Summary of the paper "HW/SW co-synthesis using CASTLE" [Plöger and Wilberg, 1999]

Kenneth Rohde Christiansen

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3, NL-9747 AC Groningen

k.r.christiansen@student.rug.nl / kenneth@gnu.org

Abstract
In this summary we discuss a paper presenting a walk-through of the CASTLE co-synthesis design environment. The CASTLE environment provides a number of design tools that makes it possible to configure application-specific design flows. In order to get acquainted with these tools, a real life example is used as a walk-though the environment. The design flow starts out with a C++ implementation of a video processing system and graduately derives a register-transfer description of the hardware component. This document assumes that the reader has followed the Technology of System Realization course given at the RUG or at least a similar course given at another research institution.

Keywords: Hardware/Software co-synthesis, Technology of System Realization, Performance, Design Tools, Computer Science.

1 Introduction

There are various things that makes it hard to develop complex systems. One of these is the fact that the design of such a system very much depends on the application domain. For some applications, like real-time video de-compression, which will be touched in this paper, the most critical design parameter is performance. In other applications domains it might be the cost or the ability to validate the design that is the most critical design parameter. In the case of a pacemaker, validation of a correct design is most critical as it can have consequences for life and death.

1.1 ASIP and VLIW-processors

The summarized paper deals with the design of so-called Application Specific Instruction-set Processors, more often known as ASIP-design. In particular a VLIW processor architecture is chosen. VLIW is short for Very Long Instruction Word, and is a class of processors exploiting parallelisms of programs at the level of machine instructions and data moves. These processors uses several execution units which enables the processor to execute several instructions at the same like, like in the following example:

1. e = a * b
2. f = c + d
3. g = e + f

It is clear that operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. On the other hand, operations 1 and 2 do not depend on other operations and they be calculated simultaneously, exploiting this parallism.

Since all the functional units are controlled with an independent chunk of control code the instruction length often becomes longer than 100 bits. It is

then the compilers obligation to keep the functional units busy at all time, which results in high processing power with little hardware controlling overhead. Additionally, the VLIW architecture includes data flow mechanisms that forms a transport-triggered architecture.

Some of the most known VLIW processors around are the Cruesoe processor from Transmeta and the Intel Itanium. These processors have a modest success due to the fact that most VLIW compilers are not nearly as mature as similar RISC compilers and that they both rely on instruction-set translators to run RISC-based code. This, on the other hand, does not keep us from the fact that VLIW processors are very capable processors.

1.2 The CASTLE environment

CASTLE employs the MOVE system developed at Delft Technical University in the Netherlands, in conjunction with the normal VLIW processor architecture. MOVE includes the generation of a compiler back-end and even extends this approach by providing a retargetable code-generation and instruction-set simulator. The compiler in MOVE is based on the known free-software GNU Compiler Suite. MOVE additionally includes an assembler, a simulator as well as a source code browser. More information on MOVE is available in [H. Corporaal, 1991].

As mentioned in the article, many case-studies show that there unlikely is one single design flow that is sufficient to cover all requirements of the different application domains. Due to this fact CASTLE uses a workbench approach that makes it possible for the developer to configure a set of tools to possible supported design flows.

1.3 The walk-thought example

The paper deals with a programmable video processor which performs realtime MPEG-I decoding. A C++ implementation of such a decoder serves as the starting point, which is then analyzed to determine specific requirements of the application domain. The result of this analysis makes it possible for the designer to specify a suitable processor structure which can be entered on a block diagram level in CASTLE. CASTLE then generates a synthesizable VHDL description of the processor, as well as a compiler backend that can compile any C++ program to this specific processor architecture.

2 The proposed design flow

The design flow, that is proposed in the paper, consists of the following steps:

- Development and/or adaption of a C++ program.

Steps using MOVE:

- Compilation of the program into sequential assembler code.
- Simulation of the sequential assembler code.

Steps using CASTLE:

- Requirement analysis.
- Design space exploration of relaxed processors.
- Data transfer analysis.
- Co-synthesis of a processor (VHDL description) and the corresponding compiler backend.

