International Journal of Parallel Programming, Vol. 35, No. 6, December 2007 (© 2007)
DOI: 10.1007/s10766-007-0046-1

Parallel Algorithms Development for Programmable Devices with Application from Cryptography

Issam W. Damaj[1]

Received November 17, 2006; accepted May 2, 2007

Reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), have been witnessing a considerable increase in density. State-of-the-art FPGAs are complex hybrid devices that contain up to several millions of gates. Recently, research effort has been going into higher-level parallelization and hardware synthesis methodologies that can exploit such a programmable technology. In this paper, we explore the effectiveness of one such formal methodology in the design of parallel versions of the *Serpent* cryptographic algorithm. The suggested methodology adopts a functional programming notation for specifying algorithms and for reasoning about them. The specifications are realized through the use of a combination of function decomposition strategies, data refinement techniques, and off-the-shelf refinements based upon higher-order functions. The refinements are inspired by the operators of Communicating Sequential Processes and map easily to programs in *Handel-C* (a hardware description language). In the presented research, we obtain several parallel *Serpent* implementations with different performance characteristics. The developed designs are tested under *Celoxica's RC-1000* reconfigurable computer with its two million gates *Virtex-E FPGA*. Performance analysis and evaluation of these implementations are included.

KEY WORDS: Parallel algorithms; methodologies; data encryption; formal models; gate array.

1. INTRODUCTION

The rapid progress and advancement in integrated circuits (ICs) technology provides a variety of new implementation options for system

[1]Dhofar University, Salalah, Oman. E-mail: i_damaj@du.edu.om

0885-7458/07/1200-0529/0

engineers. The choice varies between the flexible programs running on a general purpose processor (GPP) and the fixed hardware implementation using an application specific integrated circuit (ASIC). Many other implementation options present, for instance, a system with a *RISC* processor and a *DSP* core. Other options include graphics processors and microcontrollers. Specialized processors certainly improve performance over general-purpose ones, but this comes as a *quid pro quo* for flexibility. Combining the flexibility of GPPs and the high performance of ASICs leads to the introduction of reconfigurable computing (RC) as a new implementation option with a balance between versatility and speed.

Field Programmable Gate Arrays (FPGAs), are nowadays important components of RC-systems. FPGAs have shown a dramatic increase in their density over the last few years. For example, companies such as *Xilinx*[1] and *Altera*[2] have enabled the production of FPGAs with several millions of gates, such as in *Virtex-II Pro* and *Stratix-II FPGAs*. The versatility of FPGAs, opened up completely new avenues in high-performance computing. These programmable hardware circuits can be supported with flexible parallel algorithms design methodologies to form a powerful paradigm for computing.

The traditional implementation of a function on an FPGA is done using logic synthesis based on VHDL, Verilog or a similar hardware description langauge (HDL). These discrete event simulation languages are rather different from languages, such as *C*, *C++* or JAVA. An interesting step toward more success in hardware compilation was to grant a high-level of abstraction from the point of view of programmer. Accordingly, and recently, vendors have initiated the use of high-level languages like *Handel-C*,[3,4] *Forge*,[5] *Nimble*,[6,7] and *SystemC*.[8]

Although modern hardware compilation tools have significantly reduced the complexity of hardware design, many research opportunities are still present to study even more reduced design complexity. Accordingly, in this paper we investigate a methodology enabling high-level of abstraction in the process of hardware design. The proposed methodology is a step-wise refinement approach for developing parallel algorithms. The development will is based on higher-order skeletons exploiting possible inherent algorithmic parallelism. Algorithmic skeletons provide a promising basis for the automatic utilization of parallelism at sites of higher-order functions.[9] The correctness of the developed hardware is put forward for further discussion through in this paper.

The research presented in this paper, builds on the work of Abdallah and Hawkins[10–13] that adopts the transformational programming approach for deriving massively parallel algorithms from functional specifications (See Fig. 1). In this approach, the functional notation is used

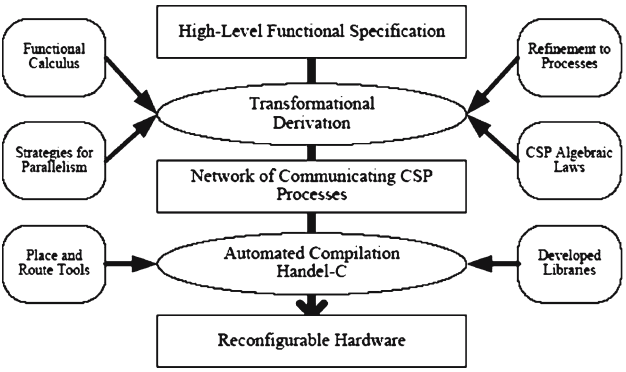

Fig. 1. An overview of the transformational derivation and the hardware realisation processes.

for specifying algorithms and for reasoning about them. This is usually done by carefully combining a small number of generic higher-order functions (such as *map*, *filter*, and *fold*) that serve as the basic building blocks for writing high-level programs. The parallelization of algorithms work by carefully composing an "off-the-shelf" parallel implementation of each of the building blocks involved in the algorithm. The underlying parallelization techniques are based on both pipelining and data parallelism. The essence of this approach is to design a generic solution once, and to use instances of the design many times for various applications.

In order to develop generic solutions for general parallel architectures it is necessary to formulate the design within a concurrency framework such as CSP.[4,11,14] Often parallel functional programs show peculiar behaviors which are only understandable in the terms of concurrency rather than relying on hidden implementation details. The formalization in CSP (of the parallel behavior) leads to better understanding of the described network of processes and allows for the analysis of its performance. The establishment of refinement concepts between functional and concurrent behaviors allows for the generation of parallel implementations for various architectures. This gives the ability to exploit well-established functional programming (FP) paradigms and transformation techniques in order to develop efficient parallel and sequential CSP processes independent from architectural details. The refinement from functional specification to CSP descriptions is reflected in Fig. 1 as transformational derivation. The transformational derivation is supported by strategies for parallelism, CSP laws, refinement rules including those for the refinement to CSP networks of processes.

