Programming Highly Parallel Reconfigurable Architectures for Public-Key Cryptographic Applications

Abstract—Tiled architectures are emerging as an architectural platform that allows high levels of instruction level parallelism. Traditional compiler parallelization techniques are usually employed to generate programs for these architectures. However, for specific application domains, the compiler is not able to effectively exploit the domain knowledge. In this paper, we propose a new programming model that, by means of the definition of software function units, allows domain-specific features to be explicitly modeled, achieving good performances while reducing development times with respect to low-level programming. Identity-based cryptographic algorithms are known to be computationally intensive and difficult to parallelize automatically. Recent advances have led to the adoption of embedded cryptographic coprocessors to speed up both traditional and identity-based public key algorithms. Custom-designed coprocessors have high development costs and times with respect to general purpose or DSP coprocessors. Therefore, the proposed methodology can be effectively employed to reduce time to market while preserving performances. It also represents a starting point for the definition of cryptography-oriented programming languages.

Keywords: identity-based cryptography, tiled architectures, parallel programming model, reconfigurable architectures, multiobjective exploration.

I. INTRODUCTION

Since traditional microprocessors are becoming increasingly complex, leading to high design and manufacturing costs, new trends in architectures are moving towards partitioned register file architectures, such as tiled architectures, which allow high levels of instruction level parallelism combined with good scaling properties. These architectures are currently considered for both general purpose and DSP applications.

The public key cryptographic algorithms are computationally intensive, so that the current research trend is oriented towards the adoption of application specific coprocessors, often based on reconfigurable hardware, to reduce time to market.

DSP-oriented tiled architectures could be used to obtain further improvements in time to market, cost and performance, provided that the parallel pipelines can be exploited intensively to limit the hardware area. To this end, new programming models are required, because standard compiler techniques are not able to extract parallelism from these algorithms at both task and instruction level.

In this paper, we propose a new programming model that, by means of the definition of software function units, allows domain-specific features to be explicitly modeled, achieving good performances while reducing development time with respect to low-level programming. We show the effectiveness of the proposed programming model by applying it to the case of computationally intensive cryptographic pairings, which are common in modern public key algorithms.

A cryptographic pairing is a bilinear map between two groups $G_1, G_2$ in which the discrete logarithm problem is hard.

$$\langle \langle G_1 \times G_1 \rightarrow G_2$$

Let $P, Q, R \in G_1$ then

$$t(P + R, Q) = t(P, Q)t(R, Q)$$
$$t(P, Q + R) = t(P, Q)t(P, R)$$

During the last few years, pairings have been successfully employed in order to resolve several open problems in cryptography such as, one-round three-way key exchange [15], identity-based encryption [6], and short digital signatures [7]. For further deepenings on the protocols that make use of pairing primitives we send back to [11], [16] and their further references. The Weil and Tate pairings on elliptic curves over finite fields represents the mathematical basics to construct identity-based cryptographic primitives. These pairings are bilinear maps from an elliptic curve group $E(F_q)$ to the multiplicative group of some extension field $F_{q^k}$. The parameter $k$ is called the embedding degree of the elliptic curve $[4], [14]$. The pairing is considered to be secure if taking discrete logarithms in the groups $E(F_q)$ and $E(F_{q^k})$ are both computationally infeasible. For optimal performance, the parameters $q$ and $k$ should be chosen so that the two discrete logarithm problems are of approximately equal difficulty when using the best known algorithms, with the order of $\#E(F_q)$ having a large prime factor $r$. The best attack known on the elliptic curve discrete logarithm problem is the parallel collision search that improves on the Pollard’s $\rho$-algorithm [34]. A pairing is considered as secure as $1024$-RSA, when $r \sim 2^{160}$, $k$ ranges from 2 to 10, depending on the application and $p^k \geq 2^{1024}$. In the wake of recent works [2], [13], [28] on pairings over general curves over pairing friendly fields of large prime characteristic, the proposed programming model will be aimed to the implementation of the Tate pairing primitive in characteristic $p$ with $k = 2$ and $p \sim 2^{512}$. The current algorithm to compute the pairing is a careful refinement of the well known BKLS/GHS algorithms as described in [4], [14], [29]. The cryptographic usage of the Tate pairing involves the application of Miller’s Algorithm [22] followed by a final exponentiation. The point
$P$ is chosen as an element of $E(\mathbb{F}_p)$ with order $r$. The point $Q$ is chosen as an element of $E(\mathbb{F}_{p^k})$ which is mapped from the twisted curve. Miller’s algorithm uses the double and add schema for elliptic curve point multiplication $rP$, with some more operations to evaluate intermediate values of the pairing that are multiplicatively accumulated to compute the output of the algorithm [27]. Miller’s algorithm performs $\lceil \log_2 r \rceil - 1$ iterations executing almost always the block of operations corresponding to a point doubling. Indeed, if a low hamming weight $r$ is used then only a few point additions will be required (e.g. 1-10). The core idea behind this work is to investigate ways to combine instruction-level parallelism that can be found in the implementation of multiprecision arithmetic operations with task-level parallelism among the finite field operations involved in the computation of pairings.

