Automated Synthesis of Streaming C Applications to Process Networks in Hardware

Sven van Haastregt and Bart Kienhuis
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
{svhaastr,kienhuis}@liacs.nl

***Abstract*—The demand for embedded computing power is continuously increasing and FPGAs are becoming very interesting computing platforms, as they provide huge amounts of customizable parallelism. However, programming them is challenging, let alone from a high level language. In [1], the ESPAM methodology was already presented to quickly obtain realizations on FPGAs from sequential C code. The realization consists of a network of processors and IP cores. In this approach, a problem was that the IP cores had to be provided manually. In this paper, we present an extension on the ESPAM methodology by incorporating the industrial high level synthesis tool PICO from Synfora Inc. In this way, we realize the automated generation of efficient hardware implementations on FPGAs from a single sequential C input specification of a streaming application. We demonstrate our approach for the Sobel and QR applications.**

I. INTRODUCTION

Real-time stream processing applications are common in many different application fields. These applications are constantly subject to improvements and extensions, leading to increased computing power demands which can no longer be delivered by a single processor. Instead, *Field Programmable Gate Arrays* (FPGAs) may provide an attractive platform for such real-time stream processing applications, as they offer the opportunity to exploit a huge amount of customizable parallelism. However, implementing efficient streaming applications for such platforms is not trivial and is currently performed mainly manually; a complex, time-consuming and error-prone process. Also, extensive knowledge about the platform is required to obtain efficient implementations satisfying performance and cost constraints. This limits access to those platforms to skilled engineers. If such platforms could be programmed from a high level language, a much broader audience could benefit from FPGA technology.

To make FPGA technology more broadly accessible, the ESPAM methodology [1] was developed to provide a fast and convenient way to implement sequentially specified applications as process networks in hardware, giving designers access to considerable speedups. Unfortunately, the designer has to provide IP cores to realize a completely functional implementation. This still limits the target audience to engineers who can either write such IP cores themselves in HDL, or have access to a rich library of IP cores. To circumvent this problem, we have extended ESPAM with the industrial PICO tool [2] from Synfora, Inc. This way, we realize the ESPAM-PICO flow that further automates quick generation of complete and efficient functional hardware implementations from high level C input describing a streaming application. The result is then mapped onto an FPGA. Converting C to HDL has been the subject of research in many projects (e.g., [3], [4], [5]). Our work differs in its focus on streaming applications and exploitation of task level parallelism.

In this paper, we present the incorporation of PICO in ESPAM. In Section II, we present our solution approach. In Section III, we give a brief overview of the Synfora PICO tool. In Section IV, we show the flow of our ESPAM-PICO approach and in Section V, we present results we have obtained with the ESPAM-PICO flow and show that we obtain competitive results from C code in matters of tens of minutes of design time instead of hours or even days. We conclude our work in Section VI.

II. SOLUTION APPROACH

To further automate the synthesis of streaming C applications to efficient hardware, we have developed the approach depicted in Fig. 1. As a first step, we extract coarse-grained parallelism from the sequential C input specification, as depicted in Fig. 1 by the "Partitioning Compiler" block. This results in a network of multiple smaller units of execution which we call *processes*. Each process can again be expressed as a C program. This C program consists of a *control part* and a *functional part*. The control part is a result of the partitioning. The functional part is taken unmodified from the original C program. Next, we synthesize a hardware processor implementation for each of these processes and connect the hardware processors according to the network topology. This is depicted by the "Synthesis" block of Fig. 1. A hardware processor is either a conventional microprocessor like a MicroBlaze (MB) or a PowerPC (PPC), or a specialized accelerator (HWN). The accelerator is an IP core with equivalent behaviour of