The initial stages of development require a back-end hardware compiler stage for realizing the developed parallel designs. In the proposed methodology, *Handel-C* is adopted as the last stage of development generating the final hardware product. Note at this point that *Handel-C* language relies on the parallel constructs in CSP to model concurrent hardware resources. Mostly, algorithms described with CSP could be implemented under *Handel-C*. *Handel-C* enables the integration with VHDL and Electronic Design Interchange Format (EDIF) and thus various synthesis and place-and-route tools. The *Handel-C* development stage is described in Fig. 1 as an automated compilation step supported by different code libraries and place-and-route tools that produces the desired hardware.

The adopted methodology is systematic in the sense that it is carried out on using step-by-step procedures. The development is yet manual and applied according to the following informal procedure:

- Specify the algorithm in a functional setting relying on high-order functions as the main building constructs wherever necessary.
- Apply the predefined set of rules to create the corresponding CSP networks according to a chosen degree of parallelism.
- Write the equivalent *Handel-C* code and complete the hardware compilation.

These steps are aided with different compilers and integrated development environments as shown in Fig. 2. The set of available mathematical rules belong mainly to the refinement to CSP stage. The automation of the development process including the creation of a preprocessor is currently under investigation.

The research related to the adopted methodology has been initiated by Abdallah, investigating the refinement from functional specifications into concurrency,[11] and presenting a calculus of decomposition of higher order functions for parallel programs derivation.[15] Hawkins and Abdallah work included the formalism for proving the refinement rules for both datatypes and processes and investigated possible *Handel-C* implementations.[13] Case studies where developed for a *JPEG* decoder, closest pair algorithm, sorting algorithms, DNA processing algorithms,[10,12,16,17] the *Kasumi* cryptographic algorithm,[18] and various parallel implementations of a matrix multiplication algorithm.[19]

The main focus of this paper is on the realization and application of the theory suggested by the development methodology. An additional focus is to test the development method and to broaden its area of use to include an industrial level application. Furthermore, it includes investigating the performance of the developed designs by carrying out a thorough

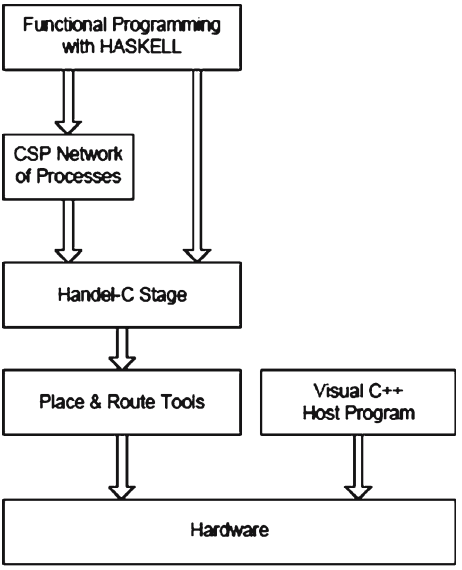

Fig. 2. Assisting tools used in the proposed development method, including, *Haskell Hugs98* compiler to test the specification, *Handel-C* for hardware compilation, *Visual C++* integrated development environment to create the host program driving the *RC-1000* device with its *FPGA*.

analysis and evaluation. This leads to critically extending, tuning, and enhancing the suggested method and its realization. In addition, the current investigation enriches the adopted method by providing libraries that supports and promotes the method for further investigation and possibly adoption by mainstream engineers.

The remaining sections of the paper are organized so that Section 2 introduces background material. In Section 3, related work is discussed. The case study from cryptography is proposed in Section 4. The analysis and performance evaluation are included in Sections 5 and 6. Section 7 concludes the paper.

2. BACKGROUND

Abdallah and *Hawkins* defined in Ref. 12 some constructs used in the adopted development model. Their investigation looked in some depth at data refinement; which is the means of expressing structures in the specification as communication behavior in the implementation.

The following parts of this section introduces briefly the proposed steps of development; the functional paradigm, CSP, and *Handel-C*. In addition, we introduce the basis of the refinement from a functional specification to networks of CSP processes. The benefits of each development step and the adopted refinement approach are stressed in Section 5.

2.1. The Functional Paradigm

Functional programming is quite different from imperative (or procedural) programming and also from object oriented programming. FP's main concern is expressions, where everything reduces to an expression. An expression, that is a collection of operations and variables, will results in a single value. Functions are the main building block in functional programs and could passed around within a program like other variables. Functional programs are usually *List* oriented, and they focus description of the problem to be solved rather than focus on the mechanism of solution. There are currently many functional languages, one of the most widely used is *Haskell*.[20]

As a brief overview, we can summarize that functions are considered as the basic unit of program development and as the major routes to reuse. In addition, strong typing is considered as an aid to development pre-implementation, during implementation and post-implementation. Some of the fundamental features in FP are powerful high-order functions, parametric polymorphism, the support provided of developing user-defined datatypes. Other features of no less importance are lazy evaluation and programming with infinite data structures. Overloading of function names are not supported in all functional languages.[21]

High-order functions are an important feature supported in functional languages. A high-order function is a function which takes another function as a parameter. The most commonly used high-order functions are *map*, *zipWith*, *fold*, and *filter*.

The functions *map* and *zipWith* are introduced in this section. The function *map* takes a list and a function as parameters, then it applies the input function to all elements of the input list, for example:

$$map \quad even \quad [1, 2, 3, 4] \quad = [False, True, False, True],$$

where *even* is a function that checks wether a number is even or not.

