1
30 Out@35 CodeLine: 35 ET:1
36 Out@35 CodeLine: 35 ET:1
33 Out@35 CodeLine: 35 ET:1
37 input1 CodeLine: 38 ET:1
38 input2 CodeLine: 38 ET:1
42 input2 CodeLine: 38 ET:1
50 input2 CodeLine: 38 ET:1
41 input1 CodeLine: 38 ET:1
46 input2 CodeLine: 38 ET:1
49 input1 CodeLine: 38 ET:1
45 input1 CodeLine: 38 ET:1
39 Com@41 CodeLine: 41 ET:1
43 Com@41 CodeLine: 41 ET:1
51 Com@41 CodeLine: 41 ET:1
47 Com@41 CodeLine: 41 ET:1
40 Out@42 CodeLine: 42 ET:1
44 Out@42 CodeLine: 42 ET:1
52 Out@42 CodeLine: 42 ET:1
48 Out@42 CodeLine: 42 ET:1
19 Com@76 CodeLine: 76 ET:1
21 Branch@78 CodeLine: 78 ET:1
5 input5 CodeLine: 53 ET:1
7 constant@7 CodeLine: 56 ET:1
8 constant@12 CodeLine: 58 ET:1
9 constant@17 CodeLine: 59 ET:1
6 constant@6 CodeLine: 56 ET:1
22 while@80 CodeLine: 80 ET:11
53 input1 CodeLine: 45 ET:1
54 input2 CodeLine: 45 ET:1
55 Com@48 CodeLine: 48 ET:1
56 Out@49 CodeLine: 49 ET:1
24 Out@112 CodeLine: 112 ET:1

Mivel a feladat sok fix iterációs számú hurkot tartalmaz, ezeket egy előfeldolgozó segítségével „kiterítettük”. Az így módosított SHSGH az iterációs hurkok megszüntetése révén jelentős mértékben elősegíti a párhuzamos végrehajtást és a pipeline szervezést.
A dekompozíciós eljárásunk eredményeként az alábbi, az eredeti megvalósításhoz [1] hasonló eredményre jutottunk.



Ebből a megoldásból kiindulva azonban még a keretrendszerünk részét képező PIPE magas szintű tervező eszköz segítségével nagyobb átbocsátási képességű pipeline feldolgozás alakítható ki az előre megadható kívánt újraindítási időre való tervezés révén.
Néhány ehhez hasonló újratervezési feladat befejezése után az eredményekből levonható következtetések alapján folyóiratcikket készítünk elő.

Budapest, 2012. november 26.

____

Dr. Arató Péter
az MTA rendes tagja
Kutatásvezető

1. Melléklet

Synthesis of a Task-dependent Pipelined Multiprocessing Structure

P. Arató[*], D. A. Drexler, G. Kocza, G. Suba

July 24, 2012

Abstract

This paper presents a method for designing a special multiprocessing structure for making the pipeline function possible as a special parallel processing even if there is no efficiently exploitable parallelism in the task description. The starting point of this synthesis method is the task description assumed to be given by a program written in a high level language (e. g. C, Java, etc). The next step is a decomposing algorithm for generating proper segments of the task-describing program. The desired number of the segments and the main properties of the processor set implementing the segments can be given as input parameters for the decomposition algorithm. The estimated communication time-demand is also taken into consideration. For constructing a beneficial pipeline structure, the high-level synthesis (HLS) methodology of pipelined datapaths is applied. The HLS methods attempt to optimize by executing the scheduling and allocation steps applied on the task-oriented input dataflow graph generated from the graph of the segments produced by the decomposition. Therefore, the resulted multiprocessing structure is not a uniform processor grid, but it is shaped depending on the task to be solved, i.e. it can be called a *task-dependent multiprocessing* or *multi-core* structure. In order to show the whole method as a framework, a specific HLS tool is applied, which generates an optimized simple arbitration-free bus system between the processing units. In this structure, there is no need for extra software to organize the communication, if the processing units can transfer input-output data directly. For illustrating and evaluating the method, the step-by-step solution is shown for two tasks as experimental results.

Keywords: program code decomposition, pipelined multiprocessor systems, high-level synthesis hardware/software co-design, system-level synthesis

[*] Budapest University of Technology and Economics, Department of Control Engineering and Information Technology, H-1117, Budapest, Magyar tudósok krt 2., IB321, Phone: +36-1-463-2196, emails: arato@iit.bme.hu, drexler@iit.bme.hu, koczagbr@gmail.com, sugergo@iit.bme.hu

1 Introduction

If a task-describing program has no efficiently exploitable parallelism (it can be called an *essentially serial* or a *sequential one*), then speeding up its execution would not be efficient by applying a uniform parallel processing architecture. In such cases, the execution speed could be increased by applying a higher performance single processing unit, but this would still exclude pipelining to speed up processing large amount of data. However, many practical tasks require and can capitalize the pipelining which is a special case of parallel processing. For constructing a beneficial pipeline structure, the high-level synthesis (HLS) methodology of pipelined datapaths can be applied. Based on proper decomposed parts (*segments*) of the serial program, a dataflow-like graph representation can be formed as the input for a HLS tool. Depending on the desired pipeline throughput (in other words: on the desired value of the *restart time* determined by the frequency of being able to receive new input data), the HLS methods attempt to optimize by executing the scheduling and allocation steps applied on the task-oriented input dataflow-like graph generated from the graph of the segments produced by the decomposition. Therefore, the resulted multiprocessing structure is not a uniform processor grid, but it is shaped depending on the task to be solved by the initial serial program, i.e. it can be called a *task-dependent multiprocessing* or *multi-core* (TDMP) structure designed for pipelined data processing. In Figure 1, the main design steps of the whole method are illustrated as a framework. In the first step, a *Sructure Description Graph* (SDG) is generated from the serial task-describing program (SP). In this paper, it is assumed that SP is written in programming language C. By choosing another high level language for task-description, the SDG generating step may be affected, but the whole design method as a framework remains applicable. As the SDG represents all *hierarchy levels* of SP, a reduced SDG is formed by restricting it to a single (usually to the highest) hierarchy level. Based on this *Single Hierarchy Level* SDG (SHSDG), a mapping algorithm is presented for constructing a *cutting matrix* (CM) to designate all such *allowable* cutting places which may result in beneficial segments by the decomposition. A cutting place in the SHDSG is considered not allowable, if it would cause a loop in the graph formed by the segments as nodes. The decomposition (i.e. an advantageous selection from the allowable cutting places) for generating the segments is then performed by applying a genetic algorithm. The desired number (P) of the resulting segments can be given as an input parameter for the decomposing algorithm. The main properties of the available set of processing units for implementing the segments and the communication time-demand can also be taken into consideration during the decomposition and in selecting from the set of processing units given in advance. The selected processing units and the communication structure between them represent already a TDMP structure enabling also the pipeline function. However, the throughput (i.e. the applicable shortest pipeline restart time) is predetermined by the properties of the processing units implementing the segments and by the communication time between them. To increase the throughput (i.e. to decrease the applicable shortest pipeline restart time), an HLS tool may be used by forming its dataflow-like input graph based on the segments as nodes. In this case, the segments are not assigned to processing units before having the result generated by the HLS tool. In order to show the whole method, the specific HLS tool PIPE [1] is applied in this paper, but other tools could be substituted by modifying the input graph, if it is necessary. The output of this HLS tool is a structure consisting of redefined and replicated processing units communicating on a simple arbitration-free bus system. In this structure, there is no need for extra software to organize the communication, if the processing units can transfer input-output data on the buses directly. The desired value of the restart time (R_d) can be given as an input parameter for the HLS tool PIPE. If the constraints caused by the communication time-demand or loops exclude applying R_d, then the shortest possible restart time (R_p) can also be calculated and implemented by PIPE. For illustrating and evaluating the whole synthesis method the step-by-step solution of two task is shown as experimental results. The modularity of the method permits that some algorithms developed and demonstrated in this paper (decomposition, forming the input for the HLS tool and the tool PIPE itself) may be replaced by other algorithms and tools depending on the properties of the target system and on the special requirements of the application.

The remainder of this paper is organized as follows. As the crucial part of the proposed method is to find a proper segmentation of the initial program SP, Section 2 summarizes some related works in the field of program decomposition. Section 3 presents the basic rules for generating the structure description graphs (SDG and SHSDG). In Section 4 the mapping algorithm for constructing the cutting matrix (CM) is described as a preprocessing step for the decomposing algorithm. The proposed genetic decomposing algorithm is presented in Section 5. In Section 6, experimental results demonstrate the whole method by solving two tasks step-by-step. On these examples it is shown how to modify the graph of segments for generating the input of the HLS tool PIPE. The resulting TDMP architectures and the arbitration-free simple bus systems provided by the extended version of PIPE are also illustrated in this section. Conclusion and further research are summarized in Section 7.

2 Related Work

The TDMP architecture can be considered as the nearest approach to the application-specific multiprocessing structures (ASMP) [2], [3], [4], where the properties of the processing units and the communication time between them are taken into consideration essentially with intuition by analysing the task. Thus, the selection from the available processing units and the task assigning to them are not algorithmized. The desired pipeline restart time is also not an input parameter in the ASMP design. Since this paper aims to present a method for designing a TDMP structure by performing the above synthesis steps as far as possible algorithmically already in the task distribution phase, therefore the most comparable related research results are in the fields characterized by pro-

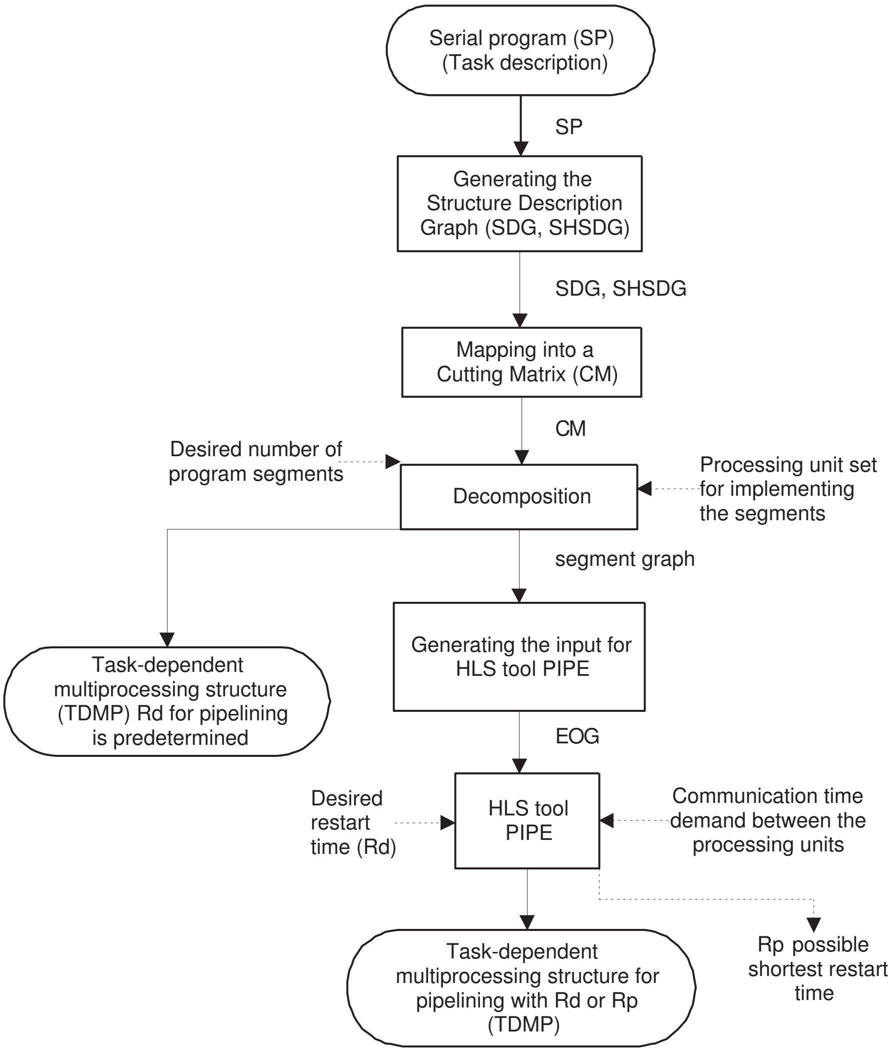

Serial program (SP) (Task description)
SP
Generating the Structure Description Graph (SDG, SHSDG)
SDG, SHSDG
Mapping into a Cutting Matrix (CM)
CM
Desired number of program segments
Decomposition
Processing unit set for implementing the segments
segment graph
Task-dependent multiprocessing structure (TDMP) Rd for pipelining is predetermined
Generating the input for HLS tool PIPE
EOG
Desired restart time (Rd)
HLS tool PIPE
Communication time demand between the processing units
Task-dependent multiprocessing structure for pipelining with Rd or Rp (TDMP)
Rp possible shortest restart time

Figure 1: The main steps of the method

gram decomposition, program slicing, code segmentation and special splitting algorithms in hardware-software co-synthesis. In this paper, the decomposition -as the base of the synthesis method- fulfills the requirements as follows:

- The task describing program is assumed to be basically serial (sequential) i.e. efficiently exploitable parallelism is not necessarily found in it.
- The desired number of code segments resulting from the decomposition can be given in advance and the effect of fewer segments can also be evaluated.
- The available set of processing units and the main properties of them can be taken into consideration in the decomposition algorithm.
- From the resulted graph of segments, the input to a HLS tool can be generated in formal steps.

In the following overview, the above requirements are also considered. There is a wide spectrum in the literature of separating a program code into segments on various purposes. Program slicing is usually used for debugging, program analysis, program understanding, code reusing, software maintenance, program integration, program differencing etc., as shown in [5, 6, 7]. The algorithms of program slicing do not necessarily result in an executable program. Program distribution covers the disassembling of a program and reassembling it in another way preserving its executability and functionality. For example, in reverse engineering [8], this method can be applied to increase the software performance [9] by changing the execution order of the software segments. In the field of hardware-software partitioning, the aim is to improve the performance by implementing the separated parts in software and hardware units respectively [10, 11, 12, 13, 14, 15, 16, 17].

Most of the methods start with a graph representation of the program to be decomposed. The graph representation idea comes from Ottenstein [18, 19] by introducing the program dependence graph. There are very few solutions which mention the way how the graph representation is made from the program to be decomposed. One of the few examples is [13], where the method of [20] is applied for graph generation and the input of the algorithm can be a program written in C, C++ or SpecC. Besides, [13] puts emphasis on finding parallelism in the code, but there is no way to specify the number of the segments or the properties of the processing units. Many program distribution algorithms attempt to find exploitable parallelism. Busherian [8] uses an actor model of the object oriented program, and looks for concurrency among actors by instruction scheduling without considering the properties of the units or the desired number of the segments. Hoffmann et. al. [21] pointed out that parallelizing can be achieved by finding spatiotemporal patterns in the code. They apply task and data partitioning in time and space. Influencing the number of resulting segments is not aimed. Johnson et. al. [22] parallelize the program along its threads and apply a min-cut based algorithm for finding the nearly optimal distribution of the task graph. It is possible here to give the set of the processing units, but the desired number of segments is not a parameter. Knudsen et. al. [12] support the procedure of hardware-software co-design by decomposing into hardware and software parts. Their dynamic programming algorithm aims to minimize the hardware area with a global execution time constraint, and the execution time with a total hardware area constraint. The algorithm detects the non-overlapping sequences. The properties like execution times, communication times are estimated, and exploiting the parallelism in the code is also targeted. Deniziak [23] aims to find the cheapest system architecture satisfying the given time constraint. Execution times and system costs can be given or estimated. The method starts from the program architecture and applies the iterative improvements of the sub-optimal solutions to gain processing speed with time constraint. Watson et. al. [10] exploit parallelism by considering the switching cost between the units and solving a multistage optimization problem. The set of the target processing units can be given. Subramanian et. al. [24] look for parallelism by using a compiler controlled program analysis. Their method is very efficient, but the result is based basically on finding exploitable parallelism. In contrary, Spacey et. al. [25] aim to start from a program without any exploitable parallelism and to give the set and the properties of the target processing units. The number of segments is obtained as a result also in this case. Wolf et. al. [11] and Mann et. al. [26, 27, 28] focus on the desired number of segments. Wolf et al. start from a system graph and reduce the number of nodes only if it is necessary to obtain the given time constraint. The properties of the processing units can also be given. Thus, these methods are the nearest approach to the basic objectives of the method presented in this paper.

3 Generating the Single Hierarchy Level Structure Description Graph (SHSDG)

For constructing the SHSDG, the structural arrangement of the task-describing program SP is to be analyzed. Let the elements of the program structure be called modules. Modules can be statements, function calls or control structures (e.g. ‘if’ condition, ‘for’, ‘while’ loops, etc.) in the program.

Modules can be divided into two groups: *separable* or *not separable* ones. A module is considered not separable, if it does not contain any other modules or it is a loop. Modules contained by the standard library in the high level programming language applied for SP (e.g. operator ‘=’, any mathematical operation, etc.) are also handled as not separable ones even if it contains other modules. Otherwise, the modules are handled as separable ones.

Modules are executed one by one essentially after each other in a basically serial (sequential) program. This sequence of modules represents an *execution order*. Modules are connected by data links. By observing the data links step-by-step from the inputs to the outputs of the program, a *data passing order* can be defined. The data passing order and the execution order can be different. In the data passing order *feedbacks* can occur. A *feedback* is the data passing order leading back to the input of a module already passed. Feedbacks are always inherent parts of a loop.

As a result of the program decomposition, *segments* are to be generated. A *segment* is a set of program modules which are assigned to a common processing unit (e.g. software or hardware). Based on the above definitions and considerations the Structure Description Graph (SDG) can be constructed for an SP. In Figure 2, the SDG of a hypothetical SP is shown as an example. Arrows represent the data links and nodes represent the modules among which both separable and not separable ones occur. Modules m_3, m_5, m_9, m_{14} are separable, because

$$\begin{aligned} d(m_3) &= \{m_1, m_2\} \\ d(m_5) &= \{m_8, m_9\} \\ d(m_9) &= \{m_6, m_7\} \\ d(m_{14}) &= \{m_{10}, m_{11}, m_{12}, m_{13}\}, \end{aligned}$$

where $d(m_i)$ denotes the set of modules that are directly called in m_i, i.e. the set of the direct sub-modules of m_i. For example, m_7 is also a sub-module of m_5, but not a direct one. If $d(m_i)$ is not separable, then formally $d(m_i) = m_i$ can be written. In this sense, m_1, m_2, m_4, m_6, m_7, m_8, m_{10}, m_{11}, m_{12}, m_{13}, are not separable. If the set $d(m_i)$ contains m_k, then m_i is the direct host module of m_k, formally: $h(m_k) = m_i$. In Figure 2:

$$\begin{aligned} h(m_1) &= h(m_2) = m_3 \\ h(m_8) &= h(m_9) = m_5 \\ h(m_6) &= h(m_7) = m_9 \\ h(m_{10}) &= h(m_{11}) = h(m_{12}) = h(m_{13}) = m_{14}. \end{aligned}$$

If m_i has no direct host, then formally $h(m_i) = m_i$ can be written. In Figure 2:

$$\begin{aligned} h(m_3) &= m_3 \\ h(m_4) &= m_4 \\ h(m_5) &= m_5 \\ h(m_{14}) &= m_{14}. \end{aligned}$$

Based on the relations $d(m)$, $h(m)$ and on the data passing order, hierarchy levels (H_n) can be defined. The first hierarchy level (H_1) consists of the set of modules, for which $h(m_i) = m_i$. The n-th hierarchy level (H_n) consists of the set of modules determined by the union of all sets $d(m_i)$ for all m_i-s on the $(n-1)$-th hierarchy level. Formally: $\cup_{n-1}[d(m_i)]$, where $\cup_{n-1}$ denotes the union of the sets on the $n-1$-th hierarchy level.

In Figure 2, the hierarchy levels are as follow:

$$\begin{aligned} H_1 &: \{m_3, m_4, m_5, m_{14}\} \\ H_2 &: \{m_1, m_2, m_4, m_9, m_8, m_{10}, m_{11}, m_{12}, m_{13}\} \\ H_3 &: \{m_1, m_2, m_4, m_6, m_7, m_8, m_{10}, m_{11}, m_{12}, m_{13}\} \end{aligned}$$

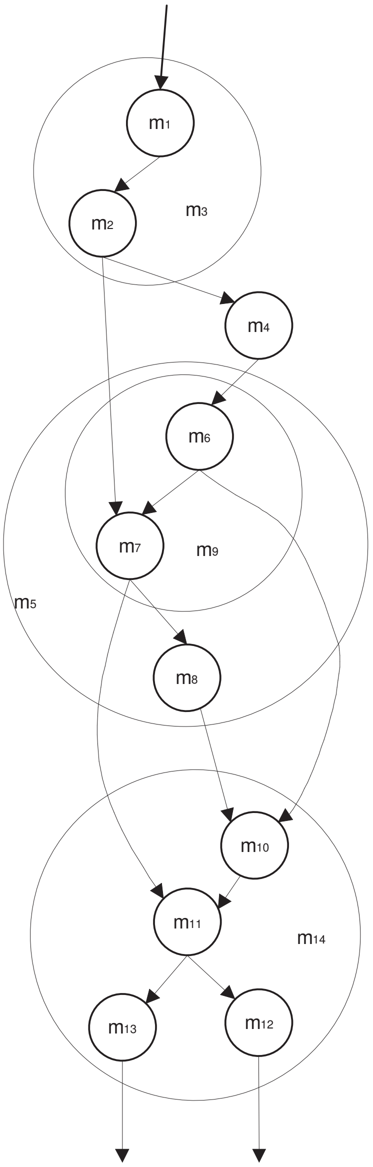


Figure 2: A Structure Description Graph (SDG)

According to the above definition, H_3 is the highest hierarchy level, because further separable modules do not exist. It is obvious that the number of modules on the n-th hierarchy level is always greater than or equal to the number of modules on the $(n-1)$-th level ($|H_n| \geq |H_{n-1}|$). Besides the hierarchy levels, generally other sets of modules may exist, which also represent correct data passing orders. For example, the set $\{m_1, m_2, m_4, m_6, m_7, m_8, m_{14}\}$, represent a correct data passing order, but it is a mixture from different hierarchy levels. If the decomposing algorithm requires checking each data link in the SDG for attempting to find the optimal cutting places in creating the segments, then the calculations should be executed obviously on the highest hierarchy level. Otherwise, some data links would be hidden from checking and evaluation. However, there might be such special requirements or limitations that prefer to consider lower (even mixed) hierarchy levels in checking the data links. As the SDG represents all hierarchy levels of SP, a reduced SDG can be formed by restricting it to the desired single (usually to the highest) hierarchy level. The decomposing algorithm performs separating the modules into segments based on this Single Hierarchy Level SDG (SHSDG). For example, an SHSDG generated from the SDG in Figure 2 for the 2^{nd} hierarchy level is shown in Figure 3.

In this paper, the SP is assumed to be written in C language as an example. Although, the SP could be written in another imperative or functional language (e.g. Java, Haskell, Erlang), it is focused only on C in the next section by presenting a specific method how to generate the SHSDG from SP.