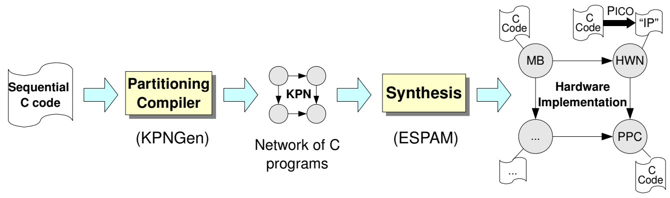
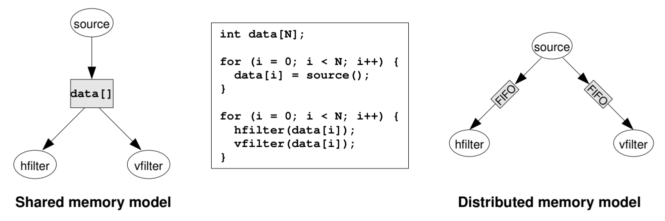


Fig. 1. Overview of our solution approach.



```
int data[N];

for (i = 0; i < N; i++) {
  data[i] = source();
}

for (i = 0; i < N; i++) {
  hfilter(data[i]);
  vfilter(data[i]);
}
```

Fig. 2. Shared vs. distributed memory.

the original C program. The novelty presented in this paper is that the IP core is now automatically synthesized using PICO, which takes the C code of the process as its input. During this step, fine-grained parallelism is extracted and exploited.

We use the *Kahn Process Network* (*KPN*) Model of Computation (MoC) to describe the network of processes and their behaviour. The KPN model has proven to be appropriate in similar cases [6]. The KPN MoC employs a distributed memory model, which is different from the shared memory model inherent to the C language. In Fig. 2, the differences between both memory models are illustrated. The shared memory implementation shown to the left of the code fragment consists of a (large) memory containing the entire array. In order to allow shared access, the amount of read and/or write ports is increased, or the memory component is time-shared across multiple entities. The first approach leads to a considerable increase in gate cost while the second approach is likely to limit throughput. The FIFO memories of the distributed memory model shown on the right do not require arbitration logic, as now only point-to-point connections are involved. Also, the sum of the FIFO buffer sizes for streaming applications is in general lower than the size of a single shared memory. An additional advantage of the KPN MoC is that high level transformations can be applied to its individual nodes [7], like unrolling and skewing, offering the possibility to explore alternative implementations of the same application.

III. PICO

PICO employs sophisticated optimization algorithms to generate a functionally equivalent RTL implementation from a specification written in PICO-C. The RTL implementation is expressed as a Pipeline of Processing Arrays (PPA), which is composed of a set of configurable architectural IP cores. The resulting PPA satisfies the constraints set by the designer, of which throughput is the most important one.

It is also possible to synthesize a Tightly Coupled Accelerator Block (TCAB). TCABs have been introduced to provide the designer with means to make hardware reuse and sharing inside a PPA possible. These TCABs typically exhibit less complex control. This puts additional restrictions on the accepted C input: only a single perfect loop nest is allowed.

Various restrictions are imposed on the PICO-C language, as it is not trivial to map concepts like pointers and recursive procedure calls onto a customizable distributed memory architecture. Another restriction on the PICO-C language is that multiple loop statements at the same nesting level inside another loop are not allowed, that is, it only accepts *perfectly nested loops*. Interestingly, our partitioning compiler KPNGEN is not affected by this perfectly nested loop requirement while the resulting KPN is always expressed as a collection of perfectly nested loop programs. Thus, each process generated by KPNGEN is always synthesizable using PICO, as long as the functional part of the process adheres to the PICO-C requirements. Another difference is that KPNGEN does a more thorough dataflow analysis, allowing a more aggressive employment of the distributed memory model, whereas PICO sometimes has to fall back to a shared memory model.