The function *zipWith* takes two lists and a function as inputs, then it applies the function on two elements; one element taken from each input list, for example:

$$zipWith \quad add \quad [1, 2, 3, 4] \quad [2, 3, 4, 5] = [3, 5, 7, 9],$$

where *add* is a function that adds two numbers.

Related work adopting FP in hardware development is introduced in Section 3, and the main benefits gained in using this paradigm in the adopted model are discussed in Section 5.

2.2. Communicating Sequential Processes (CSP)

The Communicating Sequential Processes (CSP) notation is based on events and processes. A CSP process engages in a series of events, which can be local, or perform channel-based communication with synchronization capabilities. A channel communication is an event where at most two processes participate, one acting as an input and the other as an output.

The alphabets of the components of a concurrent system determine the overall structure and interface of that system. The fact that the notation regards basic concurrency operations as primitives enables the developer to concentrate on the concurrent behavior of the system without needing to worry about the implementation of these basic functions. A synchronization or communication can be specified in one operation without any concern over how it takes place.[14]

Valuable features of CSP include its strong support for formal reasoning. CSP allows to make generic assertion about the behavior of the system, such as deadlock freedom. Moreover, specific assertions of requirements for the behavior of the process could be done. These assertions are made with reference to a number of models of the process. Two typical models are the trace and failures-divergences models.

Communicating Sequential Process also has the advantage of generality. The primitive operations of CSP are simple enough that almost any form of concurrency can be represented using them. Thus, CSP can be used to specify a wide variety of concurrent systems. Moreover, it can be used to specify the intended functionality of a message passing system at a formal level without requiring the system to be modified for a specific architecture (as may be required by implementation). Employing such features in hardware development gives the designer the freedom to choose an appropriate architecture and organization of an implementation leaving no effect on the original description. Many research projects have employed CSP in hardware design; this is discussed in Section 3.

2.3. Data Refinement

In the following, the main concern is explaining the main constructs and rules to be used in refining a possible functional specification with its description in CSP notation. Accordingly, we start by presenting some communication entities used for refining datatypes declared in the initial functional step of development; these are *Item*, *Stream*, *Vector*, and some of their combined forms. We note here that the suggested methodology relies on the message passing technique to implement parallelism.

The *Item* corresponds to a basic type, such as an Integer data type, and it is to be communicated on a single communicating channel.

The *Stream* is a purely sequential method of communicating a list of values (a list is a functional term equivalent to an array in a language like *C*). It comprises a sequence of messages on a channel, with each message representing a value. Values are communicated one after the other. Assuming the stream is finite, after the last value has been communicated, the end of transmission (EOT) on a different channel will be signaled. Given some type A, a *Stream* containing values of type A is denoted as $\langle A \rangle$.

Each item to be communicated by the vector will be dealt with in parallel. A vector refinement of a simple list of items will communicate the entire structure in a single step. Given some type A, a *Vector* of length n, containing values of type A, is denoted as $\lfloor A \rfloor_n$.

Whenever dealing with multi-dimensional data structures, for example, lists of lists, implementation options arise from differing compositions of our primitive data refinements—streams and vectors. Examples of the combined forms are the Stream of Streams, Streams of Vectors, Vectors of streams, and Vectors of Vectors. These forms are denoted by: $\langle S_1, S_2, \ldots, S_n \rangle$, $\langle V_1, V_2, \ldots, V_n \rangle$, $\lfloor S_1, S_2, \ldots, S_n \rfloor$, and $\lfloor V_1, V_2, \ldots, V_n \rfloor$.

2.4. Process Refinement

The refinement is continued by looking into the functions specified in the first stage of development. Accordingly, the refinement of the formally specified functions to processes is the key step toward understanding possible parallel behaviour of an implementation. In this section, the interest is in presenting refinements of a subset of functions—some of which are higher-order. A bigger refined set of these functions is discussed in Ref. 11.

Generally, These highly reusable building blocks can be refined to CSP in different ways. This depends on the setting in which these functions are used (i.e., with streams, vectors, etc.), and leads to implementations with different degrees of parallelism. Note that we do not use CSP in a totally formal way, but we use it in a way that facilitates the later

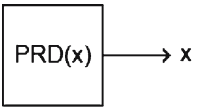

Fig. 3. The Produce process (PRD) for items.

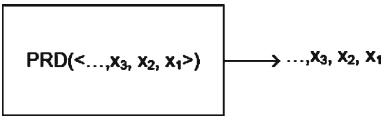

Fig. 4. The Producer process (PRD) for streams.

Handel-C coding stage. Recall for the following subsections that values are communicated through as an *elements* channel, while a single bit is communicated through another *eotChannel* channel to signal the end of transmission in the case of *Streams*.

2.4.1. Produce

The producer process (PRD) is fundamental to process refinement. It is used to produce values on the channels of a certain communication construct (*Item*, *Stream*, *Vector*, etc.). These values are to be received and manipulated by another processes.

2.4.1.1. Items. For simple, single item types (*int*, *char*, *bool*, etc.), the producer process is very simple. This is depicted in Fig. 3. Here the output is just a single channel.The definition in CSP notation is very straightforward:

$$\text{PRD (Item } a) = \text{out.element.channel ! } a \rightarrow \text{SKIP}$$

2.4.1.2. Streams. The producer process for streams is depicted in Fig. 4. As already noted, the output in this case is a pair of two other channels. One channel carries the values of the stream, and the other is a simple channel used to signal EOT.

