Quantitative Evaluation of Behavioral Synthesis
Approaches for Reconfigurable Devices

Frank Hannig, Hritam Dutta, Holger Ruckdeschel, and Jürgen Teich

Chair of Hardware/Software Co-Design,
University of Erlangen-Nuremberg, Germany.
{hannig, dutta, teich}@cs.fau.de

Abstract. State-of-the-art behavioral synthesis tools for reconfigurable architectures barely have high-level transformations in order to achieve highly parallelized implementations. If any, they apply loop unrolling to obtain a higher throughput. In this paper, we use the PARO behavioral synthesis tool which has the unique ability to perform both loop unrolling or loop partitioning. Loop unrolling replicates the loop kernel and exposes the parallelism for hardware implementation. Whereas, partitioning tiles the loop program onto a regular array consisting of tightly coupled processing elements. The usage of the same design tool for both the variants enables for the first time, a quantitative evaluation of the two approaches with help of selected computationally intensive algorithms. The processor array approach achieves in all experiments a far better throughput, up to 61% more compared with loop unrolling because of its regularity and the clustering of resources into several processing elements.

1 Introduction

Industrial, scientific, and multimedia applications are characterized by a growing need for computational power and stringent constraints on power consumption. Standard desktop, server, or embedded processors are often not suitable for these applications, because they are either expensive, have a too high power consumption, or do not provide enough performance in order to fulfill the user’s requirements. For that reason, there has always been the idea to accelerate the execution by means of special hardware support. In this regard, several different solutions have been proposed over the time. Since most computationally intensive algorithms feature a high inherent parallelism, many techniques were developed that focus on exploiting parallel processing capabilities in order to accelerate the execution of an application. All standard processors provide fine-grained parallelism by executing a small amount of machine instructions in parallel. The parallelization is done either implicitly by the processor or explicitly by the compiler for so-called very long instruction word (VLIW) processors. Furthermore, many processors provide a set of instructions that work on several data words concurrently (single instruction multiple data, SIMD). The availability of reconfigurable logic platforms like field programmable gate arrays (FPGAs) allows for implementation of custom hardware accelerators. In contrast to the general purpose architectures, these systems are usually application-specific, that is, they can only execute a single program or are only programmable to a very limited extent.
The rising importance of time-to-market considerations and time-consuming and error-prone manual implementations of special-purpose hardware accelerators motivates the need for high level synthesis tools. The major aim of such tools is to automatically generate the dedicated accelerators subject to performance and cost constraints from their corresponding algorithm description. The existing transformations in many such tools are based on compilation approaches from the DSP world. Loop unrolling is a major optimization transformation which unsheathes the data level parallelism in a loop program. Loop unrolling by factor \( n \), expands the loop kernel by copying \( n \) consecutive iterations. Then, the mapping tools schedule and synthesize the unrolled dataflow graphs for generating the hardware accelerators. This also leads to a design space exploration problem because of different factors available for partial loop unrolling [1].

Another approach for generating special class of hardware accelerators called VLSI processor arrays for computation intensive algorithms borrows ideas from parallelization in the polytope model [4, 11] and the generation of systolic architectures. These architectures consist of a regular arrangement of rather simple processor elements. The projection is an important transformation in this approach for obtaining full-size processor array descriptions from a given nested loop program. This corresponds to full loop unrolling in terms of resource usage. Partitioning is another necessary transformation for mapping loops onto reduced-size arrays in order to meet the resource constraints. Well known partitioning techniques are tiling and clustering [16]. Partitioning corresponds to partial loop unrolling.

In this paper, we investigate and compare the two different methodologies of loop unrolling from the high level synthesis approach and loop partitioning (processor array approach). In the next section, a brief overview of related work and state-of-the-art in hardware synthesis is presented. In Section 3, the PARO designflow and tool is introduced. Subsequently in Section 4, a concise description of the two loop transformations, that is, loop unrolling and partitioning and arising challenges is given. In Section 5, the quantitative analysis of the transformations with help of benchmarks is given. Finally in Section 6, a summary and outlook of the work is presented.

2 Related Work

While software compilers are used in every day life by engineers, there is only a few existing work in the field of automatic hardware synthesis from a given high level algorithm description. Commercial examples of high level design systems are Catapult-C from Mentor Graphics [12], Cynthesizer from Forte Design Systems [5], Handel-C from Celoxica [2], or PICO Express from Synfora [13, 14]. All mentioned design tools start from a subset of sequential C, C++, or SystemC code. Starting with sequential languages has the disadvantage that their semantics force a lot of restrictions on the execution order of the program and language subset. Furthermore, most existing tools do not allow high level program transformations in order to match the input program to given architecture constraints (like maximum available memory or I/O bandwidth), or only to a limited extend. Loop unrolling has been studied in detail with respect to resource usage and performance in [1].