Generating the SHSDG from C code

The *GNU Compiler Collection (GCC)* is a widely used open source compiler from C to machine code. This compiler can be applied for making the generating procedure easier. The compiler has three main parts: the frontend receiving the C source code and producing an intermediate representation (IR) called *Generic* [29], the middle end producing the register-transfer language (RTL), and the backend producing the desired machine code for the target CPU. It is advantageous to use a reduced subset of the Generic IR, called GIMPLE [29], where the expressions are given in a *single-static assignment (SSA)* form [30]. The reasons of this choice are as follow:

- Using the GCC is beneficial in any case, because the difficult processing of the source code can be performed automatically without any additional efforts (compiler steps: lexer, parser, syntax-analyses, and AST creating).
- The RTL would provide already a hardware-based representation, but this hardware would always be a CPU, which would be a very strict limitation.
- The Generic (the output of the frontend) is the first language independent IR, but it is not simple enough, because a large amount of so called *syntactic sugar* still remains in the code.
- Therefore, GIMPLE as the simpler IR seems to be more beneficial to use.

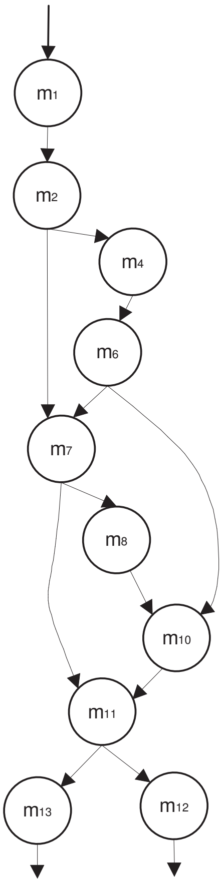


Figure 3: The SHSDG generated on the 2^{nd} hierarchy level of the SDG in Figure 2

The GIMPLE code consists of the same C functions as the source code.The functions are divided into blocks. Every block is an acyclic dataflow graph, and inside a block there are no control flows, the expressions can be evaluated just considering the data dependency. However, a control flow cannot be avoided for determining which block should start in each situation. At completing the execution of a block, it is always determined which block should start. Therefore, control lines appear between the blocks besides data links.

In GIMPLE, the function headers have the same parts as in C (return type, name and parameter list). The function bodies begin with the local variable declarations followed by the *GIMPLE instructions* [31]. The translations of these language elements will be discussed in the following.

The elements of the function parameter list are handled as virtual input nodes (modules). The advantage of these virtual nodes is that the interfaces of a module can be determined easily from the SHSDG model. Moreover, if an input parameter is used by more than one expression, a virtual input node can ensure that only one data link goes into the module (node) representing the input parameter. The outputs of functions (and of the program as well) are also handled as virtual nodes. This guarantee that the output nodes of the graph will have only such outputs that point out of the graph. This feature will be beneficial for the cutting matrix generation algorithm described in Section 4. The execution time of the virtual nodes could be 0, but the tool PIPE [1] can not handle the zero duration time. Therefore, the execution time of these virtual nodes is assumed to be 1. Of course, this causes a minimal increase in the latency, but the pipeline restart time is not affected.

All local variables will be converted to data links between modules (these will not generate any modules at all).

The most of the considerations during the translating procedure are required by the GIMPLE instructions. The correspondence between the SHSDG elements and the GIMPLE instructions are as follows:

- *gimple_label*: it defines a block in the GIMPLE code, and a block is translated into a separable module (the GIMPLE instructions inside a block require the same handling).

- *gimple_goto*: it is a jump instruction appearing in the case of a loop, branch or an explicit goto element of a C code. The direction of the jump can be forward or backward (a backward jump goes to a source code line that it has processed before, and the jump is forward otherwise). A backward jump results always in a loop (*for*, *while*, etc.) handled as a not separable module, because the current version of the applied HLS tool PIPE can not handle the separation of the body of a loop.

- *gimple_assign*: it represents atomic operations (+, -, *, <), therefore not separable module is to be generated in SHSDG.

- *gimple_cond*: it represents a branching condition evaluation, and it determines the next block to start. It is an atomic operation translated into a not separable module in SHSDG.

- *gimple_phi*: it refers to joining the outputs of different blocks. The translation of this instruction results in a not separable multiplexer (MUX) module [1] in SHSDG.

- *gimple_call*: it represents a function call. The function can be user defined or a part of a library. A user defined function is translated into a separable module in SHSDG by inlining. A library function is translated into a not separable module in SHSDG.

- *gimple_return*: it is the last GIMPLE instruction of a function. The translation into SHSDG results in a virtual output node similarly to the virtual input node discussed earlier (the execution time is also 1 here).

Since the SP is assumed to be written in C language, it is not necessary to bother with the following C++ or assembly-related elements of the GIMPLE structure (e.g. gimple_catch, gimple_try, and gimple_asm).

The main advantage of the GCC-supported generating method is that there is no need to deal with the C source code. Thus, it is not necessary to handle language elements that are difficult to process: macros, header files, various syntactic sugars (for cycle, do-while cycle and some special operators e.g. „?:", etc.).

Nevertheless, two very important C language elements, the pointer and the global variables have to be mentioned. A pointer used in a function parameter list represents a pass by reference argument. Therefore, this situation should be represented by applying two data links in the SHSDG: one for the input and one for the output direction. A pointer can refer to a statically or a dynamically allocated memory. In this paper, the dynamical case cannot be handled since the size of the referenced memory block cannot be derived in compile time, but this information should be known for calculating the communication cost for the algorithm in Section 5. Thus, avoiding the pointers with dynamical memory reference is a restriction in writing the SP. Pointers referencing to statically allocated memory are allowed, because the size of the referenced memory block can be derived in compile time according to the pointer analyses techniques [32, 33, 34].

The other problematic elements, the global variables can be removed by applying the variable-classification step described in [35].

Handling of conditional branches

For the modules in the branches, the data passing order cannot be specified by the data links only. Obviously, the actual values on the control lines also influence the data passing order. Therefore, the control lines produced by the condition calculation are also to be taken into consideration besides the data links for handling the conditional branches as separable modules during generation of the SHSDG.

The influence of the control lines on the data passing order can be illustrated by the *control-flow graph* (CFG). The CFG of a conditional branching is shown in Figure 4. a., where dotted arrows represent only the control lines prescribing the control-flow precedence of the modules. The condition evaluation (A) module (gimple_cond) and the modules in the branches (B_i-s and C_i-s) constitute the module set of the conditional branch. In Figure 4. a., only the execution order of the modules is indicated by the dotted arrows. The correspondence between the CFG and the SHSDG is illustrated in Figure 4. b., where only the assumed data passing order is indicated between the modules. The control lines (dotted) from A to all of the modules in B and from A to all of the modules in C have to be added as additional single bit data links in the SHSDG. These additional data links are not indicated in Figure 4. b. in order to maintain the simplicity of the figure. In SHSDG generation, these single bit data links are to be handled as additional input data to each module of the conditional

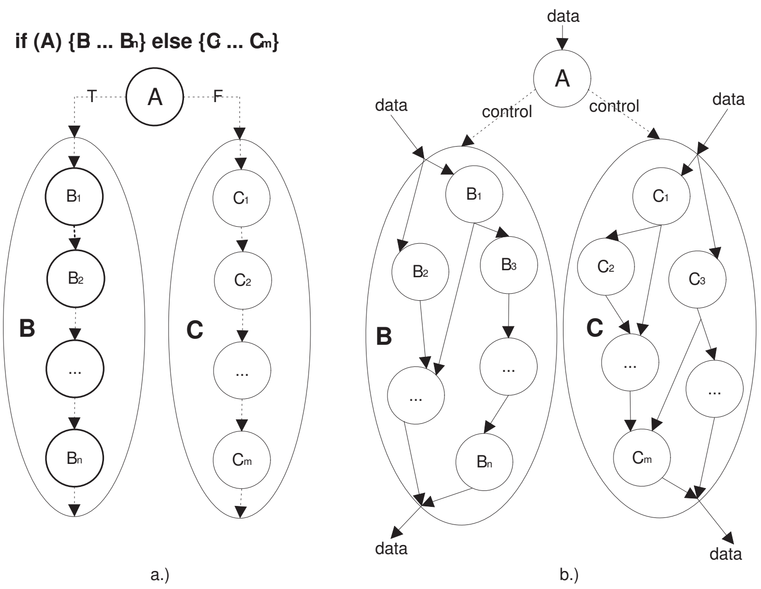


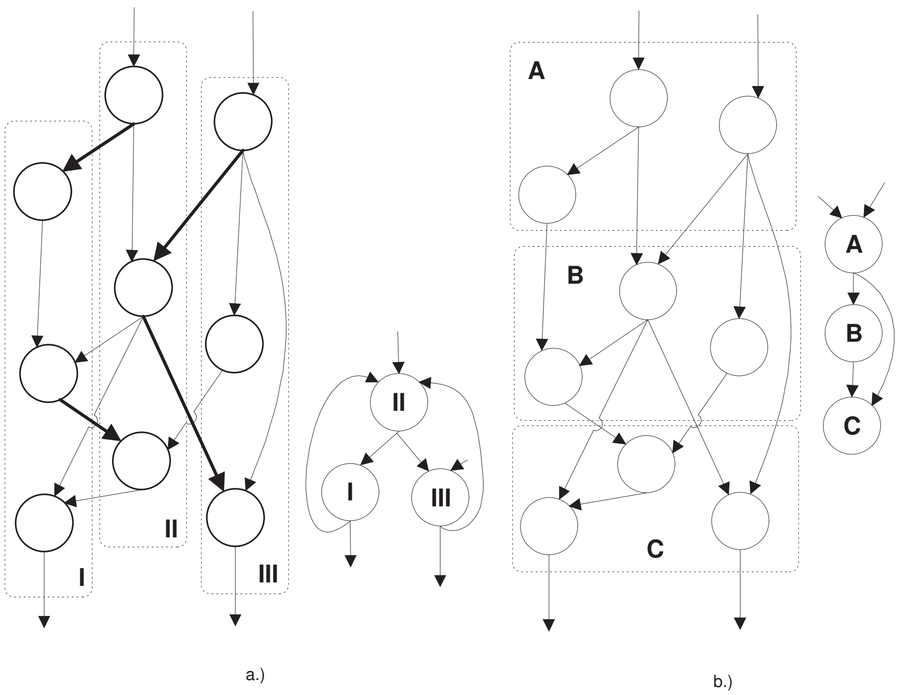
Figure 4: The control flow graph and the SHSDG of the conditional branches; a.) the control of „if" conditional branch b.) SHSDG substitution of „if" control structure

branches in order to activate its operation. These extra single bit data links can ensure the aternative execution of the branches, what may be beneficial in the allocation phase of the HLS tool.

There may be statements in the SP which are not involved in any kind of data passing only in precedence. Such statements are for example printing statements. In such cases, the control links are also considered formally as single bit activating data links when generating the SHSDG.

4 Mapping into a Cutting Matrix (CM)

The aim of the decomposition is to construct the decomposed parts of the program by distributing nodes of the SHSDG into segments. During the distribution, efforts have to be made for forming a structure having only one way communications between the resulted segments. It is important, because bidirectional data links in both directions between segments would form segment loops which are not advantageous in the further design steps. In Figure 5. a., a SHSDG and its resulted segment structure is shown, where the modules are distributed by cuttings (dotted lines) so that data links arise between the segments (thick edges) in both directions. Such *not allowable* cuttings in SHSDG should be avoided by the decomposition algorithm. It is the same problem that



a.)

b.)

Figure 5: Illustration of allowable and not allowable cuttings

occurred in [11], where the not allowable cuttings are called as 'illegal partitions'. In Figure 5. b., the same SHSDG is distributed into segments having no data links in both direction between them. Therefore, no loops would arise with the segments A, B, and C.

The preparatory step of the decomposition algorithm is to find all advantageous allowable cuts (cutting places) and to map them into a *Cutting Matrix* (CM). Generating the CM is performed by the algoritm CMGEN. After this preparatory step, the aim of the decomposition algorithm is to select from the allowable cuts by attempting to optimize according to some parameters given in advance. This can be performed by an algorithm shown in the next section. In this section, the algorithm CMGEN is described. The algorithm is similar to finding the minimal cut or the maximal flow in a weighted graph [36, 37, 38, 39]. While these algorithms are dedicated for distributing the graph into two segments, CMGEN is able to prepare the distribution into more segments generated by allowable cutting places only.

The algorithm allows considering the communication burden (time-demand) between segments during the mapping procedure by assigning proper weights to the data links. It permits that always those allowable cuts are chosen for mapping into the CM which have the smallest weight, if there are more possibilities. Weights may characterize the data quantity to be transferred on the data links, but the weight definitions and assignment can be changed depending on the various implementations of the communication between the processing units in the target system. Definitions and main steps of algorithm CMGEN are summarized in Figure 7, based on the illustration in Figure 6.

The algorithm uses the following definitions. Let those nodes of the SHSDG, which have no predecessor or successor nodes, be called input nodes and output nodes, respectively. For example, an input node is node m_1 and an output is node m_{12} in Figure 3. Let A and B denote disjoint subsets of all nodes in SHSDG. Let two disjoint subsets (X and Y) be generated in B according to the following properties:

- Subset X contains all nodes of B which send data only to nodes of set A.
- Subset Y contains all nodes of B which send data to nodes both of sets B and A.
- The remaining nodes of B are assumed to send data only to the nodes of set B.

Let the following notations be introduced:

- x_i and y_j denote the elements of sets X and Y, respectively
- $w(x_i)$ denotes the sum of weights of all data links having the drain node x_i
- $w(x_i \rightarrow A)$ denotes the sum of weights of all data links having the source node x_i
- w denotes the sum of weights of all data links from set B to set A
- t_k denotes the estimated execution time of a_k, where a_k is an element of set A
- T_A denotes the estimated execution time of the actual node set A. It is defined as the sum of all t_k-s.

The above properties and notations are illustrated in Figure 6. The CM is generated according to Figure 7. Initially, set A contains the output nodes only and set B contains the rest of the nodes. This situation is represented in CM by mapping all output nodes into the first row indicating that this cut generates two segments by cutting at the input data links of all output nodes. This cutting is always trivially allowable because the output nodes have no output data links to other nodes. Further on, each node x_i is relocated one by one from X to A. Since set X contains nodes sending data only to A, all nodes of the actual A send data only inside A. If there would be a node that has both outputs leading out of the graph, and outputs that are inputs to other nodes inside the graph, the algorithm would create a not allowable cut. However, these type of nodes can not exist in the SHSDG, because fictive output nodes are inserted during the

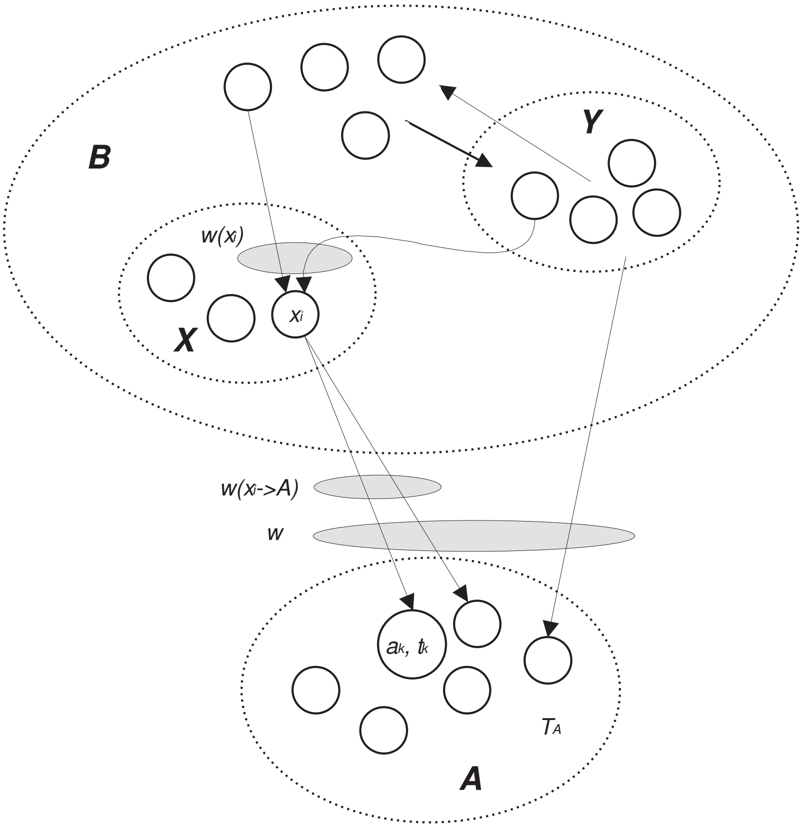


Figure 6: Illustration for the CMGEN algorithm

Input:

G directed graph

Output:

Matrix containing possible cutting places

Definitions:

G : set of nodes of the input graph
A, B : subsets of G where $B = G - A$
k : outputs of the graph
$X \in B$, $\{X\}_{B->A}$: all nodes of B which send data only to nodes of set A
$Y \in B$, $\{Y\}_{B->A}$: all nodes of B which send data to both nodes of set B and A
$w(x_i \to A)$: sum of weights of all data links having the source node x_i
$w(x_i)$: sum of weights of all data links having the drain node x_i
t_k : estimated execution time of a_k, $a_k \in A$
T_A : $\Sigma(t_k)$, $a_k \in A$
z : cycle parameter, indicates the row numbers in the matrix
$\Delta w(x_i) = w(x_{i->A}) - w(x_i)$
w : denotes the sum of weights of all data links from set B to set A

CMGEN algorithm:

```
$A \leftarrow \{k\}$
put $\{k\}$ into the $1st$ row of the matrix
$z = 2$
while($B \neq 0$)
{
    $A \leftarrow \{x_i\}$, where $x_i \in X$ and for any $x_j \in B$: $\Delta w(x_i) \geq \Delta w(x_j)$
    count $T_A = \Sigma(t_k)$
    put $z$ into row $z$ of the matrix
    put $x_i$ into row $z$ of the matrix
    put $T_A$ into row $z$ of the matrix
    $z++$
}
```

Figure 7: The definitions and main steps of algorithm CMGEN

generation procedure as described in Section 3. Therefore, the cuts formed by relocating nodes from X to A are always allowable ones. The relocated node is mapped into the next row of CM. Thus, each row of CM represents an allowable separation of SHSDG into two segments. After each relocation, sets A, B, X, Y, and all weights are actualized and recalculated. If the actual set X contains more nodes, then the selection of x_i for relocating is made by considering the effect on the communication burden between the segments generated by the relocation step. This effect can be characterized by evaluating the change in the value of w after relocating x_i. The aim is to relocate always that x_i which results in the greatest decrease in the new value of w. According to the notations, $w(x_i \to A)$ reduces the value of w, because after the relocation x_i will be already in set A. In contrary, $w(x_i)$ increases the value of w, because all data links having the drain node x_i will appear between the set B and A, i.e. between segments. Thus, the change of w caused by relocating x_i can be expressed as follows:

$$w_{new} = w_{old} - w(x_i \to A) + w(x_i),$$

where w_{old} and w_{new} are the values of w before and after the relocation, respectively. By introducing $\Delta w(x_i) = w(x_i \to A) - w(x_i)$,

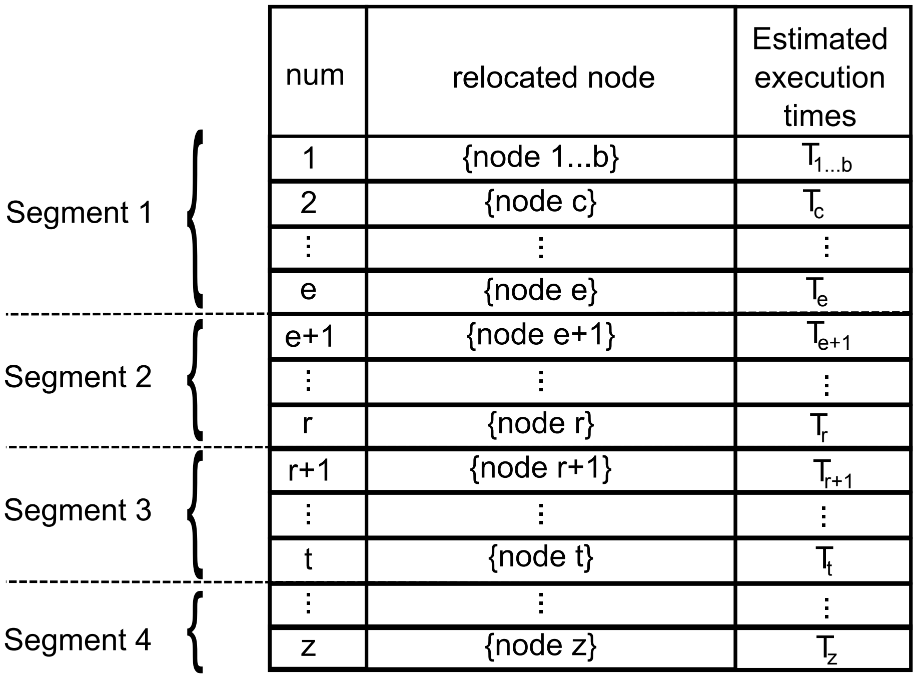
$$w_{new} = w_{old} - \Delta w(x_i).$$

Thus, that x_i should be selected, for which $\Delta w(x_i)$ is the greatest. (In the case of identical greatest $\Delta w(x_i)$ values, the selection might be arbitrary at such simple weight definition).

By continuing the above selection, relocation and recalculation steps, set B will become empty sooner or later, because all outputs of the graph are put into set A at the initialization step of algorithm CMGEN. Therefore, the algorithm relocates each node into set A sooner or later. At this point, the algorithm for mapping into CM ends. Each row represents an allowable cut in SHSDG and the actual contents of sets A and B define the two segments separated by this cut. Because of the above selection strategy for relocating, CM generally does not contain all allowable cuts. Namely, also an allowable cut would be obtained by relocating a node x_i having not the maximal value of $\Delta w(x_i)$. In this case, the order of x_i-s would change in CM, and so the content of separated segments may also be affected. However, such allowable cuts do not appear in CM, if the relocating is always performed according to the maximal value of $\Delta w(x_i)$.

The structure of CM generated by the algorithm CMGEN is illustrated in Figure 8. The first column contains row numbers, the second column identifies the node which is selected for relocation from set X to set A in that step, the third column contains the estimated execution times of the actual set A resulted by summing the execution times of all nodes in A. This is a worst case estimation, since the decreasing effect of the possible parallel or alternative node executions (e.g. conditional branches) are not considered at this point for the sake of simplicity. Of course, more accurate estimation could be done by considering possible parallel or alternative node executions.

As a summary, the properties of the CM are as follow:



num	relocated node	Estimated execution times
1	{node 1...b}	$T_{1...b}$
2	{node c}	T_c
⋮	⋮	⋮
e	{node e}	T_e
e+1	{node e+1}	T_{e+1}
⋮	⋮	⋮
r	{node r}	T_r
r+1	{node r+1}	T_{r+1}
⋮	⋮	⋮
t	{node t}	T_t
⋮	⋮	⋮
z	{node z}	T_z

Segment 1
Segment 2
Segment 3
Segment 4

Figure 8: The structure of the cutting matrix (CM) and an illustration for the segmentation

- Each row of the CM represents an allowable cutting place.
- The first row identifies all output nodes of SHSDG.
- Input nodes can occur in any rows except in the first one.
- A cut (a row) in CM identifies a decomposition of SHSDG by distributing its nodes into two segments composed by the nodes in rows above (set A) and by the nodes below (set B) this cut, respectively.

