2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors

Enabling Automatic Pipeline Utilization Improvement in Polyhedral Process Network Implementations

Sven van Haastregt, Bart Kienhuis
Leiden Institute of Advanced Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
{svhaastr,kienhuis}@liacs.nl

***Abstract*—Because of the increasing complexity of modern embedded systems, High-Level Synthesis (HLS) has gained momentum. Most HLS tools employ Control Data Flow Graph (CDFG) based approaches. An alternative route from C to RTL was presented in [1], where a CDFG based approach was augmented with a polyhedral process network based approach. This alternative route enables application of high-level transformations giving a significant increase in utilization of pipelined components, which positively affects throughput. However, increased pipeline utilization could be obtained only after manually selecting a set of transformations to apply, which is a non-trivial task. The main contribution of this paper is a reordering buffer that enables automatic improvement of pipeline utilization.**

I. Introduction

Today's embedded systems are typically implemented using multiple processing units to meet increased computational demands. Some of these processing units are specialized hardware architectures, which offer higher throughput than microprocessor-based solutions. However, manually designing such specialized architectures at the *Register Transfer Level* (*RTL*) is challenging because of complexity and tight throughput constraints. To simplify the design of specialized architectures, *High-Level Synthesis* (*HLS*) tools have gained popularity. Such tools allow algorithmic specification of a design in C code, and then generate synthesizable RTL [2]. The main operating principle of an HLS tool is to compute a static schedule for a *Control Data Flow Graph* (*CDFG*) that meets certain constraints [2].

The CDFG based approach has proven effective for synthesis of fine-grained blocks, such as kernel functions. However, when interconnecting such fine-grained blocks, a static schedule may not yield the same performance as the optimal self-timed schedule [3]. Instead, we employ a self-scheduling approach to interconnect fine-grained blocks. Our approach is based on *Polyhedral Process Networks* (*PPN*), which are a subclass of Kahn Process Networks. An alternative route from sequential C code to RTL implementations employing PPNs was explored in [1]. The authors showed that by employing the PPN model, high-level transformations such as skewing become available that can have a major impact on throughput. By manually skewing the source code, the authors reported a 4× speedup at the cost of only 27% increase in resource usage. However, such speedups cannot be obtained if the designer is not fully aware of the most appropriate transformations. This paper presents a scheduling step and a reordering buffer that enable avoiding the need for manual selection of transformations.

This paper is organized as follows. In Section II we give background information and in Section III we present our problem statement. In Section IV we discuss related solutions to the problem and in Section V we describe our solution. In Section VII we show results of experiments. We conclude in Section VIII.

II. Preliminaries

We use the flow depicted in Figure 1 to obtain RTL implementations from static affine nested loop programs specified in C. We employ the the PN compiler [4] to derive an equivalent PPN from C code. This PPN is implemented in RTL using the ESPAM tool [5]. A PPN is a directed graph $G = (V, E)$ in which V is a set of vertices, or *processes*, and E is a set of edges, or *channels*. For each function call in the C program, PN constructs a process. Each process $p \in V$ iterates over an N-dimensional domain D_p which is defined by for-loops and if-statements surrounding the function call. For every iteration point in D_p, data is read from incoming channels, that data is processed by a function F, and then the transformed data is written to outgoing channels.

A C program is shown in the upper left of Figure 2. In the bottom left of Figure 2, we show the PPN derived by the PN tool, consisting of one process and six channels. Two of these channels are selfloops, because during certain iterations, elements of x and y are both produced and consumed by function F. The remaining four channels are connected to other processes which are omitted for the sake

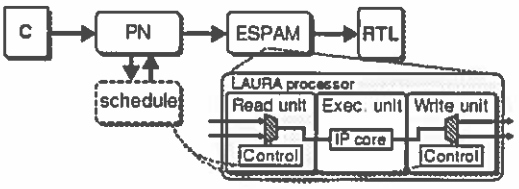

Figure 1. The PPN based C-to-RTL synthesis flow.

1063-6862/12 $26.00 © 2012 IEEE
DOI 10.1109/ASAP.2012.23

IEEE computer society

3NSDOCID <XP_____32473234A__I_>

of simplicity. The domain $D_F \subseteq \mathbb{Z}^2$ of the process derived from function F is shown on the right. Each point in the domain corresponds to an execution of function F.

Synchronization in a PPN is achieved by blocking read and write operations on the channels. Therefore, no *global schedule* expressing the firing order of the different processes is required for a PPN. A *local schedule* expresses the execution order of different iterations of an individual process. We focus on such local schedules in this work to improve throughput of processes. In current PPN implementations, D_p is always traversed in the order specified in the original C program: That is, the *original schedule* follows the lexicographical order in which the for-loops iterate through D_p, as depicted by the numbers inside the points in Figure 2.

