WAIT: Workshop on the Advances in Information Technology, Budapest, 2015

Concept of the system-level synthesis framework PipeComp

G. Suba,[1] and P. Arató[1]

[1] Department of Control Engineering and Information Technology, Budapest University of Technology and Economics, Budapest, Hungary

Abstract

In this paper, the new system-level synthesis (SLS) framework PipeComp is introduced. The purpose of PipeComp is to compile source codes to target languages of different architectures, e.g. to hardware description or software program code. PipeComp is a three-layer architecture framework, it contains frontends, middle-ends and backends. The main different between PipeComp and the existing SLS solutions is that it has a dataflow graph intermediate language among the layers. Some specific modules are already implemented in PipeComp: imperative, functional and graphical frontends, pipeline scheduler and decomposition middle-ends, program code, hardware description and visual backends, which are introduced briefly in this paper.

Categories and Subject Descriptors (according to ACM CCS): B.1.4 [Microprogram Design Aids]: Languages and compilers

1. Introduction

In this paper, the new system-level synthesis (SLS) [1] framework PipeComp is introduced. The acronym PipeComp consists of the pipeline and compiler words. The main purpose of the framework is to compile one kind of a source code to a target language, e.g. to hardware description or software program code. PipeComp allows the user to create a complex hardware-software design.

The following requirements was set preliminary for PipeComp framework:

1. **Various input languages**: the framework should be able to compile the task from different input languages, in a special case from multiple languages, in the same compiling process.
2. **Various output language**: the framework should be able to produce the result of the compiler in different software or hardware languages, to realize a hardware/software co-design.
3. **Language independent optimization**: a general optimization phase should be inserted to the compiler pipeline without any knowledge about the supported input and output languages.
4. **Task decomposition**: the framework should be able to produce the parts of the input task to different target processor architecture (e.g. CPU, DSP, GPU, FPGA).

PipeComp is an EDA (electronic design automation) kind software that is based on the following specific methodologies:

- **Electronic system-level (ESL)**[1] design: develop an electronic system in higher abstraction level.
- **High-level synthesis (HLS)** [2]: one kind of ESL design methodology, which compiles automatically an algorithm written in high-level language to a hardware description.
- **Hardware-software co-design** [3]: realizes a system, which consists concurrent hardware and software parts.
- **Reconfigurable computing** [4]: architecture and the corresponding design methods in order to build high-performance computing (HPC) systems.

The main different between PipeComp and the other open-source frameworks [5, 6] and industrial HLS software [7, 6] that it uses a dataflow graph as intermediate language between the frontend and backend of PipeComp. The dataflow graph contains the behaviour of the system in a simple, formal way. The frontend-backend separation allows the user to compile a task from different and multiple source languages to different and multiple target languages.

2. PipeComp framework

PipeComp achieves the requirements listed in the previous section by a three-layer architecture (Figure 2). The upper layer (**frontend**) is responsible for producing an intermediate language (IL) from the source code. The middle layer (**middle-end**) consists of IL transformation and optimization, and the third layer (**backend**) transforms the IL to the target code.

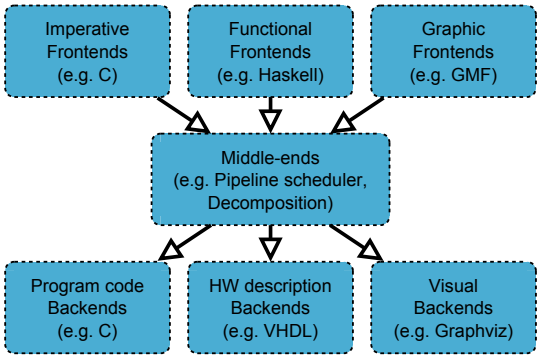

Figure 1: *Three level architecture of PipeComp*

Each supported source language is processed by a separate frontend module, thus the number of the different frontend modules equals as the number of the supported input languages.