Apart from commercial systems, there exist some academic projects. Many of those projects try to avoid the restrictions of sequential languages by using different pro-
gramming and execution models. Examples of such systems are Compaan [9], which deals with process networks, the SPARK environment [6], or the MMalpha system [15], which is based on loop parallelization in the polytope model [4, 11] similar to our own approach.

3 Design Flow Trajectory

PARO is a design system project for modeling, transformation, optimization, and hardware synthesis for the class of computationally tensive loop programs. The goal of the project is the automatic synthesis of dedicated, massively parallel processor arrays, which may be implemented on fine-grained reconfigurable platforms like FPGAs or custom application-specific integrated circuits (ASIC). The PARO design flow for application-specific processor arrays is depicted in Fig. 1. Starting point of the design flow is the new class of so called dynamic piecewise linear algorithms (DPLAs) [7]. This class of algorithms describes loop nests with a few run-time conditionals as a set of recurrence equations. A given DPLA is fed to the core of the PARO system, where first a set of equivalence preserving high-level transformations is applied. Examples of such transformations are:

- **Localization**, which replaces global data dependencies that usually introduce high hardware costs into local dependencies where data is propagated through the processor array.
– **Partitioning and Loop unrolling**, in order to match the algorithm to given hardware constraints like area and available memory or I/O bandwidth.
– **Loop perfectization** [17] to transform non-perfectly nested loop programs in perfectly nested loop programs.
– Well-known algorithmical optimizations like dead code elimination, common subexpression elimination, operator strength reduction, constant/variable propagation, and others.

During the transformation step, the program may be simulated in order to verify the algorithm and also the correctness of the transformations that were applied. Next space-time mapping is performed, that is, every computation in the algorithm is assigned a processor element and a functional unit within that processor element (space), as well as the start time for execution. A prerequisite of space-time mapping is a model of the target architecture, for example, which functional units are available and what are their execution times? Afterwards, the space-time mapped program is synthesized. The hardware synthesis step generates a completely platform and language independent register transfer level (RTL) description of the hardware. This RTL model is further optimized and, depending on the selected backend and target platform (e.g., FPGA type), finally converted into HDL code of choice.

### 4 Problem Statement

The synthesis of loop programs on hardware has been a deeply studied problem. The two different parallelization approaches for resource constrained mapping onto hardware are loop unrolling and partitioning (e.g., tiling and clustering). Fig. 2(a) shows the iteration space without data dependencies of a 4-tap FIR filter which is used to illustrate the fundamental difference between the two approaches. The pseudo-code of an \( N \)-tap filter follows on the next page.
FORALL (i >= 0 and i <= T-1) // T is the number of input samples
{ FORALL (j >= 0 and j <= N-1) // N is the number of weights
{ IF (i==0) THEN a[i,j] = a_in[j]; // read filter coefficient
ELSE a[i,j] = a[i-1,j];
IF (j==0) THEN
{ u[i,j] = u_in[i]; // read input sample
y[i,j] = a[i,j] * u[i,j];
}
ELSE
{ u[i,j] = u[i-1,j-1]; // enables data reuse
y[i,j] = y[i,j-1] + a[i,j] * u[i,j];
}
IF (j == N-1) y_out[i,j] = y[i,j]; // write output
}
}

Fig. 2(b) shows the loop unrolling approach, where the innermost loop is completely unrolled and mapped onto a processor element (PE) with 4 MUL and ADD units. The following is the pseudo-code of an N-tap FIR filter unrolled by factor of 2.

FORALL (i >= 0 and i <= T-1)
{ FORALL (j >= 0 and j <= N/2-1)
{ IF (i==0) THEN
{ a0[i,j] = a_in1[j];
a1[i,j] = a_in2[j];
}
ELSE
{ a0[i,j] = a0[i-1,j];
a1[i,j] = a1[i-1,j];
}
IF (j==0) THEN
{ u0[i,j] = u_in[i];
y0[i,j] = a0[i,j] * u0[i,j];
}
ELSE
{ u0[i,j] = u1[i-1,j-1];
y0[i,j] = y1[i,j-1] + a0[i,j] * u0[i,j];
}
{ u1[i,j] = u0[i-1,j];
y1[i,j] = y0[i,j] + a1[i,j] * u1[i,j];
IF (j == N-1) y_out[i,j] = y1[i,j];
}
}

The IF conditional in the pseudo-code is to enable data-reuse within the generated hardware. This helps combat the memory bottleneck for compute intensive applications.

