Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Model-based new method for system-level synthesis of heterogeneous multiprocessor systems

PhD Thesis Book

Gergely Suba

Supervisor:
Dr. Péter Arató
Professor emeritus, Member of Hung. Acad. of Sci.

Budapest, 2018

1 Introduction, prelude

Nowadays, the developing of FPGA (field-programmable gate array) and ASIC (application-specific integrated circuit) systems are usually executed, by using HDL (hardware description languages). The Verilog, VHDL and their advanced variants are the most common used HDLs. These languages represent the digital circuits modularly as logic operations, memories and the wires (bitvectors) among them. The timing aspects of the circuits are also modeled by the HDL languages by explicit description of the clock signals.

The high-level synthesis (HLS) [NSP+16] has been introduced as a next step of the electronic design automation as a new method among the abstraction levels. The aim of the HLS is to generate the hardware description automatically from a purely algorithmic task description being devoid of hardware concepts and timing parameters. During this generation, optimization steps (synchronization, scheduling and allocation) serve to ensure the quality of the hardware description output. In the 90's, the HLS was considered as a generic tool in the hardware design workflows [MPC90, Cam90]. After widely using the FPGAs, the purpose of the HLS has been modified a bit, and became an important area of the FPGA based embedded system development [CGMT09, MST+09, GAGS09, CNNV11, CDL11, MVG+12, SL16].

The HLS is useful for the software engineers, because they can develop efficient hardware accelerators or full application specific hardware written in a general-purpose software language without hardware knowledge. Furthermore, the HLS is beneficial for hardware designers, because they can start to implement the hardware from a higher abstraction level than the HDLs.

Increasing the abstraction level of the design methods, the ESL (electronic system-level synthesis) has been introduced. Using the ESL design, the systematic development of a system containing hardware and software may start from a high-level behavior description. The automatic ESL design workflow can be referred as ESL synthesis, or simply SLS (system-level synthesis) [GHP+09].

The HLS and the SLS may help also in various field. These are suitable for fast and automated prototype implementation, even in designing digital application specific hardware, or in compiling for special supercomputer architectures including FPGAs, as well. The latter research fields are the RC (reconfigurable computing) [CDW10, HDA08, TCW+05] and the HPC (high-performance computing) [VN14, Awa09].

The hardware description language (i.e. the outputs of the HLS tools) can be considered as a dataflow [WP94, NLG99, JHM04] language, because the dataflow computing model and the structural description of the digital hardware are analogous. The structural hardware description includes the instances of the operation

modules and the bitvectors (wires) between them. The former corresponds to the operators (process, actors), the latter corresponds to the channels (flows, datalinks).

The most of the current HLS tools process the imperative and object-oriented C and C++ based input source codes. The products of the largest EDA (electronic design automation) companies Vivado HLS, CatapultC, CtoS, Synphony C, and the academic software products LegUp, Bambu, GAUT, etc. [NSP+16, AANS+14, MVG+12, CNNV11] can be mentioned as examples. There are fewer HLS input languages, which are not imperative and are known from other area of the software development field: the Lava [BCSS98] and the CλaSH[1] [BKK+10] based on the Haskell[2] functional language.

The high proportion of the C language among the HLS source languages can be explained by the fact that the HLS is applied in most cases in domains where the development basing on C is dominated (hardware systems, low level algorithms, embedded systems). Furthermore, a large number of algorithms have become available in C language in the past decades of the software engineering. These can be applied in HLS in a simple way, because transforming between the languages can be avoided.

The imperative paradigm differs on a great extent from the hardware description languages. A HDL contains concurrent operations or modules created by them, while the imperative language describes a sequential code. This essential difference is the reason why the HLSs with imperative input are complicated [Edw05]. In the following, the three biggest challenges will be detailed that are important when imperative languages are used for HLS: global variables, pointers and control structures.

The states are described as variables in the higher level imperative languages. The statements may change the values of the variables; moreover, it can happen in any point of the program in case of global variables. In hardware synthesis, it is an especially large problem, because the memory operations can be handled only by expensive wire buses (similarly to the processor systems that runs the imperative language descriptions).

The pointers are typical elements of the imperative languages. They point to specific memory places and can be used to read from or write to an arbitrary address. In contrast, the purpose of the HLS is to use the more, as possible distributed memories, because it is beneficial for running the codes in parallel. Therefore, it is a negative effect if a pointer would access to the whole memory content.

Another challenge is the control structures of the imperative programs. For example, the return statement can be called in any point of the function, and it performs immediate exit from the function. In case of processor computing, this is

[1] `http://www.clash-lang.org/`
[2] `https://www.haskell.org`

appropriate and can be implemented easily, but in a parallelly computing hardware (and also in a pure dataflow model), the jump statements cannot be handled. Similar problems are caused in loops by the break and continue statements and by the goto statement allowing over great freedom. The goto is usually forbidden by the coding standards or guides.

Although, the functional languages in HLS are rarely used, in the field of the software development they are becoming more and more popular. In the last years, there was a trend that the functional language elements (e.g. lambda expressions, data streams, immutable variables) are became the parts of the commonly used basically imperative and object-oriented languages. For example, the C# supports the lambda expressions from the version 3 (2007), the C++ from the version 11 (2011) and the Java from the version 8 (2014).

