ABSTRACT
We present an generator-based approach for efficient design space exploration and high-level synthesis of cryptographic designs. It allows the directed search for new algorithmic variants.

The approach combines high-level synthesis features like automated scheduling and allocation with performance considerations down to the register-level which are crucial for the performance of cryptographic designs.

We demonstrate the feasibility of this approach on an cryptographic algorithm based on elliptic curves.

KEY WORDS
Design Automation, Design Space Exploration, Variant Selection, Cryptography

1 Introduction
The increasing complexity of modern designs, together with reduced time-to-market, leads to the requirement of more and more automated processes and higher abstraction levels in hardware design.

In the recent past a trend towards High-Level Synthesis (HLS) approaches emerged. These approaches are based on high-level programming languages and abstract from a detailed hardware view. The low-level hardware implementation is capsuled inside an automated process and transparent to the designer.

We present a novel high-level design flow with the intention to optimize the process of variant selection with the help of genetic algorithms for design space exploration. This allows the identification of Pareto-optimal design variants. The design flow is focused on the cryptographic domain which constitutes several properties, thus influencing the design flow.

2 Related Work
The field of low-level synthesis is long known and widely researched. Commercial synthesis tools like Synplicity [15] work on concurrent Hardware Description Languages (e.g., VHDL). Other approaches, however, introduce own description languages to perform synthesis [11]. These low-level approaches leave scheduling and partitioning to the developer.

The structure of cryptographic algorithms allows a multitude of design variants. [14] presents different parameters to span up a design space and demonstrates that the selected design variant has significant impact on the obtained hardware implementation.

Most existing HLS design flows rely on C[6] or C-based dialects like Handel-C [1], Impulse C [13], SystemC [12] and many others. Variant selection is realized by manual interaction before starting the scheduling process.

This novel design flow addresses the problem of variant selection by the use of genetic algorithms for design space exploration. Interesting design variants are generated (i.e. selected) automatically and can be used for detailed implementations.

3 Design Flow
In figure 1 the design flow of existing HLS approaches is summarized on the left hand side. It is derived from the Behavioral Compiler from Synopsis [6], so other approaches may differ in details. On the right hand side the structure of the HLS is given, which will be detailed in this paper. After a short presentation of both design flow alternatives the detailed steps of the presented approach are addressed in the subsequent sections.

Both design flows start with a design entry step. We use a Control Dataflow Graph (CDFG) as the description means, to prevent the potential loss of information about parallelism as described in section 3.1. In the next step the design has to be partitioned into separate hardware modules. These modules will be handled as atomic blocks in the subsequent scheduling process. Depending on the design flow they either be represented with code annotations or by instantiating components for each block. Section 3.2 describes briefly how such a partitioning can be found.

After the initial partitioning of the code further constraints and annotations have to be given in classical HLS. These directives select one variant out of the existing design space. This design variant will then be used in the following hardware generation. If the result does not meet the requirements, another design variant is selected in an
We present an alternative approach using a genetic algorithm. It allows the parallel evaluation of a large set of design variants because it tries to find and to generate a set of Pareto-optimal design alternatives. Out of this set the preferred solution can be selected as described in section 4. An iterative step to select another design variant, like it is depicted in the left hand of figure 1 is not necessary as all relevant design variants will be present in the Pareto-set.

For the selected variant synthesizable VHDL code is generated automatically. The generator has a special focus on efficiency, which is crucial for cryptographic engines. In section 4.2 some related implementation details and results are presented.

### 3.1 Design Entry

One of the main tasks of HLS is the scheduling and allocation of operations and resources respectively. In other words, the operations are placed in a fixed, sequential order and mapped onto corresponding resources. The more knowledge about inherent parallelism of the original specification is available in this step, the better results can be obtained.

Most HLS approaches for hardware design are based on sequential high-level programming languages. This requires the analysis of the sequential code to extract information about parallel structures. To the current day, there is no satisfying methodology for this automated extraction of parallelism.

To circumvent any potential loss of information in this analysis step, we advocate a Control Data Flow Graph for design entry such that the graph-based models do not introduce an implicit sequential order of operations. For a more generalized domain, the representation of complex specifications in form of a CDFG might be difficult. This is not the case here, because of the simple and clear structure of algorithms in the cryptographic domain we refer to.

### 3.2 Partitioning

In the next step the partitioning of code segments into functional blocks has to be determined. These blocks will serve as basic elements for the subsequent scheduling.

For the performance of any HLS process, the number of basic operations is crucial. It depends directly on the granularity of the basic operations. So, the designer has to find a partition with an appropriate granularity.

A fine-granular approach would, in theory, allow optimal resource utilization. However, the large number of basic operations spans up a design space, which is much too large for complex design space exploration or scheduling.