The rest of this paper is organized as follows. Section II introduces tiled architectures and their interconnection structure. Section III outlines the proposed programming model. Section IV provides an experimental evaluation of the proposed programming model. Finally, Section V draws the conclusions and suggests future research directions.

II. Tiled Architectures

Recent trends in microprocessor design are moving towards partitioning processor resources such as register files, cache banks and pipelines. In Very Long Instruction Word (VLIW) architectures, a single program counter controls several pipelines that access the same register file. However, this structure does not scale well, since large register files are impractical. Tiled architectures, such as Raw [33], Wavescalar [31] and TRIPS [26], represent an evolution of VLIWs, partitioning the register file so that each pipeline or cluster of pipelines (called a tile, a computational node or simply a node) can access a private register bank. While this allows smooth scaling, it poses communication problems, as data need be moved among the different pipelines, moreover, since the register file is partitioned, communication must take place on an interconnect network, called a scalar operand network [32]. These issues must be dealt with by the compiler, which is in charge of scheduling instructions not only in time, but also in space – that is across different nodes. Explicit communication instructions must be issued to synchronize the register file partitions.

Tiled architectures aim at addressing critical problems in high performance processor design, especially design complexity and manufacturing fault rates, by replacing complex processors with smaller and simpler replicated processing elements. The typical applications range from general purpose (for high-end tiled architectures with private data caches for each node) to DSP (for more compact designs, with centralized data cache or streaming data access). Tiled architecture fill a niche between the static general purpose and DSP processors, and the FPGA-based reconfigurable systems. They often expose a degree of reconfigurability in the scalar operand network, allowing the communications among clusters to be tailored to suit the application.

In this work, we focus on DSP-oriented tiled coprocessors with a single control flow, since they are the direct competitor of the FPGA and ASIC solutions for public key cryptographic algorithms. More complex nodes, such as those of Raw (a MIPS pipeline with private data and instruction caches) would be orders of magnitude larger and more costly than the industry standard solutions.

A tiled architecture is an array of nodes, where each node is a computing element accessing its own register file and exposing a set of private function units. When all the nodes have the same type of function units, the architecture is homogeneous, and heterogeneous otherwise. The migration of the operands among clusters is demanded to a word-level communication network and is controlled by special instructions – like and or cv – executed by the nodes themselves, or by dedicated hardware. This kind of architectures belongs to the family of Scalar Operand Networks (SON), and can be characterized by the AsTrO taxonomy [32], which specifies whether the assignment of the instructions, the transport of the operands and the ordering of the instructions are statically or dynamically performed.

DSPFabric [8], by STmicroelectronics, is a tiled architecture specifically designed for modulo scheduling computationally intensive loops of multimedia applications. With respect to the AsTrO taxonomy, it is a Static-Static-Static SONs, which means that the assignment of the instructions, the displacement of the copies and the scheduling passes are compiler tasks.