The functional languages are parts of the declarative languages family. In these languages, the execution order of the statements is not given explicitly, in contrast with the imperative languages, where the order is defined. Therefore, the jump statements and the higher structures of that (conditional statements, loops) cannot be described in a functional language. Instead of conditional statements, conditional expressions should be used, and instead of the loops, recursion is used.

The other important feature of the functional languages is the side-effect free behavior. In a pure functional language, the variables can be assigned only once, which is a large difference compared with the imperative languages. The variables of the imperative languages which can be assigned any times, are called mutable [But14], while the variables used in functional languages (which are assigned only once) are called immutable.

The functional languages are very similar to the dataflow languages in handling the side-effects. In both cases, the outputs depend only on the inputs (this is called referential transparency [But14] feature). This feature simplifies the HLS from functional source language in comparison with the imperative case.

However, certain features of functional languages differ from the dataflow languages, and also the hardware descriptions. E.g. the recursive functions, the first-class property of the functions, and the higher-order functions (where the parameters can be functions) are examples of such features.

The compilers of the programming languages are often constructed as three layer architectures [Mer03, LA03, LA04]. The frontend processes the source language, and produces the intermediate representation (IR). The middle-end optimizes the IR, and the back-end generates the output target code. This architecture has a large advantage. To handle a new source language, only a new frontend module has to

be implemented. Thus, the original middle-end and back-end can be used without changes. Similarly, a new target code generating requires only to develop a new back-end module. The optimization algorithms can be implemented independently from the source and target languages; it has to be implemented in the middle-end.

For these programs, choosing an appropriate intermediate representation is a crucial architectural decision. This determines which optimization algorithm can be performed by the middle-end, and what kind of transformation has to be performed in the frontend and backend layers.

The current HLS systems support typically only one source languages, or only one paradigm (i.e. similar languages). The reason of this is that the integrated handling of very different languages (e.g. imperative or functional) is a difficult task. Namely, the compilers use completely different IR, since the source languages are essentially different [Mer03, TC11].

2 Purposes

In my research, I have looked for the answer to the question: What kind of features have to be included into the HLS intermediate representations for helping the beneficial construction of a proper three-layer compiler architecture? This research aim could help to design and develop such a HLS framework that can process more source languages, and more target languages. Including different source languages to the HLS is important, because in the field of software engineering there are a lot of source languages (and in the future more and more languages will appear) that are usable in hardware synthesis tools. (Therefore, it was not my goal to work out a novel source language, because as I think, it is beneficial to use already existing, well known and widely used languages in HW synthesis. In this way, the synthesis system will be more usable by the engineers) One purpose of the dissertation is to work out a new HLS intermediate language that can handle both the imperative and the functional paradigm specialty. Hereby, the phases of the HLS and SLS workflow (decomposition, scheduling and allocation) can also be performed by using this intermediate language.

My next purpose has been to develop a compiler based on the HLS IR, for handling both imperative and functional language inputs. As far as I know, there is no such HLS system, which handles more language paradigms (e.g. imperative and functional).

A further purpose has been to work out a pipeline scheduling algorithm in the HLS IR for increasing the effectiveness of the whole design process.

3 New research results

In the following three subsections, I will present my new scientific research results summarized in three theses.

3.1 New intermediate language for high-level and system-level synthesis

First, I will consider the languages that can be used as intermediate languages (or part of them) in high-level and system-level synthesis: the dataflow graphs, the software modelling and generic graph languages in addition the imperative and functional programming language toolkits.

The properties of the dataflow graphs (side-effect free, explicit syntax for the datalinks) and the available scheduling algorithms [Chi12, AVJ01, TC98] are beneficial respecting the hardware synthesis and the hardware description generator. However, there are serious drawbacks. They do not support the hierarchic graphs, so they do not show the structural and modular construction of the source languages. The conditions are represented by control flow (CFG) edges usually. Thus, the graph loses its pure dataflow nature. Such graphs are called usually control and dataflow graph, CDFG [NRE04]. In CDFG, the loops are represented only by backward edges. Thus, the loops can be found only by expensive algorithms looking for circles.

A further drawback is that the dataflow graphs do not support the mutual data. In these cases, Load and Store nodes should be introduced [SK86], but thereby the pure dataflow nature is lost as well. Moreover, in these graphs the components and component instances are not shown, so writing common codes (routines, functions) generally are not supported.

Because of these drawbacks, using the dataflow graphs as a proper HLS IR is not beneficial. In spite of that, it can be stated that the basis of the new language would be worth constructing on the dataflow model principle.

Nowadays, the most widespread software modeling language is the UML (Unified Modeling Language) [OMG15]. The UML activity diagram can describe the dataflow graphs, the control flow graphs or a mix of these (CDFG). It supports the hierarchic construction, which is a big advantage in specifying a system or in forming the architecture. The UML activity diagram is generic software modeling tool for users, but it is not dedicated to intermediate representation for HLS problems. Therefore, it cannot be considered as a DSL (which has reduced number of elements) realizing the intermediate representation of compilers.

As the generic graph languages (GraphML, GXL) [BEH$^+$, WKR02] fit into the

graph problems well, they can describe the dataflow graphs in some degree. The drawback is, that they are too general, so the elements of the HLS domain are not contained. Nonetheless, some features of the languages (hierarchy, hyper edges) can be considered in constructing the HLS intermediate languages.