By the help of the CM, the distribution of the SHSDG into segments can be performed. In the next step of the decomposition algorithm, an optimal selection from the advantageous allowable cutting places is attempted. The desired number of the resulting segments (P) can be given in advance. It means that P sets of rows should be generated in CM. In Figure 8, $P = 4$ is assumed, so three rows of the CM as cutting places are to be determined. An assumed decomposition result is illustrated by the thick horizontal lines in Figure 8 as selected cutting places. The nodes contained by the four segments are assigned by the selected three cutting places. An algorithm for selecting the cutting places is described in the next section.

5 The decomposition algorithm

Various practical conditions may be formulated to influence the decomposition procedure. One of the practical aims can be to prescribe the desired maximal number (P) of segments as an input parameter for the decomposition algorithm. Let $S : \{s_1, s_2, \dots, s_k\}$ denote the set of segments resulted by the decomposition algorithm. Besides $k = P$, the algorithm may propose a segmentation in which $k < P$, if the same or more beneficial solution can be achieved by generating less segments. The $k > P$ case is not considered, since there always must be an upper limit for the number of segments. Usually this upper limit is defined by the number of available processing units for the implementation. Just this could be the reason why P may be prescribed as input parameter in advance.

In order to attempt an optimal segmentation in some sense, a cost function should be constructed that enables to apply a global optimization algorithm. Applying global optimization algorithms (e.g. ant colony optimization [40] or genetic algorithm [41]) is not unfamiliar in partitioning problems, especially in hardware software codesign. The cost function should be defined by considering the purpose of the segmentation and the properties of the target system. In the decomposition algorithm proposed in this paper, two alternatives for the cost function are shown as examples.

The first alternative is to consider the processing units in which the segments will be implemented. Different processing units may have different communication speed, execution speed and capacity. Let a single unique identifying integer from the set $\{1, 2, \dots, P\}$ be assigned to each processing units that is available to implement the segments. Hereby, the assignment of available processing units to segments can be formally represented by the assignment identifying set of integers $U : \{u_1, u_2, \dots, u_P\}$. For example, if segment s_i is assigned to processing unit 2, then $u_i = 2$. Each processing unit is characterized by the following attributes: the speed of execution, the maximal execution time of the segment the can be allocated into the processing unit (also called the capability of the processing unit), and the communication cost between each pair of processing units. Identical types among the available processing units are represented by identical attributes. For attempting to find an optimal solution, the most advantageous processing unit should be selected for each segment. The algorithm presented in this paper does not assign more segments to the same processing unit. Otherwise, the graph of the segments might be affected, even loops might arise in the resulted architecture similarly to the not allowed cuts. Thus $u_i = u_j$ cannot occur. In the second alternative, only the segmentation is performed without assigning the segments to processing units. This alternative is more suitable for applying a HLS tool after the segmentation in order to allocate the segments for establishing advantageous pipeline architecture. In this case, the optimization is attempted by choosing a Pareto-optimal solution that varies between fast communication and fast execution of the segments. The two alternatives outlined above and the relation between P and k represent four cases:

- Case 1: Segmentation and assignment to processing units, $k = P$ only.
- Case 2: Segmentation and assignment to processing units, $k < P$ also allowed.
- Case 3: Segmentation only, $k = P$ only.
- Case 4: Segmentation only, $k < P$ also allowed.

The decomposition algorithm proposed in this paper is illustrated for these four cases as a framework. The first step is to formulate a proper cost function for attempting to find the optimal segmentation in each of the four cases. As an example, let the cost function (f) be introduced as follows:

$$f = \sum_{i=1}^{P-1} \left(G_{u_i}^{-1} T_i + \max_j \left(C_{u_i \to u_j} Com_{i \to j} \right) + \sigma\left(i, u_i\right) \right) + G_{u_P}^{-1} T_P + \sigma\left(P, u_P\right)$$

where

- u_i: is the identifying integer of the processing unit assigned to segment s_i
- G_{u_i}: is the processing speed of the processing unit identified by u_i,
- T_i: is the execution time of segment s_i,
- $C_{u_i \to u_j}$: is the communication cost between processing units identified by u_i and u_j,
- $Com_{i \to j}$: is the communication burden (number of bits transferred in the examples) between segments s_i and s_j,
- $\sigma\left(i, u_i\right)$: is an overfill function for the processing unit identified by u_i assigned to s_i.

Further on, all values of execution time, processing speed and communication time are handled as relative values without considering the dimensions. Note that the sum in the cost function goes from 1 to $P-1$, since $P-1$ cuts are to be found for creating P segments. The capability of the processing unit is used to set a certain limit to the segment assigned to the processing unit. In this example, this measure is defined by the maximum value of the total execution time of the segment assigned to a given processing unit. Note that other limits may also be introduced instead of this, e.g. maximal memory size. Let the total execution time of segment s_i be denoted by T_i. Thus, an overfill function can be defined as follows:

$$\sigma\left(i, u_i\right) = \epsilon\left(T_i - Cap_{u_i}\right)\left(e^{\alpha\left(T_i - Cap_{u_i}\right)} - 1\right)$$

where Cap_{u_i} is a threshold defined for each processing unit u_i referred to as the capability of the processing unit, and α is an appropriately chosen constant, while $\epsilon(\cdot)$ stands for the Heaviside function. The Heaviside function provides the zero overfill function value, if the execution time of the segment to be implemented in u_i is under the limit defined by Cap_{u_i}, but increases exponentially, if this limit is exceeded. The function $\sigma(i, u_i)$ could also be defined as an ideal one that provides zero, if the execution time of s_i is less than or equal to Cap_{u_i}, and provides an infinite value, if the the execution time of s_i exceeds this limit. The function used here approximates this ideal function by a continuous change. The parameter α determines the approximation distance between the ideal function and the continuous one. The greater the α is, the closer the function is to the ideal one. The reason of using a continuous function is to help the convergence of the gradient-like optimum search algorithms.

The cost function basically considers the execution times of the resulting segments that highly depend on the speed of the processing units in which the segments are implemented. The communication cost is added to the execution time inspiring the fact that the communication affects the execution time of the whole SP. The third component in the cost function represents the size-limits of the segments for avoiding useless solutions.

Minimizing the above cost function results in minimal latency, since it is the sum of relative time values for all segments. The minimization of the pipeline restart time will be attempted by the HLS tool, as it is shown in the further sections.

In this paper, the optimization is attempted to be solved by applying a Single-population Genetic Algorithm (SGA) with real population as an example. Each individual is composed of $2P-1$ genes, as follows:

$$\begin{bmatrix} c_1 & c_2 & \dots & c_{P-1} & u_1 & u_2 & \dots & u_P \end{bmatrix}$$

The first $P-1$ genes determine the place of the cuts (the rows of the CM), i.e. c_i determines where segments s_i and s_{i+1} are separated. The values of c_i are chosen from the set $\{1, \dots, J-1\}$, where J is the number of rows of the cutting matrix CM. Note that P has to be chosen such that $P \leq J$ must hold. This may be a limit in prescribing the value of P and it turns out only at this stage. The set of nodes contained in a segment are determined by the second column of the CM that contains the relocated nodes. The first segment is composed of the nodes that are between the 1^{st} row and the c_1-th row of CM ($c_1 = e$ in the example in Figure 8), the second segment is composed of the nodes that are between (c_1+1)-th row and c_2-th row of CM ($c_2 = r$ in the example in Figure 8), and so on. The last segment is composed of the nodes that are between the $(c_{P-1}+1)$-th row and the J-th row of the cutting matrix. The process of creating segments requires that $1 \leq c_1 < c_2 < \dots < c_{P-1} < J$ must hold. Each of the last P genes determines a processing unit u_i, in which segment s_i is implemented, respectively. As the formal representation of the identifying integers, the value of u_i is chosen from the set $\{1, 2, \dots, P\}$, and $u_i \neq u_j$ must hold for all $i \neq j$, and so the last P genes are permutations of the elements of the integers $1, 2, \dots, P$.

In the standard genetic algorithm applied in this paper, only the recombination and mutation operators have to be changed to suit the problem. The recombination of these individuals requires a careful treatment. The first $P-1$ genes are recombined using standard intermediate recombination, however the last P genes cannot be generated by this method, since the last P genes need to be the permutations of the finite series $1, 2, \ldots, P$, i.e. the identifiers of the processing units. If intermediate recombination were applied, then the result would not necessarily be a permutation of that series. Therefore, the recombination of the last P genes is done by creating two offsprings with the same genes at the first $P-1$ position, but different genes at the last P position inherited from both parents. Then, another group of offspring is created by adding random permutations at the last P genes. The best offspring is chosen according to the cost function. The mutation operator acts similarly. The first $P-1$ genes are mutated in standard manner, however the last P genes are mutated by creating a random permutation of the series $1, 2, \ldots, P$.

The minimal value of the cost function is the upper limit of the latency of the best individual, if the capability limits of the processing units are not exceeded. Namely, the latency is the sum of the execution and communication times.

The decomposition framework presented above is suitable for handling each of the four cases. For example, a simple modification in the procedure can make it suitable for Cases 2 and 4, when the result may contain less than P segments, if the minimization of the cost function were more beneficial in this way. In order to ensure this feature of the optimization process, the value of zero can be assigned to any of the last P genes of the individuals. The zero value of a gene in one of the last P places means that the segment is not realized at all, thus it does not exist. This zero value can be interpreted as a fictive processing unit, with zero processing speed ($G_0 = 0$), zero communication cost ($C_{0 \to u_j} = 0$), and infinite capability threshold ($Cap_0 = \infty$). During the optimization procedure, this zero assignment can be attempted and evaluated for any number of genes at the last P places of the individuals to simulate as if the corresponding c_i genes were neglected as cutting places.

In Cases 3 and 4, the result of the segmentation serves as input for a HLS tool. Therefore, the types and the properties of the processing units will be determined by the allocation result of a HLS tool. The cost function applied above for a general case can be fitted easily to these cases by defining formally the same type for each processing unit. The speed and the communication burden of this single type processing unit determine the trade-off between optimizing the minimal execution time of a segment and the communication time between segments. The capability of the single type processing unit serves as an upper bound on the maximal execution time of a segment in this case.

6 Experimental results

The main steps and design cases of the method described in the previous sections are illustrated on an example C code used in audio synthesis. The program consists of three functions, the main function and two other user defined functions. The nodes associated to the commands of the program code are denoted in the comments. Pure variable declarations are handled as the inputs of the program. Thus the program code (called SP in the previous sections) is as follows:

```
#include<stdio.h>
#include<math.h>

//            //Node 11            //Node 12     //Node 13
void genSweep(float *buffer,long numSamples,int sampleRate,...
      //Node 14      //Node 15
float  minFreq,float maxFreq){
   float start=2.0*3.14*minFreq; //Node 16
   float stop=2.0*3.14*maxFreq; //Node 17
   float tmp1=log(stop/start); //Node 18
   long s=0; //Node 19
   while(s < numSamples) //Node 20
   {
           float t;
           t=(float)s/numSamples;
           float tmp2;
           tmp2=exp(t*tmp1)-1.0;
           buffer[s]=sin((start*(float)numSamples*tmp2)/...
           ((float)sampleRate*tmp1));
           s++;
   }
   // Node 21 is a fictive out node
}
//                //Node 22        //Node 23       //Node 24
void fadeInOut(float *buffer, long numSamples, int sampleRate,...
//Node 25
float fadeTime)
{
   long numFadeSamples=fadeTime*sampleRate; //Node 26
   if(numFadeSamples > numSamples)  //Node 27
   // Node 28 for branch node
   {
           numFadeSamples=numSamples; //Node 29
   }
   long s=0; //Node 30
   while(s < numFadeSamples) //Node 31
   {
       float weight;
       weight=0.5*(1-cos(3.14*s/(numFadeSamples-1)));
       buffer[s]=buffer[s] * weight;
       buffer[numSamples-(s+1)]=buffer[numSamples-(s+1)]*weight;
       s++;
   }
   //Node 32 is a fictive out node
}

 //                     //Node 4       //Node 1       //Node 2
float* top_module(float *buffer, int sampleRate, float duration,..
      //Node 5       //Node 6       //Node 7
float  minfreq,float maxfreq, float fadetime) {
   long numSamples=duration*sampleRate; //Node 3
   genSweep(buffer,numSamples,sampleRate,minfreq,maxfreq);//Node 8
   fadeInOut(buffer,numSamples,sampleRate,fadetime); //Node 9
   return buffer; //Node 10
 }

// main function for the test on CPU
int main() {
   float buffer[1];
   top_module(buffer, 1, 1, 440, 4400, 10);
   for (int i=0; i<1; i++)
   {
          printf("%f\n",buffer[i]);
   }
 }
```

The SHSDG belonging to the highest (in this case the 3^{rd}) hierarchy level is shown in Figure 9. The labels in the nodes consist of two rows: the first one is the identifying index of the node, and the second one refers to the estimated execution time of the node. In this simple illustration, the estimated execution times are assumed to be 1 for each node which does not represent a loop. The nodes representing loops (nodes 20 and 31) are assumed to have the estimated execution times determined by the sum of execution times of the nodes forming the loop (as if the loops were executed only once). Of course, this simplified estimation is made only for demonstrating formally the steps of the synthesis framework presented in this paper. Realistic estimations might be elaborated by considering the type and character of the loop. Such considerations and the loop handling are crucial in high level synthesis [1], [42], [43], [44]. To develop loop rules dedicated specifically to the synthesis method presented in this paper is one of the aims for further research. In this particular case, the number of loop cycle depends on input variables *duration* and *sampleRate*, these are assumed to be 1. The execution time of both loops is $4j$ (since there are 4 nodes in the loop), where $j = duration * sampleRate$. In this example $j = 1$. This simplification would be justified, if the loop in the C code is assumed to get all the data in the buffer array variable, and each loop cycle is executed on a single element of the array independently. In this case, the whole buffer array can be loaded and handled in pipeline mode. In consequence, the whole program would be executed for each element of the buffer. Thus, some program parts (which should be executed only once for a specific buffer array) would be executed multiple times (e.g. nodes 5, 6, 14, 15, 16, 17, 18 in Figure 9). However, this solution implies that the size of the buffer array does not affect the structure of the resulting system, but it determines the pipeline restart time.

The Cutting Matrix resulting from the SHSDG can be seen in Figure 10. The first column of the matrix (num) contains the identifying indices of the cutting places. The c_i genes in the genetic algorithm refer to these indices. The second column contains the nodes that are actually relocated to the set containing the nodes above a given index in the matrix. The third column contains the cumulated estimated execution times of the nodes forming the segment above the cutting place. These cumulated estimated execution times are recalculated at each step by adding the estimated execution time of the node defined in the second column. For the sake of simplicity, it is a worst case estimation, since it does not consider the possible parallel or alternative node executions.

Further on, some results are shown in the four design cases defined in the previous section.

Case 1

Let two different types of processing units be assumed, and they will be called hardware and software. Thus, this problem is analogous to the well-known hardware-software splitting. Let the relative properties (represented by integers) of the units be assumed as follows:

- The processing speed of the software is $G_s = 1$
- The processing speed of the hardware is $G_h = 6$
- The communication cost between hardware and software is $C_{h \to s} = 3$
- The communication cost between two hardware is $C_{h \to h} = 2$
- The communication cost between two software is $C_{s \to s} = 5$
- The capability of the software is $Cap_s = 15$
- The capability of the hardware is $Cap_h = 10$

The decomposition is done first with $k = P$, using 3 software and 3 hardware units, i.e. $P = 6$. The result of the decomposition is represented by an adjacency half matrix in Figure 11. Each row and each column is labeled by a segment s_i. The number in the cell corresponding to s_j and to s_k represents the number of bits that need to be transferred from segment s_j to segment s_k. The communication cost between the processing units is not indicated in the adjacency half matrix.

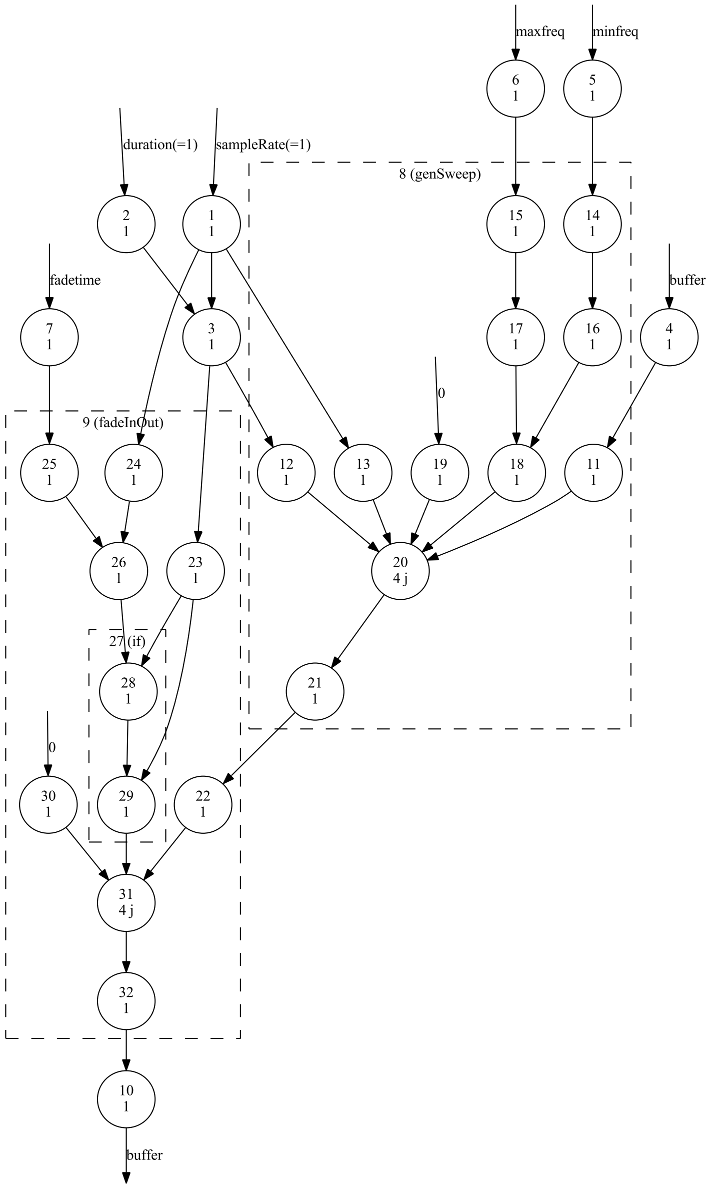


Figure 9: The SHSDG of the program code on the highest hierarchy level

num	relocated node	Estimated execution times
1	10	1
2	32	2
3	31	6
4	30	7
5	29	8
6	22	9
7	21	10
8	28	11
9	23	12
10	26	13
11	25	14
12	7	15
13	24	16
14	20	20
15	19	21
16	11	22
17	4	23
18	12	24
19	3	25
20	2	26
21	13	27
22	1	28
23	18	29
24	17	30
25	16	31
26	15	32
27	6	33
28	14	34
29	5	35

Figure 10: The Cutting Matrix of the SHSDG in Figure 9

$$\begin{array}{cccccc}
s_2 & 32 & & & & \\
s_3 & 48 & 0 & & & \\
s_4 & 0 & 0 & 32 & & \\
s_5 & 32 & 0 & 64 & 0 & \\
s_6 & 0 & 0 & 144 & 0 & 0 \\
 & s_1 & s_2 & s_3 & s_4 & s_5
\end{array}$$

Figure 11: The adjacency half matrix of the decomposition result with $P = k = 6$, 3 hardware and 3 software units

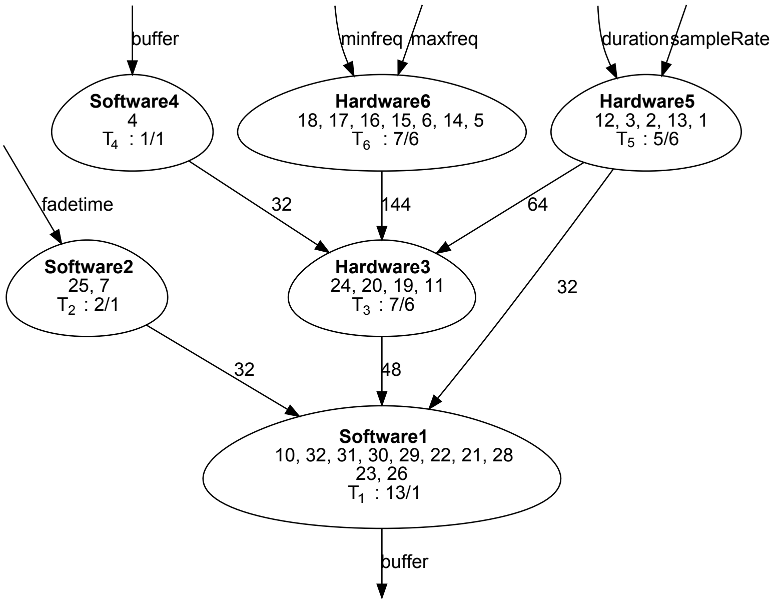


Figure 12: The result of the decomposition for Case 1, using 3 software and 3 hardware units

The resulting segments contain the following nodes (with execution times T_i):

$$
\begin{array}{llr}
s_1 = \{ \; 10 \;\; 32 \;\; 31 \;\; 30 \;\; 29 \;\; 22 \;\; 21 \;\; 28 \;\; 23 \;\; 26 \; \} & T_1 = 13 \\
s_2 = \{ \; 25 \;\; 7 \; \} & T_2 = 2 \\
s_3 = \{ \; 24 \;\; 20 \;\; 19 \;\; 11 \; \} & T_3 = 7 \\
s_4 = \{ \; 4 \; \} & T_4 = 1 \\
s_5 = \{ \; 12 \;\; 3 \;\; 2 \;\; 13 \;\; 1 \; \} & T_5 = 5 \\
s_6 = \{ \; 18 \;\; 17 \;\; 16 \;\; 15 \;\; 6 \;\; 14 \;\; 5 \; \} & T_6 = 7
\end{array}
$$

Segments 1, 2 and 4 are implemented in software, and segments 3, 5 and 6 are implemented in hardware processing units. The best value of the cost function delivered by the decomposition algorithm is $f = 508$. The result of the decomposition is also depicted as a directed graph in Figure 12. The segments are the nodes of the graph. The labels of the nodes consist of three rows. The first row is the name of the processing unit type implementing the segment. The second row is the list of SHSDG nodes allocated into the segment, and the last row is the execution time divided by the relative processing speed of the processing unit implementing the segment.

$$\begin{array}{cccc}
s_2 & 48 & & \\
s_3 & 32 & 64 & \\
s_4 & 0 & 144 & 0 \\
 & s_1 & s_2 & s_3
\end{array}$$

Figure 13: The adjacency half matrix of the decomposition result with $k < P = 6$ (3 hardware and 3 software units), the result consists of 3 hardware and 1 software units.

Case 2

In this case, $k < P$ is also allowed. Let the number and types of processing units be assumed as in Case 1. Let the cost function be modified by an extra $+\lambda k$ term, where $\lambda > 0$ is a parameter for quantifying how small number of segments would be favored by the user. By executing the decomposition for various values of parameter λ, the user may tune the result according to several requirements in the target system. Thus the cost function used in Case 2 (and also in Case 4 later on) is

$$f = \sum_{i=1}^{k-1} \left(G_{u_i}^{-1} T_i + \max_j \left(C_{u_i \to u_j} Com_{i \to j} \right) + \sigma\left(i, u_i\right) \right) + G_{u_k}^{-1} T_k + \sigma\left(k, u_k\right) + \lambda k$$

where the indices go from 1 to k instead of 1 to P and the extra $+\lambda k$ term is added.