In the paper the authors start out with an MPEG-I decoder found on the internet. The code is adapted and compiled into RISC code using the compiler distributed with MOVE. The sequential RISC assembler code can, together with a machine description file, be simulated with the MOVE simulator. The simulator outputs basic block profiles for the various test data. It is these block profiles that serve the basic for the analysis phases in CASTLE.

Lets look a bit more detailed into what happens with CASTLE. In the *requirement analysis* step the designer isolate the most demanding functions, the so-called hot spots. Optimizing these will yield the best acceleration possibilities. This can be accomplished by either dedicating special hardware to these hot spots or optimizing them using algorithmic transformations. Optimizing them using algorithmic transformation increase their potential instruction level parallelism according to the paper [Bau and Fisher, 1993]. After optimizations have been performed, the gains can immediately be measured by simulating the transformed code. The result of this requirement analysis is used to find the adequate functional units for the data path. Once these have been found we can move to the phase of *design space exploration*. The design space is still quite large because of the many variables, such

as register file size, the functional unit types, the topology etc.

CASTLE makes it possible to simulate different data paths and get information about the numbers of clock-cycles used. CASTLE does this at different levels of details and in that way answers questions about the parallelizability of the code and the utilization of the different functional units. According to the paper, this is of special interest since most other performance measuring tools only looks at functions and statements, where as CASTLE takes it a level lower and measures performance at the level of the basic blocks.

A minimal and a maximal machine description is used as a starting point for defining the worse and best achievable clock-cycle count. Furthermore, the numbers of adders and multiplexers is varied to look for saturation. A resource is saturated when the clock-cycle count is not reduces further by adding additional functional units of its kind. Finding the saturated numbers result in a justified guess for numbers of the various functional units needed.

In the *data transfer analysis* step, the interconnect and bus utilization is examined to find the right topology for the interconnected network.

When these steps have been completed it is possible for the designer to combine the results into a suitable VHDL model. This is mostly a manual task of combining parameterized, synthesizable VHDL components from a library of already made components. The designer is not left entirely on his own, but guided by the CASTLE so-synthesis schematic editor. When finished CASTLE generates a structural VHDL description as well as a compiler backend that can produce machine code for the synthesized processor.

3 The CASTLE programs

CASTLE consists of various tools to deal with these steps. A simple BLTGRAPH tool that simply visualize tabular numerical data. A VIEWMANAGER source code and performance browser, as well as SCHEMER which is a VLIW sensitive schematic editor accombinated with a VHDL component library. And last but not least, the tool ARCHITEX which provides a graphical user interface for exploring different VLIW-architectures.

4 The experiment

The goal of the experiment is to answer how an application-specific instruction-set processor look like, which is able to decode an MPEG-stream in realtime based on an already existing C++ implementation.

According to the paper, a sequential architecture will not live up to the requirements and a VLIW-architecture is required in order to take advantage of the parallelism.

The initial implementation is the C++ program called `mpeg_play` developed by Patel et al [K. Patel and Rowe, 1993]. As stated before the application has to be adapted a bit. In particular, this means:

1. Remove global variables
2. Make all communication explicit
3. Exclude external function calls from relevant profiling parts
4. Replace -if possible- pointers by array select methods.

The original sources have been adapted in the above manner and all external calls to low level X11 display routines have been removed as well. Since the Inverse Cosine Transformation in the MPEG algorithm is highly optimized in the code, it is also replaced by a less optimal solution, as the authors believe that such an optimization is an integral part of the hardware/software co-design.

4.1 Requirement analysis

With the use of MOVE, VIEWMANAGER and the visualization tool, sequential simulation is performed with various test data and the most demanding function is found. Various data sets were used to insure that the choice of function was data independent. The function `j_rev_dct` is found to be a good candidate for optimization, as it consumes 18% of all clock-cycles. The authors decide to concentrate on this function exclusively.