The upmost widespread imperative compiler is the GCC (GNU Compiler Collection) [EGH+05], which supports various source languages and target platforms. In industrial and also research projects, the other widespread popular compiler system is the LLVM [LA04], that has a new architecture and it has an intermediate representation with an external language. Both known compiler system has three layers (frontend, middle-end, backend), and the intermediate representation is based on the SSA (static single assignment) [CFR+91, BP03].

Although, these languages describe in deed the scalar variables in SSA form, this is not true for arrays and structures, that is the SSA based modeling of the data structures is not resolved. Disadvantageously, the conditions and loops still existing in the C language vanish in this representation and jump statements appear instead of them.

The GCC and LLVM IR are developed for the demand of imperative compilers dedicated to traditional processors, therefore they rather resemble the assembly languages, than a dataflow description. Besides, the conditional and unconditional jumps in the end of the basic blocks (BB) are strange from the dataflow approaches.

The SSA form fits well the dataflow graphs, thus it can be applied in hardware synthesis. However, the SSA form can be used only with scalar variables. Furthermore, these languages are basically imperative (it is reasonable, because they were made for processor systems), therefore, they are built strongly on the control flow principles.

From the imperative language toolkit, the mutable variable should be built in the new language, because there are problems handling by only immutable data would be difficult. The structural elements of the imperative languages, the conditions and loops should be handled in a HLS IR, as well.

The intermediate representations of the functional languages are mostly based on the lambda calculus. In these, the functions are first-class types, that is operations can be performed on functions, and the functions can be used as parameters of other functions (so higher-order functions can be handled). Obviously these cannot be represented directly by a hardware description; thus they have to be eliminated from a language based on lambda calculus. This is done usually by beta reduction resulting code duplications and increasing the IR size. In my solution, the function types and types represented in hardware description are also the parts of the language, but in totally separated form (thus, the functions are not first-class types anymore).

The further difficulty with the functional languages is that all the problems which normally could be solved by loops, will be solved by recursions. Another drawback is, that they do not handle mutable data.

The HIG (HLS Intermediate Graph) I have proposed is based on dataflow graphs. Thus, the graph nodes are the operations and the edges represent data links. The operations are instances of predefined components. A component can be elementary or complex. The elementary components describe atomic operations; the complex components can be defined by dataflow graphs. This forms a hierarchical construction. The loop and conditional components are also complex components (these are the structures of the higher-level languages). The components have given number of input and output ports, which are inherited by instantiating. The components can be parametrized by other components (similarly to the functional languages, where a function can be parametrized by other functions). The HIG can be modeled the information for scheduling with dependency and resource edges.

Thesis 1. *I worked out a new, dataflow graph based hierarchic graph model (HIG), which can be used as intermediate representation in the electronic system-level and high-level synthesis. The model allows the handling both the immutable and the mutable task descriptions in contrast with the known dataflow graphs. Thereby, it can be adapted to the source languages with both pure functional and imperative paradigms. In contrast to the usual graph models, it represents the nested loops and conditions without control flows. At the same time, the components, operations (component instances), ports and the data links (that appears in the hardware description abstraction layers) are also parts of the HIG. Furthermore, it models also the parameters for the decomposition, scheduling and allocation, thus it can be applied in the whole system level synthesis workflow. I have defined the HIG by formal grammar and metamodel definition. [1, 2, 3]*

3.2 System-level synthesis starting with functional and imperative languages

Comprehensive studies on commercial and open source HLS systems are available in the literature time to time [NSP+16, SL16, MVG+12, CDW10, MST+09]. The internal architecture of commercial software is in most cases considered industrial secrecy. Therefore, I have analyzed the open source compilers, because obviously more information is available. Based on the studies, it can be said that most of the existing open source HLS and SLS systems use the front end of the software compilers. For example, BAMBU [PF13], GAUT [CCB+08] and HercuLeS [KM13]

use GCC, LegUp [CCA+11] and Trident [TGP07] use LLVM, CλaSH [BKK+10] uses GHC.

It can be seen that these programs provide solutions for only one language paradigm. So, e.g. the GCC and LLVM provide solutions to imperative languages (mainly C based), and the CλaSH specialized on a Haskell functional language. In contrast, the methods presented in the thesis integrate both the imperative and the functional input languages.

The immutable and mutable data are important to handle during the whole process of system-level synthesis.

On the one hand, some problems are more reasonable to handle in the source language as immutable, while others as mutable data. The advantage of this is that a variable is only assigned once. Thus, it is easier for the developer or tester to see from the source code that a variable on a particular point of the program which operator results is using. In case of mutable data, the algorithms can be written by using loops, thus it is not necessary to express by recursion. It provides usually a simpler, more intuitive description.

On the other hand, the efficiency of the synthesized hardware is greatly influenced by the decision between the immutable and mutable data. In case of immutable data, the possibility of the parallelization increases, since only the single node writing the immutable data and the operations reading it become dependent. In the case of mutable data however, each such node is dependent, which writes or reads this mutable data. This affects the parallelizing possibilities negatively. However, the M-data have the advantage in implementing such algorithms in which the small data parts in a structure should be rewritten many times (this occurs often in handing arrays). In such cases, the solution using M-data can be implemented more effectively, because there are no needs for successive copying.