The backends are also language dependent. The framework is able to produce multiple output codes: e.g. hardware descriptions for FPGA and ASIC, program codes for CPUs and DSPs. For each destination language a different backend module is applied, which generates the output for the destination architecture from the IL.

The typical corresponding functions of the three layers:

- **Frontend**: lexer and parser for textual frontends, syntactic and semantic analyzers, language dependant optimization and generation of the IL
- **Middle-end**: language independent optimization, input synchronization of operations, pipeline scheduling, decomposition
- **Backend**: optimization depends on the target device or target language, synchronization, scheduling, allocation, target code generation

The IL between the frontend and middle-end, and between the middle-end and backend is the novel dataflow graph representation HIG (HLS intermediate graph), which will be detailed in the next section.

2.1. HIG - HLS intermediate graph

HIG, as the intermediate language of PipeComp can be considered as a multi-rate extension of the dataflow graph EOG (elementary operation graph) [8]. The main advantage of HIG contrary to EOG is its nested-loop representation capability.

A vertex of the HIG is an operation, an edge between two vertices is a dataflow channel (shortly dataflow) between operations.

All operations have the following properties:

- **Operation class**: the base type of the operation. In HIG, the following classes are defined: elementary operation, constant node, input and output ports, complex and loop operations.
- **Execution time**: defines the duration time of an operation in the time unit of the system (e.g. clock period if the target is hardware). If the execution time is 0, the operation can be represented by a combinatorial logic, otherwise by a sequential logic.
- **Number of the inputs and outputs**: the number of the input operands and output results defined by the given operation.
- **Side effect free (SEF) property**: an operation is SEF, if the output depends on nothing but the inputs. Contrary, if the output of an operation depends on an inner state, the operation has side effect, thus the SEF property is not present.
- **Requirement of input synchronization (RSI) property**: if an operation is RSI, it starts the execution, only if all input dataflow channels of the operation has valid data. Otherwise, only one input data is enough to start the execution.

The properties of the dataflow channels:

- **Data type**: the type of the value transported by the dataflow. It can be considered as the generic variants of the C-based types. An essential parameter of the data type is the number of bits it can be represented in hardware/software.
- **Source node**: each dataflow has exact one source node (the edge starts from the source node of the dataflow).
- **Destination node**: each dataflow has exact one destination node (the edge ends at the destination node of the dataflow).

In Figure 2.1, an example is shown to demonstrate the usage of HIG. The six possible operation classes (each has its own color in the figure) will be detailed in the following:

- **Elementary operation** (dark green rectangles): an operation considered as atomic. It has accurately one output and one or more inputs. The behaviour of the elementary operation is either represented in the target language (e.g. simple arithmetic operations addition, substraction, multiply, compare, etc.), or it is defined by a complex module, function or subprogram (depends on the language termi-

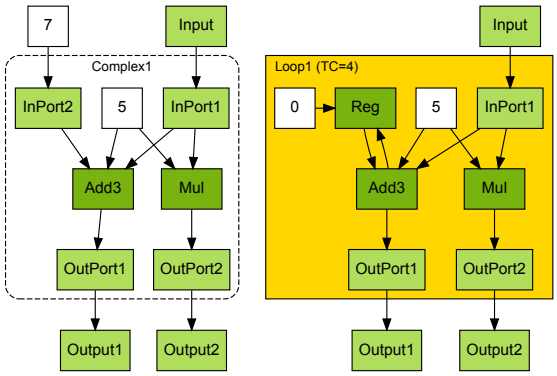

Figure 2: *Two example HIG graphs*

nology) in the target language. The duration time has to be defined before compiling process.