III. Motivation & Problem Statement

In high-performance streaming applications, the function F of a process is typically implemented using a pipelined IP core [5] to satisfy throughput requirements. This pipelining allows for overlapped execution of multiple points of D_p. A high degree of overlapped execution is desirable, since this leads to higher throughput of the process. Whether overlapped execution actually takes place depends on multiple factors, such as pipeline depth and the chosen local schedule. In this paper, we assume pipeline depth is a given. Therefore, we want to focus on the local schedule.

The original schedule does not always yield the highest degree of overlapped execution. For our example of Figure 2, data dependences require that iteration $(1,1)$ executes before iterations $(1,2)$ and $(2,1)$. Similarly, $(1,2)$ should execute before $(1,3)$. When implementing F using a P-stage pipeline and following the original schedule, execution of the first four iterations takes $3P+1$ clock cycles, as depicted in Figure 3. However, if we execute the first four iterations in the order $(1,1)$, $(1,2)$, $(2,1)$, $(1,3)$, we still respect data dependences but execution takes only $3P$ cycles. Although in this simplified scenario the gain is only one cycle, we show that more substantial gains are achievable for real applications in Section VII.

IV. Related Work

A natural way to overlap execution of process iterations is to perform loop parallelization, using for example Feautrier's

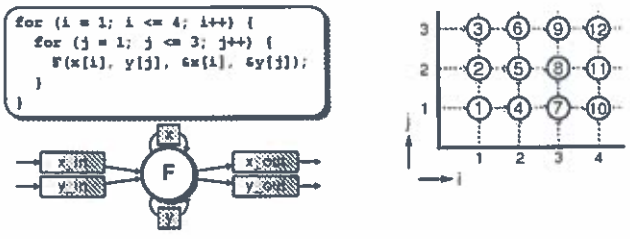

Figure 2. Example C program and corresponding PPN and dependence graph. The domain is traversed according to the original schedule.

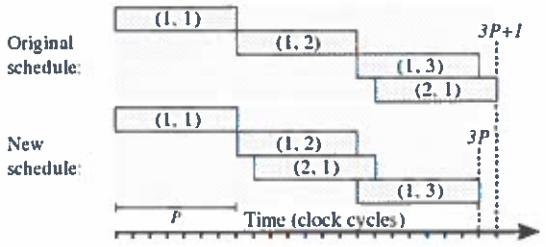

Figure 3. Pipeline behaviour for two different schedules. As shown in the dependence graph of Figure 2, iterations $(2,1)$ and $(1,2)$ do not depend on each other, which allows their executions to be overlapped.

algorithm [6]. Feautrier's algorithm has a high computational complexity, which motivated Feautrier to apply the algorithm to sets of communicating regular processes [7]. Unfortunately, Feautrier does not elaborate on the implications of the new schedule for the communication channels between processes. In Section VI we show that these implications cannot always be ignored. Another way to address the computational complexity of Feautrier's algorithm and control flow overhead of the resulting schedules was presented in [8]. They limit the possible schedule coefficients resulting in simpler schedules. This leads to more scheduling dimensions, which may counteract the benefits of simpler schedules. Feautrier's algorithm is employed by e.g. the MMAlpha tool [9] to generate hardware from algorithms specified in the Alpha language.

In [1], [10], the authors apply a skewing transformation by rewriting the source code. This is an effective way to increase overlapped execution, and consequently, improve pipeline utilization. However, identifying the skewing transformation parameters, such as the loop to skew, requires thorough studying of the application. This motivated us to investigate an automated approach to improve pipeline utilization without requiring designer decisions.

V. Solution Approach

Both the PPN model and Feautrier's scheduling algorithm are based on the polyhedral model. This allows us to incorporate a scheduling pass in the PPN derivation flow, such that pipeline utilization is improved without any effort from the designer. Our solution involves adding a scheduling step to the C-to-RTL synthesis flow, as depicted by the schedule block in Figure 1. The precise details of the scheduling step are beyond the scope of this paper. Changing a local schedule affects inter-process communication, which is the topic of the remainder of this paper.