The decision that a data structure is implemented as immutable or as mutable data is already determined by the source language in the solutions known nowadays. E.g. the strings are mutable in C language. In Java, the strings are immutable, while the arrays are mutable. In Haskell, the most of the data structures are immutable, but it is possible to handle data structures as mutable data (e.g. MArray). In Scala[3], Kotlin[4] or Xtend[5] the *var* keyword declares mutable, while *val* declares immutable data.

With the solutions of current compilers, the problem is that the decision has to be made very early in the time of the source code writing, whether the implementation is immutable or mutable data. In my solution, the decision between immutable

[3] `https://www.scala-lang.org`
[4] `https://kotlinlang.org`
[5] `https://www.eclipse.org/xtend`

and mutable is arbitrary in the source language level. Thus, the user can freely choose those descriptions that better fit the language and the problem. During the implementation, however, those solutions that suit better to the conditions of the hardware synthesis (e.g. time and area requirements) can be implemented.

The HIG describes the immutable and mutable data in a completely analogous way, the only difference is the storage identifier represented by the datalinks which allows using multiple data connections for the same memory location.

The system-level synthesis method (PipeComp) that I have proposed to handle both functional and imperative inputs complemented by the immutable/mutable datalink decision is shown in Figure 1.

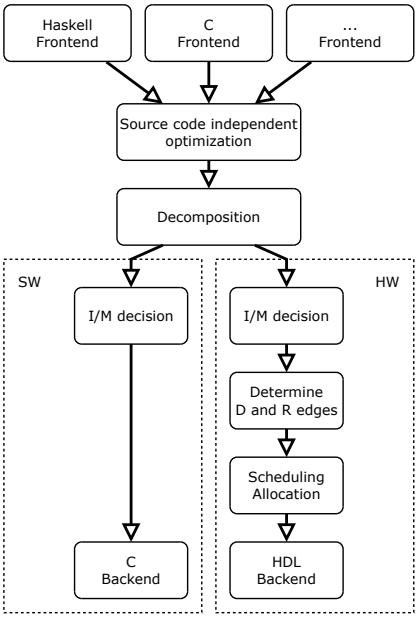


Figure 1: Functional and imperative language system-level method

During the decomposition, it is not necessary to consider whether the source code description contains immutable or mutable data. If the decomposition separates two data-dependent nodes into two separate partitions, then the data transfer between the partitions has to be ensured. It is irrelevant whether immutable or mutable data connection it is, in both cases, the entire data structures must be moved between the partitions.

After the decomposition phase has determined which nodes of the graph should

be allocated into which processing unit, the decision about immutable and mutable datalinks can be made by considering the properties of each processing unit. Regarding the software path, it is worthy to utilize the benefits of mutable data, since the operations cannot be parallelized in this case anyway, but the implementation with mutable data requires less memory. Besides, copying is not required in this case, which would be typical with immutable data.

Regarding the hardware path, the decision about immutable and mutable data affects the performance due to the above reasons. Derivation of dependency (D) and resource (R) edges based on datalinks depends on the immutable/mutable (I/M) decision, so these calculations have to be postponed after the I/M decision.

Thesis 2. *I worked out a system-level synthesis method (PipeComp) based on HIG, which can handle both functional and imperative languages in an integrated way. At the time of writing this dissertation, there has not been known such system-level or high-level synthesis tool which takes into consideration the characteristics of both the functional and the imperative paradigm languages. The algorithms of the PipeComp have been described by model transformation rules. [1, 4, 5, 6, 7, 8]*

- 2.1: *I have developed a method for an integrated uniform handling of immutable and mutable data in PipeComp synthesis. In contrast with methods available in the literature, it should not be decided at the source code level whether the implementation will be performed using immutable or mutable data connections. Thus, this decision effect on the lower abstraction levels it is not required to consider already in task describing. Therefore, the user has a greater freedom in choosing the most suitable describing method for designing the algorithms. Thus, the decisions at the lower level can be postponed that affects positively the automated optimum search. During the implementation, algorithmic decisions on immutable/mutable datalinks become possible for satisfying predefined constraints (e.g. time and space requirements).*

- 2.2: *For the PipeComp imperative frontend, I have defined a new SSA (static, single assignment) based structured language, which unlike the intermediate representations of the C compilers, does not contain explicit control flows. This makes it more suitable for both HLS and SLS tasks. Besides, finding and analyzing the parallelizable operations becomes simpler. The language can also be used as IR of the imperative software compilers and so for simplifying the dataflow optimizations.*

3.3 Pipeline scheduling of nested loops and mutable data

In the application of high-level and system-level synthesis – especially in DSP (digital signal processing) systems – the throughput is an important property of the efficiency. One of the methods for increasing the throughput is the pipeline scheduling of the system [AVJ01].

The HLS and SLS scheduling phase [Chi12] determines the start time of the graph nodes. The scheduling can occur in either at compile or run time. The former is called static scheduling; the latter is called dynamic scheduling. The advantage of static scheduling is that the duration time and the start time of each node is determined at compile time, that easier to implement the hardware. The disadvantage is that if the duration time of the operations is unknown in compile time or cannot be deduced by algorithm, then the scheduling cannot be performed.