Moreover, DSPFabric is characterized by coarse-grained reconfigurable data-paths. The compiler must select a subset of feasible node connections for data flowing, and emits at compile time the reconfiguration instructions that activate the selected wires. These reconfiguration instructions change at runtime the network topology, tailoring it to the specific code.

The reconfiguration space – the space of feasible topologies – is tailored by the constraints given by the availability of I/O ports with respect to the total number of connecting wires. In the DSPFabric organization, each node can be potentially connected to all the others, exploiting a hierarchical interconnection schema, based on different levels of MUXes. Effective limitations are given by the MUXes capacity. We describe in the following the DSPFabric architecture, focusing the attention on the structure of the interconnections.

A. DSPFabric Architecture

Figure 1 gives an overall picture of a 64 nodes DSPFabric coprocessor. At level 0 it can be seen as an array of four 16-issue processors (clusters), communicating through a collection of multiplexers, which implements a multi input/output switch. Each cluster has $N$ input wires and $N$ output wires, where the output wires are possibly connected to all the others. On the contrary, the input wires can be connected to only one source. Figure 2 shows a feasible data path at level 0, assuming $N$ equal to 4.

At level 1, the spatial structure replicates itself inside each cluster, again with an array of 4-issue processing elements, connected together by multiplexers with capacity $M$. The last
level is composed by the computation nodes connected through a reconfigurable crossbar, which takes as input the internal connections and $K$ of the wires outgoing from level 1. Each computation node has two ingoing wires and one outgoing wire.

The computation nodes are single issue pipelined processors, accessing their own register file and functional units. Since DSPFabric has been specifically designed as a loop accelerator coprocessor for multimedia applications, each node is equipped with hardware features for better executing modulo scheduled code [23]; e.g., the node contains support for instruction predication and rotating registers. Precisely, the application is scheduled using the Kernel Only Modulo Scheduling [23] technique, which fully predicates loop prologue and epilogue. Thus, branches are not allowed and the execution is controlled by a cyclic program counter.

The copies between different register files are controlled by the receive primitive executed by the destination node. Two regions of the register file are organized as input buffers, which push on top the incoming values, but can be read randomly by the receiver.

The coupling with the main memory subsystem is demanded to a programmable DMA. Each node can generate an address request, which is directly sent to DMA without consuming inter-clusters communication patterns. Only a limited number of requests can be served at the same time, i.e. 8 requests, thus the compiler must ensure that the amount of simultaneous requests does not exceed that limit. Since the memory requests have no unary latency, the DMA engine provides input and output FIFOs – of depth equal to the serving time – for handling high memory pressure. When a value is ready it is directly loaded in the requesting cluster register file.

## III. Programming Model & Compiler Techniques

In this Section, we discuss the limitations of the compiler techniques for scheduling the target algorithms on tiled architectures, and propose a new programming model to deal with these issues. We apply the proposed programming model to the case of the Tate pairing computation.

### A. Compiler Techniques for Tiled Architectures

Tiled architectures are specifically designed for the execution of computationally intensive kernels of multimedia architecture. A typical scenario is to employ such machines as innermost loop accelerators – implemented as coprocessors and coupled with the central processing system.

Multimedia applications spend most of their execution time in few kernel algorithms, i.e. Inverse Cosine Discrete Transform, interpolation and deblocking filters. These loops are characterized by largely independent operations and low memory aliasing, exposing a high degree of potential Instruction Level Parallelism (ILP). Moreover these kernels are usually quite small – in the range from 100 to 1000 instructions in the loop body.

The compiler is typically driven by in-code pragmas, which select the loops to map onto the multiclustered coprocessor. As intermediate representation the loop is described by its Data Dependency Graph (DDG), where each node represents a native instruction and each edge introduces a data dependence between instructions.

The behaviour of the compiler back end is to assign the instructions to the clusters and to schedule them, compatibly with the communication net topology, the data dependencies and the resource constraints. The compiler tries to extract the maximum degree of parallelism and, at the same time, to limit the penalties due to explicit inter-cluster operand copies. Different approaches have been proposed for performing cluster assignment and scheduling, considering both 2-phases and unified techniques [9], [10], [12], [20], [21].