In a more general case, the structure of the values which the stream is carrying is not necessarily known. These may be simple items, but may also be streams or vectors. Generally, producing a stream could be described as:

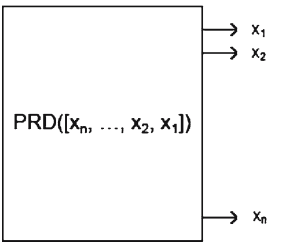

Fig. 5. The Producer process (PRD) for vectors.

$$\begin{aligned}&\text{PRD } (\langle s \rangle) = \\ &(;)_{i=1}^{i=\text{length}(s)} \\ &(\text{PRD } s_i)[\text{out.elements.channel/out}]); \\ &\text{out.eotChannel ! eot} \rightarrow \text{SKIP}\end{aligned}$$

This description defines PRD as a process that produces items sequentially (this is described using the sequential execution operator ";"). The number of items is equal to the length of the stream. After all elements are produced, an end of transmission signal will be produced on the *eotChannel* channel.

2.4.1.3. Vectors. For vectors of size n, n instances of the producer process are composed in parallel, one for each item in the vector. The output here is an array of channels. This is depicted in Fig. 5. A general definition is given below:

$$\begin{aligned}&\text{PRD } (\lfloor v \rfloor_n) = |||_{i=1}^{i=n} \\ &(\text{PRD } v_i)[\text{out.elements}_\text{i}.\text{channel/out}].\end{aligned}$$

The operator $|||_{i=1}^{i=n}$ is used to indicate that n copies of the process PRD v for producing items will be running concurrently. PRD is described as a processes that runs concurrently n instances (of a processes that produces single items).

A process *STORE* stores a communication construct in a variable. We use this process to store items, vectors, streams, or combinations of vectors and streams. A subscript letter is used with the processes PRD and *STORE* to indicate the type of communication. We sometimes omit this subscript if the communication structure is clear from context.

2.4.2. Feeding Processes

The feed operator in CSP models function application. The feed operator is written $\triangleright$. The feed operator takes two processes, composes them together in parallel, and renames both the output of the first and the input of the second to a new name, which is then hidden. Given the lifted concepts of CSP channel renaming and hiding, the definition can remain the same regardless of the type of the communicating construct (*Item*, *Stream*, *Vector* or any combination).

$$P \triangleright Q =$$
$$(P[\text{mid/out}] \parallel Q[\text{mid/in}]) \setminus \{\text{mid}\}.$$

2.4.3. Formal Process Refinement

Given the definition of a feed operator that operates on processes, a formal definition of process refinement could be delivered. Consider a function f, which takes input values of type A and returns values of type B. Assume that the data refinement step has already been performed, such that A and B are both types of some transmission value:

$$f :: A \rightarrow B.$$

Then, consider a potential refinement for a function f, a process F. The operator $\sqsubseteq$ denotes a process refinement, where the left-hand side is a function, and the right-hand side is a process. To state that f is refined to F, or in other words, the process F is a valid refinement of the function f, the following may be used:

$$f \sqsubseteq F.$$

The rules of refinement were proven once in Ref. 11 and applied in this paper refine a functional specification into a network of communicating processes.

2.4.4. MAP the Process Refinement of the Higher-order Function map

Now the attention is turned to the refinement of the widely used higher-order function *map*.[12] Employing this function in stream and vector settings is presented. The refinement for combined structures is to be made in a similar way.

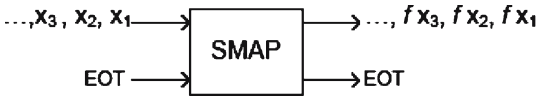

Fig. 6. The SMAP process for streams.

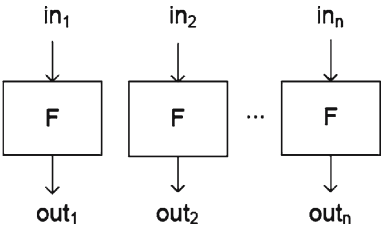

Fig. 7. The VMAP process for vectors.

2.4.4.1. Streams. A process implementing the functionality of *map* f in stream terms should input a stream of values, and output a stream of values with the function f applied (See Fig. 6).

In general, the handling of the EOT channels will be the same. However, the handling of the value will vary depending on the type of the elements of the input and output stream.

$$
\begin{aligned}
&SMAP(F) = \\
&\mu X \bullet in.eotChannel\ ?\ eot \rightarrow \\
&out.eotChannel\ !\ eot \rightarrow SKIP \\
&\Box \\
&F[in.elements.channel/in, \\
&out.elements.channel/out];\ X
\end{aligned}
$$

2.4.4.2. Vectors. In functional terms, the functionality of *map* f in a list setting is modeled by *vmap* f in the vector setting. Consider F as a valid refinement of the function f. The implementation of $VMAP$ can then proceed by composing n instances of F in parallel, and directing an item from the input vector to each instance for processing (See Fig. 7). In CSP we have:

$$
\begin{aligned}
&\mathrm{VMAP}_n(F) \qquad\qquad = \\
&|||_{i=1}^{i=n} F[in_i/in, \mathrm{out}_i/\mathrm{out}]
\end{aligned}
$$

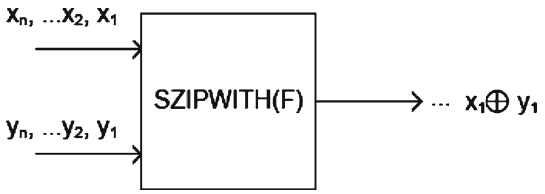

Fig. 8. The SZIPWITH process for streams.