- **Constant node** (white rectangles): produce a constant value in run-time.
- **Input port** (light green rectangles): an input node of the HIG. A HIG is only able to get information through an input port from the outside context. In other words, an algorithm written in HIG gets its parameters from its input ports.
- **Output port** (light green rectangles): an output node of the HIG. A HIG can produce information only through an output port to its context.
- **Complex operation** (white, rounded rectangles): an operation, which is represented by an inner graph. The inner graph is also a HIG (which also contains operations and dataflow channels). An inner HIG gets its parameters through the input ports, and the results are produced through the output ports.
- **Loop operation** (yellow rectangles): a complex operation, which represents an algorithmic loop. The difference between the loop and complex operation is that the operations in a loop are executed more than ones (the number of the iterations is the **trip count**).

3. PipeComp modules

In this section, the specific modules are introduced, which is already implemented in PipeComp: imperative, functional and graphical frontends, pipeline scheduler and decomposition middle-ends, furthermore program code, hardware description and visual backends.

3.1. Imperative frontends

Considering an arbitrary computer language, the compiler has the following steps, no matter if it is an imperative, functional or any other paradigm language. The **lexer** processes the original source code, and produce a token list (e.g. in C: keywords, operations, identifiers, constants and literals). The next process is the **parser**, which gets the token list, and produces the abstract syntax tree (AST).

The AST of an imperative language (excluded the object oriented structures) consists of the typical structures: assignments, operations, branches, loops, function definitions, type definitions, etc. The AST is transformed to a simpler structure: limited number of branch and loop structures, single assignments for represent the operations.

An advanced compiler has many optimization possibilities. Most of them are dataflow optimization methods, where it is necessary to convert the assignments to SSA (static single assignment) form.

One solution for compiling C to HIG is the **GIMPLE frontend** [9] based on GCC [†] (GNU Compiler Collection). GIMPLE is in SSA form, but it has also huge disadvantages: it does not contain structural aspects (type and variable definitions), and it is not high-level enough. The loops and the branches are substituted by jumps, therefore a control flow processing method is needed to create the loop-nest hierarchy (LNH). Furthermore, it contains CPU specific decisions, e.g. memory offset parameters in case of arrays.

A new approach is applying the ANTLR [‡] to produce the AST, which eliminates the listed disadvantages of GIMPLE frontend, but the parser has to be developed from scratch.

The GIMPLE and ANTLR based approach is summarized in Figure 3.1 on the left and the right respectively.

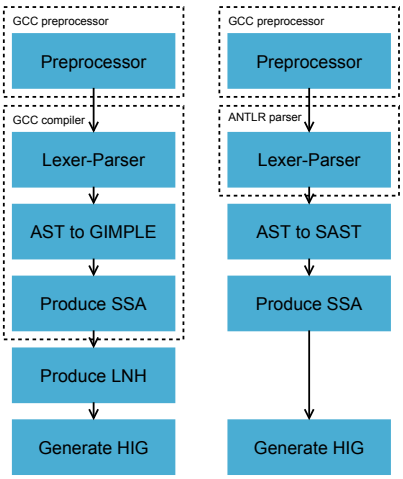

Figure 3: *GIMPLE and ANTLR based C frontends*

[†] https://gcc.gnu.org/
[‡] http://www.antlr.org/

3.2. Functional frontends

The first part (lexer and parser) of the functional frontends is similar to the imperative case detailed previously.

The **Haskell frontend** [10] of PipeComp is based on GHC [§] (Glasgow Haskell Compiler). The GHC parser produces an AST based on lambda calculus with the typical structures: lambda abstraction and application, variable binding and usage, pattern matching, type definition.

The AST of Haskell is evidently already SSA, as the functional languages set a specific variable only once.

3.3. Graphical frontends

Some aspects of tasks can be modeled graphically much simpler than in textual way, for example the connections of modules or the pipeline architecture algorithms. In **GMF frontend** [11] of PipeComp, the Eclipse based GMF [¶] (Graphical Modeling Framework) is applied to transform the inner model to HIG.

It is important to note that GMF is able to be used also for workflow editor, which controls the whole compiling process.