For example, let the effects of $\lambda = 90$ be examined. The result of the decomposition is 4 segments, 3 of them implemented in hardware, and one of them in software units. The adjacency half matrix is shown in Figure 13.

The resulting segments contain the following nodes (with execution times T_i):

$$\begin{array}{ll}
s_1 = \{\ 10\quad 32\quad 31\quad 30\quad 29\quad 22\quad 21\quad 28\quad 23\quad 26\quad 25\quad 7\ \} & T_1 = 15 \\
s_2 = \{\ 24\quad 20\quad 19\quad 11\quad 4\ \} & T_2 = 8 \\
s_3 = \{\ 12\quad 3\quad 2\quad 13\quad 1\ \} & T_3 = 5 \\
s_4 = \{\ 18\quad 17\quad 16\quad 15\quad 6\quad 14\quad 5\ \} & T_4 = 7
\end{array}$$

Segment 1 is implemented in software, segments 2, 3, and 4 are in hardware units. The resulted best value of the cost function is $f = 941.1677$ in this case. This value is greater than in Case 1, however the comparison would be realistic by considering the extra $+90k$ component in the cost function. The impact of the original components can be compared, if $90k$ is subtracted from the value of the cost function obtained in this case, i.e. $941.1677 - 90 \cdot 4 = 581.1677$, which is greater than the cost value resulted in Case 1. Thus, Case 1 provides a better solution, if the effect of less processing units is not considered (see later Figure 18). For example, the decomposition algorithm has been executed in order to check whether $k < P = 3$ could offer a smaller value for the cost function. All of the three processing units are assumed to be hardware type now. The best

$$\begin{array}{cccccc}
s_2 & 32 & & & & \\
s_3 & 32 & 16 & & & \\
s_4 & 0 & 0 & 32 & & \\
s_5 & 32 & 0 & 64 & 0 & \\
s_6 & 0 & 0 & 144 & 0 & 0 \\
 & s_1 & s_2 & s_3 & s_4 & s_5
\end{array}$$

Figure 14: The adjacency half matrix of the result of decomposition in Case 3

result is obtained with 3 segments, i.e. $k = P = 3$ in this case. The resulted cost function value for comparison is $1435 - 90 \cdot 3 = 1165$. Thus, $k = 4$ in the former case seems to be a better solution, if the price of more processing units is not considered (see later Figure 18).

Case 3

In this case only segments are created without distributing them into processing units, because the result of the segmentation is assumed to be the input of a high-level synthesis tool in order to generate the proper allocation for establishing a pipeline system. The same cost function as the one defined in the previous section may be used also in this case. Formally only one type of fictive processing units is assumed. The ratio between the processing speed and the communication cost properties of this fictive processing unit is a tool to scale whether the low communication time or the low execution time is preferred. Let the processing speed of the processing unit be assumed $G_s = 1$, so the relative value of the communication cost defines this ratio. In this example the value of the communication cost is chosen as $C_{s \to s} = 2$. Let the capability of the fictive processing unit be $Cap_s = 15$. First the case $k = P = 6$ is considered. The resulted adjacency half matrix is shown in Figure 14. The resulted best value of the cost function is $f = 419$.

The segments contain the following nodes (with execution times T_i):

$$\begin{array}{llr}
s_1 = \{ \; 10 \;\; 32 \;\; 31 \;\; 30 \;\; 29 \;\; 22 \;\; 21 \;\; 28 \;\; 23 \; \} & & T_1 = 12 \\
s_2 = \{ \; 26 \;\; 25 \;\; 7 \; \} & & T_2 = 3 \\
s_3 = \{ \; 24 \;\; 20 \;\; 19 \; \} & & T_3 = 6 \\
s_4 = \{ \; 11 \;\; 4 \; \} & & T_4 = 2 \\
s_5 = \{ \; 12 \;\; 3 \;\; 2 \;\; 13 \;\; 1 \; \} & & T_5 = 5 \\
s_6 = \{ \; 18 \;\; 17 \;\; 16 \;\; 15 \;\; 6 \;\; 14 \;\; 5 \; \} & & T_6 = 7
\end{array}$$

The result of the decomposition is depiceted as a directed graph in Figure 19.

Case 4

Now let $k < P$ be also allowed. The cost function is modified by the $+\lambda k$ term with $\lambda = 90$ as in Case 2. The adjacency half matrix of the result is shown in Figure 15. The value obtained for the cost function is $f = 773$. In

$$\begin{array}{cccc}
s_2 & 16 & & \\
s_3 & 64 & 16 & \\
s_4 & 0 & 0 & 64 \\
 & s_1 & s_2 & s_3
\end{array}$$

Figure 15: The adjacency half matrix of the result of the decomposition in Case 4

order to compare this result with the value in Case 3, the cost function value obtained in Case 4 should be modified by subtracting the extra λk term, i.e. it is $773 - 90 \cdot 3 = 503$, that is greater than the result of Case 3. Thus, Case 3 provides a better solution, if the price gain of applying less processing units is not considered (see later Figure 18).

The segments contain the following nodes:

$$\begin{array}{ll}
s_1 = \{\ 10\ \ 32\ \ 31\ \ 30\ \ 29\ \ 22\ \ 21\ \ 28\ \ 23\ \ 26\ \ 25\ \ 7\ \} & T_1 = 15 \\
s_2 = \{\ 24\ \} & T_2 = 1 \\
s_3 = \{\ 20\ \ 19\ \ 11\ \ 4\ \ 12\ \ 3\ \ 2\ \ 13\ \ 1\ \ 18\ \ 17\ \ 16\ \} & T_3 = 15 \\
s_4 = \{\ 15\ \ 6\ \ 14\ \ 5\ \} & T_4 = 4
\end{array}$$

The effect of the segment weighting factor λ

Most of the parameters in the decomposition algorithm (processing speed, communication cost, etc.) are the properties of the processing units. However, the role of parameter λ is to influence the algorithm by the user. There is no straightforward way to determine a proper value for λ weighting the number of segments during the decomposition, but some experimental considerations for a justifiable choice are presented in the following. Let the value of λ be varied from 1 to 196 with step size 5, thus $\lambda = \{1, 6, 11, \ldots, 196\}$ in Case 4. Since the decomposition is based on a genetic algorithm with finite generations, the result may be different at each run. Therefore, the further illustration is based on the decomposition executed -for example- 18 times for each above value of λ. In Figures 16 and 17 the mean of the resulted number of program segments ($E\{k\}$) and the mean of the cost function values ($E\{f\}$) are depicted. Based on these figures, two trends can be observed by increasing the value of λ: the number of segments is decreasing, and the value of the cost function is increasing.

The first trend is trivial, since the greater the value of λ is, the less number of segments is favored in the decomposition. The second trend is also easy to explain, because increasing λ leads to worse results regarding the original optimality criteria components (execution times and communication costs). This is also evident in a multicriteria optimization, since the introduction of the new criterion weakens the effect of the other criteria.

Considering the above trends, the question can be formulated: how to find a beneficial value of λ and how to consider the effect of k? The mutual effect of these parameters can be observed by analyzing the results in Figure 18.

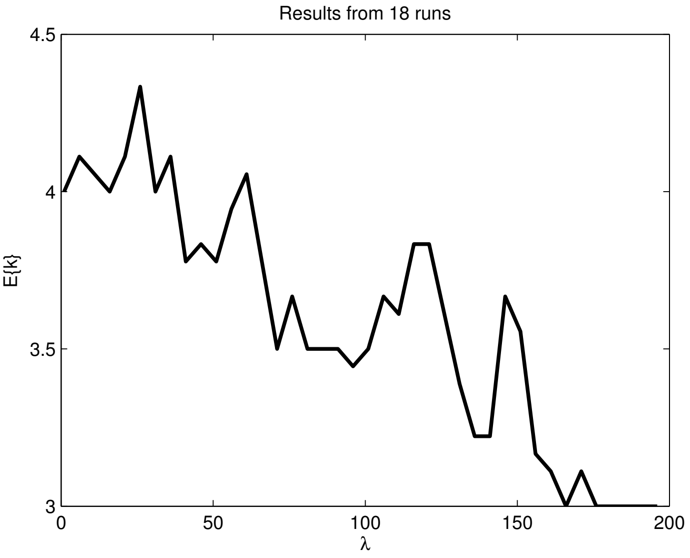


Figure 16: The mean of the number of segments for different values of λ after 18 runs of the decomposition algorithm

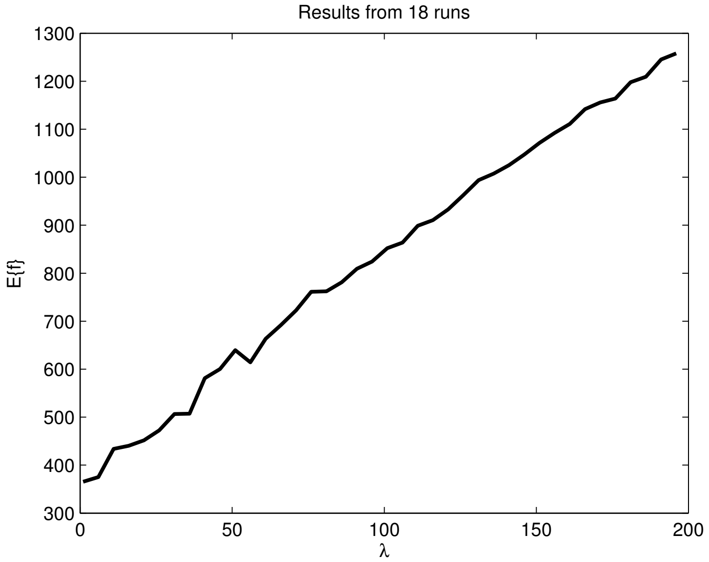


Figure 17: The mean of the cost function values for different value of λ after 18 runs of the decomposition algorithm

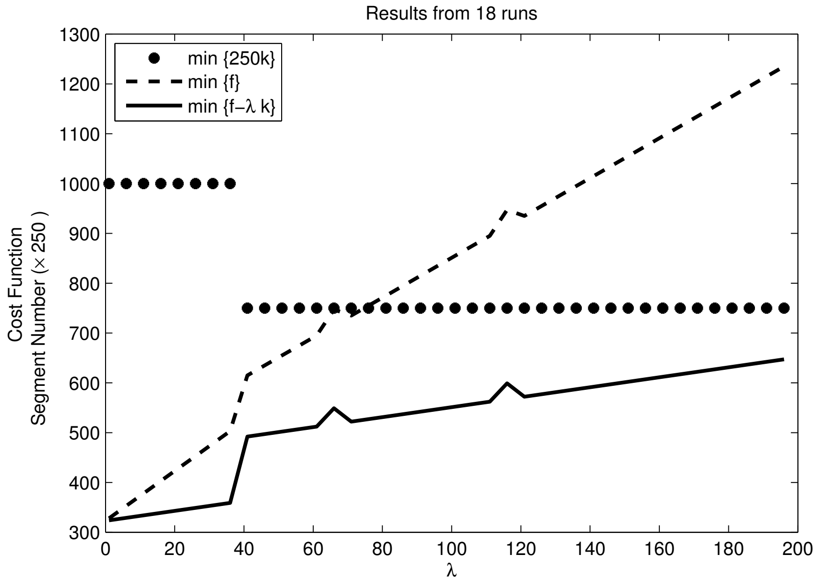


Figure 18: The dashed line depicts the minimal values of the cost function provided by the solutions for which the number of segments (dots) is minimal. The solid line demonstrates the minimal values of the cost function decreased by λk

In this figure, the minimal values of the cost functions (instead of the mean as in Figure 16 and Figure 17) belonging to the minimal numbers of segments at a certain value of λ are depicted. The dots indicate the number of segments obtained from the decomposition algorithm, and their value is multiplied by a factor of 250 for easy visualization together with the cost values. The trend of the cost function values is quasi-linear with a significant change at that value of λ, where the minimal number of segments changes as well (around $\lambda = 40$ in Figure 18). Thus, a beneficial choice seems to be the smallest value of λ, where the smallest value for the number of segments occurs, since this value may bring the smallest distortion in the original cost function belonging to Cases 1 and 3. However, the significant change in the cost function value should be taken into consideration even in this case, too.

Generating the input for the HSL tool PIPE

The result of the segmentation algorithm in Cases 3 and 4 from the previous section can serve as an input to any HLS tool after appropriate transformations. In this section, the use of the HLS tool PIPE [1] is demonstrated.

The PIPE operates on the so called Elementary Operation Graphs (EOGs) that have some special attributes. A node of the EOG can have at most two inputs and one output. Other properties of EOG elements can be found in [1]. In order to feed the result of the decomposition to the tool PIPE, the resulted graph of segments must be transformed into an EOG. It is obvious from the examples in the previous section that the segments resulted after decomposition may have any number of inputs and outputs, thus the graph of segments need to be transformed to another graph that is consisted of nodes with at most two inputs and one output. This implies that a procedure should be formulated to reduce the number of outputs and inputs of the segment graph elements. This is done by introducing a special node type called Data Uniting Node (DUN), that will be considered as an extra EOG node with execution time 1, and that has two inputs and one output. The function of the DUN is the following: given two inputs a and b, with number of data bits b_a and b_b, the output c is the concatenation $c = [a, b]$ and has number of data bits $b_c = b_a + b_b$. This means that the two different data link are united for handling as one data link. Thus the data link reduction can be performed easily by inserting DUNs into the proper places of the segment graph. Let the resulted segment graph in Case 3 in Figure 19 be considered. Each node of the graph is a segment, and the first row of the node label is the segment name, the second row is the list of allocated SHSDG nodes, and the third row is the excecution time of the segment. This graph cannot be mapped directly into an EOG. The problems are:

1. Segment 5 has 2 outputs
2. Segment 3 has 3 inputs
3. Segment 3 has 2 outputs
4. Segment 1 has 3 inputs

The solution of the problem caused by more than two inputs can be easily solved by inserting extra DUN nodes into the graph. However, if there are multiple outputs, the DUN unit has to be inserted (and implemented) inside the segment. Thus, the listed problems are solved in Figure 20 as:

1. DUN5 is inserted into segment 5 to unite the two outputs of segment 5 into one output
2. DUN45 is inserted to unite the output of segment 4 and DUN5. Thus, segment 3 has two inputs, one of them is the output of DUN45, and the other is the output of segment 6.
3. DUN3 is inserted into segment 3 to unite the two outputs of segment 3.
4. DUN23 is inserted to unite the output of segment 2 and DUN3. Thus, segment 1 has two inputs, one is the output of DUN5, and one is the output of DUN23.

These formal transformations can always be performed on any segment graph resulted by the decomposition method outlined in this paper. Thus, a formal

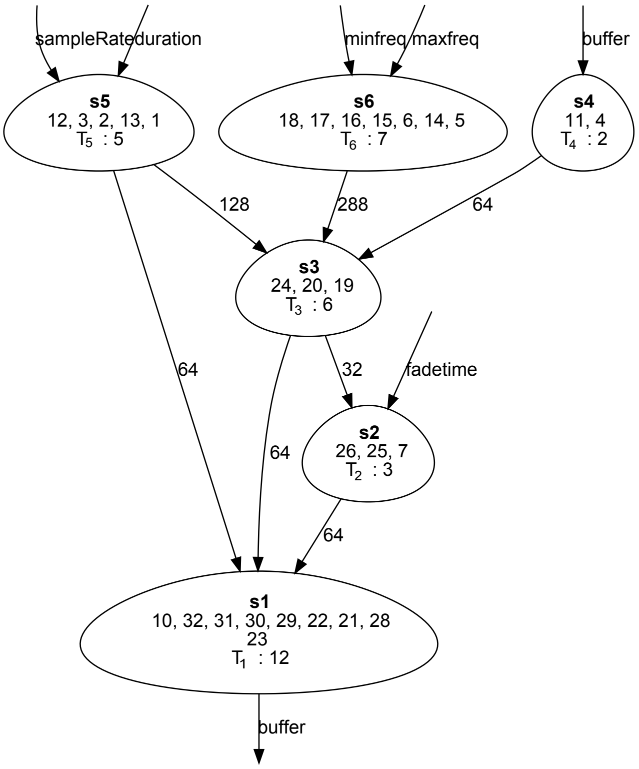


Figure 19: The segmentation result in Case 3 with $P = 6$

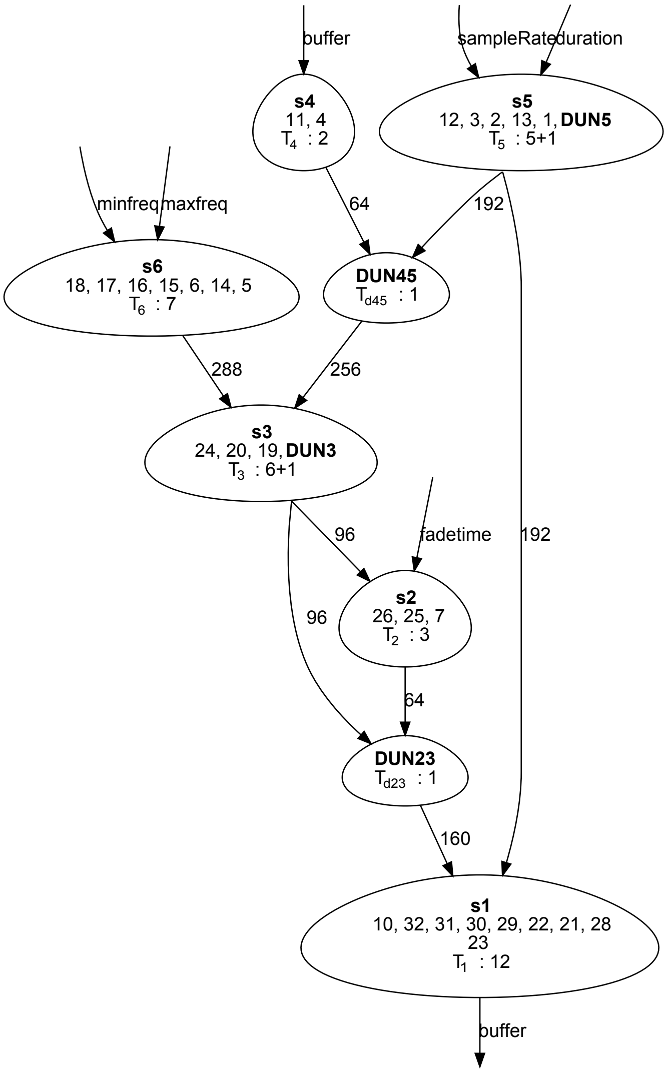


Figure 20: The transformed graph serving as input to the HLS tool PIPE

EOG can always be obtained and it can serve as the input of the HLS tool PIPE. For the further analysis, PIPE provides a simple measure of complexity as a function of the desired restart time R_d. The complexity (c_x) is calculated as follows:

$$c_x = \sum_{i=1}^{n} p_i t_i,$$

where p_i denotes the processing units in the structure generated by PIPE (segments, copies of manifold implemented segments, inserted buffers and synchronizing shift registers). DUN units are not considered, because their influence on complexity does not depend on R_d. The execution time of p_i is denoted by t_i and n is the number of processing units.

In Figure 21, this complexity function resulted by the EOG of Figure 20 is shown for each R_d between 1 and 19. It can be observed that three beneficial R_d values occur in this case: 4, 8 and 12. Namely, these R_d-s are the shortest at certain complexity domains. By starting the extended version of PIPE [1] with each of these beneficial R_d values, the task-dependent multiprocessing (TDMP) structures are generated communicating on time-shared arbitration-free bus systems as illustrated in Figures 22, 23 and 24, respectively. The horizontal lines symbolize the buses numbered with #. Cirles labeled by S or D represent the segments and DUN units, respectively. Rectangles labeled by bff or $sbff$ symbolize the inserted buffers and the synchronizing shift registers, respectively. In $sbff$ symbols, the number after the character _ indicates the number of bits of the synchronizing shift register. Circles enclosed by dotted rectangles are the simplified representations of manifold implemented segments. The number of copies [1] are depicted after x. The polygon symbols stand for inputs and outputs.

Nevertheless, these R_d values could be applicable only if the communication times are neglectable compared with the execution times of the processing units, or if the processing units can transfer input-output data directly via the arbitration-free buses. If it is not the case, then the communication times represent formally the execution times of fictive extra nodes between the segments. This may affect the applicability of R_d and the synchronisation. In this case, the shortest possible restart time (R_p) and the updated synchronisation should be recalculated by PIPE.