2.4.5. ZIPWITH the Process Refinement of the Higher-order Function zipWith

Recall another higher-order function, namely *zipWith*. This function is used to zip two lists (taking one element from each list) with a certain operation. Formally:

$$
\begin{aligned}
&zipWith :: \\
&(A \to B \to C) \to [A] \to [B] \to [C] \\
&zipWith\ (\oplus)\ [x_1, x_2, \ldots x_n][y_1, y_2, \ldots y_n] = \\
&[x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_n \oplus y_n]
\end{aligned}
$$

2.4.5.1. Streams. The process implementation of $(zipWith\ f)$ in stream terms should input two streams of values, and output a stream of values with the function f applied (See Fig. 8).

Again, the handling of the EOT channel will be the same. Nevertheless, the handling of the value will vary depending on the type of the input and output streams elements.

$$
\begin{aligned}
&SZIPWITH(F) = \\
&\mu X \ \bullet \text{in.eotChannel ? eot} \to \\
&\text{out.eotChannel ! eot} \to SKIP \\
&\Box \\
&F[in_1\text{.elements.channel/in}_1, \\
&\text{in}_2\text{.elements.channel/in}_2, \\
&\text{out.elements.channel/out}]; X
\end{aligned}
$$

2.4.5.2. Vectors. To implement the data parallel version of this higher-order function, we refine it to a process *VZIPWITH* that takes two vectors as input and zips the two lists with a process *F*; *F* is a refined process from the function $(\oplus)$. This is depicted in Fig. 9.

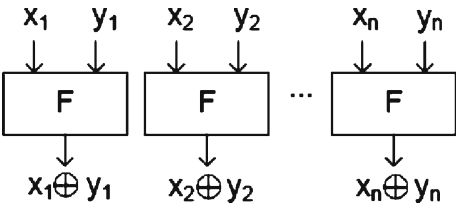

Fig. 9. The VZIPWITH process for vectors.

$$vzipWith\ (\oplus) :: \lfloor A \rfloor_n \rightarrow, \lfloor B \rfloor_n \rightarrow \lfloor C \rfloor_n$$

$$VZIPWITH\ (\oplus) =$$
$$|||_{i=1}^{i=n} F[\mathrm{out}_i/\mathrm{out}, c_i/\mathrm{in}_1, d_i/\mathrm{in}_2].$$

2.5. Handel-C as a Stage in the Development Model

Based on datatype refinement and the skeleton afforded by process refinement, the desired reconfigurable circuits are built. Circuit realisation is done using *Handel-C*, as it is based on the theories of CSP[14] and *Occam*.[22]

From a practical standpoint, each refined datatype is defined as a structure in *Handel-C*, while each process is implemented as a *macro procedure*. We divide the constructs corresponding to the *CSP* stage into two main categories for organisation purposes. The first category represents the definitions of the refined datatypes. The second category implements the refined processes. The refined processes are divided into different groups. The *utility processes* group contains macros responsible for producing, storing, sinking, broadcasting data, etc. The *basic processes* group contains macros that correspond to simple arithmetic and logical operations. These basic processes could be simple addition, multiplication, etc. The *higher-order processes* group contains the macros realising the CSP implementations corresponding to the higher-order functions. A separate group contains the macros that handle the FPGA card setup and general functionality. The reusable macros found in these groups serves as building blocks used for constructing a certain specified algorithm (Fig. 9).

2.5.1. Datatypes Definitions

The datatypes definitions are implemented using structures. This method supports recursive as well as simple types. The definition for an

Item of a type *Msgtype* is a structure that contains a communicating channel of that type.

```
#define Item(Name, Msgtype)
  struct {
     chan Msgtype   channel;
     Msgtype       message;
     } Name
```

For generality in implementing processes the type of the communicating structure is to be determined at compile time. This is done using the *typeof* type operator, which allows the type of an object to be determined at compile time. For this reason, in each structure we declare a *message* variable of type *Msgtype*.

A stream of items, called *StreamOfItems*, is a structure with three declarations a communicating channel, an EOT channel, and a *message* variable[12]:

```
#define StreamOfItems(Name, Msgtype)
  struct {
     Msgtype       message;
     chan Msgtype   channel;
     chan Bool     eotChannel;
     } Name
```

A vector of items, called *VectorOfItems*, is a structure with a variable *message* and another array of sub-structure elements.[12]

```
#define VectorOfItems(Name, n, Msgtype)
  struct {
     struct {
        chan Msgtype   channel;
        } elements[n];
     Msgtype    message;
     } Name
```

Other definitions are possible, but it affects the way a channel is called using the structure member operator (.). Examples of different extended definitions are as follows (the first definition reuses the *Item* structure, while the second one employs channel arrays supported in *Handel-C*):

```
#define VectorOfItems(Name, n, Msgtype)
  struct {
     struct {
```

```
        Item(element, MsgType);
        } elements[n];
      } Name

#define VectorOfItems(Name, n, Msgtype)
  struct {
     chan Msgtype   channel[n];
     Msgtype       messages;
     } Name
```

2.5.2. Utilities Macros

The utility processes used in the implementation are related to the employed datatypes. The *Handel-C* implementation of these processes relies on their corresponding CSP implementation. An instance of these utility macros is shown in the following code segment:

```
macro proc ProduceItem(Item, x){
  Item.channel ! x;}

macro proc StoreItem(Item, x){
  Item.channel ? x;}
```

2.5.3. Higher-Order Processes Macros

An example for an implementation in *Handel-C* of the CSP refinement of a higher-order function (*map*) is done as follows. The process runs through a loop which terminates when the variable *eot* is set to true. At each step of the loop, the process enters a wait state until either the EOT or the value channel of the input stream is willing to communicate. If the EOT channel is willing to communicate, the input is consumed from it and stored in the variable *eot*, then output an EOT message for the output stream. If the value channel of the input stream is willing to communicate, the value is consumed then F is applied to it giving the result on the output stream channel.

```
macro proc
 SMAP (streamin, streamout, F){
  Bool eot;
  eot  = False;
  do{
   prialt{
   case streamin.eotChannel ? eot:
```

```
      streamout.eotChannel ! True;
      break;
    default:
      F(streamin.elements,
        streamout.elements);
       break;
     }} while (!eot)}
```