For HLS a higher abstraction level is required. This allows a smaller number of operations and therefore the use of more complex and dedicated algorithms.

In [5] the performance of the exploited genetic algorithm has been analyzed. It has been shown that the algorithm provides acceptable performance up to several hundred operations or tasks (e.g., 200 sec for 200 tasks). So, this step of the design flow has to find a partitioning with a granularity, detailed enough to benefit from scheduling on the one hand, and abstract enough to allow the execution of genetic algorithms in realistic runtimes, on the other hand.

As cryptographic algorithms consist of complex, non-standard operations, a library-based approach is preferred over annotation-based partitioning. This allows an easy replacement and encapsulation of cryptographic operations when the implementation of these operations may change.

This library-based approach allows a top-down structure of the partitioning algorithm. It starts with the implementation of a high-level algorithm. Sub-procedures will be specified step-wise until the optimal granularity is reached. All remaining operations are regarded as library components and have to be provided by the developer (e.g. as netlists).

Algorithm 1 denotes the steps how to create a suitable partition. It aims towards a library-mapping, such that the target algorithm needs just about 10 to 500 calls to basic operations.

The partitioning process is based on a simple manual interaction by the developer. Especially the selection of
loop body. This is required, if the number of operations in the unrolled design variant may exceed the capabilities of the genetic algorithm. The outer loop can then be handled in a second HLS flow, where the first design serves as a library component.

For this work we will focus on the loop body of EC-Mult to illustrate the features and advantages of this new design flow. In the second step of the partitioning the sub-functions of the target algorithm will be unfolded until an appropriate complexity is reached. We unfold MAdd and MDouble with the algorithms 3 and 4. This leaves the operations \(mult_{ff}, add_{ff}, \) and \(square_{ff}\) as basic operations, which have to be provided as library components.

### Algorithm 3 MAdd

**Input:** EC point \((x_0, z_0)\).

**Output:** EC point \((x, z)\).

```plaintext
1: \(x := square_{ff}(x_0)\)
2: \(ta := square_{ff}(z_0)\)
3: \(z := mult_{ff}(ta, x)\)
4: \(ta := mult_{ff}(ta, x)\)
5: \(x := square_{ff}(x)\)
6: \(x := add_{ff}(x, ta)\)
7: return \((x, z)\)
```

Beside the loop-statement, the if-statement requires a special handling as a control-flow operation. Here, both branches perform the same operations, only the order of operands is modified. This allows the encapsulation of the control flow inside an additional Decider component.

Figure 2 presents the resulting Data Flow Graph for the ECC 2P-Algorithm. In addition, a list of available resources, their execution times, and the resource graph are defined. Due to the limited space they are not detailed in this paper.

### 4 Design Space Exploration and Variant Selection

The next synthesis step is an exhaustive design space exploration to find suitable design variants. These variants differ in the allocation of resources (e.g., how many hardware multipliers are available), binding (on which of the

---

**Algorithm 1 Partitioning**

**Input:** An Algorithmic Specification \(f\)

**Output:** An Partitioned Algorithm

1: repeat
2: \[\textbf{while} \ f \text{ contains unrolled loops do} \]
3: \[\text{Unroll}_1 \text{ Loop or Step}_1 \text{ Inside} \text{ Loop} \]
4: \[\textbf{end while} \]
5: \(n := \text{Number of Base Operations} \)
6: \[\text{Choose an arbitrary sub-function } f' \in f \]
7: \[\textbf{if} \ n < 10 \text{ then} \]
8: \[\text{Unfold } f' \]
9: \[\textbf{else} \]
10: \[\text{Step Inside } f' \text{ (Set } f := f') \]
11: \[\textbf{end if} \]
12: until \(10 < n < 500 \)

---

**Algorithm 2 ECMult**

**Input:** EC point \((x, y)\) and a scalar \(k\) of bitwidth \(l\).

**Output:** EC point \((x, y)\).

```plaintext
1: \(x_1 := x; z_1 := 1; x_2 := x^4 + \text{const.}; z_2 := x^2\)
2: for \(i = l - 1 \text{ downto } 0 \) do
3: \[\textbf{if } k_i = 1 \text{ then} \]
4: \( (x_1, z_1) := MAdd(x_1, z_1, x_2, z_2) \)
5: \( (x_2, z_2) := MD\text{ouble}(x_2, z_2) \)
6: \[\textbf{else} \]
7: \( (x_2, z_2) := MAdd(x_2, z_2, x_1, z_1) \)
8: \( (x_1, z_1) := MD\text{ouble}(x_1, z_1) \)
9: \[\textbf{end if} \]
10: \[\textbf{end for} \]
11: \[\textbf{return} \ \(x, y) = \text{MAffine}(x_1, z_1, x_2, z_2) \]
```

An important element of the partitioning operation is the handling of loops. This work is limited to static loops (i.e. loops which are performed a constant and predefined number of rounds), which allow loop unrolling. For cryptographic algorithms a support of dynamic loops is not required, an important domain-specific property.