Since these architectures are conceived for loop acceleration, they typically provide hardware features to enhance Modulo Scheduling [23] compiler technique, like support for predicated execution and rotating registers [24].
Cryptographic algorithms that use multi-precision integer arithmetic are representative of a class of applications that present peculiar properties in terms of available parallelism and program structure. Specifically, computationally intensive public key cryptographic algorithms such as the Tate pairing implementation in [3], [19], [30] can be parallelized at task level (TLP), as proven by a wide range of literature on the design of hardware implementations that typically use replicated modular arithmetic circuits to exploit this type of parallelism [17], [18]. The design of the individual modular arithmetic circuits highlights the availability of a significant amount of instruction-level parallelism (ILP): the parallel operations in hardware can be transposed to parallel instructions in software implementation. On the other hand, loop-level parallelism (LLP), that is the opportunity to perform different iterations of the same cycle on different computational elements, is less easily found in this type of application, due to the need to propagate loop carried data dependencies (such as the carry propagation for the integer or mod p arithmetic) across the iterations of a given loop. Since LLP is the type of parallelism most easily exploited by compilers, while TLP is especially difficult to extract by means of a compiler, these algorithms prove particularly difficult to parallelize automatically.

To tackle this issue, our method highlights TLP and ILP in the target algorithms, by mirroring typical hardware design concepts, such as specialized arithmetic hardware. Specifically, in the proposed model, the target algorithm is written using a library of software components that perform the same operations as specialized hardware function units for multi-precision integer arithmetic. The code software function units are optimized for the target architecture, customizing the connections between tiles of the architecture to fit their data propagation schemata. Since carry propagation flows one-way from the least significant word to the most significant one, it makes for a very regular structure that can be easily mapped to the configurable connections between computational nodes, as each node needs to synchronize only with its neighbours.

Each software functional units is, on a given target architecture, characterized by two parameters: the schedule length and the resource usage, in terms of number of computational nodes. This characterization mirrors closely the area and latency parameters of an hardware functional unit. Therefore, a top-down approach can be used, applying well-known methodologies for the design of the controller datapath. In this way,
the high-level representation of the algorithm is mapped to the software functional units by means of a list-based scheduling algorithm [5].

C. Case Study: Modular Arithmetics

The goal of this Section is to describe the design of a basic multiprecision arithmetic library. The Montgomery multiplier is the main element of any such library. To this end, we need to first develop basic function units such as the modular adder and the word-by-vector multiplication, with the aim of composing them to implement the main loop of the Montgomery multiplier as described in Algorithm III.1.

Table I shows the basic schema for a modular adder. Each column of the table represent the schedule of a single computational node. For each word of the multi-precision operands to add, a pair of nodes is used to speculatively execute both the case with carry and with no carry. The table considers the case of only three words multiprecision operands, but the extension to larger sizes is straightforward.

![Figure 3: Time/space scheduling of a 128-bit modular addition](image)

Figure 3 shows how the adder can be further optimized to reduce resource usage: the modular adder unit is shown on the left, while on the right the pipelined operations have been compacted onto 8 computational nodes only, without penalty for the performance. This kind of optimization, while have been demonstrated only for a 128-bit modular adder, seamlessly scales to larger input sizes, requiring only 8 nodes and 2n + 6 clock cycles, where n is the number of words of the input.

Table II provides an implementation of the basic 32 bit multiplier unit using 16 bit multipliers provided in the target architecture. The word-by-vector multiplication is obtained by juxtaposing 32 bit multipliers, followed by a multiprecision (non modular) adder unit that handles carries. This method of obtaining larger units by composing smaller ones is fully developed in the generation of hierarchic software function units: an adder and the word-by-vector multiplier are used to design the Montgomery multiplier.

The Montgomery multiplier is based on the core loop shown in Algorithm III.1, where A and B are the input operands, while N is the modulus, w is the size of the word, b = 2^w and N_0 is the least significant word of the modular inverse of N, modulo the Montgomery radix. In this implementation, the number of iterations performed is n + 2 to bound the result in the range [n, 2n] for multiplicands up to 2n. This is achieved by eliminating the final subtraction in the original Montgomery algorithm and, as a consequence, after the inputs are converted in the Montgomery domain, the operations of the high-level algorithm are all performed therein.