However, a static pre-scheduling may be reasonable when the specific implementation is accomplished by dynamic scheduling. In this case,

- preliminary estimate can be given for the duration and start time of the operations
- in pipeline mode, the replications of the operations and the required restart times (iteration interval) can be specified
- the decisions on resource and time parameters can be made at compile time by estimating the expected results

Pipeline scheduling may have efficiency gains in case of loops, i.e. if the system or part of it is to be run repeatedly several times with different data. Scheduling can be particularly complicated, if the system construction contains nested loops or operations dealing with mutable data.

The maximum throughput of a system can be increased by looking for the components (bottlenecks) that effectively limit the throughput (i.e. the restart time) and then make changes on them. There are several ways to increase the efficiency:

- in case of loops, pipeline scheduling of the loop body
- replication of the component
- replacing the component by more effective implementation (this will not be discussed, assuming that the chosen hardware implementation is sufficient)

In today's HLS tools, finding bottlenecks and making decisions about the optimization solutions belong usually to the responsibilities of the user (one method is the profiling). In the Department of Control Engineering and Information Technology there is research for more than two decades for automated optimization in

order to achieve the desired throughput (more precisely restart time). However, the research has not revealed the pipeline optimization of nested loops and the mutable data.

Hereinafter, I will outline the results of my research in pipeline scheduling of the nested loops and mutable data.

The main features of scheduling:

Definition 3.1 *Pipeline mode: A HIG component is in pipeline mode, if it starts to process the next input data, before all of the output results are produced under processing the previous data.*

Definition 3.2 *Latency time: The latency time ($L(G)$) of a HIG component is the duration time* [6] *between the earliest consume time of graph inputs and the latest produce time of graph outputs (i.e. for multiple inputs and multiple outputs: the time between the maximum of the output time and the minimum of the input time).*

Definition 3.3 *Restart time: The restart time ($R(G)$) of a HIG is the duration that is the minimum time between two successive processing of the input data.*

To determine the restart time of a given HIG, the following input information is required:

- for each operation, the duration ($t_D(n_i)$) and busy ($t_B(n_i)$) time of the component instance
- busy time of all resource edges ($t_B(r_i)$)

$R(G)$ is defined in the following definitions for each of the three cases:

Definition 3.4 *Minimal restart time of HIG without pipeline scheduling (none-pipeline mode):*
$R_{NP}(G) = L(G)$

Definition 3.5 *Minimal restart time of HIG by pipeline scheduling without replication (pipeline mode, not replicated):*
$R_{PNR}(G) = \max\Big(\max_{\forall n_i \in N}\big(t_B(n_i)\big), \max_{\forall r_i \in E_R}\big(t_B(r_i)\big)\Big)$

[6] the duration time is measured in clock cycles

Definition 3.6 *Mininal restart time of HIG by pipeline scheduling with replication of operations and resources:*

$$
\begin{aligned}
R_{MIN}(G) &= \max\Big(\max_{\forall n_i \in N}\big(t_{MINBN}(n_i)\big), \max_{\forall r_i \in E_R}\big(t_{MINBE}(r_i)\big)\Big) \\
t_{MINBN}(n_i) &:= \begin{cases} 1 & \textit{if } n_i \textit{ can be multiplied} \\ t_B(n_i) & \textit{if } n_i \textit{ cannot be multiplied} \end{cases} \\
t_{MINBE}(r_i) &:= \begin{cases} 1 & \textit{if } r_i \textit{ can be multiplied} \\ t_B(r_i) & \textit{if } r_i \textit{ cannot be multiplied} \end{cases}
\end{aligned} \tag{1}
$$

To determine the minimum restart time of HIG, parameters t_D and t_B of the component instances are required (t_D should be calculated because of $L(G)$), so in the following I will give the calculation method for each component type.

The duration (t_D) and busy (t_B) time of the elementary components are considered as given input parameters, so I will not deal with these. For HIG components, the duration time equals to the latency time of the inner graph, while the busy time equals to the restart time of the inner graph.

$$t_D(G) = L(G) \tag{2}$$

$$t_B(G) = R(G) \tag{3}$$

This means that in the case of nested HIGs, the busy times of all the inner HIG components have to be determined before the pipeline restart time of the HIG start to compute. For the busy time, the restart time of the inner graph has to be computed. In the tree hierarchy of the HIG components, the calculation should start with the leaves, i.e. it is a bottom-up recursive algorithm.

In case of selection component, the duration and busy time can be calculated by the maximum of the inner HIG parameters:

$$
\begin{aligned}
t_D &= \max(t_D(C_i)) \\
t_B &= \max(t_B(C_i))
\end{aligned} \tag{4}
$$

For a loop component, the iteration count of the loop (tc), the busy time (t_B) and duration time (t_D) of the loop body have to be considered:

$$t_B = t_D = (tc-1)t_B{}'(B) + t_D(B) \tag{5}$$

In the Expression 5, there is $t_B{}'(B)$ instead of the loop body busy time ($t_B(B)$), since in this case, it is not enough to consider the busy time of the HIG component. In this case, the minimal pipeline restart time is limited also by the loop carried

dependencies (LCD) [STL04]. The busy time extended by the feedbacks is the maximum of the component busy time and the feedback duration time:

$$t_B{}'(B) = \max\left(t_B(B), \max_{\forall i \in [1,n]} \left(t_{PT}(y_i) - t_{CT}(f_i)\right)\right) \tag{6}$$

According to the expression, the feedback duration is given by the difference of the output and input start time. Therefore, it is necessary to have at disposal the t_{CT} and t_{PT} port attributes specified in the precondition of the transformation.