VI. Communication

Ideally, a producer process produces tokens in the same order as the consumer process consumes them. Such *in-order* communication allows the channel from producer to consumer to be realized using a relatively inexpensive FIFO

BNSDOCID: <XP_____32473234A__I_>

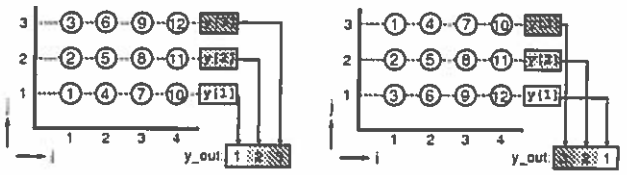

Figure 4. Example scheduled using its original schedule (left) and a different schedule introducing out-of-order communication (right).

buffer. However, the schedule computed by Feautrier's algorithm does not necessarily enforce in-order communication of data between processes. Thus, after applying the schedule, the order in which tokens are produced by the producer process may be different from the order in which tokens are consumed by the consumer process, and vice versa. For such *out-of-order* communication, channels can no longer be realized using FIFO buffers. Instead, more sophisticated interconnects in the form of *reordering buffers* are required. Such reordering buffers store incoming tokens in order in a private memory and contain reordering logic which outputs the stored tokens in the order required by the consumer.

In Figure 4, we show two different process domain traversals for the running example in which the `x[i]` accesses have been removed from function call `F`. On the left, we follow the original schedule. Channel `y_out` receives tokens in the order `y[1]`, `y[2]`, `y[3]`. On the right, we follow another valid schedule in which the inner loop is traversed in the reverse direction. As a result, channel `y_out` receives tokens in the order `y[3]`, `y[2]`, `y[1]`, which is different from the order resulting from the original schedule. If we assume that the schedule of `y_out`'s consumer process is not modified, the tokens would arrive in reverse order if `y_out` would be implemented using a FIFO buffer. To respect the correct token order, channel `y_out` has to be implemented using a reordering buffer.

In [11], different realizations of reordering buffers have been proposed, such as linear, pseudo-polynomial, and Content Addressable Memory (CAM) based implementations. The authors showed that these reordering buffer designs have a considerable negative impact on performance and resource usage. For example, read and write operations of a CAM implementation take four and two clock cycles [12], respectively, which would counteract improvements gained from rescheduling.

To avoid counteracting the benefits of a better schedule because of possible reordering communication, we have developed a new reordering buffer. The primary difference with previous work is that read and write operations now take only one clock cycle. This means that replacing a FIFO buffer with a reordering buffer increases resource usage, but does not introduce additional delay cycles.

Our reordering buffer is composed of a Write Address Generator (WAG), a Read Address Generator (RAG), and a private memory. The memory is dual-ported, with one

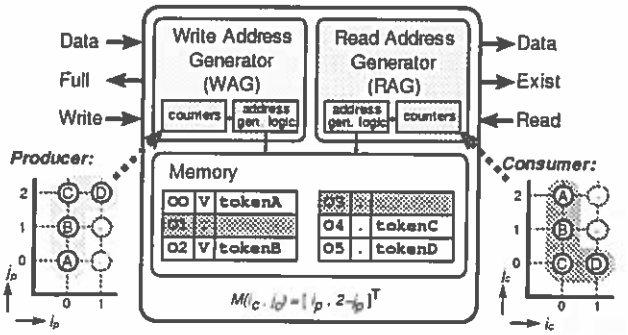

Figure 5. Reordering buffer.

port being addressed by the WAG and the other port being addressed by the RAG. The WAG and RAG both contain a set of counters which iterate through domains associated to the channel. These counters are used by the address generation logic to compute the next write and read addresses. To avoid delay cycles, the counters and address generation logic are implemented in a pipeline fashion. To minimize the latency of the address generation logic, we employ a linear addressing scheme. This addressing scheme is based on conventional linearization of an n-dimensional array into a 1-dimensional array. As such, the resulting address expressions are linear polynomials that can be realized efficiently in hardware.

The interface of the reordering buffer resembles a point-to-point FIFO buffer interface. This allows straightforward integration of reordering buffers in ESPAM-generated PPN implementations. That is, when rescheduling introduces out-of-order communication, we do not have to modify the interfaces of the processes involved in the out-of-order communication. The interface is depicted in Figure 5. The outgoing slave interface exposes an output data bus, an exist signal to indicate if a token is available, and a read signal to acknowledge a read operation. The incoming master interface exposes an input data bus, a full signal to block write operations when the buffer is not ready to accept them, and a write signal to acknowledge a write operation.

We illustrate the memory organization of our reordering buffer at the bottom of Figure 5. In the bottom left, we show a producer domain consisting of four points $(0,0)$, $(0,1)$, $(0,2)$, and $(1,2)$. The producer produces four tokens in the order A, B, C, D. We store these tokens according to a linear addressing scheme at address

$$wAddr(i_p, j_p) = i_p + 2 \cdot j_p. \qquad (1)$$

The slot for each token is shown in the memory of Figure 5. For example, token C is produced in iteration $(0,2)$ and is therefore stored at address 04. Because of the linear addressing scheme, some addresses may remain unused for non-rectangular domains. In our example, this occurs for addresses 01 and 03. The consumer domain shown on the

NSDOCID: <XP_____32473234A__I_>

bottom right consumes the four tokens in the order C, D, B, A. To retrieve these tokens in the correct order from the memory, we compute

$$rAddr(i_c, j_c) = wAddr(M(i_c, j_c)) \quad (2)$$

for each point in the consumer domain. That is, we first apply the channel relation M as found by the PN compiler. This gives the point (i_p, j_p) in the producer domain that corresponds to the point (i_c, j_c) in the consumer domain. We then compute $wAddr(i_p, j_p)$ to obtain the address from which the token should be read.