An alternative approach would be to step inside the loop body. This is required, if the number of operations in the unrolled design variant may exceed the capabilities of the genetic algorithm. The outer loop can then be handled in a second HLS flow, where the first design serves as a library component.

---

**Algorithm 3 MAdd**

**Input:** EC points \((x_0, z_0), (x_1, z_1)\).

**Output:** EC point \((x, z)\).

```plaintext
1: \(x := mult_{ff}(x_0, z_1)\)
2: \(z := mult_{ff}(z_0, z_1)\)
3: \(ta := mult_{ff}(x, z)\)
4: \(z := add_{ff}(z, x)\)
5: \(z := square_{ff}(z)\)
6: \(x := mult_{ff}(z, \text{const.})\)
7: \(x := add_{ff}(x, ta)\)
8: \[\text{return} \ (x, z) \]
```
resources should two operands be multiplied), and scheduling (in which order should the operations take place). The variants span the available design space.

The design space does not contain just one unique optimal solution, but a set of so-called Pareto-optimal solutions. All solutions in this set are better than any other solution in at least one optimization criterion (such as runtime or resource allocation). In order to obtain good implementation results the selected design variant has to be part of the Pareto-set.

Classical HLS approaches require the developer to select a design variant very early in the design flow. A list of Pareto-optimal solutions is not available and the variant selection is thus based on personal experience only.

An exploration of multiple design variants to evaluate the quality of the selection would therefore be valuable. However, although this process may be supported by some extend by a tool, it requires a substantial effort.

We therefore propose an approach for variant selection based on genetic algorithms. Genetic algorithms work on a whole generation of solutions simultaneously. They do not only produce one solution, but generate a whole set of possible solutions. If the quality of parameters for the genetic algorithm are sufficient, these solutions will be part of, or at least close to the set of Pareto-optimal solutions and an automated design space exploration has thus been performed. Figure 3 outlines the proposed exploration flow. [8] gives detailed descriptions of the implemented genetic algorithms.

Now, the developer can choose his preferred design variant out of the generated solution set. In addition, he also has an overview of other design alternatives that might be interesting. As another advantage, genetic algorithms allow the simple extension in terms of additional design criteria. The current version uses runtime and resource usage as optimization criteria. In [4] this additional expandability has been demonstrated for the consideration of dependability. Due to the large bitwidth of cryptographic cores, the available memory bandwidth turned out to be an important design optimization criterion as well.

In the current version the bandwidth is not modeled as a dedicated optimization objective. Instead, a limited bandwidth is modeled by extending the DFG from figure 2 by memory access nodes on each edge. Each node represents one memory access with a certain bitwidth. By allocating multiple resources, the selection of an arbitrary bandwidth is realized. The genetic scheduling will then find solutions with the minimal number of required resources, including all memory access resources.

### 4.1 Synthesis Results

In table 1 the results of the design space exploration for the algorithm of figure 2 are given. It lists the execution time and costs of all Pareto-optimal design variants found by the genetic algorithm. Execution time is given in clock cycles and resource costs are obtained from the resource requirements of all allocated library elements (\(\text{square}\), \(\text{mult}\), and \(\text{add}\)). The table was computed with no limitation for the memory bandwidth. Notice that the original DFG with 14 tasks already leads to 8 different design variants.

To illustrate the impact of memory bandwidth we created another version with an explicit bandwidth allocation of one operand per clock cycle. In this case the Pareto-set was reduced to only two variants. Both allocate one operand to each memory access.

![Figure 3. Outline of the genetic Design Space Exploration](image-url)
add_ff and one square_ff, but the first variant allocates one mult_ff and the second two mult_ff modules. The allocation of additional resources does not result in additional Pareto-solutions due to the limited bandwidth.

For the following steps we selected two design variants with limited memory access. The figures 4 and 5 present their (slightly simplified) operation scheduling. The limited bandwidth forces the introduction of idle cycles as observable between mult1 and mult3 in figure 4. They are required to store the result into the memory and get it back. Please note that this is only necessary when the result is required as an operand in a direct subsequent operation. In other cases, as between mult2 and mult1, the operands for mult1 can be fetched in parallel to the execution of mult2.

4.2 Implementation Results

In order to complete the design flow a VHDL-based code-generator has been developed which generates a synthesizable top-level controller for a selected design variant. Special care was taken with respect to the performance of the generated code.