As a second example a basically serial program has been analyzed. The task-describing program is based on the block diagram of an image processing algorithm used for evaluating mammographic images [45]. Because of the rather big size of the program and the large amount of data, the SDG has been generated only from a reduced characteristic version of the program. The SHSDG of the 1^{st} hierarchy level is shown in Figure 25. Each node represents a separable function in the code. The upper numbers in the nodes are the identifiers and the lower numbers represent the executiton times as the necessary number of clock periods for execution. The decomposition has been done on SHSDG of the 2^{nd} hierarchy level, i.e. all separable nodes of Figure 25 have been separated. To save space, this large and complicated SHSDG is not depicted, only the result

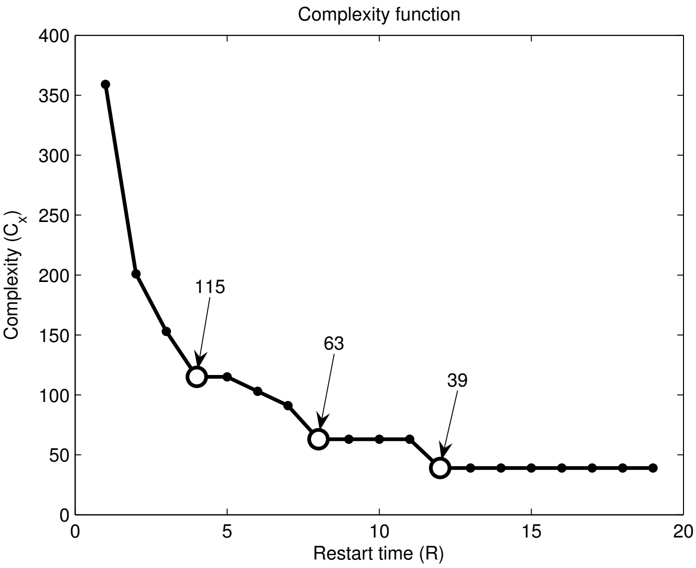


Figure 21: The complexity as a function of the desired restart time R_d for the EOG in Figure 20

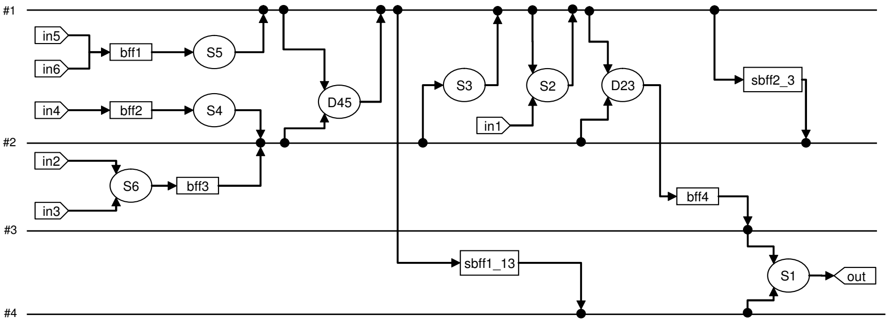


Figure 22: The TDMP structure obtained for the EOG in Figure 20, if $R_d = 12$

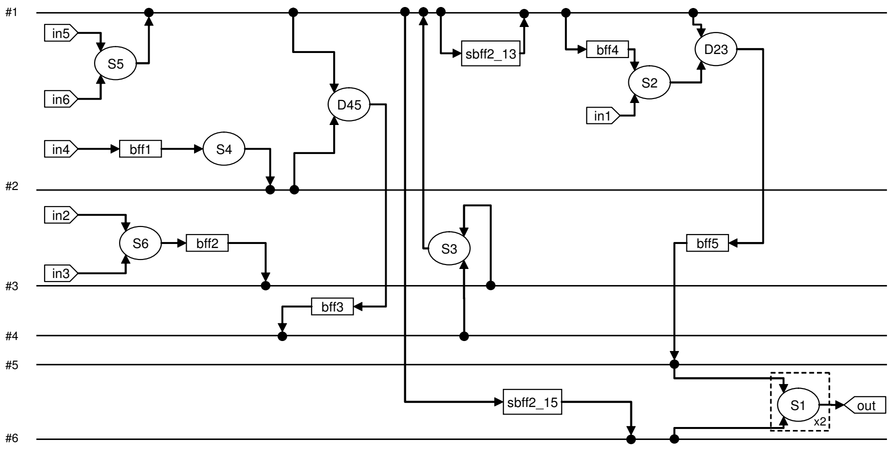


Figure 23: The TDMP structure obtained for the EOG in Figure 20, if $R_d = 8$

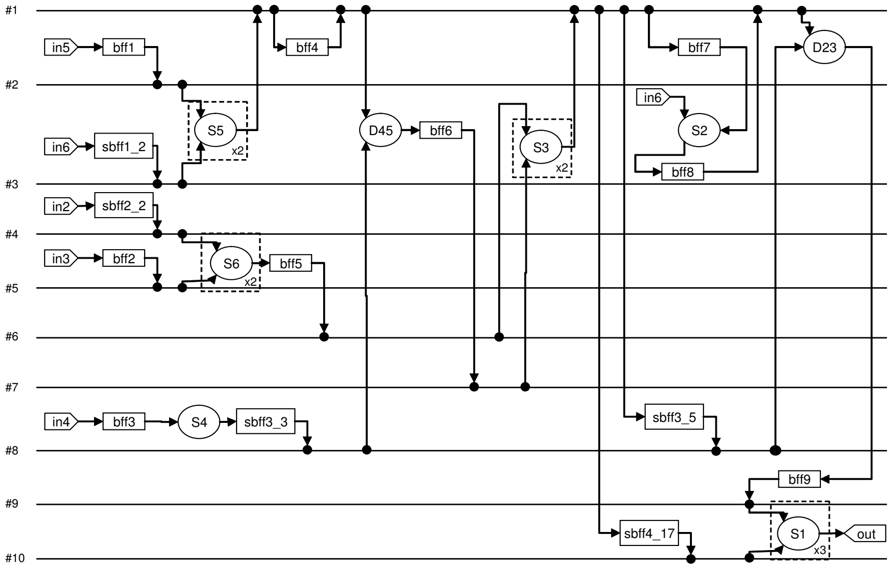


Figure 24: The TDMP structure obtained for the EOG in Figure 20, if $R_d = 4$

of the decomposition is illustrated in Figure 26. This result has been obtained for $P = 6$ in Case 4 ($k < P$ also allowed). For providing only the segmentation (not assigning the segments to processing units yet), the decomposition algorithm handles formally fictive processing units of identical type as shown in Cases 2 and 4. The segmentation resulted in Figure 26 has been obtained by assuming that the fictive processing units have a processing speed of $G_s = 1$, communication cost of $C_s = 2$, and capability as $Cap_s = 60$.

The decomposition resulted in 5 segments. The execution times of the segments are:

$$T_1 = 48 \quad T_2 = 44 \quad T_3 = 19 \quad T_4 = 47 \quad T_5 = 45$$

The execution times are distributed almost evenly, except for s_3. The adjacency half matrix is in Figure 27. The number of bits to be transferred is beneficially low at almost all cuts.

In Figure 28, the EOG is shown by inserting the necessary DUN units for PIPE.

The complexity function provided by PIPE is illustrated in Figure 29. In this case, $R_d = 10$ can be considered as a beneficial desired restart time. For this value, PIPE provided the arbitration-free bus system shown in Figure 30.

For the applicability of $R_d = 10$ and for calculating an R_p value, the same analysis of the communication properties is necessary as it was outlined in the first example.

The decomposition has been executed also with assignment to processing units in Case 2 for $P = 12$ ($k < P$ also allowed). Four types of processing units have been assumed: two software types and two hardware types. The available set of processing units has been assumed to consist of four units of each type. The following properties of the processing units have been assumed:

- The processing speed of software type 1 is $G_{s_1} = 1$
- The processing speed of software type 2 is $G_{s_2} = 2$
- The processing speed of hardware type 1 is $G_{h_1} = 8$
- The processing speed of hardware type 2 is $G_{h_2} = 10$
- The communication cost between software type 1 and software types 1 and 2 is $C_{s_1 \to s_1} = C_{s_1 \to s_2} = 10$
- The communication cost between software type 1 and hardware types 1 and 2 is $C_{s_1 \to h_1} = C_{s_1 \to h_2} = 8$
- The communication cost between software type 2 and software type 2 is $C_{s_2 \to s_2} = 8$
- The communication cost between software type 2 and hardware types 1 and 2 is $C_{s_2 \to h_1} = C_{s_2 \to h_2} = 6$

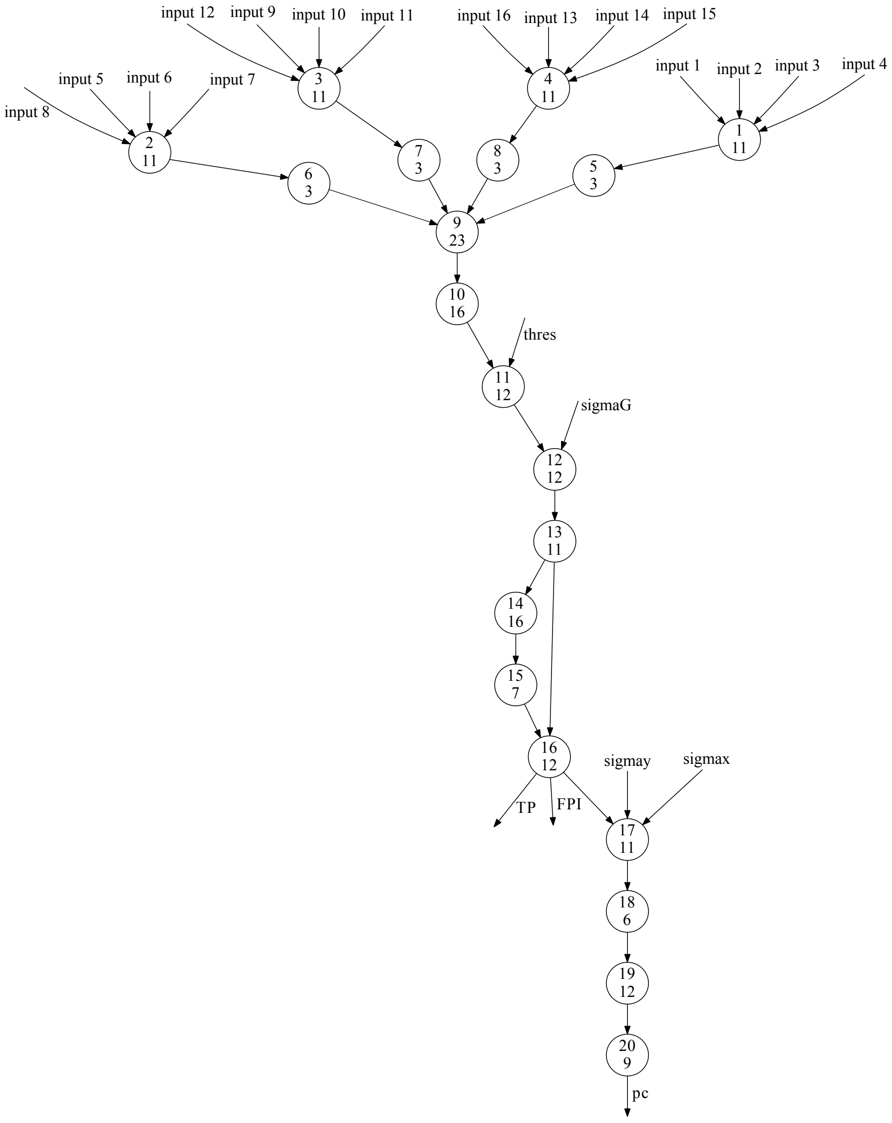


Figure 25: The SHSDG on the 1^{st} hierarchy level

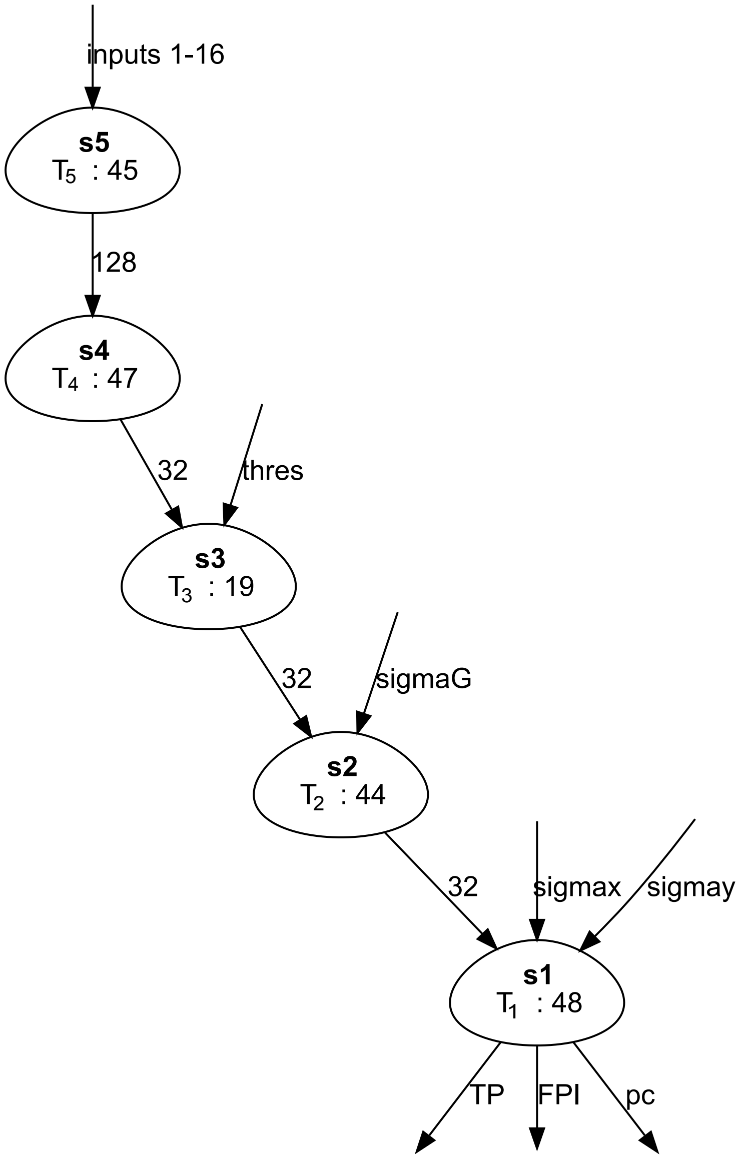


Figure 26: The result of the decomposition on the 2^{nd} hierarchy level in Case 4 with $P = 6$

$$\begin{array}{ccccc}
s_2 & 32 & & & \\
s_3 & 0 & 32 & & \\
s_4 & 0 & 0 & 32 & \\
s_5 & 0 & 0 & 0 & 128 \\
 & s_1 & s_2 & s_3 & s_4
\end{array}$$

Figure 27: The adjacency half matrix of the decomposition shown in Figure 26

- The communication cost between hardware types is $C_{h_1 \to h_1} = C_{h_1 \to h_2} = C_{h_2 \to h_2} = 2$
- The capability of each processing unit is $Cap_{s_1} = Cap_{s_2} = Cap_{h_1} = Cap_{h_2} = 40$

The decomposition algorithm has resulted in eight segments as shown in Figure 31. The label of the nodes consists of two rows. In the first row $H1$ stands for hardware type 1, $H2$ for hardware type 2, $S2$ for software type 2 processing units. The number in the brackets is the index of the segment implemented by the processing unit (i.e segments 1, 3 and 4 are assigned to software type 2 processing units, segments 5 and 6 are assigned to hardware type 1 processing units, and segments 2, 7 and 8 are assigned to hardware type 2 processing units). The second row of the labels refers to the relative execution time of the segment divided by the processing speed of the assigned processing unit.

The resulted execution times obtained for the segments by assigning them to processing units are:

$$T_1 = 8 \quad T_2 = 4 \quad T_3 = 13 \quad T_4 = 10 \quad T_5 = 1.375 \quad T_6 = 2 \quad T_7 = 3.4 \quad T_8 = 4$$

As it is mentioned in Section 1, such decomposed structures assigned to processing units can be considered also as TDMP structures. In most cases, the pipeline function is also applicable, but the shortest value of R_p is predetermined by the set of processing units and by the communication time between them. Namely, the assignment of the segments to the processing units generates already a fixed target system without proper flexibility for scheduling and allocation by applying a HLS tool. However, the predetermined shortest value of R_p can be calculated by a HLS tool. For example, the HLS tool PIPE would provide the value of R_p for the structure in Figure 31 by the following calculation, if the synchronizing was solved and the properties of the nodes were satisfying the conditions of the EOG [1]. If the communication time between the processing units is assumed not to be longer than 10 clock period, then the maximal busy time ($Qmax$) is represented by the node $S_2(3)$ in the graph [1]:

$$Q_{max} = \frac{26}{2} + 10 = 23.$$

Thus, the shortest $R_p = Q_{max} + 1 = 24$ [1].

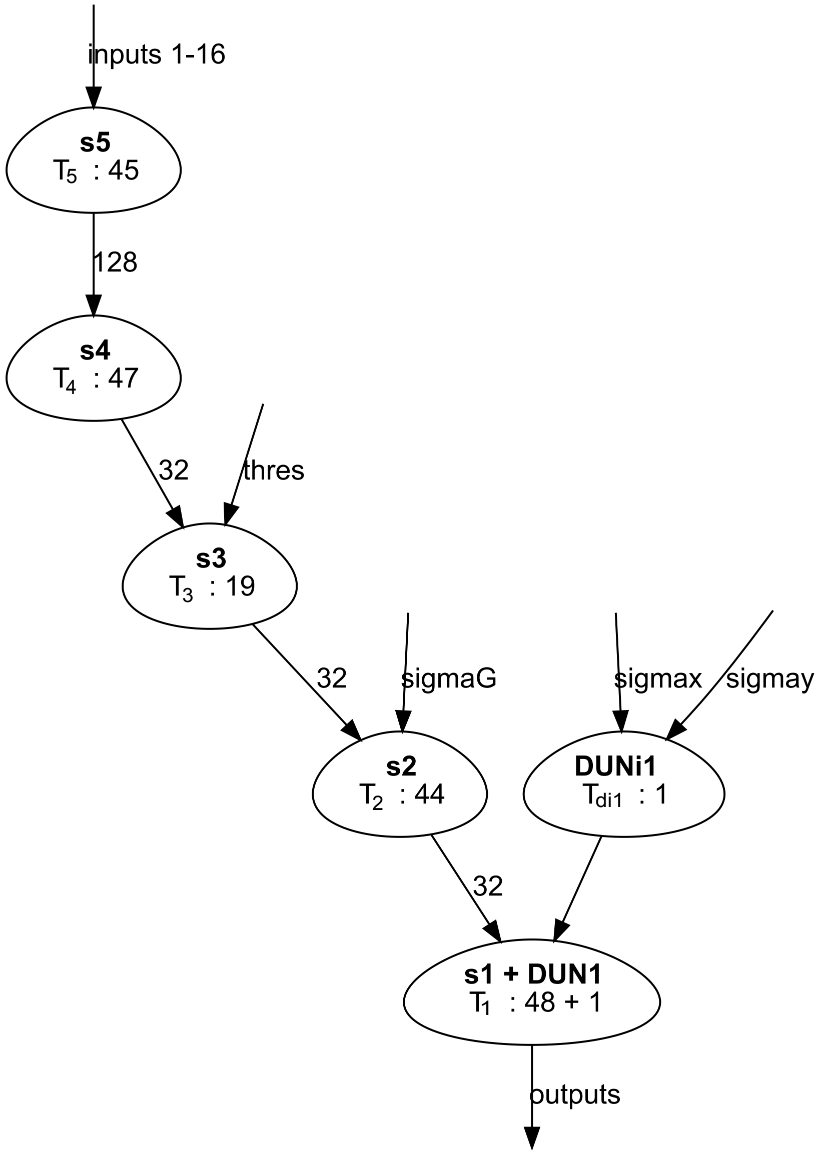


Figure 28: The EOG generated from the graph shown in Figure 26

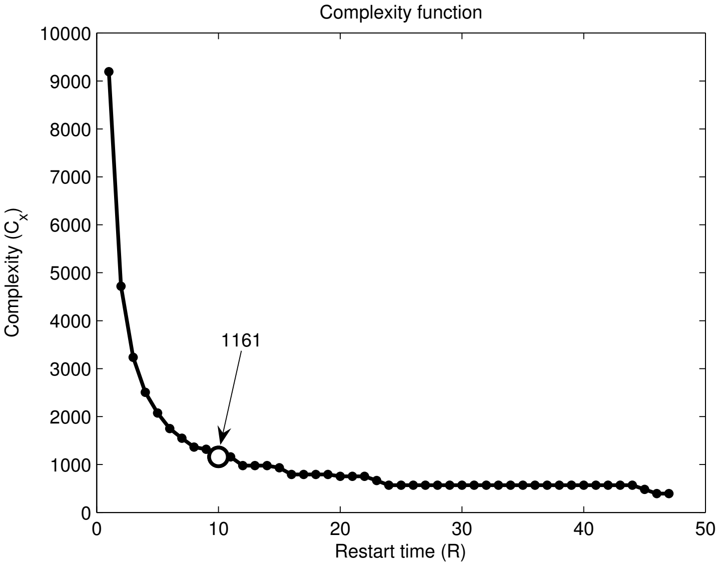


Figure 29: The complexity as a function of the desired restart time R_d for the EOG in Figure 28

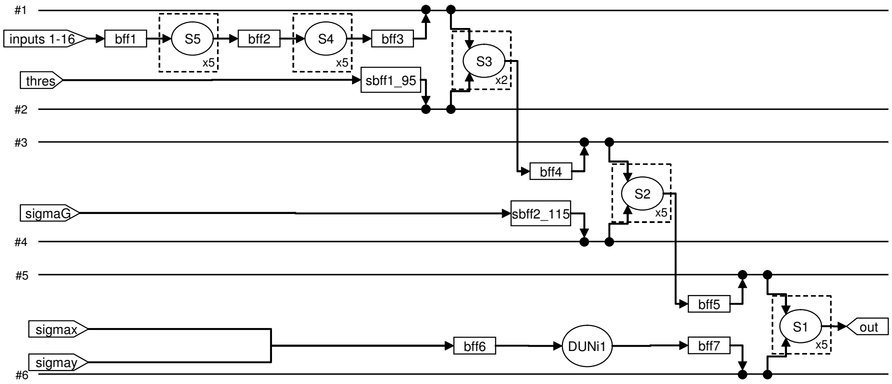


Figure 30: The TDMP structure obtained for the EOG in Figure 28, if $R_d = 10$

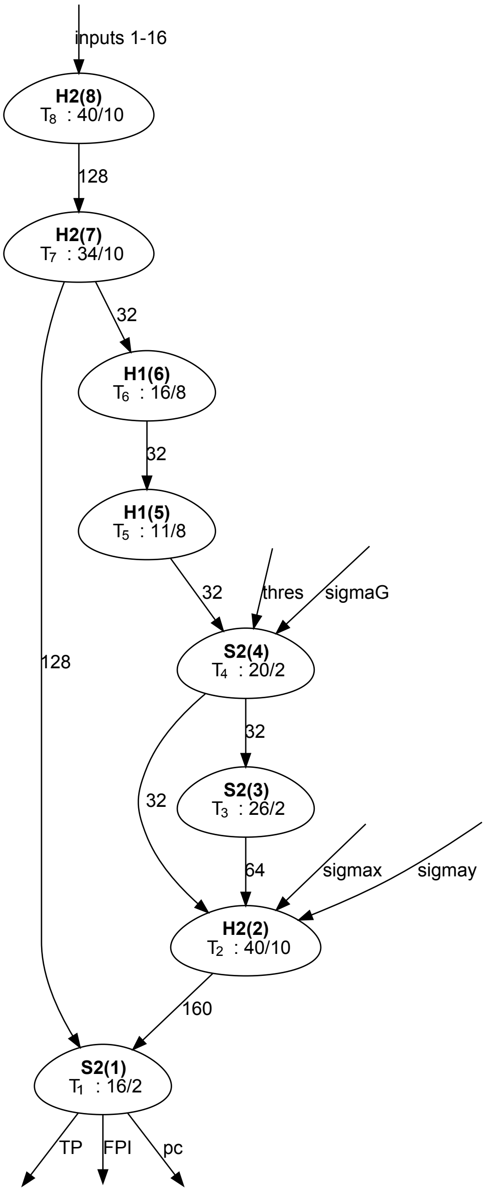


Figure 31: The result of the decomposition on the 2^{nd} hierarchy level in Case 2 with $P = 12$

$$\begin{array}{cccccccc}
s_2 & 160 & & & & & & \\
s_3 & 0 & 64 & & & & & \\
s_4 & 0 & 32 & 32 & & & & \\
s_5 & 0 & 0 & 0 & 32 & & & \\
s_6 & 0 & 0 & 0 & 0 & 32 & & \\
s_7 & 128 & 0 & 0 & 0 & 0 & 32 & \\
s_8 & 0 & 0 & 0 & 0 & 0 & 0 & 128 \\
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7
\end{array}$$

Figure 32: The adjacency half matrix of the decomposition shown in Figure 31

7 Conclusions and further research

The framework presented in this paper for synthetizing a specific (task-dependent) multiprocessing structure demonstrates that the pipeline function can be realized as a special parallel processing even if there is no efficiently exploitable parallelism in the task description. Due to its modular structure, the synthesis method presented in this paper offers an easy way to replace the decomposition algorithm and the HLS tool by other ones focusing on the actual properties and requirements of the target system. The problems solved as examples illustrate the framework character of the method by emphasizing the special consequences of the applied algorithms and tools. In the present stage, the method may be a starting point for further developement to adopt it to specific requirements of various practical applications. In this sense, some aims of further research are as follow:

- Testing and evaluating the efficiency of various algorithms by considering typical properties of target systems (e.g. mode of communication between the processing units, applying IPs or special-purpose processing units).
- Analysing the effect of various more sophisticated weight-assingments and of more accurate estimation of segment's execution times (regarding parallel or alternative nodes and loops) during the CM mapping and the decomposition algorithm.
- Developing a loop handling strategy for more realistic module definitions and execution time estimation by analysing the various loop types (e.g. cycles, recursions, single- and multirun executions, multirate structures etc.)
- Redesigning some existing multiprocessor or multicore systems by applying the method presented in this paper for comparison as benchmarks.
- Testing and evaluating the efficiency of the method, if the task-describing program is not a basically serial one, but it contains significant parallelism.
- Analysing the effect of forming the SHSDG on lower (or even mixed) hierarchy levels.