IV. IMPLEMENTATION

The realization of the high level flow given in Fig. 1 is given in Fig. 3. This figure shows the flow of our ESPAM-PICO approach. The user specifies the application as a C file, which is translated into a KPN by KPNGEN [8]. The input to KPNGEN is restricted to Static Affine Nested Loop Programs (SANLPs). By translating sequential code to a KPN representation, KPNGEN automatically obtains an implementation based on the distributed memory model shown in Fig. 2. Next, ESPAM-PICO takes this KPN, together with a platform specification describing the amount and types of processors in the system and a mapping specification which maps the processes of the KPN onto the processors. It then synthesizes an RTL implementation of the specified multiprocessor system by producing a Xilinx Platform Studio (XPS) project. ESPAM-PICO relies on the XPS tool to generate the final bitstream that configures an FPGA.

A processor can be a conventional microprocessor like a PowerPC or MicroBlaze, or it can be a hardware accelerator. In the conventional ESPAM flow, an IP core library is required to implement the functional part of a hardware accelerator. In this section, we present two new models, which incorporate PICO to synthesize completely functional hardware accelerators: the PPA hardware node model and the TCAB hardware node model.

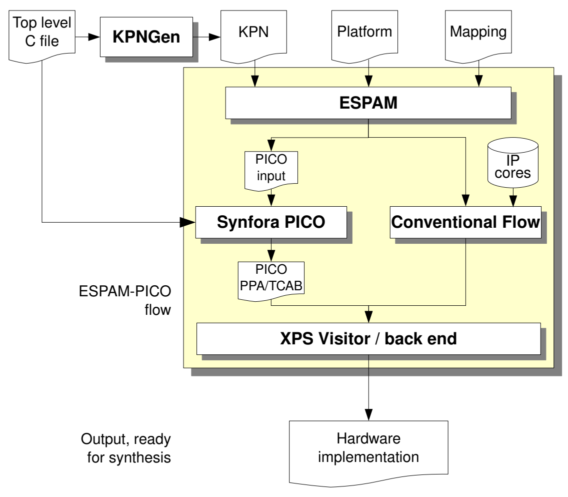


Fig. 3. The ESPAM-PICO design flow.

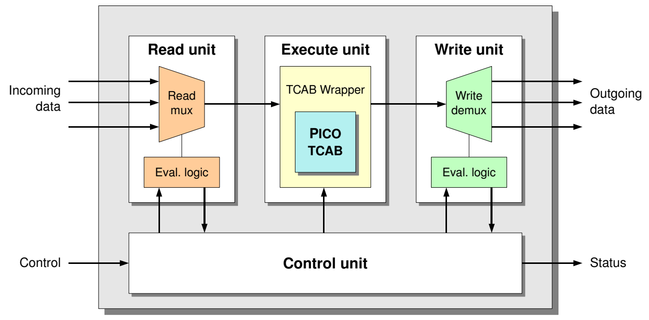


Fig. 4. The TCAB hardware node model.

A. PPA Hardware Nodes

As a first model, we use PICO to synthesize the C code belonging to a process into a PPA. This means that the PPA is responsible for both the control part (loop nest control, data input and output operations) and the functional part (the actual computation of the node). This PPA is integrated into a process network by using the stream interfaces offered by PICO-C, allowing a straightforward integration in the KPN model. However, we found that the PPA hardware node model is not suitable for KPNs containing feedback loops (cycles) or nodes with self-loops. This is because the entire PICO PPA is stalled when data is not available on one or more input streams.

B. TCAB Hardware Nodes

Because of the limitations of the PPA hardware node model, we propose a second model which builds further upon the LAURA processor model [9]. This processor model wraps an IP core in such a way that it can be integrated in a network of processors. A LAURA processor consists of a read and write unit which handle data communication from and to the right FIFO channels, and a control unit which synchronizes the different units. The functional part is implemented in the execute unit, and is where the actual computations of a node take place. Instead of integrating an IP core taken from a library in the execute unit, which was the default action, we now use a TCAB generated by PICO. The resulting TCAB hardware node model is shown in Fig. 4. This model is more robust than the PPA hardware node model, because the read and write units are now completely decoupled. Therefore, the TCAB hardware node model can much better handle deeply pipelined IP cores and the asynchronous behaviour of a KPN. As a result, the model can cope with self-loops and avoids undesired node stalls in the case of input unavailability. Furthermore, the node can be parameterized at runtime. For example, in case of imaging, the dimensions of a frame can be used as parameters of which the values are set or even changed at runtime.