The Expression 5 can be explained as follows. While the loop component is executed once, the inner graph is performed tc-times, so between the first and last restart there is $(tc-1)*t_B{}'(B)$ clock time. After the last restart, the loop component is still busy for $t_D(B)$ clock periods (this is the time taking for the last input to be fully processed). Thus, the duration time of the loop component is given by the sum of these two values. The busy time is given by the resulted duration time.

An important requirement for a system can be to achieve a specified throughput. In other words, to accomplish the duration time of the successive input data. This is the desired restart time (R_D). To fulfill this requirement, the system has to be modified, and it may result in increasing the number of the hardware resources. In case of simple (not hierarchic) graphs, the desired throughput can be achieve by buffer insertion and operation replication. In the following, I will present the system design process for the desired pipeline restart time with the specialties of the HIG (mutable data, hierarchic graph, Loop and Sel component).

In case of elementary components, the method to reduce the busy time (t_B) is the replication of the operation. To reduce the busy time of the HIG component can be done by pipeline scheduling. If R_D is smaller than the resulted R_{NP}, then the components and resources limiting it, the R_D (i.e. $t_D > R_D$) should be modified according to the component type. In case of selection component, each inner component has to be interpreted by the desired restart time separately.

In case of loop component, there is a more efficient solution: to pipeline the loop instead of replicate the entire component. By this method, a limit for the loop body (B) can be determined based on the maximum busy time (C):

$$R_D(B) = \max\left(\left\lfloor \frac{R_D(C) - t_D(B)}{tc(C) - 1} \right\rfloor, R_{MIN}(B)\right) \tag{7}$$

If the feedbacks also limit the busy time, then the duration time of the concerned operations (which are in the critical path between the input and output ports of the feedbacks) should be reduced.

In order to get the least costly solution, calculating the extra cost required for reducing the unit duration time should be calculated for each node in the critical paths. Then, that node should be chosen that has the lowest cost for reducing the unit duration time. By this method, the duration times must be reduced until the execution time of the critical path reaches the desired value. (Reducing the unit duration time may actually result in a longer implementation time change, similarly to reducing the restart time one-by-one).

Reducing the duration time of the components can be done for each component type in the following ways:

- Elementary component: the only way to reduce the duration time is to choose a new implementation, so I will not deal with this case in the following.
- HIG components: the duration time of the critical path should be reduced, analogously to the solution described above.
- Sel component: the duration time of the inner HIG component should be reduced. The resulted cost is the sum of the costs of each HIG components.
- Loop component: pipeline scheduling can reduce the duration time, as it is detailed before.

If the required restart time is limited also by resource edges created by the mutable data, then there are two options. Either the duration time of the critical path should be reduced between the source and target operations of the resource edge, or resource replication has to be created with resource selection datalinks. The details are presented in the previous chapter.

Thesis 3. *I have developed a method for the static, hierarchical pipeline scheduling of the HIG operations, by performing changes in the required levels of the graph hierarchy in order to reach the desired restart time (iteration interval). By this method, the scheduling of the tasks containing immutable and mutable data and data independent (or data dependent, but limited trip count) nested loops can be performed. During the first step of the method, the restart time for each HIG component is calculated without pipeline mode. A particular HIG component will be pipeline scheduled, if it becomes necessary to fulfill the desired restart time of the entire system. If the pipeline scheduling is not enough for fulfilling the requirements, then replicating the operations can be performed. When loop feedback also limits the restart time, then the pipeline scheduling of inner loops can provide the solution. [2, 8]*

4 Application of the new results

The best known and the most widely used benchmark set in the field of high-level synthesis is the CHStone [Har08]. The benchmark set contains twelve codes written in C, all of which implement a well-known algorithm for commutation protocols, encryption, media files and arithmetic fields. I am presenting the practical application of the theses on the example of the SHA (Secure hash algorithms) of CHStone that includes all the components required for illustrating the pipeline scheduling method sufficient transparently.

Based on the methods presented in the theses, I have designed and implemented the HLS system PipeComp. In this work I have applied the following technologies:

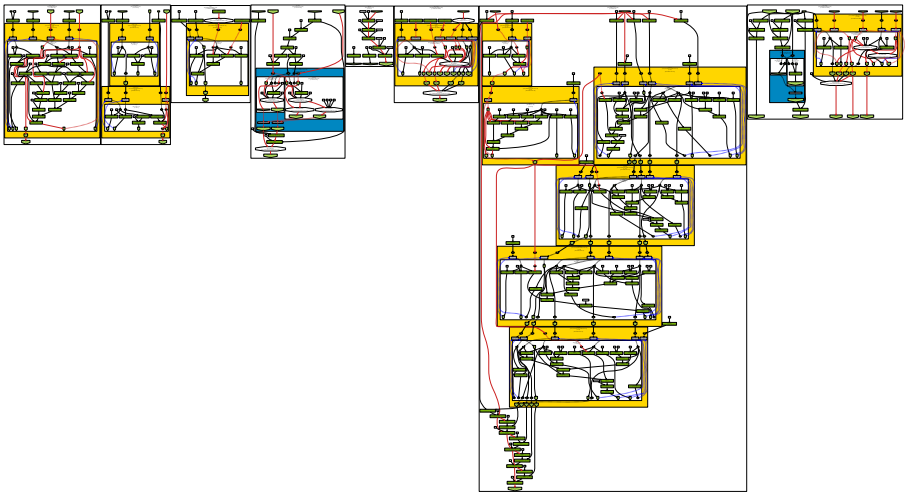
- Java 8: most of the code is written in Java, based on version 1.8
- ANTLR[7]: parser generator for syntactic analysis of C source codes
- EMF (Eclipse Modeling Framework)[8] [SBPM09]: a framework for model-based development
- VIATRA[9] [Újh16]: for the validation and transformation of graph-based models
- Xtend[10]: for model modification and code generation
- Xtext[11]: for processing textual DSLs
- GraphViz[12]: graphic rendering of graphs

The HIG generated from the SHA benchmark obtained by model transformations is shown in the upper part of the Figure 2. The summary table and the cost diagram are shown in the lower part of the figure.

4.1 Industrial application

The C language frontend of the PipeComp compiler system summarized in Thesis 2, and the SSA based imperative model have been applied in the practice by Prolan Zrt. The company develops – among other things – safety critical railway control systems, in which it has been necessary to process and to analyze the C language source code of embedded software in order to generate memory descriptions.

[7] `https://www.antlr.org/`
[8] `https://www.eclipse.org/emf`
[9] `https://www.eclipse.org/viatra`
[10] `https://www.eclipse.org/xtend`
[11] `https://www.eclipse.org/Xtext`
[12] `https://www.graphviz.org`



R	C	main	shastream	hig62	hig21	hig141	hig134	loop63node3
392842	959	-	-	-	-	-	-	-
392822	959	P	-	-	-	-	-	-
390032	959	P	P	-	-	-	-	-
390030	959	P	P	P	-	-	-	-
339738	959	P	P	-	P	-	-	-
323612	959	P	P	-	P	P	-	-
323610	959	P	P	P	P	P	-	-
315930	959	P	P	-	P	P	P	-
157965	1320	P	P	P	P	P	P	2
105310	1681	P	P	P	P	P	P	3
78983	2042	P	P	P	P	P	P	4
63186	2403	P	P	P	P	P	P	5

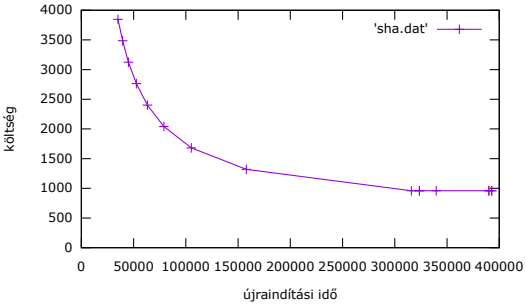


Figure 2: Summarization of the SHA pipeline scheduling results

5 Publications

Own publication

[1] Péter Arató and Gergely Suba. Data Flow Graph Generation of Nested Loops from Imperative High Level Description. *SCIENTIFIC BULLETIN of The POLITEHNICA University of Timişoara, Romania*, 59(73)(2):123–130, 2014.

[2] Gergely Suba. Hierarchical Pipelining of Nested Loops in High-Level Synthesis. *Periodica Polytechnica Electrical Engineering and Computer Science*, 58(3):81–91, 2014.

[3] Gergely Suba and Péter Arató. A new data flow graph model extended for handling loops and mutable data in high level synthesis. *Workshop on the Advances of Information Technology*, pages 1–5, 2017.

[4] Péter Arató and Gergely Suba. A data flow graph generation method starting from C description by handling loop nest hierarchy. In *IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 269–274. IEEE, may 2014.

[5] Gergely Suba and Péter Arató. A new method for transforming algorithm into VHDL by starting from a Haskell functional language description. In *Middle-European Conference on Applied Theoretical Computer Science*, pages 10–11, 2016.

[6] Gergely Suba and Péter Arató. Concept of the system-level synthesis framework PipeComp. In *WAIT 2015 : Workshop on the Advances of Information Technology*, Budapest, 2015.

[7] Péter Arató, István Loványi, Gergely Patai, Gergely Suba, Timotity Milan, and Tamás Torba. Compiling High Level Flow Languages into Hardware. Technical report, 2010.

[8] Péter Arató, Péter Dóbé, Daniel Andras Drexler, András Ercsényi, Balázs Goldschmidt, Tamás Horváth, Richárd Péter Kápolnai, István Loványi, György Pilászy, György Rácz, Balázs Simon, Ágoston Srp, and Gergely Suba. Feladatfüggő felépítésű pipeline többprocesszoros rendszerek tervezési módszerének kidolgozása és alkalmazása nagy sebességigényű beágyazott célrendszerekben - TÁMOP-4.2.2.C-11/1/KONV-2012-0004 projekt dokumentáció. Technical report, Pannon Egyetem, 2015.