The generated designs according to figure 4 and 5 have been compared to existing hand optimized implementations. The arithmetic operations (mult_ff, add_ff, and square_ff) have been extracted and are utilized as library components for this work. So, all performance differences in the results can be traced back to the top-level controller which is the main focus of this work.

Table 2 presents some of the obtained results. The first two columns ([3] and [10]) refer to implementations, which were developed without high-level tools. The results in [3] refer to the Xilinx XC4085XLA hardware platform, while [10] refers to the Xilinx Virtex2-Pro 7 which was utilized in this work as well. The remaining three columns refer to different design variants obtained by means of this work.

4.2.1 Implementation Results

<table>
<thead>
<tr>
<th>Design Variant</th>
<th>Execution Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>34</td>
<td>152</td>
</tr>
<tr>
<td>v1</td>
<td>35</td>
<td>149</td>
</tr>
<tr>
<td>v2</td>
<td>36</td>
<td>139</td>
</tr>
<tr>
<td>v3</td>
<td>37</td>
<td>138</td>
</tr>
<tr>
<td>v4</td>
<td>38</td>
<td>136</td>
</tr>
<tr>
<td>v5</td>
<td>42</td>
<td>133</td>
</tr>
<tr>
<td>v6</td>
<td>59</td>
<td>123</td>
</tr>
<tr>
<td>v7</td>
<td>61</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 1. Design Variants

The execution time of MDouble and MAdd (alg. 3 and 4) was reduced from 68 to 60 clock cycles. This is justified by the parallel utilization of add_ff and square_ff with mult_ff operations. This has been avoided in the manual implementation, as it significantly increases potential scheduling errors in the top-level controller and therefore the time for debugging and evaluation. These numbers demonstrate the quality of the presented design space exploration and code generation as no additional overhead in terms of clock cycles is introduced. The achievable clock frequencies are basically the same for both, the hand implemented and the generated design as the critical path delays lay inside library components ([3] is implemented on a different hardware platform and therefore not comparable in this aspect).

The first code generator exploits a register based approach to save intermediate data. This allows full parallel operand access without limiting memory bandwidth. The number of registers was minimized using the Left-Edge algorithm [7]. When looking at the implementation results this approach turned out to require significantly more logic resources compared to the equivalent hand optimized design in [10] (70% vs. 46%).

The additional resources were identified in the register-based top-level controller (which required 33% out of the 70% percent). To reduce the size of this controller we modified the code generator to use a RAM based approach for the storage of intermediate data. This significantly re-
<table>
<thead>
<tr>
<th>Design</th>
<th>[3]</th>
<th>[10]</th>
<th>Register Based (1 (mult_{ff}))</th>
<th>RAM Based (2 (mult_{ff}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(mult_{ff})</td>
<td>8 cycles</td>
<td>8 cycles</td>
<td>7 cycles</td>
<td>7 cycles</td>
</tr>
<tr>
<td>MDouble / MAdd</td>
<td>68 cycles</td>
<td>68 cycles</td>
<td>58 cycles</td>
<td>60 cycles</td>
</tr>
<tr>
<td>Frequency</td>
<td>33 MHz</td>
<td>114 MHz</td>
<td>114 MHz</td>
<td>117 MHz</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>n/a</td>
<td>11%</td>
<td>37% (3744)</td>
<td>11% (1291)</td>
</tr>
<tr>
<td>Logic</td>
<td>76%</td>
<td>46%</td>
<td>70%</td>
<td>35%</td>
</tr>
<tr>
<td>Top-Level</td>
<td>33%</td>
<td>5%</td>
<td></td>
<td>9%</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>37%</td>
<td>30%</td>
<td>67%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Implementation Results

Produced the required logic in the top-level controller (from 33% to just 5%) and the overall logic requirement from 46% to 35% compared to the manual implementation. The performance is reduced slightly from 58 to 60 clock cycles since the instantiated RAM can not provide the bandwidth of multiple registers. So two cycles for memory access had to be inserted as visible from figure 5.

5 Conclusion

We presented a novel and domain specific design flow for high-level synthesis, HLS which is based on genetic algorithms. It is aimed towards the special requirements and constraints of the cryptographic domain.

This approach allows an efficient design space exploration. The developer does not need to find an optimal design variant by himself, but he or she can select his/her preferred variant out of the generated set of Pareto-optimal solutions.

Alternative design variants can comfortably be selected and the corresponding top-level controllers may be generated within minutes. In case of a classical HLS this requires the modification of algorithms and manual code annotations and takes thus much more time.

Compared to any manual implementation the automated process with its code generation is much less error-prone. Especially the concurrent execution of operations bears the potential for scheduling errors which can be eliminated by means of the proposed design flow.

References