Partitioning is a well known transformation which covers the index space of computation using congruent hyperplanes, hyperquaders, or parallelepipeds called tiles [3,16]. Well known partitioning techniques are multiprojection, LSGP (local sequential global parallel, often also referred as clustering or blocking) and LPGS (local parallel global sequential, also referred as tiling). Fig. 2(c) shows the partitioning approach with help
of clustering, where the iterations within the tile are processed in parallel by 4 PEs. Each PE contains one MUL and one ADD unit. The tiles are processed globally sequentially. The following is the pseudo-code for the partitioned FIR filter.

\[
\text{FORALL } (i \geq 0 \text{ and } i \leq T-1) \\
\{ \text{FORALL } (j \geq 0 \text{ and } j \leq N/2-1) \\
\{ \text{FORALL } (j1 \geq 0 \text{ and } j1 \leq 1) \\
\{ \text{IF } (i==0) \text{ THEN } a[i,j,j1] = a_{in}[j+j1]; \\
\quad \text{ELSE } a[i,j,j1] = a[i-1,j,j1]; \\
\quad \text{IF } (j==0) \text{ THEN} \\
\quad \{ u[i,j,j1] = u_{in}[i]; \\
\quad \quad y[i,j,j1] = a[i,j,j1] \times u[i,j,j1]; \\
\quad \text{ELSEIF } (j1==0) \text{ THEN} \\
\quad \{ u[i,j,j1] = u[i-1,j-1,j1+1]; \\
\quad \quad y[i,j,j1] = y[i,j-1,j1+1] + a[i,j,j1] \times u[i,j,j1]; \\
\quad \text{ELSE} \\
\quad \{ u[i,j,j1] = u[i,j-1,j1-1]; \\
\quad \quad y[i,j,j1] = y[i,j,j1-1] + a[i,j,j1] \times u[i,j,j1]; \\
\quad \} \\
\} \\
\} \\
\}
\]

The major question that needs to be answered on basis of several algorithms is:

- What is the quantitative trade-off in terms of hardware cost and performance between loop unrolling and partitioning?
- What should be the optimal granularity of resources in parallel processors for efficient mapping?

In the next section, we answer the above questions by quantitative analysis of reconfigurable hardware generated for the transformed loop programs by PARO design system. The resource constrained scheduling problem for the dataflow graph for both the variants is solved by the same mixed integer linear programming (MILP) approach \cite{8} which leads to minimal latency.

## 5 Experiments and Quantitative Evaluation

In this section, we compare the two methods described before with respect to resource usage and performance (clock frequency and throughput). All synthesis results were obtained from Xilinx ISE 8.2 on a Xilinx Virtex 4 FPGA (xc4vlx100-12ff1513). The usage of DSP48 slices, BRAMs, and resource sharing was disabled throughout all experiments excepting for the matrix multiplication where we allowed BRAMs for the storage of intermediate data.

In the first experiment a 8-bit 64-tap FIR filter is considered. The coefficients of the filter are reconfigurable, that means, they were implemented as inputs. The results are shown in Table 1, where the first column denotes the unroll factor $u$ in case of the
Table 1. Resource usage and performance of different 64-tap FIR filter implementations.