References

[AANS+14] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrian Cristal, and Mikel Lujan. An empirical evaluation of High-Level Synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, sep 2014.

[AVJ01] Péter Arató, Tamás Visegrády, and István Jankovits. *High Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Ltd, New York, USA, 2001.

[Awa09] Mariette Awad. FPGA supercomputing platforms: A survey. In *2009 International Conference on Field Programmable Logic and Applications*, pages 564–568. IEEE, aug 2009.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98*, volume 34, pages 174–184, New York, New York, USA, jan 1998. ACM Press.

[BEH+] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML Progress Report.

[BKK+10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. CLaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, sep 2010.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, may 2003.

[But14] Paul Butcher. *Seven Concurrency Models in Seven Weeks.* Cambridge University Press, Cambridge, 2014.

[Cam90] R. Camposano. From behavior to structure: High-level synthesis. *Design & Test of Computers, IEEE*, 7(5):8–19, oct 1990.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, page 33, New York, New York, USA, 2011. ACM Press.

[CCB+08] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. GAUT: A High-Level Synthesis Tool for DSP Applications. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 147–169. Springer Netherlands, 2008.

[CDL11] Alexandre Cornu, Steven Derrien, and Dominique Lavenier. HLS tools for FPGA: Faster development with better performance. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6578 LNCS, pages 67–78, 2011.

[CDW10] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing. *ACM Computing Surveys*, 42(4):1–65, jun 2010.

[CFR+91] 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, oct 1991.

[CGMT09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, jul 2009.

[Chi12] M Chinnadurai. Survey On Scheduling And Allocation In High Level Synthesis. *International Journal of Computer Science & Engineering Survey*, 3(5):43–51, oct 2012.

[CNNV11] Jason Cong, Stephen Neuendorffer, Juanjo Noguera, and Kees Vissers. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, apr 2011.

[Edw05] Stephen A. Edwards. The challenges of hardware synthesis from C-like languages. In *Proceedings -Design, Automation and Test in Europe, DATE '05*, volume I, pages 66–67. Ieee, 2005.

[EGH+05] D Edelsohn, W Gellerich, M Hagog, D Naishlos, M Namolaru, E Pasch, H Penner, U Weigand, and A Zaks. Contributions to the GNU Compiler Collection. *IBM Systems Journal*, 44(2):259–278, 2005.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design*, volume 35. Springer US, Boston, MA, 2009.

[GHP+09] Andreas Gerstlauer, Christian Haubelt, A.D. Pimentel, T.P. Stefanov, Daniel D. Gajski, and J. Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, oct 2009.

[Har08] Yuko Hara. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195. IEEE, may 2008.

[HDA08] Scott Hauck, André Dehon, and DeHon André. *Reconfigurable Computing - The Theory and pratice of FPGA-BASED computing.* 2008.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, mar 2004.

[KM13] Nikolaos Kavvadias and Kostas Masselos. The HercuLeS high-level synthesis environment. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–1. IEEE, sep 2013.

[LA03] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. *GCC Developers Summit*, page 121, 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, number c, pages 75–86. IEEE, 2004.

[Mer03] Jason Merrill. Generic and gimple: A new tree representation for entire functions. *Proceedings of the 2003 GCC Developers' Summit*, 2003.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[MST+09] Grant Martin, Gary Smith, Don Tho, Mario Barbacci, and Alice Parker. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, jul 2009.

[MVG+12] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, aug 2012.

[NLG99] Walid a. Najjar, Edward a. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, dec 1999.

[NRE04] R. Namballa, N. Ranganathan, and A. Ejnioui. Control and data flow graph extraction for high-level synthesis. In *IEEE Computer Society Annual Symposium on VLSI*, pages 187–192. IEEE Comput. Soc, 2004.

[NSP+16] Razvan Nane, Vlad-mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):;article–PhD1591–1604, oct 2016.

[OMG15] OMG. OMG Unified Modeling Language (Version 2.5). Technical report, 2015.

[PF13] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, sep 2013.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternoostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[SK86] J Sargeant and CC C Kirkham. Stored data structures on the Manchester dataflow machine. *ACM SIGARCH Computer Architecture News*, 14(2):235–242, jun 1986.

[SL16] Hayden Kwok Hay So and Cheng Liu. *FPGAs for Software Programmers*. Springer International Publishing, Cham, 2016.

[STL04] H. Styles, D.B. Thomas, and W. Luk. Pipelining designs with loop-carried dependencies. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, pages 255–262. IEEE, 2004.

[TC98] Giri Tiruvuri and Moon Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):162–180, apr 1998.

[TC11] Andrew Tolmach and Tim Chevalier. An External Representation for the GHC Core Language. Technical report, The GHC Team, 2011.

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods, 2005.

[TGP07] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From High-Level Language to Hardware Circuitry. *Computer*, 40(3):28–37, mar 2007.

[Újh16] Zoltán Újhelyi. *Program Analysis Techniques for Model Queries and Transformations*. PhD thesis, 2016.

[VN14] Mario Vestias and Horacio Neto. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, sep 2014.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. Software Visualization. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 324–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[WP94] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, jan 1994.