By using VIEWMANAGER, it is possible to look further at the instruction mix and that way map this to the overall data path structure. The conclusion of this analysis states that there are abundant adder operations and that there is a proportional relation between the number of `ALU`, `SHIFT`, `LOAD` and the rest of the operations like 4:2:1:1.

4.2 Design space exploration

The question is now how many of these functional units that can be used in parallel. The ARCHITEX tool is used to do disjoint analysis, thus varying only one parameter (for instance the number of adders) at the time and in that way determine the saturation point. After disjoint analysis is performed, conjoint analysis takes place, thus varying for instance the adders and shifts at the same time to explore their dependencies on each other. At the end variations of the register files are explored as well. Unfortunately, it is impossible to tell what is the maximum cycle count due to the fact that the compression algorithm depends on old state information in order to decode the picture. This state space is roughly proportional to the picture size, so an exausive exploration is not realistic.

The authors conclude that the disjoint and conjoint analysis were not very conclusive with the used implementation of the Inverse Cosine Transformation and started exploring other implementations. Another reason for this, was that the parallelism faded away too quickly and that the amount of cycles were too high according to the authors. We suggest the interested reader to look at the original paper for more information on this.

4.3 Data transfer analysis

After playing with different algorithmic variations a good candidate was found on which the disjoint analysis was performed.

The cycle count has this far been measured for a fully connected network which is pricey and not needed. Data transfer analysis is used to reduce the interconnect to a more realistic one. MOVE has the possibility to show the most frequent data moves. This gives a good idea on how to connect the different units to avoid write back to register files.

4.4 Co-synthesis and code generation

After finishing the analysis steps the developer has a good ideas on the kind of units needed, the amount and how they must be connected. CASTLE does not design the processor automatically, but instead offers a schematic editor with a component framework. It is then up to the developer to design the processor manually. From the paper it is hard to tell how well-suited this tool is for the task.

When the processor has been designed, the schematic editor SCHEMER will generate a VHDL file that can be simulated with a tool as VHDLAN. Unfortunately, the processor is not worth much without the code to run on it, so CASTLE also provides a retargetable assembler backend that translates sequential MOVE code into *opcode* of the designed VLIW-processor.

5 Conclusion

The authors of the paper looked into how to do co-synthesis starting from a simple C++ program, and described a co-design platform that they developed called CASTLE. Used in combination with MOVE, CASTLE makes it possible to analyze and find a suited VLIW processor design on which to run the assembler code generated from the C++ program.

The authors came to the conclusion that starting the design from a C++ program is indeed a practical and well-working solution. Other conclusions made are that the most optimized C++ code might not result in the highest performance with respect to instruction-level parallelism, which is to be expected, and that a hand coded implementation is often outperformed by an algorithmically transformed one.

The paper gives a good look into the various aspects of co-synthesis, but could have been better written. Many details of the experiment are mentioned without really giving the reader a real feeling of the experiment, which we find a pity. Their tools are also mentioned at several occasions, but only little is mentioned on how they actual work and how precisely they fit in with the MOVE suite. Unimportant details are often mentioned, as for instance that the tools are implemented in TCL and almost one whole page is dedicated to explaining how they organized the directory structure for performing their tests. This unfortunately just confuses the reader and keeps the reader away from the real essence of the article. We find this a great shame as the paper serves as a good introduction into co-synthesis, as well as shows a practical way of attacking the design problem.

6 Acknowledgements

We want to thank our teacher Sietse Achterop for an interesting and worthwhile course.

References

Bau, B. and Fisher, J. (1993). Instruction-level parallel procesing: History, overview and perspective. *Journal of Supercomputing*, **7**(2), 9–50.

H. Corporaal, H. M. (1991). Move: A framework for high-performance processor design. *Proceeding of Supercomputing '91, Albuquerque*.

K. Patel, B. S. and Rowe, L. (1993). Performance of a software mpeg video decoder. *Proceeding of 1st ACM International Conference on Multimedia, Anaheim, CA*.

Plöger, P. G. and Wilberg, J. (1999). Hw/sw cosynthesis using castle. *Separate paper*.