<table>
<thead>
<tr>
<th>( u )</th>
<th>HLS #PE ([\text{no.}])</th>
<th>HLS LUTs ([\text{no.}])</th>
<th>HLS FFs ([\text{no.}])</th>
<th>HLS Clock ([\text{MHz}])</th>
<th>HLS Throughput ([\text{MB/s}])</th>
<th>HLS LUTs Diff. ([\text{no.}])</th>
<th>HLS FFs Diff. ([\text{no.}])</th>
<th>HLS Clock Diff. ([\text{MHz}])</th>
<th>HLS Throughput Diff. ([\text{MB/s}])</th>
<th>Processor Array #PE ([\text{no.}])</th>
<th>Processor Array LUTs ([\text{no.}])</th>
<th>Processor Array FFs ([\text{no.}])</th>
<th>Processor Array Clock ([\text{MHz}])</th>
<th>Processor Array Throughput ([\text{MB/s}])</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>250</td>
<td>167</td>
<td>194</td>
<td>5.8</td>
<td>295</td>
<td>184</td>
<td>10.2</td>
<td>221</td>
<td>13.9</td>
<td>6.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>419</td>
<td>273</td>
<td>186</td>
<td>11.1</td>
<td>439</td>
<td>300</td>
<td>9.9</td>
<td>219</td>
<td>17.7</td>
<td>13.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>802</td>
<td>533</td>
<td>191</td>
<td>22.8</td>
<td>807</td>
<td>556</td>
<td>4.3</td>
<td>219</td>
<td>14.7</td>
<td>26.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1559</td>
<td>1051</td>
<td>199</td>
<td>47.4</td>
<td>1540</td>
<td>1064</td>
<td>1.2</td>
<td>220</td>
<td>10.6</td>
<td>52.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>3250</td>
<td>2192</td>
<td>178</td>
<td>84.9</td>
<td>3011</td>
<td>2092</td>
<td>-4.6</td>
<td>212</td>
<td>19.1</td>
<td>101.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>7189</td>
<td>4666</td>
<td>158</td>
<td>150.7</td>
<td>5895</td>
<td>4387</td>
<td>-6.0</td>
<td>255</td>
<td>61.4</td>
<td>243.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. The throughput for different unroll factors and numbers of processor elements, respectively, is shown in the upper half of the figure. In the lower half, the throughput is normalized by the gate count of the designs.

Standard high-level synthesis (HLS) approach and in case of a processor array implementation the number of processing elements. This number corresponds also to the total number of available multipliers and adders for the data-path implementation in both variants. The results for the HLS approach follow in the next four columns and for the processor array implementations in the last four columns of the table. Two columns depict the costs in terms of number of look-up tables (LUTs) and slice flip-flops (FFs), and the other two columns represent the performance metrics clock frequency and throughput. Next to each column (LUTs, FFs, and clock) of the processor array implementations, the relative difference compared with the HLS approach is given. Since the throughput is proportional to the clock frequency, the relative difference is the same and is omitted in the table. For the lowest resource usage \( (u = 2) \), the processor ar-
ray implementation is almost 14% faster than the unrolled variant but also for a higher price, 18% more LUTs and 10% more flip-flops. It can be noticed that for increasing \( u \), the clock frequency of the HLS approach is decreasing whereas for the processor array implementations it is almost constant except for the full size case (\( u = 64 \)) where it is considerably higher. This performance boost is because of the fact, that no loop control is needed anymore. Why the same argument does not apply for the HLS approach could only be speculated. It seems that the place and route routines do not perform so well for larger designs (flattened register-transfer circuits). Whereas, the clustering of operations into several processor elements performs much better in terms of clock speed. In Fig. 3, the throughput itself and normalized by the gate count is shown. Since, the difference for the \( u = 64 \) implementations was tremendous, several other fully unrolled/full size versions have been studied. The results are shown in Table 2 and Fig. 4.

Table 2. Fully unrolled and full size processor arrays, respectively, for different numbers of FIR filter taps.

<table>
<thead>
<tr>
<th>LUTs</th>
<th>FFs</th>
<th>Clock [MHz]</th>
<th>Throughput [MB/s]</th>
<th>LUTs Diff.</th>
<th>FFs Diff.</th>
<th>Clock Diff.</th>
<th>Throughput [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>747</td>
<td>528</td>
<td>188</td>
<td>179.3</td>
<td>748</td>
<td>0.1%</td>
<td>24.2</td>
</tr>
<tr>
<td>16</td>
<td>1513</td>
<td>1043</td>
<td>176</td>
<td>167.8</td>
<td>1484</td>
<td>-1.2%</td>
<td>24.1</td>
</tr>
<tr>
<td>24</td>
<td>2324</td>
<td>1588</td>
<td>175</td>
<td>166.9</td>
<td>2218</td>
<td>-4.6%</td>
<td>24.1</td>
</tr>
<tr>
<td>32</td>
<td>3182</td>
<td>2153</td>
<td>163</td>
<td>155.4</td>
<td>2953</td>
<td>-7.2%</td>
<td>24.1</td>
</tr>
<tr>
<td>48</td>
<td>5093</td>
<td>3338</td>
<td>165</td>
<td>157.4</td>
<td>4430</td>
<td>-13.0%</td>
<td>24.2</td>
</tr>
<tr>
<td>64</td>
<td>7189</td>
<td>4666</td>
<td>158</td>
<td>150.7</td>
<td>5895</td>
<td>-18.0%</td>
<td>24.2</td>
</tr>
</tbody>
</table>

Fig. 4. Throughput of different fully unrolled and full size FIR filter implementations.