V. EXPERIMENTS & RESULTS

To demonstrate how our methodology performs in terms of throughput and resource usage, we have examined two different applications: Sobel edge detection and QR decomposition. The first is a common image processing operation, the latter finds applications in adaptive beamforming systems for example. In the experiments, we used the XUP-V2P board, containing a Virtex-II Pro 30 FPGA (XC2VP30). Our implementations run at a clock frequency of 100 MHz.

A. Sobel Edge Detection

In Table I, the results of our experiments with the Sobel application are shown. The first column contains the name of the experiment. Next, the amount of FPGA slices and BRAMs needed for implementation are given. The last two columns show the absolute and relative amounts of 280×200 images ("frames") that could be processed in one second by the particular implementation.

The first four experiments in the table are implemented using the ESPAM-PICO flow presented in this paper. In the Sobel-PPA experiment, we implement the Sobel application using the PPA hardware node model. In the Sobel-PPA-2 and Sobel-PPA-4 experiments, the nodes of the network are unrolled by a factor of 2 and 4, respectively, to demonstrate that the unrolling transformation allows us to increase throughput in exchange for increased hardware resource usage. In the Sobel-TCAB experiment, we implement the Sobel application using the TCAB hardware node model. Both our models achieve the same throughput but the TCAB hardware node model requires 23 percent more resources than the PPA hardware node model. Part of this can be attributed to the presence of infrastructure for runtime parameter adjustment, which is not present in a regular PPA. Also, the extensions for more flexible pipeline behaviour supporting self-loops lead to increased slice usage.

The remaining experiments use approaches that already existed. This allows us to compare our work to other approaches. In the Sobel-PICO-naive experiment, we implement the application by providing the unmodified sequential C code of Sobel to PICO. This code does neither adhere to the PICO coding recommendations, nor does it make use of any PICO specific constructs like internal streams. This experiment yields the smallest implementation in terms of slice count. However, a huge memory component is required, which additionally limits throughput due to its single access port, as explained using Fig. 2. In the Sobel-PICO-hand experiment, a PICO hand design that makes use of a line buffer is evaluated. This design yields a small implementation, both in terms of slice count and memory usage, but manual rewriting of

TABLE I
RESULTS FOR SOBEL ON A 280×200 GRAYSCALE IMAGE.

Setup	Device utilization		Throughput	
	Slices	BRAMs	(frames/sec)	%
Sobel-PPA	1226	7	1784	100
Sobel-PPA-2	2768	14	3567	200
Sobel-PPA-4	5860	28	7129	400
Sobel-TCAB	1507	7	1784	100
Sobel-PICO-naive	665	27	181	10
Sobel-PICO-hand	895	4	1784	100
Sobel-ESPAM	1641	7	897	50

the input C code was required. Once buffers and pipelines are filled, the Sobel-PPA, Sobel-TCAB and Sobel-PICO-hand implementations deliver one pixel of the result per cycle. The Sobel-ESPAM experiment is the result of previous research [1] in which handwritten IP cores were used. Because Sobel-TCAB and Sobel-ESPAM have the components of the LAURA processor in common, the small difference in slice usage shows that the PICO-generated TCAB is an efficient IP core. Sobel-ESPAM requires about two times more clock cycles than the Sobel-PPA and Sobel-TCAB implementations, which is caused by the use of a memory unit instead of purely stream-based I/O.

B. QR Decomposition

In Table II, the results of our experiments with the QR application are shown. The columns of this table are similar to those of Table I, except that the last two columns now show the amount of 21×7 matrices that could be processed in one second. The trigonometric functions used in QR are implemented using a lookup table (LUT) or using the Maclaurin/Taylor series expansions (TA). The different realizations for the trigonometric functions could easily be expressed in the C program of QR. Due to the presence of self-loops in the KPN of the application, an implementation using the PPA hardware node model was not possible. All experiments are therefore done with the TCAB hardware model.