We turn the attention to providing a definition in *Handel-C* for the behaviour of the process VMAP. Here we can employ *Handel-C*'s enumerated *par* construct to place n instances of the process F in parallel. Each instance is passed to the corresponding channels from both the input and output channels.

```
macro proc
 VMAP (n, vectorin, vectorout, F) {
  typeof (n) c;
  par (c = 0 ; c < n ; c++){
    F(vectorin.elements[c],
     vectorout.elements[c]);}}
```

2.6. Evaluation Tools and Performance Metrics

Different tools are used to measure the performance metrics used for the analysis. These tools include the design suite (DK) from *Celoxica*, where we get the number of *NAND* gates for the design as compiled to (EDIF). The DK also affords the number of cycles taken by a design using its simulator. Accordingly, the speed of a design could be calculated depending on the expected maximum frequency of the design.

To get the practical execution time as observed from the host computer, the *C++* high-precision performance counter is used. The counter probes the execution of the design after loading the image of the design into the FPGA till termination. Practically, the probation comes directly after writing a control signal to the FPGA enabling execution. The counter stops immediately after receiving a signal through reading the status register. According to this measurement the speed of execution is calculated.

The information about the hardware area occupied by a design, i.e., number of Slices used after placing and routing the compiled code, is determined by the ISE place and route tool. In the current investigation the only used metrics are the number of Slices and the Total Equivalent Gate Count for a design.

3. RELATED WORK

In this section we define four perspectives, not necessarily mutually exclusive or unconnected, to be considered for relating our work with its global literature:

- Purpose: Related to frameworks created for refining correct hardware implementations.
- Implementation Framework: Related to the use of the Functional Paradigm in hardware development. Related work in this area might also meet the purpose of developing correct reconfigurable hardware.
- Description: Related to the use of CSP in hardware development.
- Application: Related to the use of FPGAs in implementing the Serpent cryptographic algorithm.

The idea for deriving implementations from the specification through correct well defined refinement steps has been motivated by many technical facts. For instance, the limitations in commonly used synthesis tools and formal verification techniques utilized in equivalence checking between the synthesized hardware and the abstract specification.[23] Many frameworks for developing correct hardware has been brought out in the literature.[23–26] Our work meets these multi-stage frameworks in their aim of refining correct hardware from specification.

The Provably Correct Systems project (ProCoS) suggested a mathematical basis for the development of embedded and real-time computer systems. They used FPGAs as a back-end hardware for realizing their developed designs.[24]

In Ref. 26, a formal approach to correctly generate an architecture-level model of a system from its specification model is proposed. The proposed approach relies on formal transformations to refine a specification model into a provably correct architectural model. Tools have been created to support automatic generation of refined models.[23]

The attractions for using the functional paradigm in hardware development incited many researchers. This triggered many investigations in this area, such as *Lava*,[27] *Hawk*,[28,29] *Hydra*,[30] *HML*,[31] *MHDL*,[32] *DDD* system,[33] *SAFL*,[34] *MuFP*,[35] *Ruby*,[36] and *Form*.[37]

The compiled *Occam* into *FPGAs*[38,39] and the *Handel-C* compiler[3] are considered as the major work introducing CSP in hardware development. Susan Stepney at the University of York[4,40] investigated ways to translation between CSP and *Handel-C*. *Handel-C* compiler is used to map designs onto FPGAs. The suggested translation uses FDR2 as a front-

end specification and proof tool, then automatically translates the formal designs into executable *Handel-C*.

Many efforts have been put to efficiently implement the *Serpent* in hardware. R. Anderson proposed in Ref. 41 the *Serpent* algorithm and evaluated its performance under different processing systems. Adam *et al*. Ref. 42 presented an FPGA implementation and performance evaluation of the *Serpent*. Multiple architecture options of the *Serpent* algorithm were explored with a strong focus being placed on high-speed implementations. Bora in Ref. 43 investigated the possibilities of realising the *Serpent* using *FLEX10K ALTERA FPGAs* series. The implementations of this algorithm was introduced in Ref. 44 with an effort to determine the most suitable candidate for hardware implementation within commercially available FPGAs.

4. CASE STUDY: THE SERPENT CRYPTOGRAPHIC ALGORITHM

The *Serpent* algorithm is chosen as a test case for the proposed development model. The motivation behind choosing the *Serpent* is its proven strength and suitability for hardware implementation.[41] The *Serpent* algorithm is a 32-round substitution-permutation (SP) network operating on four 32-bit words. The algorithm encrypts and decrypts 128-bit input data and a key of 128, 192 or 256 bits in length. The *Serpent* algorithm consists of three main blocks an initial permutation (IP), A 32-round block, and a final Permutation (FP). One round function is comprised of three operations occurring in sequence. These are bit-wise XOR with the 128-bit round key, substitution via 32 copies of one of eight S-boxes, and data mixing via a linear transformation. These operations are performed in each of the 32 rounds with the exception of the last round. In the last round, the linear transformation is replaced with a bit-wise XOR with a final 128-bit key.

This section develops parallel implementations of the *Serpent* algorithms showing all stages of development and the results of testing. The following subsections presents the functional specification, followed by the refinement and the implementation in *Handel-C*. Various designs with different degrees of parallelism are investigated. Different solutions are presented to some realization pitfalls. The final section presents the results of running the compiled designs with comparison among different processing systems.