As second algorithm a DCT width 8-bit I/O and internally up to 16 bits datapath was studied. In Table 3, the results for different numbers of available multipliers are shown. Unlike the FIR filter, the DCT has no loop carried data dependencies. The loop body of the DCT was unrolled four times and a processor array consisting of 4 processing elements was considered, respectively. The unrolled loop body contains 96 multiplications therefore the results of the last experiment where 96 multipliers were available are quite
Table 3. Resource usage and performance of different DCT implementations.

<table>
<thead>
<tr>
<th>no. of mult.</th>
<th>LUTs [no.]</th>
<th>FFs [no.]</th>
<th>Clock [MHz]</th>
<th>Throughput [MB/s]</th>
<th>LUTs Diff. [no.]</th>
<th>FFs Diff. [no.]</th>
<th>Clock Diff. [MHz]</th>
<th>Throughput Diff. [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4704</td>
<td>2012</td>
<td>139</td>
<td>50.5</td>
<td>4486</td>
<td>-4.6%</td>
<td>2045</td>
<td>1.6%</td>
</tr>
<tr>
<td>32</td>
<td>5305</td>
<td>3315</td>
<td>178</td>
<td>113.2</td>
<td>5056</td>
<td>-4.7%</td>
<td>3123</td>
<td>-5.8%</td>
</tr>
<tr>
<td>48</td>
<td>5421</td>
<td>4125</td>
<td>223</td>
<td>212.7</td>
<td>5127</td>
<td>-5.4%</td>
<td>3905</td>
<td>-5.3%</td>
</tr>
<tr>
<td>96</td>
<td>5045</td>
<td>5439</td>
<td>246</td>
<td>469.2</td>
<td>5034</td>
<td>-0.2%</td>
<td>5443</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

However, if the multipliers have to be shared by several operations, the processor array implementations achieve 11 to 21% higher throughput.

Lastly, an algorithm for the multiplication of two $64 \times 64$ matrices was considered. Using the processor array approach, again a higher clock frequency and throughput as compared with loop unrolling approach using the same number of resources (multipliers and adders) is achieved (see Table 4). The higher area costs in terms of LUTs for the processor array approach is due to control overhead caused by the partitioning onto several processor elements. 8 BRAMs are instantiated for data reuse for both the variants.

Table 4. Resource usage and performance of different matrix multiplication for $N = 64$ implementations.

<table>
<thead>
<tr>
<th>n</th>
<th>#PE [no.]</th>
<th>LUTs [no.]</th>
<th>FFs [no.]</th>
<th>Clock [MHz]</th>
<th>Throughput [MB/s]</th>
<th>Processor Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>784</td>
<td>474</td>
<td>148</td>
<td>9.2</td>
<td>895</td>
<td>14.2%</td>
</tr>
<tr>
<td>4</td>
<td>1459</td>
<td>812</td>
<td>145</td>
<td>18.2</td>
<td>1629</td>
<td>11.7%</td>
</tr>
<tr>
<td>8</td>
<td>3049</td>
<td>1724</td>
<td>123</td>
<td>30.8</td>
<td>3030</td>
<td>-0.1%</td>
</tr>
<tr>
<td>16</td>
<td>5937</td>
<td>3166</td>
<td>125</td>
<td>62.4</td>
<td>5895</td>
<td>-0.1%</td>
</tr>
</tbody>
</table>

6 Summary and Future Work

In existent behavioral synthesis tools, loop unrolling is applied in order to achieve higher throughput. Unique to the PARO system is the ability to consider both loop unrolling and processor array designs. The usage of the same design tool enabled, for the first time, a fair quantitative evaluation of the two approaches for a set of computational intensive algorithms. Due to its regularity and the clustering of resources into several processing elements, the processor array approach achieves in all experiments a far better throughput, up to 61% more compared with loop unrolling.

Furthermore, the PARO design flow is also capable of targeting a new class of reconfigurable architectures, the so-called weakly programmable processor arrays (WPPA) [10]. WPPAs are a generic architecture template consisting of a tightly coupled, reconfigurable VLIW processor arrays. In this context, we would like to study the trade-off between VLIW word size and number of processors. In the future, we would like to study more algorithms in order to automatically elaborate when loop unrolling and when the processor array approach should be applied. This is important to obtain for
small as well as for large problem instances and different requirements (throughput, cost, etc.) optimal designs.

References