For token C, which is consumed in iteration $(0,0)$, the *rAddr* function yields address 04 which is the same address that was computed by the WAG. However, a token may not have been written by the producer yet. For example, token C may not be available yet at address 04. Therefore, we introduce an additional valid bit for each memory location. The valid bit is set once a token has been written to its address. To comply with the blocking read semantics of the PPN model, the RAG blocks until the token corresponding to the current consumer iteration is written. In the memory of Figure 5, tokens A and B have been written, as indicated by the "V"s, whereas tokens C and D have not been written yet, as indicated by the "."s.

VII. Experimental Results

To assess the impact of rescheduling on resource usage and performance, we have experimented with the following applications: QR decomposition, which is used in e.g. wireless receivers and radar; grid, taken from [10]; and matrix-matrix multiplication (mmm). We target a Xilinx Virtex-5 XC5VLX110T-2 FPGA and use Xilinx ISE 13.1 for synthesis. We instruct the synthesis tools to report the maximum achievable clock frequency.

In the second and third columns of Table I, we show the number of out-of-order channels and the total number of channels for the original and rescheduled implementations, respectively. For example, the rescheduled PPN of QR contains twelve channels, of which three require a reordering buffer. We found that the linear realization of our reordering buffers provides a good balance between control overhead and memory usage. In each experiment, the reordering buffers are not part of the critical path limiting clock frequency and each reordering buffer requires only a single 2KB block memory primitive.

Table I
Impact of Applying Feautrier's Algorithm on PPNs.

Application	#OO / #Channels		ΔLUTs	Speedup
	Orig.	Resched.	Resched.	Resched.
grid	0 / 6	0 / 6	+106 (+27%)	2.7
mmm	0 / 6	4 / 6	+292 (+45%)	2.8
QR	0 / 12	3 / 12	+871 (+68%)	2.7

VIII. Conclusions

In this paper, we have presented an approach that enables obtaining improved pipeline utilization in polyhedral process networks in an automated way. Our approach consists of leveraging Feautrier's optimal multi-dimensional scheduling algorithm to improve overlapped execution in a pipelined design by altering local schedules of processes. We take into account the impact of a new schedule on inter-process communication using a new reordering buffer design that has a lower access latency than existing solutions. This enables automatic generation of improved RTL implementations from applications written in C. In experiments with three applications we have obtained speedups of 2.7× without requiring additional effort from the designer.

References

[1] S. van Haastregt and B. Kienhuis, "Automated Synthesis of Streaming C Applications to Process Networks in Hardware," in *Design, Autom. and Test in Europe (DATE'09)*, April 2009.

[2] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Des. Test*, vol. 26, pp. 8–17, July 2009.

[3] E. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," in *Global Telecommunications Conference*, 1989, pp. 1279–1283.

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

[5] H. Nikolov, T. Stefanov, and E. Deprettere, "Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM," *EURASIP Journal on Embedded Systems*, vol. 2008.

[6] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time," *Intl. J. of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec 1992.

[7] ——, "Scalable and Structured Scheduling," *Intl. J. of Parallel Programming*, vol. 34, no. 5, pp. 459–487, October 2006.

[8] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative Optimization in the Polyhedral Model: part II, Multi-dimensional Time," in *Programming Language Design and Implementation*, 2008, pp. 90–100.

[9] A.-C. Guillou, P. Quinton, and T. Risset, "Hardware Synthesis for Multi-Dimensional Time," in *Application-Specific Systems, Arch. and Processors (ASAP)*, 2003, pp. 40–50.

[10] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances," in *Hardware/software codesign (CODES)*, 2002, pp. 7–12.

[11] A. Turjan, B. Kienhuis, and E. Deprettere, "Realizations of the extended linearization model," in *Domain-Specific Embedded Multiprocessors*. Marcel Dekker, Inc., 2003, ch. 9, pp. 171–191.

[12] C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "Solving Out of Order Communication using CAM Memory: an Implementation," in *Circuits, Systems and Signal Processing (ProRISC)*, 2002.

BNSDOCID: <XP_____32473234A__1_>