Only small FIFO sizes are involved, so all FSL components are implemented in logic only leading to zero BRAM usage. A first observation is that the lookup table based implementation is more efficient than the Taylor series based implementation, both in terms of device utilization and throughput. This is due to the higher complexity of the QR-TA implementation, which leads to deeper pipelines. Due to the flow dependencies in the QR network, the pipelines in the various hardware processors are underutilized which leads to limited throughput. By applying the skew transformation, we reschedule asynchronously the moments on which those dependencies occur. This makes it possible to keep the pipelines better filled, resulting in higher throughput, as shown in Table II for both QR-LUT and QR-TA.

In Table III, we show tool running times for two of our experiments. The translation from a sequential C specification into a parallel KPN representation takes only a couple of seconds. The amount of time needed by ESPAM-PICO depends greatly on the number of PICO hardware nodes in the network, as the PICO tool is invoked for each node in a KPN that is implemented by a PPA or TCAB hardware node. Synthesis into an FPGA bitstream still consumes most of the time for both experiments.

TABLE II
RESULTS FOR QR ON A 21×7 MATRIX.

Setup	**Device utilization**		**Throughput**	
	Slices	BRAMs	(matrices/sec)	%
QR-LUT	1417	0	43365	100
QR-LUT skewed	1798	0	191570	442
QR-TA	2705	0	23781	100
QR-TA skewed	3075	0	125313	527

TABLE III
RUNNING TIMES OF THE TOOLS.

Step	**Compile time**	
	Sobel-PPA	QR (LUT)
1. KPNGEN	5 sec.	5 sec.
2. ESPAM-PICO	6:50 min.	5:05 min.
3. Synthesis using XPS	12:45 min.	16:25 min.
Total	19:40 min.	21:35 min.

VI. CONCLUSIONS

In this paper, we have presented the ESPAM-PICO flow that further automates synthesis of streaming C applications written as static affine nested loop programs to process networks in hardware. The obtained results are efficient, because our flow exploits automatically different levels of parallelism and employs more aggressively a distributed memory model, without requiring source code annotations. Our ESPAM-PICO flow allows us to obtain an implementation in matters of tens of minutes. Next to our PPA hardware node model, our TCAB hardware node model proved to be an interesting addition, as it can be parameterized and handles self-loops.

The results shown in Table I and II have been obtained by only specifying C code. No HDL has been written or had to be inspected to get the results. This shows that in principle someone only familiar with C can obtain efficient implementations of streaming applications on FPGAs. Our ESPAM-PICO flow shows a promising collection of principal technologies and concepts that can help to provide a broader audience with tools to use FPGA technology.

REFERENCES

[1] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multi-processor System Design, Programming, and Implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 3, March 2008.

[2] Synfora Inc., "PICO Technology," http://www.synfora.com/, last accessed: 2008-06-11.

[3] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized Generation of Data-Path from C Codes," in *Design Automation and Test Europe (DATE'05)*, March 2005.

[4] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations," in *International Conference on VLSI Design*, January 2003.

[5] Y.D. Yankova et al., "DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator," in *Proc. of the 17th Intl. Conference on Field Programmable Logic and Applications (FPL'07)*, 2007, pp. 697–701.

[6] E. de Kock, "Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study," in *Proc. of the 15th International Symposium on System Synthesis (ISSS'02)*. ACM Press, 2002, pp. 68–73.

[7] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances," in *Proc. of the tenth international symposium on Hardware/software codesign (CODES'02)*. ACM Press, 2002, pp. 7–12.

[8] S. Verdoolaege, H. Nikolov, and T. Stefanov, "PN: a Tool for Improved Derivation of Process Networks," *EURASIP Journal on Embedded Systems*, 2007.

[9] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "LAURA: Leiden Architecture Research and Exploration Tool," in *Proc. of the 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, September 2003, pp. 911–920.