- Testing and evaluating the efficiency of applying a functional language for the task description.

Acknowledgements

The research work presented in this paper has been supported by the Hungarian Scientific Research Fund OTKA 72611 and by the "Research University Project" TAMOP IKT T5 P3.

References

[1] P. Arató, T. Visegrády, and I. Jankovits. *High-level Synthesis of Pipelined Datapaths*. John Wiley & Sons, 2001.

[2] Ahmed A. Jerraya and Wayne Wolf. *Multiprocessor Systems-on Chip*. Systems on Silicon. Morgan Kaufmann Publishers (Elsevier), 2005.

[3] Jiang Xu, Wayne Wolf, Joerg Henkel, and Srimat Chakradhar. A design methodology for application-specific networks-on-chip. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):263–280, May 2006.

[4] Damien Lyonnard, Sungjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the 38th annual Design Automation Conference*, pages 518–523, 2001.

[5] David W. Binkley and Keith Brian Gallagher. Program slicing. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 43, pages 1 – 50. Elsevier, 1996.

[6] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[7] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, March 2005.

[8] Omid Bushehrian. Automatic actor-based program partitioning. *Journal of Zhejiang University SCIENCE C*, 11(1):45–55, 2010.

[9] J. C. Huang. State constraints and pathwise decomposition of programs. *IEEE Transactions on Software Engineering*, 16(8):880–896, 1990.

[10] Daniel W. Watson, John K. Antonio, H. J. Siegel, and Mikhail J. Atallah. Static program decomposition among machines in an SIMD/SPMD heterogeneous environment with non-constant mode switching costs. In *Proceedings of the Heterogeneous Computing Workshop*, pages 58–65, 1994.

[11] Junwei Hou and Wayne Wolf. Process partitioning for distributed embedded systems. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 70–76, Washington, DC, USA, 1996. IEEE Computer Society.

[12] Peter Voigt Knudsen and Jan Madsen. Pace: A dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 85–92, Washington, DC, USA, 1996. IEEE Computer Society.

[13] Shunsuke Sasaki, Tasuku Nishihara, Daisuke Ando, and Masahiro Fujita. Hardware/software co-design and verification methodology from system level based on system dependence graph. *Journal of Universal Computer Science*, 13(13):1972–2001, 2007.

[14] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation Electronic Systems*, 10(1):136–156, January 2005.

[15] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38 – 43, April 2003.

[16] Péter Arató, Sándor Juhász, Zoltán Ádám Mann, and András Orbán. Hardware/software partitioning in embedded system design. In *Proceedings of the IEEE International Symposium on Intelligent Signal Processing*, pages 197–202, 2003.

[17] M.B. Abdelhalim, A.E. Salama, and S.E.D. Habib. Hardware software partitioning using particle swarm optimization technique. In *Proceedings of the 6th International Workshop on System-on-Chip for Real-Time Applications*, pages 189 –194, December 2006.

[18] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19:177–184, April 1984.

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[20] K. Tanabe, S. Sasaki, and M. Fujita. Program slicing for system level designs in specc. In *Proceedings of the International Conference on Advances in Computer Science and Technology*, pages 252–258, 2004.

[21] Henry Hoffmann, Anantl Agarwal, and Srinivas Devadas. Partitioning strategies: Spatiotemporal patterns of program decomposition. In *Proceedings of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems*. International Association of Science and Technology for Development, November 2009.

[22] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 59–70, New York, NY, USA, 2004. ACM.

[23] Stanislaw Deniziak. Cost-efficient synthesis of multiprocessor heterogeneous systems. *Control and Cybernetics*, 33(2):341–355, 2004.

[24] R. Subramanian and S. Pande. Efficient program partitioning based on compiler controlled communication. In *Proceedings of the Fourth International Workshop on High Level Parallel Programming Models and Supportive Environments (in conjunction with IPPS 1999)*. Springer-Verlag, April 1999.

[25] Simon A. Spacey, Wolfram Wiesemann, and Wayne Luk Daniel Kuhn. Robust software partitioning with multiple instantiation. INFORMS Journal on Computing, July 2011.

[26] Zoltán Ádám Mann, András Orbán, and Péter Arató. Finding optimal hardware/software partitions. *Formal Methods in System Design*, 31(3):241–263, December 2007.

[27] Péter Arató, Zoltán Ádám Mann, and András Orbán. Extending component-based design with hardware components. *Science of Computer Programming*, 56(1 - 2):23 – 39, 2005.

[28] Péter Arató, Zoltán Ádám Mann, and András Orbán. Time-constrained scheduling of large pipelined datapaths. *Journal of Systems Architecture*, 51(12):665–687, December 2005.

[29] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*. Red Hat, Inc., 2003.

[30] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[31] R. M. Stallman and GCC Developer Community. Gnu compiler collection internals. for gcc version 4.8.0., 2010.

[32] L. Séméria and G. De Micheli. Spc: synthesis of pointers in c application of pointer analysis to the behavioral synthesis from c. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers.*, pages 340 – 346, nov 1998.

[33] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in c with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems - System Level Design*, 9(6):743–756, December 2001.

[34] Jianwen Zhu and Silvian Calman. Context sensitive symbolic pointer analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):516–531, 2005.

[35] Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, and Soonhoi Ha. Conversion of reference c code to dataflow model h.264 encoder case study. In *Proceedings of the Asia and South Pacific Conference on Design Automation, 2006.*, pages 152–157, January 2006.

[36] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.

[37] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, SODA '92, pages 165–174, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[38] David R. Karger and Clifford Stein. An $o(n^2)$ algorithm for minimum cuts. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 757–765, New York, NY, USA, 1993. ACM.

[39] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, February 1992.

[40] Mouloud Koudil, Karima Benatchba, Said Gharout, and Nacer Hamani. Solving partitioning problem in codesign with ant colonies. In *Proceedings of the International Work-Conference on the Interplay Between Natural and Artificial Computation - IWINAC (2)'05*, pages 324–337, 2005.

[41] Madhura Purnaprajna, Marek Reformat, and Witold Pedrycz. Genetic algorithms for hardware-software partitioning and optimal resource allocation. *Journal of Systems Architecture*, 53(7):339–354, July 2007.

[42] Martin Palkovic. *Enhanced Applicability of Loop Transformations*. PhD thesis, T.U.Eindhoven, 2007.

[43] Harald Devos. *Loop Transformations for the Optimized Generation of Reconfigurable Hardware*. PhD thesis, Ghent University, 2008.

[44] Martin Griebl and Christian Lengauer. On the space-time mapping of while-loops. In *Parallel Processing Letters*, pages 677–688. Springer-Verlag, 1994.

[45] Yufeng Zheng. Breast cancer detection with gabor features from digital mammograms. *Algorithms*, 3:44–62, 2010.

2. Melléklet

Communication Time Estimation in High Level Synthesis

György Pilászy, György Rácz, Péter Arató
Department of Control Engineering and Information Technology
BME, H-1117, Magyar tudósok krt 2., Building I

Abstract

The high level synthesis (HLS) tools may result in a multiprocessing structure, where the time demand of the interchip data transfer (briefly the communication) between the processing units (hardware or software) is determined exactly only after the task-allocation. However, a realistic preliminary estimation of the communication time would help to shape the scheduling and the allocation procedures just for attempting to minimize the communication times in the final structure. Compared to the task-execution times of the processing units, especially significant communication times are required by the serial communication interfaces which are frequently used in microcontroller systems. This paper presents an estimation method by analysing four well-known serial communication interfaces (SPI, CAN, I^2C, UART).

Keywords:
communication time estimation, HLS, CAD, microcontroller, multiprocessing, embedded systems, serial communication interfaces

1. Introduction

In high-level design, the task-specification can be transformed into some kind of data flow graphs. Various HLS (High Level Synthesis) algorithms and tools are available for optimizing the schedule and allocation of the graph. [3, 7, 9]. The HLS framework presented in [7] for synthesizing a specific (task-dependent) multiprocessing structure demonstrates that how important is a realistic preliminary communication time estimation already in the decomposition phase. The HLS tools are rarely dealing with the communication between the nodes of the data flow graph; it is generally considered with zero-time execution. This solution is appropriate within an intra-FPGA (Field Programmable Gate Array) communication, but in case of more than one IP (Intellectual Property Unit or Intelligent Processor) it is not always applicable. In IPs containing microcontrollers or microprocessors, generally integrated communication peripherals are applied. Compared to the task-execution times of the processing units, especially significant communication times are required by the serial communication interfaces which are frequently used in microcontroller systems. If the task-specification allows, each communication channel (line) can be represented by an extra node with an estimated or determined execution time representing the communication time.

A realistic preliminary estimation of the communication time would help to influence the scheduling and the allocation procedures in order to reduce the communication complexity between the processing units in the final structure.

Further on, we present an estimation method by analyzing four well-known serial communication interfaces (SPI, CAN, I^2C, UART).

2. Calculating the communication time

Various types of serial channels are often used, because of the small number of pins and other resource constraints. In the following sections, we examine the frame structures of four frequently integrated serial communication interfaces.

2.1. Analysis of the Serial Peripheral Interface (SPI)

The SPI implements master-slave type synchronous communication. It uses three signals: clock (CLK), serial data input (SDI) and serial data output (SDO) for data transmission between the master and slave unit. A fourth signal called slave select ($\overline{SS}$) is used for the selection of the slave unit.

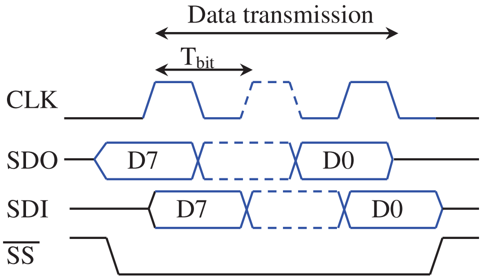


Figure 1. SPI data transmission

If more slaves are used, then each slave has a separate selection line, but in some cases, the slave units can be chained as well. The communication is bidirectional. In each clock period, one bit is transmitted. The integrated peripherals of the microcontrollers are usually configurable, the sampling points (clock polarity) and the clock phase can be set. In Figure 1, the SDO output is valid at the rising edge, while the SDI input is sampled at the falling edge of the CLK. The smallest unit of the transmitted data is eight bits. There are no restrictions on the maximum number of transmitted data.

Estimation of the SPI transmission time

The SPI is fit for the creation of a point-to-point link, it is established for the case when the timing of the data is exactly known (this is provided by the scheduling phase of an HLS tool). Then the data transmission time (T_k) of n bits can be calculated as follows:

$$T_k = b \cdot T_{bit} + T_{SS} \quad (1)$$

where b is the number of bits to be transmitted, the T_{bit} is the clock period and the T_{SS} is the sum of the selection/deselection (enable and disable) time. Since the enable and disable events are often scheduled by the clock signal, so we can use the following simplified estimation:

$$T_{SS} \cong T_{bit} \quad (2)$$

Based on (1) and (2) the SPI communication time can be estimated by the following formula:

$$T_k = (b+1) \cdot T_{bit} \quad (3)$$

In case of transmitting N number of bytes, the communication time is the following:

$$T_k = (N \cdot 8 + 1) \cdot T_{bit} \quad (4)$$

2.2. Analysis of the I²C interface

The Inter-IC bus was developed in 1992, its application is still widespread in the microcontrollers [8]. For the communication, the I²C bus uses a clock (SCL) and a data signal (SDA) line.

The bus can be a multi-master bus, and of course several slave units can be connected. The communication is framed with well-defined START and STOP conditions. The transmission of a 7 or 10-bits address follows the START condition. The address is followed by the read/write control bit and the actual data transfer. Figure 2 shows the structure of such a frame [4].

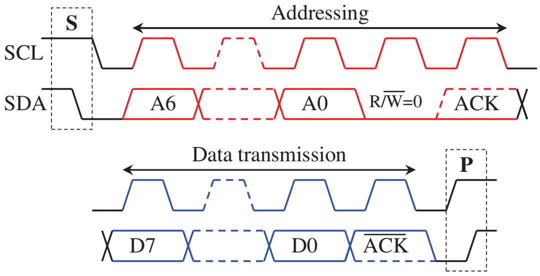


Figure 2. I2C data transmission with 7 bit addressing

Estimation of the I2C transmission time

Assuming that we fix the time of the START and STOP bits as one bit period each, we can state concerning the number of the bits to be transmitted the following:

	7bit address	10bit address
framing	2 bit	2 bit
address	9 bit	9 + 9 bit
databits	N·9	N·9
Total transmitted bits	11+ N·9	20+ N·9

Table 1.

N means the number of bytes in the message.

Typical I^{2}C bus speeds [4] are shown in Table 2a.

Speed mode	Max. bitrate [kbit/s]	Bit time (Tbit) [µs]
Normal	100	10
Fast	400	2.5
Fast +	1000	1
High speed	3400*	0.29
Ultra high speed	5000*	0.2

Table 2a. Typical I^{2}C speeds and bit times

Note:

* The "high speed" modes require special handling [4].

Interface	**T$_k$**
I2C-7 bits address	$(N \cdot 9 + 11) \cdot T_{bit}$
I2C-10 bits address	$(N \cdot 9 + 20) \cdot T_{bit}$

Table 2b. Estimated communication times of the I^{2}C bus

Note:

These data are valid only for write operation.

Table 2b shows the estimated communication times of the I^{2}C bus. Assuming 100 kHz clock frequency and normal 1-8 data bytes, the message transmission periods is shown in the Table 3.

Databytes (N)	7 bit address	10 bit address
1	200µs	290µs
2	290µs	380µs
3	380µs	470µs
4	470µs	560µs
5	560µs	650µs
6	650µs	740µs
7	740µs	830µs
8	830µs	920µs

Table 3.

2.3. Analysis of the UART interface

The UART (Universal Asynchronous Receiver Transmitter) interface makes possible the serial transmission of data bits with asynchronous framing. The transmitted data are linked to a start bit and, in general 1, 1.5 or 2 stop bits as well [10]. Optionally, a parity bit (even or odd) can be transmitted, too. The UART output circuits are usually connected to a line driver through the transmission medium. Depending on the driver circuit, the wiring and the higher level protocol, point-to-point or point-to-multipoint networks can be formed. The direction of the communication can be full duplex, half duplex or simplex.

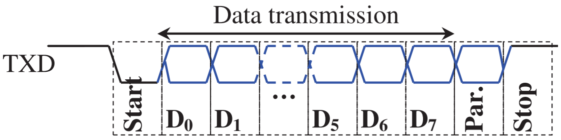


Figure 3. UART data transmission

Estimation of the UART transmission time

The communication time can not be separated from the number of additional bits, so the following formula can be used:

$$T_k = (b + 1 + p + s) \cdot T_{bit} \qquad (5)$$

Where b is the number of data bits to be forwarded (5..8),

p is number of parity bits to be forwarded (0 or 1),

s is number of stop bits to be forwarded (1, 1.5 or 2)

T$_{bit}$: forwarding time of one bit

If N bytes are transmitted, the communication time is as follows:

$$T_k = N \cdot (1 + 8 + p + s) \cdot T_{bit} \qquad (6)$$

If the least possible amount of additional bits is used (no parity bits, 1 stop bit), then the communication time is:

$$T_k = N \cdot 10 \cdot T_{bit} \qquad (7)$$

2.4. Analysis of the CAN interface

The CAN (Control Area Network) is a serial communication protocol which supports the real-time distributed control, at up to 1 Mbit/s speed. There are two types of CAN protocols: CAN V2.0A (standard format) which uses 11 bits, and the CAN V2.0B (extended format) which uses 11+18 = 29-bits as identifier in the transmission of messages. The devices, which can use the extended format, are able to communicate in the standard mode too. Thus, the different devices can work together, but only in the standard mode.

The structure of CAN messages are carefully detailed in [1] and [2].

Figure 4 shows the structure of the CAN data frame. At the three interfaces described above, the length of the message did not depend on the content. Because of the so called bitstuffing applied in the CAN system we can give only a worst case estimation for the communication time. We must estimate also the maximum bit number of the longest CAN message.

Calculation of the maximum CAN message size

The length of the CAN message can vary depending on the transmitted data and the stuffed bits. The method of the bitstuffing is as follows: if the transmitter detects five consecutive bits of identical value in the bit stream to be transmitted it automatically inserts a bit of opposite value in the actual bit stream before the transmission [2]. The frame segments, start of frame, arbitration field, control field, data field and CRC sequence are coded by the method of bit stuffing [2].

To estimate the maximum number of the inserted bits, ignore the fixed control bit values (eg.: IDE, RB0, RB1, etc). The frame starts with a zero SOF bit, so we start also the sample sequence identifier with 0 bits. After five 0 bits (including the SOF), a 1-bit value will be inserted. Afterwards, the test series continues with 4 additional 1-bits, then again a 0 bit, and so on. Figure 5 shows the first 16 bits of the test sequence.

Of the above test series can be seen that for the transmission of 16 bits 4 stuffed bits were needed. The "#" character marked the inserted bits.

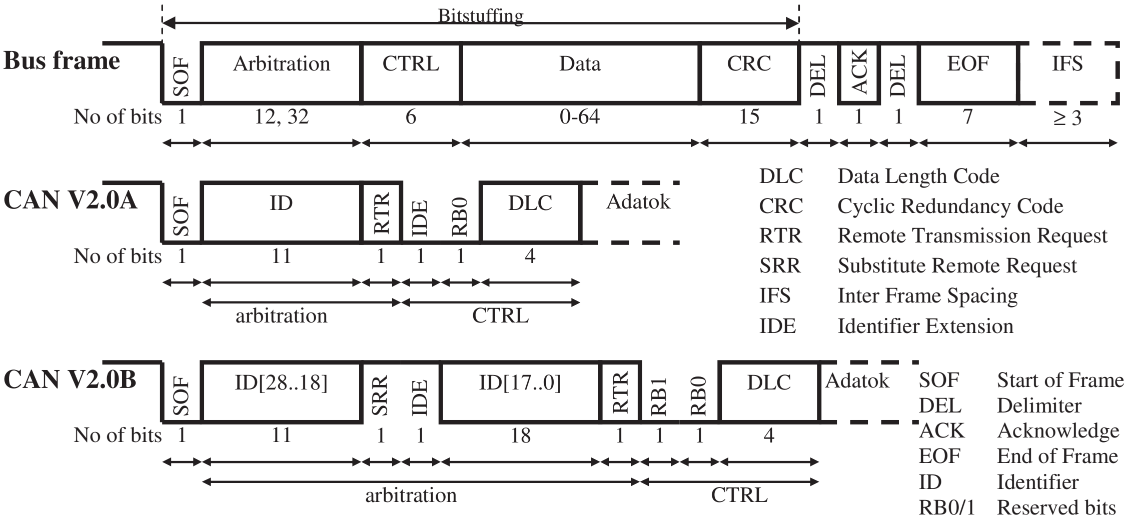


Figure 4. Structure of CAN dataframes [1], [2]

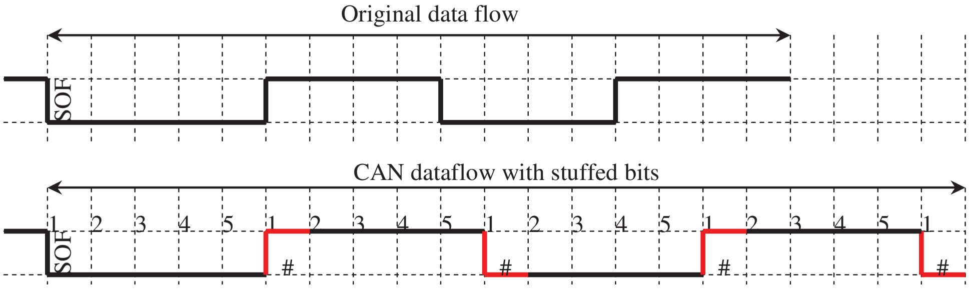


Figure 5. Bitsuffing

The maximum number of stuffed bits (s) can be estimated by dividing the bit number of the message bits by 4, and then the result is rounded up.

$$s = \left\lceil \frac{No\ of\ databits}{4} \right\rceil = \left\lceil \frac{FM + AM}{4} \right\rceil \qquad (8)$$

FM: bit number of fix fields
AM: bit number of variable data field
AM = 8·N, where N is the number of data bytes.
According to the above considerations, the total number of bits in the frame (db) is as follows:

$$db = FM + AM + s + E \qquad (9)$$

E: number of non stuffed bits after the CRC at the end of the frame $E \geq 13$.

Taking into account the bit number of the fields (in the fix header and the variable lengths of data and the empty space (13 bits) at the end of frame and the stuffed bits), the following maximum number of bits can be calculated:

	CAN2.0A	CAN2.0B
Fix fields	34 bits	54 bits
N [byte]	Total length [bit]	
1	53+13	78+13
2	63+13	88+13
3	73+13	98+13
4	83+13	108+13
5	93+13	118+13
6	103+13	128+13
7	113+13	138+13
8	123+13	148+13

Table 4. Maximal bit number of CAN messages

Transmission time estimation of the CAN interface

The maximum transmission time can be estimated, if we know the bit-time of the CAN interface. For this kind of estimation, the previously calculated message length is multiplied by the transmission time of one bit. Reordering the equation (9):

$$db = FM + \left\lceil \frac{FM}{4} \right\rceil + E + AM + \left\lceil \frac{AM}{4} \right\rceil \quad (10)$$

As AM field size can be 0…8 bytes, the inserted bits can be maximum 2 of each byte. When N is the number of data bytes, the total bit number of the data field is:

$$AM + \left\lceil \frac{AM}{4} \right\rceil = N \cdot (8 + 2) = N \cdot 10 \qquad (11)$$

For example: at 100kbit/s data transfer rate from Table 4 is shown in Table 5.

Message size [byte]	CAN2.0A	CAN2.0B
1	660μs	910μs
2	760μs	1010μs
3	860μs	1110μs
4	960μs	1210μs
5	1060μs	1310μs
6	1160μs	1410μs
7	1260μs	1510μs
8	1360μs	1610μs

Table 5. Maximal CAN transmission time

By using the closed forms, the results presented above are summed up in Table 6.

CAN	**Maximal number of bits**	**Maximal transmission time (T_k)**
2.0A	$56 + N \cdot 10$	$(56 + N \cdot 10) \cdot T_{bit}$
2.0B	$81 + N \cdot 10$	$(81 + N \cdot 10) \cdot T_{bit}$

Table 6. Length estimation of CAN messages

N denotes the number of data bytes (N: 0…8), and $T_{\mathbf{bit}}$ is the bit time

Notes:

1. If it is expected that the message will be repeated, then the above times must be multiplied by the number of repetitions. In error-free channels there is no need for repeating messages.
2. The model can be further refined if we consider the effect of fixed value bits.

3. Using the results in HLS design systems

The HLS tools usually represent the task to be solved by forming a data flow graph (DFG). The Figure 6 shows a simple example, how to represent the communication channel (e_3) between the e_1 and e_2 elementary operations. In Figure 6, t_1 and t_2 represent the duration of the operation e_1 and e_2.