3.4. Optimization middle-ends

Pipeline scheduling [8] is an essential methodology to increase the throughput of a system, especially digital signal processing ones. In case of hierarchical dataflow graphs pipeline scheduling needs special considering [12], which is implemented in the **Pipeline scheduler middle-end** of PipeComp.

The purpose of the other significant part **Decomposition middle-end** [13] is to cut the task definition HIG into several parts automatically. After this process the specific parts can be synthesized to different processors.

3.5. Visual backends

The visual backends are used for documenting the inner dataflow model. In our case a Graphviz backend [11] is used. Graphviz is an open source graph visualization tool [‖]. Among others, in this paper this output is used to visualize the HIG representation.

3.6. Program code backends

The purpose of program code backends is to produce a compilable and runnable program code for CPUs, DSPs or any other classic processor architectures. The Java-based Xtend [**] is applied to generate the target code.

3.7. Hardware description backends

In this case the target domain is hardware structure descriptions rather than software program codes. Since the dataflow aspect is analogous to the hardware circuits, generating a HDL code from the HIG is obvious. The dataflow vertices are transformed to module instantiations or atomic operation keywords, and the edges are transformed to bit vectors.

The **VHDL backend** [10] is the only existing HDL backend of PipeComp now, but the development of a Verilog generator would not be a big challenge after the VHDL one is completed.

4. Summary

In this paper the SLS framework PipeComp was introduced briefly. Four requirements was set preliminary, which are achieved by a frontend - middle-end - backend architecture. The main different between PipeComp and the other similar frameworks that it uses a dataflow graph as intermediate language between the frontend and backend. The intermediate language in this case is the hierarchical intermediate graph with dataflow aspect.

PipeComp has specific modules already implemented, C, Haskell and GMF frontends, pipeline scheduler and decomposition middle-ends, furthermore program code, hardware description and visual backends.

Acknowledgements

The research work presented in this paper has been supported by the Hungarian Scientific Research Fund OTKA 72611, by the "Research University Project" TAMOP IKT T5 P3 and the research project TAMOP- 4.2.2.C-11/1/KONV-2012-0004.

References

1. A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. D. Gajski, and J. Teich, "Electronic System-Level Synthesis Methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.
2. G. Martin, G. Smith, D. Tho, M. Barbacci, and A. Parker, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009.
3. J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," in *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, May 2012, pp. 1411–1430.
4. T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," p. 193, 2005.

[§] https://www.haskell.org/ghc/

[¶] http://eclipse.org/modeling/gmp/

[‖] http://www.graphviz.org/content/dot-language

[**] https://eclipse.org/xtend/

5. S. Ravi and M. Joseph, "High-Level Test Synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 19, no. 4, pp. 1–27, Aug. 2014.

6. J. Cong, S. Neuendorffer, J. Noguera, and K. Vissers, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

7. A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in *Lecture Notes in Computer Science*, vol. 6578 LNCS, 2011, pp. 67–78.

8. P. Arató, T. Visegrády, and I. Jankovits, *High Level Synthesis of Pipelined Datapaths*. New York, USA: John Wiley & Sons, Ltd, 2001.

9. P. Arató and G. 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)*. IEEE, May 2014, pp. 269–274.

10. G. Suba and P. 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*, 2013.

11. G. Suba, P. Dóbé, B. Simon, and R. P. Kápolnai, "A magas szintû programnyelvi feladatleírás alapján adatfolyam jellegû gráf szisztematikus képzése," Budapest University of Technology and Economics, Budapest, Tech. Rep., 2015.

12. G. Suba, "Hierarchical Pipelining of Nested Loops in High-Level Synthesis," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 58, no. 3, pp. 81–91, 2014.

13. P. Arató, D. A. Drexler, and G. Kocza, "Loop-free Decomposition in High-Level Synthesis," *SCIENTIFIC BULLETIN of The POLITEHNICA University of Timižoara, Romania*, vol. 59(73), no. 2, pp. 99–104, 2014.