![Algorithm III.1: Montgomery multiplier core loop](image)

Note that the composition of the larger function unit takes into account the shape of the scheduled code of the component units: by compacting the pipelined computations, it is possible to achieve a performance gain that would not be possible were the components implemented as functions. C functions either require call mechanisms that enforce a barrier synchronization between the two computation steps, or inline mechanisms that would lead back to the explosion in the nodes number of the dataflow graph.

D. High-level Scheduling

Given the software functional units described in Section III-C, in order to implement a public key cryptographic primitive, we need to encode it in terms of the software functional units. Then, we can explore the possible high-level schedules by means of automatic scheduling tools, such as those presented in [5].

For the Tate pairing algorithm in characteristic p, Figure 4 shows the dataflow graph of the doubling step of the core loop body. The nodes are arranged so that high-level parallelism is emphasized, following an ASAP scheduling policy with no resource constraints, thereby showing the maximum available parallelism at any given time. The figure highlights the presence of a significant amount of parallelism, making the exploration of performance vs. area tradeoffs worth being conducted.

The typical structure of the Miller’s algorithm, upon which the implemented Tate pairing algorithm [27] is based, includes a conditional branch that is only taken when the scan of binary expansion of the scalar r (see Section I) meets a 1.
The implementations ensure that the Hamming weight of \( r \) is minimal – in the range of 1 to 10. Since this operation is rarely executed (less than 1% of the iterations), it is handled in a tiled architecture such as DSPfabric by the intervention of the controller processor, which causes the coprocessor control to flow from the main iteration loop to a secondary code that is optimized for the branch execution. The alternative of predicing the branch code is feasible, but the size of the secondary code and the fact that the primary path is much faster (it has no instructions to execute) would cause the predicated code to negatively affect the performance.

IV. Experimental Results

In this Section we provide experimental evidence to support the effectiveness of the proposed approach. First, we gauge the complexity of the software function units in terms of both area (that is, number of CPUs) and latency. Table III summarizes the complexity data for the simpler units, while Tables IV and V show the complexity of two different implementations of the Montgomery multiplier. Analytically, these data can be derived from Equations 1 and 2, where Equation 1 represents the basic version of the Montgomery multiplier, while Equation 2 refers to the area-optimized version of the same unit that splits the \( A,B \) word-by-vector multiplication in Algorithm III.1 to execute it in parallel with \( tN \) and the subsequent addition, to reduce the number of processors used.

\[
T = (n + 2) \frac{52n}{\text{cpu}} + (2n + 1) + 5
\]

(1)

\[
8 \leq \text{cpu} \leq 4n
\]

\[
T = (n + 2) \left( \frac{26n}{\text{cpu}} + \max \left\{ \frac{26n}{\text{cpu}}, (2n + 1) \right\} + 5 \right)
\]

(2)

\[
16 \leq \text{cpu} \leq 2n
\]

In these equations, \( n \) is the number of input words, while \( \text{cpu} \) is the total number of nodes in the tiled architecture.

In order to evaluate the effectiveness of the high-level scheduling, we perform a multiobjective exploration of the design space defined by the architectural parameters, that is the number of Montgomery multipliers, modular adders and shifters available in the system, as well as the implementation...
TABLE IV
EXECUTION TIME AND TIME/AREA PRODUCT FOR THE SOFTWARE IMPLEMENTATION OF THE MONTGOMERY MULTIPLIER AS A FUNCTION OF INPUT WORDS AND NUMBER OF EMPLOYED CPUs