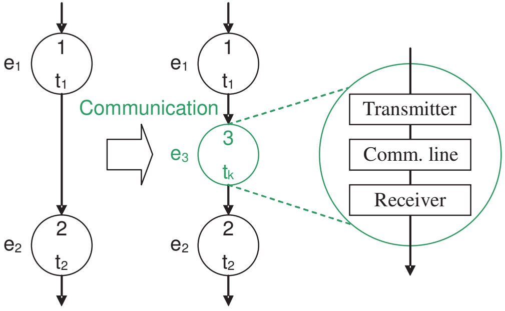


Figure 6. DFG representation

The execution time of the e_3 communication operation is t_k. However, the communication time (T_k) presented in the previous chapters depends on the bit rate, so we have to transform it into the timing system used by the particular HLS tool. The communication time is characterized by the $T_{\mathbf{bit}}$ and T_k. In the HLS tools the time is generally modelled by the number of the clock periods. The two time domains should be scaled as follows:

$$t_k = \frac{T_k}{T} \qquad (12)$$

where T denotes the length of the clock period and t_k is T_k expressed by the number of clock periods.

As an example for the practical usage of this method, we have chosen a sound source localisation structure from the reference [6]. Let a CAN communication network be assumed between the microcontrollers. In order to optimize the graph structure, we used the HLS tool PIPE [3]. The Elementary Operation Graph (EOG) of the task is shown in Figure 7.a. By applying the tool PIPE for scheduling and allocation, the allocated DFG is illustrated in Figure 7.b.

The execution times of the operations are assumed as shown in Table 7.

Processor	Operation	t_i	pieces
P1	FFT	23	4
P2	SC	1	5
Pk	**CAN**	**3**	**4**
P4	HT	25	1

Table 7. The input parameters

In the example, it has been assumed that a restart time (initialisation period) R=50 clock period ensures the desired pipeline throughput. To fulfil this requirement, the necessary numbers of operations are summarized in Table 8.

Processor	Operation	t_i	pieces
P1	FFT	23	2
P2	SC	1	3
Pk	**CAN**	**3**	**1**
P4	HT	25	1

Table 8. Resources after the allocation

The results show that the HLS tool PIPE provided only two FFT blocks and only one CAN interface at the specified restart time.

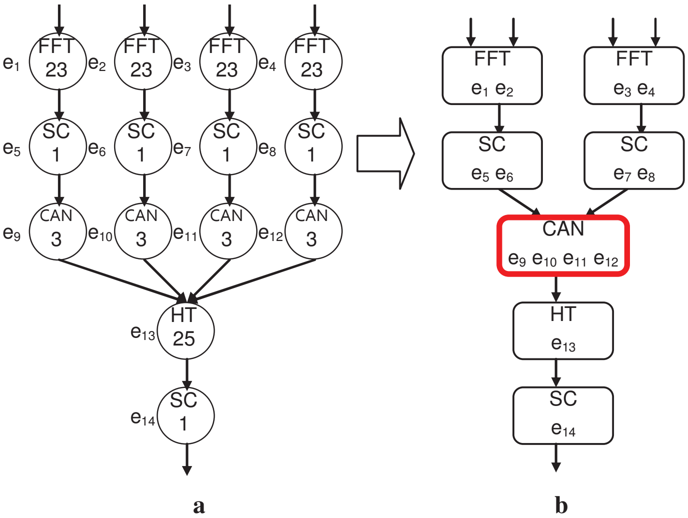


Figure 7. The EOG and the allocated DFG

4. Results

The method presented in this paper can be applied to estimate the communication time in four frequently serial communication interfaces. Since such interfaces are byte-organized in most cases, so it is advisable to indicate the data to be forwarded (N) in bytes.

Interface	T_k
SPI	$(N \cdot 8 + 1) \cdot T_{bit}$
I2C-7 bits address	$(N \cdot 9 + 11) \cdot T_{bit}$
I2C-10 bits address	$(N \cdot 9 + 20) \cdot T_{bit}$
UART (1 stop, 0 parity)	$N \cdot 10 \cdot T_{bit}$
UART (1 stop, 1 parity)	$N \cdot 11 \cdot T_{bit}$
CAN2.0A	$(N \cdot 10 + 56) \cdot T_{bit}$
CAN2.0B	$(N \cdot 10 + 81) \cdot T_{bit}$

Table 9. Communication time estimation

The results are summarized in Table 9. N is the number of bytes (N=1..8), T_{bit} is the bit time. The N $\leq$ 8 limit is needed, because the maximum size of CAN messages can be 8 bytes. Figure 8 shows the message transmission time of various communication interfaces at T_{bit} =10μs bit time. The presented method can be applied in other types of interfaces as well.

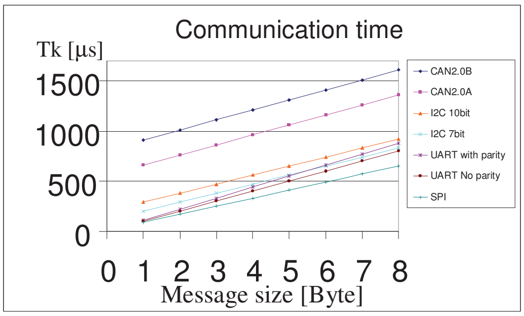


Figure 8. Comparison of the communication times at T_{bit}=10μs

5. Acknowledgement

The support of the Hungarian Scientific Fund (OTKA K72611) and New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002) IKT-P5-T3 are gratefully acknowledged. This work is also belonging to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project.

6. References

[1] CAN specification 2.0A (http://www.can-cia.org/index.php?id=441)

[2] CAN specification 2.0B (http://www.can-cia.org/index.php?id=441)

[3] Péter Arató, Tamás Visegrády, István Jankovits: High Level Synthesis of Pipelined Datapaths, John Wiley & Sons, New York, ISBN: 0 471495582 4, 2001

[4] I2C-bus specification and user manual Rev. 4 — 13 February 2012 (http://www.nxp.com/documents/user_manual/UM10204.pdf)

[5] Pilászy György, Móczár Géza, Remote control of modular microcontroller systems, Microcad 2001, In: Measurement and Automation. Miskolc, Hungary, 2001.03.01-2001.03.02. pp. 45-50.

[6] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos: Energy-Optimal Software Partitioning in Heterogeneous Multiprocessor Embedded Systems, DAC 2008, June 8–13, 2008, Anaheim, California, USA

[7] P. Arató, D. Drexler, G. Kocza, G. Suba: Synthesis of a Task-dependent Pipelined Multiprocessing Structure, submitted to the International Journal of Circuit theory & Applications (Wiley)

[8] The I2C Bus specification version 2.1, January 2000, http://www.nxp.com/documents/other/39340011.pdf

[9] Philippe Coussy, Daniel D. Gajski, Michael Meredith, Andres Takach, An Introduction to High-Level Synthesis, IEEE Design & Test of Computers, 2009

[10] 8251A Programmable communication interface, 1993, Intel Corporation, Document number: 205222-003

3. Melléklet

Hierarchical pipelining of nested loops

Dr. Péter Arató, Gergely Suba

November 21, 2012

Abstract

Pipelining of the nested loops is a very important way to increase the throughput of a system developed by a high-level synthesis tool. The most of the pipelining methods require that the input of the method should be a single loop. Therefore, nested loops have to be converted into a single loop by the method called loop flattening. Nevertheless, in this way the sequential loops can not be pipelined in separate pipeline stages. This constraint limits the throughput. In this paper, we present a novel pipeline scheduling method of nested loops for implementing the sequential loops in separate pipeline stages. There is another advantage of the method: the desired restart time of the whole system can be given as an input parameter of the method. A novel multi-rate dataflow graph is introduced for modelling the nested loops in an easy and abstract way.

1 Introduction

High-level synthesis is based on many optimization methods in order to ensure a desired performance in speed, area and cost. The loops are essential parts of the algorithms to be implemented by a high-level synthesis tool. Therefore the proper loop handling is unavoidable in increasing the throughput in a pipeline system. In achieving a given pipeline throughput (initiation interval or restart time[1]), even the loops with constant trip count may set limit if no efforts have been made for decreasing the latency or restart time of the loop.

In this paper a novel method is presented for increasing the pipeline throughtput of nested loops with constant trip count. By this method, the pipeline scheduling is performed on more than one level of the loop hierarchy simultaniusly in contrary with the usual solutions. Compared with the previous work, this simultaneous scheduling has advantage, if the loop hierarchy contains also sequential loops. The method can map the sequential loops into successive pipeline stages[2], which may increase the throughput of a system significatly.

In contrary to the methods applied by the most HLS tools, the desired restart time (initiation interval) can be given as an input parameter for the method presented in this paper.

The method can be used if every loop in the loop nest has constant trip count (number of iterations). Otherwise a transformation is needed to make the trip counts constant. For example, if an upper bound can be defined or estimated for the trip count of a loop, this bound can be used as constant trip count. In this case some control solution is maded to ensure that the loop body should be executed in the same number of times as the original trip count. Although this transformation increases the latency, and so it can cause some performance loss, in many cases the benefits of pipelining sequential loops is greater than the drawback of the

The support of the Hungarian Scientific Fund (OTKA K72611) and New Hungary Development Plan (Project ID: TMOP-4.2.1/B-09/1/KMR-2010-0002) IKT-P5-T3 are gratefully acknowledged. This work is also belonging to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project.

[1] It can be called also initiation interval, which is the reciprocal of the pipeline throughput. These values are interpreted by the number of clock periods.

[2] Let P_j^k denote the data processed by the operation e_j at the k-th initialisation (restart). S_i is a disjoint subset of operations (nodes) called pipeline stage, if $P_s^k = P_r^k$:$\forall k$ and $\forall e_s, e_r \subset S_i$. Further the restart time (R) and Initiation Interval (II) will be used as synonyms.

performance loss.

It is beneficial to perform the pipeline scheduling method on a dataflow representation, because these are formal and abstract models of a system. The operations inside a loop are executed more times than the program itself, therefore a so-called multi-rate dataflow model [13, 9, 3] can be applied to represent nested loops. The well known multi-rate dataflow graphs (e.g. SDF, discussed later) can not represent the nested loops in hierarchical way. Therefore, a novel dataflow graph based on existing single-rate dataflow models is introduced.

The paper is organized as follows. In Section 2 the previous works in the area of nested loops pipelining and the related dataflow models are reviewed. In Section 3 a novel dataflow graph, the so-called MR-HSDFG is introduced. In Section 4 the latency and restart time calculation is presented in the MR-HSDFG model. Section 5 discusses the optimization of these two parameters. In Section 6 experimental results are presented. The conclusion is summarized in Section 7.

2 Previous Work

In this section the most relevant methods are evaluated regarding the pipeline scheduling of nested loops. Further on, the dataflow graph representations are compared, which are suitable for modeling nested loops.

One of the approaches is the hierarchical reduction [11] method. In this way the program (represented in dataflow graph) is scheduled hierarchically, starting with the innermost loop. After this scheduling, the whole loop is substituted by a single operation. The same scheduling method will be executed for an outer loop if it does not contain inner loops, only operations (every loop has already been substituted by operations). At the end of these procedure, the entire program is reduced to a single operation. The paper [11] contains only the main concept of the reduction, and it doesn't discuss the precise algorithm.

An other hierarchical approach is the hierarchical pipelining [2]. In this method four level of the hierarcy is defined: system, behaviour, loop and operation. The description of the levels is not unified; each level has its own description graph (CFG, CDFG and DFG), therefore the handling of the levels of the hierarchy is different. The given time constraint determines the length of the stages. The stages are filled by the nodes one after the other. A node is assigned to the same stage as the predecessor node, if the latency of the nodes doesn't exceeds the length of the stage. This method is the nearest approach to the basic objectives of the method presented in this paper.

Another approach to perform pipelining is to flatten [5] the loop nest first, then the resulted single, non-nested loop can be pipelined using the single loop pipelining methods [10, 11]. The main idea of the loop flattening is to emulate the execution of the original loop nest by a single (called flattened) loop, where the trip count is equal to the total sum of inner loop trip counts. The method calculates that which iterations of the original loops should be executed for each iterations of the flattened loop. Several commercial HLS tools apply this approach, e.g. Calypto Catapult C [4] or Xilinx Vivado.

The advantage of the loop flattening is that it can handle also some types of loops having nonconstant trip counts. The disadvantage of the loop flattening regarding the pipelining is that the original inner loops desappear and so multilevel pipeline scheduling cannot be applied (e.g. the sequential loops can not be mapped to successive pipeline stages). Therefore the throughput of the whole system may be decreased.

Another way is use a self-timed ring representation to perform the pipeline scheduling of nested loops [6]. This method is dedicated to asynchronous pipeline mode where a handshake control is assumed in each stage. This solution makes the run-time pipeline loop scheduling easier, but the necessary control overhead is significant. The self-timed ring method can be applied only for asynchronous systems, which is out of the scope in this paper.

The method presented in this paper is based on the hierarchical reduction, i.e. in contrast with the flattening way is made level by level in successively in the loop hierarchy.

In handling the nested loops the dataflow model apply for the representation is crucial. The main types of such models can be classified as follows.

Hereinafter the most commonly used dataflow models are reviewed. The models are characterized how it can be used to represent nested loops.

For digital signal processing, the application of

Synchronous Data Flow (SDF) [12] is typical. In SDF the nodes represent operations (called actors), and the edges represent communication channels realized by FIFO queues. The FIFO-s connect to the ports of the actors. For each port of an actor the number of produced or consumed tokens are defined. An input port consumes tokens from the predecessor FIFO, and an output port produces tokens to the successor FIFO. The number of produced and consumed tokens of an actor can differ, becaouse the frequency of the fires (executions) of the actors may vary. Therefore the SDF can be considered as a so-called multi-rate [13] dataflow graph.

The Homogeneous SDF (HSDF) is a simplified variant of the SDF, where the number of produced and consumed tokens of each actor is always 1 Therefore the HSDF can be considered as a so-called single-rate [13] DFG.

As the HSDF is a very simplified model, it is easier to analyze than the generic SDF model. Besides the SDF model has disadvantage that the most analyzing and optimizing algorithms have to start with transforming the SDF to a HSDF. These transformation increases the number of nodes significantly. [12] Therefore, the SDF representation is practically not applicable in case of great number of operations.

An approach resembling the HSDF is the so called Elementary Operation Graph (EOG) [1]. This model is also single-rate, but it contains also timing parameters in constrast with the HSDF. The nodes are considered as elementary (not separable) operations and for each operation the duration (execution) time.

For the elementary operations of the EOG, the following assumptions are made:

- operation v_i is started only after having finished $\forall v_j \in V$, where v_j is the direct predecessor operation of v_i and V is the set of the operations
- operation v_i requires all its input data during the whole duration time of the v_i
- operation v_i may change its output during the whole duration time
- after the output of v_i shows, holds its actual output stable until its next start

The EOG supports the description and developing of the pipeline scheduling algorithms. For calculating and optimizing the restart time there are algorithms in [1], that will be reused in this paper. These algorithms handle the loops as single elementary operations, without defining the inner behavior of them. The aim of this paper is to represent and handle each level of the loop hierarchy separately.

3 MR-HSDFG - a novel multi-rate dataflow graph

In this section an extended dataflow model, the MR-HSDFG is introduced, which is based on the EOG model. The main purpose of the model is to represent nested loops in an abstract and formal way that can be used effectively to perform pipeline scheduling methods. The reviewed dataflow models in the previous section do not comply with these requirements. The HSDF and the EOG are single-rate DFG, therefore they can not represents nested loops in worthwhile way. (in a single-rate DFG every operation executes in the same rate, but in case of a loop the operations inside the loop body need to run more times than the loop itself) Although the SDF can represent nested loops, the analyzing and optimizing methods convert the model to HSDF first, hereby increasing the number of the nodes significantly.

The novel multi-rate homogeneous synchronous dataflow graph (MR-HSDFG) is the improvement of the HSDFG and the EOG. The MR-HSDFG is a finite and directed graph. Each node represents an operation, which has zero[3] or more input and zero or one output (except the case when both are zero), and each edge is a dataflow channel, which transports data from an operation to an other. Each operation has a duration and a restart time parameter.

The model has five types of operations:

- simple elementary operation: an operation, which is considered as atomic (indivisible) - in EOG this is the only operation type

[3] It's a different between MR-HSDFG and EOG, because in EOG every operation has at least one input and one output.

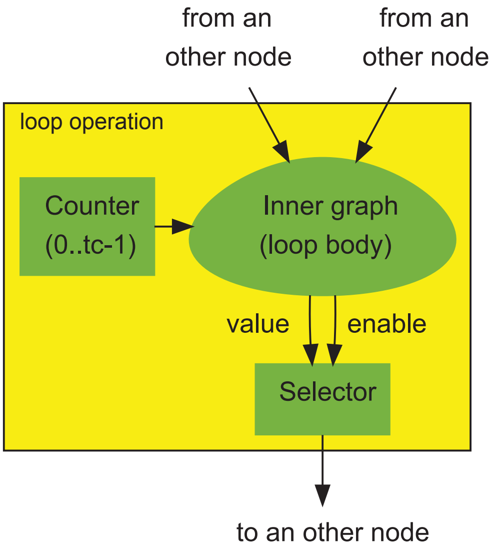


Figure 1: General construction of the loop operation (in this figure the loop operation has two predecessor nodes)

- loop operation: an operation that represents a loop. The behavior of the loop body is defined by an inner dataflow graph (discussed later), which is also a MR-HSDFG, therefore it can also contains loop operations.
- constant operation: an operation, which produces a constant value in runtime (it has no input dataflow channel)
- input operation: represents an input of the dataflow graph (it has no input dataflow channel)
- output operation: represents an output of the dataflow graph (it has no output dataflow channel)

The duration time of the constant, input and output operations are 0 by definition.

The loop operations are essential to ensure the model can handle the nested loops, because it contains the control of the loop iterations.

The general construction of the loop operation is shown in figure 1. A loop operation consists of special fixed operations (Counter and Selector) and a task dependant subgraph, called inner graph, which represents the loop body. This inner graph is illustrated by a single node in figure 1. The Counter can be considered as an input operation of the inner graph, which counts the number of times the loop body has to run in the task description. The Selector is a special downsampling operation, which has a value and an enable input. When the enable input is on, the value will be buffered to the output of the Selector node, and will be held until the next enable sign. The Selector can be considered as the output operation of the inner graph. The predecessor operations (Pred_1 and Pred_2) need to hold stable output state during the loop operation, therefore these predecessor operations can be considered as constant operations from the perspective of the inner graph.

The loop operation can be formally defined as a pair: $C = \langle G, tc \rangle$, where:

- G is the inner graph (MR-HSDFG), which represents the loop body
- $tc \in \mathbb{N}$ is the trip count of the loop, where $tc > 1$ (otherwise there wouldn't be a loop). While the loop operation performs once, the G should be restarted tc times, that is tc data is sent to the input of the G.

The MR-HSDFG fits to the nested loops written in a programming language. The top-level function can be described by an MR-HSDFG. The loops inside the function can be mapped to loop operations in MR-HSDFG. The other parts of the function body can be represented as simple elementary and constant operations, because these parts run just once in each running of the function. The function parameters and the returning value can be represented as input and output operations.

The tc is the trip count of the loop, which is used by the Counter operation as the upper bound of the counting. If the trip count is data dependant, a designer decision is needed to fix an upper bound for this value, which will be used as constant trip count. Among others the commercial softwares (e.g. Catapult C) also use manually given upper bound of a loop, if it has data dependant trip count.

The MR-HSDFG can be illustrated by an example. Let's consider the following expression:

$$f(a,n) = \prod_{i=1}^{n}(a+i), n \in [1, n_{max}] \qquad (1)$$

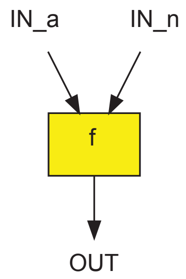


Figure 2: EOG representation of the task

The task is to calculate the value of the function f in case of a given a and n inputs repeatedly. Inside the f function there is a loop that represents the $\prod_{i=1}^{n}$ operation.

In EOG the f function can be only represented as an elementary operation, where the inner behavior is hidden, therefore pipeline scheduling can not be performed for the inner behavior in this model. This representation is illustrated in figure 2. In MR-HSDFG the f function is represented as a loop operation, where the inner graph represents the behavior of the loop body, which can be seen in figure 3. The inner graph (without its environment) can be represented as MR-HSDFG, as it is shown in figure 4. As this inner graph doesn't contain loop operation, it can be considered as a regular EOG, therefore the analyzing and optimizing algorithms of the EOG can be performed on it. The following sections will introduce these extended algorithms.

4 Calculation of the minimal restart time in MR-HSDFG

In this section we introduce an algorithm, which calculates the minimal restart time of the system in three cases: if scheduling is performed in each level of the loop hierarchy 1) sequentially, 2) using pipeline mode without any other optimization (operation replication [1], loop unrolling [7, 8], etc.), 3) using pipeline mode with operation replication.

If the MR-HSDFG does not contain loop operation, the model will be similar to the EOG. The calculation of the latency and the restart time in EOG is written in [1], which will be reviewed in

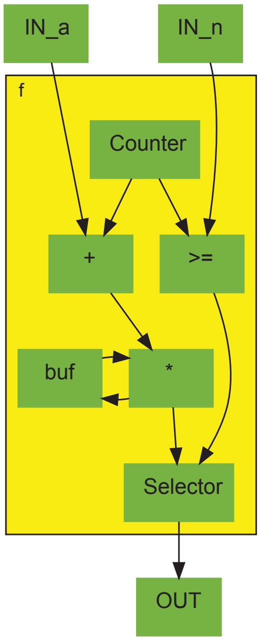


Figure 3: MR-HSDFG representation of the task (the $+$, $>=$, $*$ and buf operations represent the loop body, called inner graph)

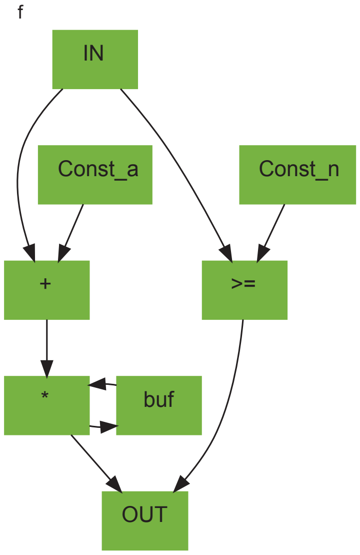


Figure 4: MR-HSDFG representation of the inner graph

the following, then they will be extended in case of loop operations in order to calculate the latency and restart time in MR-HSDFG.

The dataflow graph needs to be acyclic, therefore the operations inside each each recursive loop[4] are to be substituted by a combined operation first.

The latency of the G dataflow graph is $L(G)$, which is the summarize of the duration time of the longest path in G. The calculation of the restart time of a G dataflow graph in the cases described above are given in the following expressions [1]:

- **Sequential scheduling:** the system is restarted when the previous input data is reached the output. The throughput is low, but this scheduling allows area decreasing in the course of the allocation.

  $$R_{seq}(G) = L(G) \tag{2}$$

- **Pipeline scheduling without operation replication:** the graph is unchanged (same as in the case of sequential scheduling), but the system is restarted more often. The restart period can be less than in case of sequential scheduling. This scheduling allows less area decreasing in the course of the allocation compared with the sequential scheduling.