4.1. Formal Functional Specification

Two main building blocks construct the *Serpent*, the key scheduling block and the encryption (decryption) block. The key scheduling block inputs the private key and outputs the desired 132 subkeys. The encryption block inputs data segments representing the plaintext and outputs the corresponding ciphered data segments. The formal functional specification employs the following names used for clarifying types definitions.

```
type Private   = [Bool]
type SubKey    = [Bool]
type DataBlock = [Bool].
```

The following subsections present the specification of the *Serpent* algorithm. The implementation of the specification under *HUGs98 Haskell* compiler is tested at the unit, component and integration levels.

4.1.1. Key Scheduling

Two main steps are carried out to generate the required 132 32-bit subkeys for the *Serpent*. The algorithm for generation is as follows:

- Generate an intermediate list *ws* by:
  - Padding the input key to 256-bit if necessary.
  - Then, partitioning the key into eight segments of equal length (32-bit) $ws_0, .., ws_7$.
  - Then, expanding these to intermediate prekeys $ws_8, .., ws_{139}$ by the following recurrence: $ws_i := (ws_{i-8} \oplus ws_{i-5} \oplus ws_{i-3} \oplus ws \oplus 9e3779b9_{hex} \oplus (i - 8)) <<_{11}$ where $(<<_n)$ is the n-element left circular shift operator.
- The round subkeys *ks* are now calculated from the prekeys *ws* using the S-boxes as follows:

$$
\begin{aligned}
\{k_0;\ k_1;\ k_2;\ k_3\} &= S3(w_0;\ w_1;\ w_2;\ w_3)\\
\{k_4;\ k_5;\ k_6;\ k_7\} &= S2(w_4;\ w_5;\ w_6;\ w_7)\\
\{k_8;\ k_9;\ k_{10};\ k_{11}\} &= S1(w_8;\ w_9;\ w_{10};\ w_{11})\\
\{k_{12};\ k_{13};\ k_{14};\ k_{15}\} &= S0(w_{12};\ w_{13};\ w_{14};\ w_{15})\\
\{k_{16};\ k_{17};\ k_{18};\ k_{19}\} &= S7(w_{16};\ w_{17};\ w_{18};\ w_{19})\\
&\ldots\\
\{k_{124}; k_{125}; k_{126}; k_{127}\} &= S4(w_{124}; w_{125}; w_{126}; w_{127})\\
\{k_{128}; k_{129}; k_{130}; k_{131}\} &= S3(w_{128}; w_{129}; w_{130}; w_{131})
\end{aligned}
$$

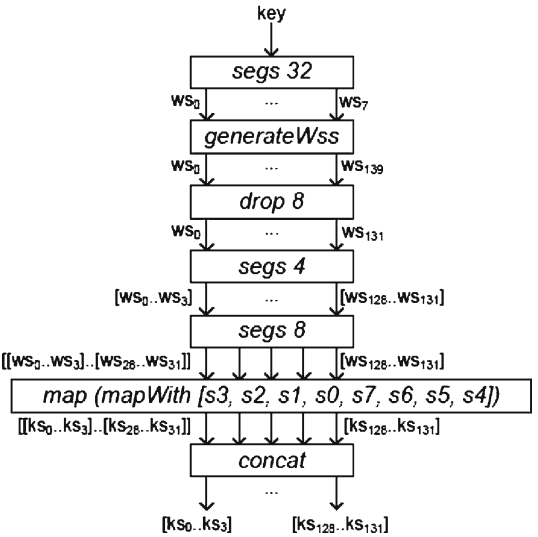

Fig. 10. Steps for *Serpent* subkeys generation.

The function *keySchedule* formally specifies the above algorithm. This function inputs the private key and outputs the desired subkeys following the steps clarified in Fig. 10. This figure also shows the format of the final output as ordered for later use in the functions specifying the encryption.

```
keySchedule :: Private -> [[SubKey]]
keySchedule key = concat kss
where
ws = drop 8 (generateWs 8 (segs 32 key))
kss = map (mapWith [s3, s2, s1, s0,
                    s7, s6, s5, s4])
                    (segs 8 (segs 4 ws))
```

The application of the S-boxes is done by mapping the function *(mapWith [s3, s2, s1, s0, s7, s6, s5, s4])* over the prepared segmentation of *ws [segs 8 (segs 4 ws)]*. Note that the length of the list *ws* at this point is 132 elements. Grouping this list into segments of four and then of eight, will give four lists each of eight four-element sublists, covering 128 elements from *ws*. The remaining four elements constitutes a final list of four elements. With the lazy evaluation property found in functional programming, the final mapped *mapWith* only applies the function *s3* to the remaining list. This will give the desired output list of lists representing the 132 round subkeys.

The *generateWs* responsible for generating the prekeys is specified as follows:

```
generateWs :: Int -> [[Bool]] -> [[Bool]]
generateWs i ws
  | ((i < 140) && (i > 7)) =
       (generateWs (i+1) (ws ++ [wsD]))
  | otherwise = ws
  where
   wsD = (shift 11 (foldr1 fullexor
   [(ws!!(i-8)), (ws!!(i-5)),
   (ws!!(i-3)), (ws!!(i-1)),
   const, (itob (i-8))]))

   const = concat
   (map itob.htoi ["9e37", "79b9"])
```

The S-boxes are specified using the logic functions *fullexor*, *fullOR*, *fullAND*, and *fullComplement*. These corresponds to the full-word bitwise version of *XOR*, *OR*, *AND*, and *NOT* logic operations. For instance, the first S-box is specified as the function *s0* with a list of list of *bool* as input and output. The input list elements *[a, b, c, d]* are distributed to different operations computing for the final output list *[w, x, y, z]*. Temporary variables used to compute the final output list are grouped to be zipped with their operation using the higher-order function *zipWith*. The current specification does not reflect the order that these operations should be carried out. A dependency analysis has to be done aiding the later refinement. Note that the decryption inverse S-boxes are specified in a similar way. In the following we show the specification of the *s0* function.

```
s0 :: [[Bool]] -> [[Bool]]
s0 [a,b,c,d] = [w, x, y, z]
  where
  [t01, t03, z, t06, y,
   t12, t13, t15, t17, x ]=
   zipWith
     fullexor [b, a, t02, a, t09,
               c, t07, t06, w, t12]

              [c, b, t01, d, t08,
               d, t11, t13, t14, t17]

  [t05,t07, t02] =
   zipWith fullOR [c, b, a] [z, c, d]
```