<table>
<thead>
<tr>
<th>Input size n</th>
<th>Number of CPUs</th>
<th>Time [clk]</th>
<th>Time × Area [clk×#CPU]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>200</td>
<td>1600</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>155</td>
<td>2160</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>399</td>
<td>3192</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>259</td>
<td>4144</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>666</td>
<td>5328</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>432</td>
<td>6912</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>315</td>
<td>10080</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>2414</td>
<td>19312</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>1330</td>
<td>24480</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>1088</td>
<td>34816</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>867</td>
<td>55488</td>
</tr>
</tbody>
</table>

TABLE V
EXECUTION TIME AND TIME/AREA PRODUCT FOR THE SOFTWARE IMPLEMENTATION OF THE MONTGOMERY MULTIPLIER USING HIGH LEVEL PARALLELIZATION, AS A FUNCTION OF INPUT WORDS AND NUMBER OF EMPLOYED CPUs

<table>
<thead>
<tr>
<th>Input size n</th>
<th>Number of CPUs</th>
<th>Time [clk]</th>
<th>Time × Area [clk×#CPU]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>315</td>
<td>5040</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>1088</td>
<td>17408</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>867</td>
<td>27744</td>
</tr>
</tbody>
</table>

of the Montgomery multipliers employed, as described in Tables IV and V.

Figure 5 sketches the Pareto frontier for the multiobjective exploration problem of finding the best configurations in terms of both area and latency.

The notion of Pareto optimality, states that a solution is optimal if it is impossible to find a solution which improves on one or more of the objectives without worsening any of them. If one solution is better in one objective than another solution and not worse in any other objectives, the latter is dominated by the former, which is always preferred. This set of solutions is called the Pareto frontier and is guaranteed to contain all optimal solutions, whatever way the individual objectives are weighted relative to each other. To put it in other words: the Pareto frontier exactly captures the available trade-offs between the different objectives.

Note that the Pareto frontier shown in Figure 5 gives a set of possible solutions. Then, time or area constraints should be applied to select the best solution. If no constraint is specified, then it is possible to observe that the optimum point using a time × area figure of merit is the architecture with 4 Montgomery multipliers, each implemented on 16 nodes, plus one shifter and one adder, which needs just over 1 million cycles to perform the entire pairing primitive.

However, if the goal is to optimize time, then, by employing large hardware resources, it is possible to cut down the execution times by 30%. On the other hand, if a compact device (e.g., 48 CPUs) is required, there is slowdown by a factor of 2 with respect to the time × area optimum.

The exploration also allows to better evaluate the implementations of the individual units. In our case, it shows that the 16 CPUs Montgomery multiplier implementation is superior to the equivalent implementations on 32 or 8 CPUs.

Comparing our approach with FPGA competitors is difficult, since related works [17], [25] are based on different arithmetic, while the current trend is to employ mod p-based cryptosystems (see Section I). Moreover, while for processors it is possible to obtain area estimates, the measurement of the physical area of FPGA implementations is widely dependent on CLB interconnection and pin layout. Therefore, the CLB count of an FPGA implementation gives no clue on the actual area occupied by the design. For the proposed implementation, coprocessors based on DSPfabric size at a 7 mm² die for 64 nodes, which is a mean figure with respect to the range of possibilities illustrated in the experimental evaluation.

On the other hand, a comparison can be given with a high-end embedded processor such as the 32-bit StrongARM, which is reported to execute the same pairing computation in over 60 million cycles [27].

With respect to competitor technologies, tiled architectures using the proposed methodology give the following advantages: smooth scalability (tiled architecture provide excellent scalability properties w.r.t. standard VLIW or superscalar architectures); quick development cycle (almost as fast as software development).

V. CONCLUDING REMARKS

In this paper, we propose a novel programming model for tiled architectures, suitable for computationally intensive public key cryptographic applications.

Our proposal is supported by a case study on the DSPfabric reconfigurable tiled architecture, focusing on the implementation of the Tate pairing primitive, which is at the core of all identity based cryptographic protocols.

Results prove that large amounts of parallelism can be extracted and exploited, yielding speedups of one order of magnitude with respect to state of the art software implementations.
As a future development, the methodology developed in this work could be fully automated, by designing a dedicated programming language and its compiler toolchain and integrating the scheduling algorithm within the compiler backend.

REFERENCES

ewline