  In the following this minimal restart time is called simple pipelined restart time, and denoted by R_{sp}.

  $$\begin{aligned} &R_{sp}(G) = max(q_{v_i}) \\ &\forall v_i \in V \end{aligned} \tag{3}$$

- **Pipeline scheduling with operation replication:** the minimum restart time that can be reached with operation replicating. In most cases this restart time is 1, but the value can be higher, if some operation is not allowed to be replicated.

  $$\begin{aligned} &R_{min}(G) = max(q'_{v_i}) \\ &q'_{v_i} := \begin{cases} 1 & \text{if } \mathrm{v_i} \text{ can be replicated} \\ q_{v_i} & \text{if } \mathrm{v_i} \text{ can not be replicated} \end{cases} \\ &\forall v_i \in V \\ &L_{min}(G) = \sum_{w \in W} t'_w \\ &t'_w := \begin{cases} t_w + 1 & \text{if w can be replicated} \\ & \text{and } \mathrm{q_{v_i}} > \mathrm{R} \\ t_w & \text{otherwise} \end{cases} \\ &W = \{v_{j_1}, v_{j_1} \ldots v_{j_n}\} \text{ longest path operations} \end{aligned} \tag{4}$$

where V is the set of the operations, E is the set of the dataflow channels (the edges of the graph), and $t_{v_i} \in \mathbb{N}$ is the duration time (number of clock periods required for execution) of the v_i operation in terms of clock periods. For the various restart times the following is true:

$$R_{seq}(G) >= R_{sp}(G) >= R_{min}(G) \tag{5}$$

For the expression (2), (3) and (4), the duration time of each operation is needed. These duration times are given by parameters generally. The only exception is the loop operation, where calculation is needed to get the duration time of the operation. The inner behavior of the v_i loop operation can be defined by the $C_i = (G_i, tc_i)$ pair, and the duration time of the loop operation (which is equal to the busy time in case of loop operation) can be calculated in terms of the elements of the pair:

$$q_{v_i} = t_{v_i} = (tc_i - 1) * R(G_i) + L(G_i) \tag{6}$$

where $R(G_i)$ is the restart time of the inner graph inside the v_i loop operation, and $L(G_i)$ is the latency of that. The correctness of the expression: the inner graph of the loop operation is also MR-HSDFG, and its restart time $R(G_i)$ can be also defined by the expression (2), (3) and (4). While the loop operation is executed once, the inner graph is restarted tc times, therefore between the first and last restart $(tc_i - 1) * R(G_i)$ time elapsed. After the last restart, the loop operation is busy for $L(G_i)$ time (as the time needed to change the output value after the last input data is sent), therefore this value should be added to the expression, to get the overall duration time of the loop operation.

The calculation method introduced in this section deals with the nested loops in bottom-up way,

[4] Cycle in the graph, that is directed path goes from a node to the same node.

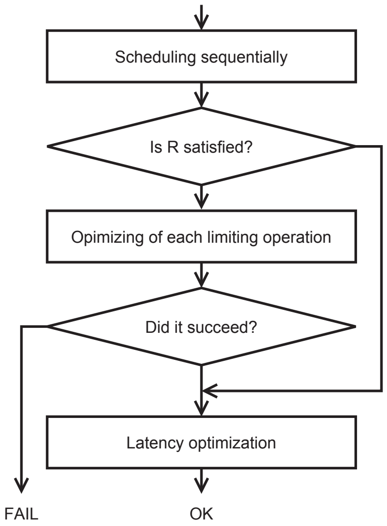


Figure 5: Optimizing of each operation

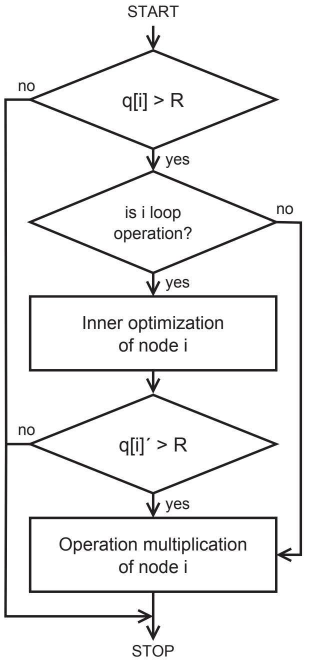

START
q[i] > R
no
yes
is i loop operation?
no
yes
Inner optimization of node i
no
q[i]´ > R
yes
Operation multiplication of node i
STOP

Figure 6: Optimizing of each operation

beginnings with the innermost loop. (the algorithm can be implemented by a recursion)

5 Achieve desired restart time in MR-HSDFG

An optimization method [1] are known for the EOG, which decrease the minimal restart time to a given value. In the first step the method inserts buffers to achieve the goal. If the insertion doesn not satisfy, the operation replication is perform to achieve the goal. In MR-HSDFG the modified version of this optimization method can be used, which will be introduced in this section.

The RESTART algorithm of the MR-HSDFG starts dealing with the top level of the loop hierarchy.

The overview of the achieving algorithm is introduced in figure 5. The first step is to calculate the duration times of each loop operations using sequential scheduling (see section 5.1).

If the resulted R restart time satisfies the desired value, the whole algorithm can be stopped and sequential scheduling is performed in the source graph. If it is not the case, optimizing of each limiting operation (see section 5.2) is needed. This step may be failed, in this case the whole algorithm is failed. Finally the optional latency optimization can be performed.

The method introduced in this section deals with the nested loops in top-down way, and the algorithm can be implemented by a recursion.

5.1 Scheduling sequentially

The first step is to calculate the duration times of each loop operations, in order to get the restart time of the system in case of sequential scheduling. (sequential scheduling can result the least resource using and the largest amount of allocating) This calculation can be performed as described in Section 4, using the expression (6).

5.2 Opimizing of each limiting operation

In case of R is not satisfied after the sequential scheduling, each of the operations, which limits the restart time, need to be optimized one by one.

The overview of the operation optimizing algorithm can be seen in figure 6. The algorithm deals with one operation, therefore it should be run for each operation of the graph, where the running order is irrelevant.

The first condition detects wheather the given node limits the desired restart time. If not, there is no need to do anything with that operation. In case of an operation that limits the desired restart time two steps can be performed in order to satisfy the requirement, the cheaper inner optimization (for only loop operations) and the more expensive operation replication (for any operations). If the operation is a loop operation, first the inner optimization (see in section 5.2.1) is performed, as it is the cheaper. If this step can not decrease the duration of the loop operation to the desired value, or the operation is not a loop operation then replication is needed.

5.2.1 Inner optimization of node i

In case of a loop operation, which doesn't satisfy the desired R restart time, an effective way is to pipeline schedule the inner graph of the loop operation. This inner scheduling can decrease the duration time of the loop operation. In the following this inner graph pipeline scheduling will be discussed (deducing).

The following expression should be complete, that the v_i loop operation desires the global restart time R [1]:

$$q_{v_i} \leq R \tag{7}$$

Just in case of this condition can be ensured the desired R restart time is fulfilled. To substitute the expression (6) reviewed in the previous section to the expression (7):

$$(tc_i - 1) * R(G_i) + L(G_i) \leq R \tag{8}$$

whereby:

$$R(G_i) \leq \frac{R - L(G_i)}{tc_i - 1} \tag{9}$$

Because $R(G) \in \mathbb{N}$ and the minimum restart time of G_i is $R_{min}(G_i)$ (see expression (4)), the desired restart time can be expressed:

$$R_{des}(G_i) = max\left(\left\lfloor \frac{R - L(G_i)}{tc_i - 1} \right\rfloor, R_{min}(G_i)\right) \tag{10}$$

The inner graph needs to be optimized by the calculated desired restart time $R_{des}(G_i)$. As the inner graph is an MR-HSDFG, the pipeline scheduling of the inner graph can be performed also in the method overviewed in figure 5. (actually the same optimization algorithm is performed in case of each loop of the loop nest)

When the inner graph G_i is optimized, the duration time t_{v_i} of the loop operation v_i can be calculated by the expression (6), where $R(G_i) = R_{des}(G_i)$.

5.2.2 Operation replication of node i

If the operation is not a loop operation, or the inner optimization step (section 5.2.1.) may not satisfy the desired restart time, the only way to decrease the restart time is to replicate them. The number of copies can be calculated by the following expression:

$$c_i = \left\lceil \frac{t_i}{R} \right\rceil \tag{11}$$

6 Experimental results

In the following section two example is introduced. The first will demonstrate the advantage of the calculation method (section 4) and the second example demonstrate the advantage of the restart time minimization method (section 5).

6.1 First example - edge detection

As first example an edge detection algorithm based on the Roberts operator [14] will be introduced as the input of the reviewed methods.

The algorithm transforms a 64x64 pixel size grayscale bitmap to a same size bitmap, which shows the edges of the original bitmap. The data interface of the algorithm is two streams, one for the input (original) bitmap and one for the output (transformed) bitmap.

For calculating the intensity of an output pixel $P_{out}(x, y)$, four input pixels p_1-p_4 are needed:

$$\begin{aligned} p_1 &= P(x-1, y-1) \\ p_2 &= P(x+1, y-1) \\ p_3 &= P(x-1, y+1) \\ p_4 &= P(x+1, y+1) \end{aligned} \tag{12}$$

The calculation of the output pixel via the Roberts operator:

$$p_{out}(x, y) = max\left(|p_1 - p_4|, |p_2 - p_3|\right) \qquad (13)$$

As it can be seen, for calculating the pixels in a given row, the two adjacent rows has to be used, therefore 3 rows should be stored simultaneously. (the 3. row is the given row that will be used calculating the next row) As the access of the bitmap is sequentially (the data is got from a stream), a buffer is needed to store these 3 rows at the same time.

The Catapult C code of the algorithm is shown in figure 7. In the source code there are three for loops, the I, which iterates through the rows (between 0 and 63), the J1 and J2, which iterate through the columns (between 0 and 63 too). In every iteration of the I, one input row is processed, and one output row is produced. The J1 loop is responsible for reading the input stream, and store the data to a temporary buffer. The J2 loop calculates the next output pixel one-by-one, using the values stored in the temporary buffer.

We created the MR-HSDFG of the code, which is shown on Figure 8. The loop I is represented by a loop operation, which contains inside a Counter operation, five other elementary operations and the inner loops J1 and J2 as two loop operations. The loop operation J1 contains the stream reading and buffer writing operations, and the loop operation J2 contains the operations performing the edge detection and the stream writing.

```
#include "ac_fixed.h"
#include "ac_channel.h"
#include "type.h"

#pragma hls_design top

void test (
ac_channel<int> &data_in,
ac_channel<int> &data_out)
{
  int buf[4][64];

  I: for (int i=0; i<64; i++)
  {
    int is = i % 4;
    int ia = (i-1) % 4;
    int ib = (i-3) % 4;

    J1: for (int j=0; j<64; j++)
    {
      buf[is][j] = data_in.read();
    }

    int a1=0, b1=0, a2=0, b2=0;

    J2: for (int j=0; j<64; j++)
    {
      int a = buf[ia][j];
      int b = buf[ib][j];

      int atlo1 = a-b2;
      int atlo2 = b-a2;
      if (atlo1<0) atlo1 = -atlo1;
      if (atlo2<0) atlo2 = -atlo2;

      a2 = a1; a1 = a;
      b2 = b1; b1 = b;

      int el = atlo1 > atlo2
                 ? atlo1 : atlo2;

      data_out.write(el);
    }
  }
}
```

Figure 7: Catapult C source code of the task

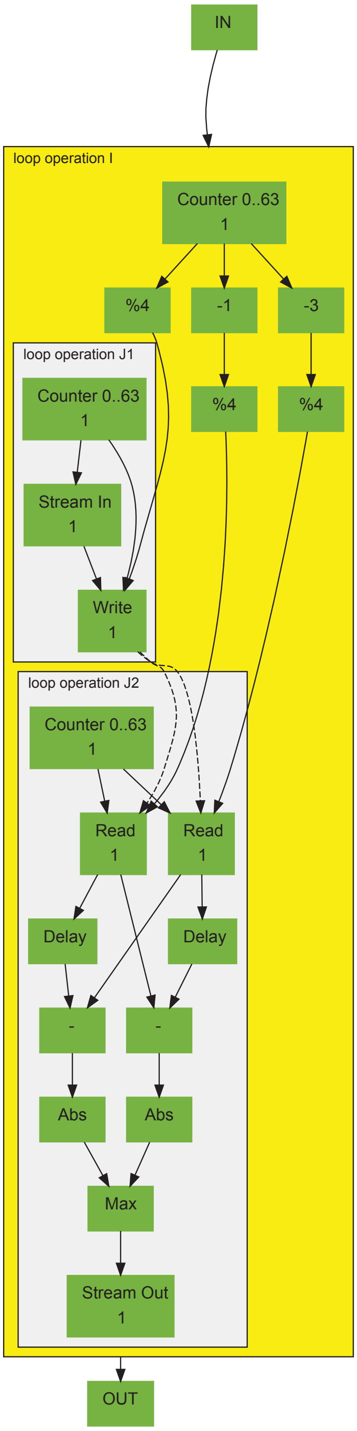


Figure 8: MR-HSDFG of the edge detection algorithm

The results is summarized in table 1. In the first four rows (Catapult results) the restart time of the outer loop I equals to the sum of the duration time of the operations inside the loop body:

$$R_3 = \sum t_{v_i} \tag{14}$$

where the duration time t_{v_i} of the loop operations can be calculated via the expression (6). In this case $R_3 = (63R_1 + L_1) + (63R_2 + L_2) + L'_3$, where L'_3 is negligible (the latency of the simple elementary operations inside the loop body of I).

For the second and third rows we used flattening in Catapult C, therefore the J1 and J2 loops disappeared. The parameter $II = 1$ in the third row can not be satisfied, because the minimal restart time of the original (before flattening) inner J2 loop is 2. In the fourth row the inner loops are optimized one by one, where the desired restart time of J1 is 1, and for J2 is 2. This is the best achieved solution using Catapult C.

The last row represents the result of our method. In this case J1 and J2 are put to different pipeline stages, therefore better result (8384 cycle instead of 12609, which is the best Catapult solution) is achieved compared with the Catapult C.

6.2 Second example - divisor function

The second example is a computation algorithm, which first calculates the average of 64 integer input values, than calculates the number of divisors of this average. The Catapult C code represents the algorithm is shown in figure 9. The code contains two for cycles, the first one (J1) deals with the average calculating, and the second one (J2) calculates the number of divisors.

The algorithm can be described in MR-HSDFG as the figure 11 shows, where the J1 and J2 loops are represents by loop operations. According to the section 4, the minimal restart time of the loop operation J1 is $(64-1)*1+3 = 66$ (for the inner graph, R=1 can be satisfied). For the loop operation J2, the minimual restart time is $(256-1)*7+7 = 1792$ (for the inner graph, R=7 can be satisfied without replication). The expression 3 can be used to calculate the R_{sp}, which is 1794 in this case. This situation is shown in figure 10(a).

Suppose the developer likes to schedule with R=68 constraint, which is the minimal value where loop operation J1 is not needed to replicate (actually it can't be replicated, because the Stream in operation can be executed once in a system clock period). According to the algorithm written in section 5, in the top level of the hierarchy only the loop operation J2 is to be scheduled newly. The desired restart time of the inner graph inside the loop operation J2 can be calculated by the expression (10):

$$R_{desired} = max\left(\left\lfloor\frac{68-7-2}{256-1}\right\rfloor, 1\right) = 1 \tag{15}$$

Inside the inner graph, only the % operation is to be replicated, where the number of copies is $\lceil\frac{5+2}{1}\rceil = 7$. In this case the $R_{desired} = 1$ can be satisfied and the duration time of the loop operation J2 will be $(256-1)*1+7 = 262$. After this, the loop operation J2 has to be replicated $\lceil\frac{262+2}{68}\rceil = 4$ times to satisfy the R=68 constraint. This case is shown in figure 10(b).

The Catapult C and MR-HSDFG results are summarized in table 2. The 1. row shows the sequential scheduling, in this case none of the loops is scheduled in pipeline mode. In the 2. solution, the J2 loop is pipeline scheduled (we chose the J2 for pipelining, because this was the bottleneck in the 1. solution) The 3. solution is pipeline scheduled in one level higher (in the main loop), which is achieved by flattening the main loop. The result is very similar than the previous one, because pipelining the loop J1 is ineffective. To increase the throughput additionally loop unrolling method is performed (4-6. rows), but in these cases the area cost will be duplicated. The 6. solution can not be satisfied, because the J1 loop can not be unrolled by 2.

7 Conclusion

In this paper we presented a novel pipeline scheduling method of nested loops. The greatest advantage of the method, that the pipeline scheduling can be performed in each level of the loop hierarchy simultaneously. Compared with the flattening based approaches this method can achieve more increasing in performance, as the restart time of the system

N	Results	J1 loop (tc_1=64)	J2 loop (tc_2=64)	I loop (tc_3=64)	Thrtpt. Cycles
1.	Cat. without opt	R_1=2, L_1=2	R_2=4, L_2=4	R_3=386	24705
2.	Cat. II_I=2	flattened	flattened	R_3=2 (tc_3=8128)	16260
3.	Cat. II_I=1	flattened	flattened	-	*can not be satisfied*
4.	Cat. II_{J1}=1, II_{J2}=2	R_1=1, L_1=2	R_2=2, L_2=4	R_3=197	12609
5.	MR-HSDFG R=R_{sp}	R_1=1, L_1=3	R_2=2, L_2=4	R_3=131	8384

Table 1: Results (R_i is the inner restart time, L_i is the inner latency of the given loop body)

N	Results	J1 loop	J2 loop	Throughput Cycles	Area
1.	Cat. without opt	tc_1=64, R_1=1, L_1=1	tc_2=256, R_2=6, L_2=6	1603	589
2.	Cat. II_{J2}=1	tc_1=64, R_1=1, L_1=1	tc_2=256, R_2=1, L_2=6	329	2880
3.	Cat. II_M=1	tc_1=64, *flattened*	tc_2=256, *flattened*	(L_M=8) 319	3071
4.	Cat. II_{J2}=1, U_{J2}=2	tc_1=64, R_1=1, L_1=1	tc_2=128, R_2=1, L_2=6	202	5733
5.	Cat. II_M=1, U_{J2}=2	tc_1=64, *flattened*	tc_2=128, *flattened*	(L_M=10) 191	5968
6.	Cat. II_M=1, U_M=2	tc_1=32, *flattened*	tc_2=128, *flattened*	*can not be satisfied*	-
7.	MR-HSDFG R=R_{sp}	tc_1=64, R_1=1, L_1=3	tc_2=256, R_2=7, L_2=7	1794	622
8.	MR-HSDFG R=264	tc_1=64, R_1=1, L_1=3	tc_2=256, R_2=2, L_2=4	264	2904
9.	MR-HSDFG R=68	tc_1=64, R_1=1, L_1=3	tc_2=256, R_2=2, L_2=4	68	11615

Table 2: Results (R_i is the inner restart time, L_i is the inner latency of the given loop body)

can be minimized, for example in case of sequential loops.

The novel dataflow graph model MR-HSDFG is also introduced, where the main purpose is to represent nested loops in an abstract and formal way that can be used effectively to perform pipeline scheduling methods.

We presented experimental result, where the advantage of the method is shown. Contrary to the Catapult C commercial development environment, the method creates the pipeline scheduling optimization automatically.

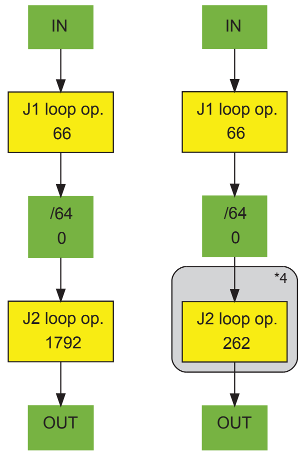
Further working: the array handling is very important to determine, whether the method can pipeline the sequential loops.

```
#include "ac_fixed.h"
#include "ac_channel.h"
#include "type.h"

#pragma hls_design top

int test(ac_channel<char> &data_in)
{
    int sum=0, count=0;
    for (int j=0; j<64; j++) {
        sum += data_in.read();
    }
    int number = sum / 64;
    for (int i=1; i<256; i++) {
        if (number%i==0 && i<=number)
            count++;
    }
    return count;
}
```

Figure 9: Catapult C source code of the divisor algorithm



(a) R=R_{sp} (b) R=68

Figure 10: MR-HSDFG of the divisor algorithm

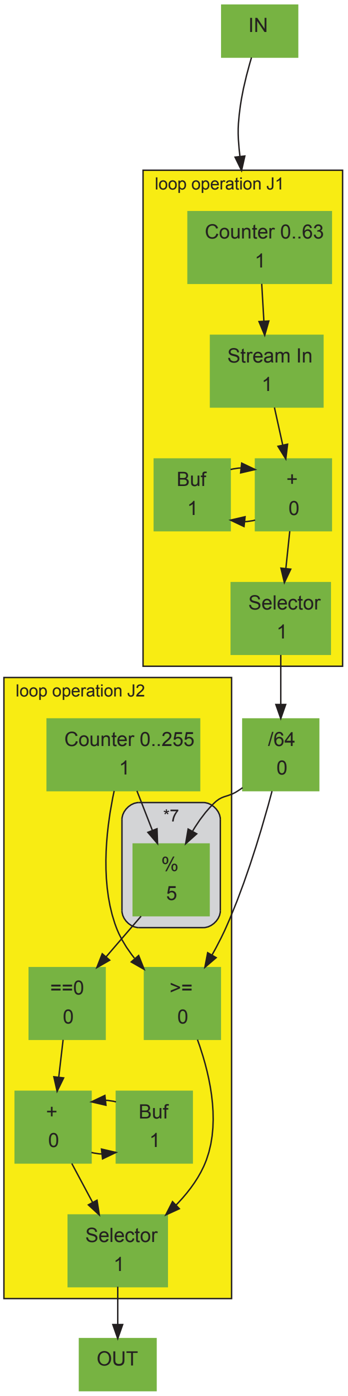


Figure 11: MR-HSDFG of the divisor algorithm

References

[1] Peter Arato, Visegrady Tamas, and Istvan Jankovits. *High Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[2] Smita Bakshi and Daniel D. Gajski. Performance-constrained hierarchical pipelining for behaviors, loops, and operations. *ACM Trans. Des. Autom. Electron. Syst.*, 6(1):1–25, January 2001.

[3] N. Chandrachoodan, S.S. Bhattacharyaa, and K.J.R. Liu. An efficient timing model for hardware implementation of multirate dataflow graphs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, volume 2, pages 1153 –1156 vol.2, 2001.

[4] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[5] Anwar M. Ghuloum and Allan L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 58–67, New York, NY, USA, 1995. ACM.

[6] Gennette Gill, John Hansen, and Montek Singh. Loop pipelining for high-throughput stream computation using self-timed rings. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 289–296, New York, NY, USA, 2006. ACM.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[8] J.C. Huang and T. Leng. Generalized loop-unrolling: a method for program speedup. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*, pages 244 –248, 1999.

[9] K. Ito and K.K. Parhi. Determining the iteration bounds of single-rate and multi-rate dataflow graphs. In *Circuits and Systems, 1994. APCCAS '94., 1994 IEEE Asia-Pacific Conference on*, pages 163 –168, dec 1994.

[10] R.B. Jones and V.H. Allan. Software pipelining: a comparison and improvement. In *Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on*, pages 46 –56, nov 1990.

[11] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[12] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[13] K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proceedings of the IEEE*, 77(12):1879 –1895, dec 1989.

[14] Lawrence G. Roberts. *Machine perception of three-dimensional solids*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1963.