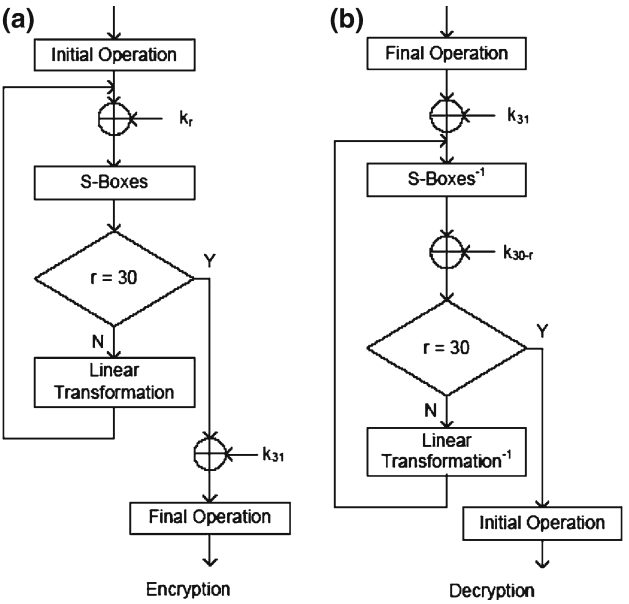

Fig. 11. Serpent encryption (a) and decryption (b) flowcharts.

```
[t08, t09, t11, t14] =
 zipWith fullAND [d, t03, t09, b]
                 [t05, t07, y, t06]

w = fullComplement t15
```

4.1.2. Serpent Block Cipher

Flowcharts showing the steps to carry out the encryption and the decryption are shown in Fig. 11. Decryption is different from encryption in that the inverse of the S-boxes must be used in the reverse order, as well as the inverse linear transformation and reverse order of the subkeys.

A functional specification formulates *Serpent* encryption as a function *serpentEncrypt*. This function works by firstly inputting a list of lists of data blocks. Then, it maps the function *serpentEncryptSeg*, responsible for a single 128-bit data block encryption, with the input private key to all the input list elements. The functional specification of *serpentEncrypt* is as follows:

```
serpentEncrypt :: [[DataBlock]] -> Private
                  -> [[DataBlock]]

serpentEncrypt inputs key =
map (serpentEncryptSeg(keySchedule key))
  inputs
```

The formalised function *serpentEncryptSeg* inputs the generated round subkeys in a form of a list of lists, besides, the 128-bit plaintext input data block. The first 31 rounds subkeys are taken from the input list of subkeys and zipped in a list of pairs with the corresponding S-box number. The higher-order function *foldl* is used with the function *serpentFold* to fold the input data block over the zipped list of pairs. In other words, the function *foldl* replicates the required 31 rounds in a pipelined fashion. The final round is carried out by XORing the output from the 31st round with the 32nd set of subkeys *(sKeys!!32)*, at this point the result is passed to the function *s7*. The final ciphered output is the result of XORing the output from the function *s7* with the last set of subkeys *(sKeys!!32)*. The suggested formal functional specification is as follows:

```
serpentEncryptSeg :: [[SubKey]] ->
        [DataBlock]->[DataBlock]

serpentEncryptSeg sKeys input =
 zipWith fullexor (sKeys!!32) (s7 xorOut))
 where
  xorOut = zipWith fullexor (sKeys!!31) roundsOut

  roundsOut = foldl serpentFold input
   (zip (take 31
    (concat (copy1 [0,1,2,3,4,5,6,7] 5)))
     (take 31 sKeys))
```

A *Serpent* fold, specified as the function *serpentFold*, inputs a data block and a pair corresponding to a list of four subkeys and the corresponding S-box number employed in that fold. The subkeys are zipped with the input, passed to the corresponding S-box, and finally linearly transformed using the function *lTransfrom*. The input S-box number is used to choose one of the available S-boxes listed in the list of functions *s*. A possible formalisation is as follows:

```
serpentFold :: [DataBlock] ->
     (Int, [SubKey]) -> [[Bool]]
```

```
serpentFold input (i,skey) =
  lTransform ((s!!i)
     (zipWith fullexor skey input))
  where
  s = [s0, s1, s2, s3,
      s4, s5, s6, s7]
```

The function *lTransform* linearly transforms a list of four inputs into a list of four outputs. the transformation uses the left circular shift function *shift* and the left shift function *lshift* as follows:

```
lTransform :: [[Bool]] -> [[Bool]]
lTransform [x0, x1, x2,x3] =
  [y0, y1, y2, y3]
 where
  [y0i, y2i, y0, y1, y2, y3] =
  mapWith [(shift 13), (shift 3),
           (shift 5), (shift 1),
           (shift 22), (shift 7)]
        [x0, x2, y0ii, y1i, y2ii, y3i]

  [y1i, y3i, y0ii, y2ii] =
     zipWith fullexor
        (zipWith fullexor
           [x1, y2i, y0i, y2i]
           [y0i, (lshift 3 y0i),
                        y1, y3])
        [y2i, x3, y3, (lshift 7 y1)])

lshift :: Int -> [Bool] -> [Bool]
lshift n ls =
(drop n ls) ++ (copy False n)
```

4.2. Algorithms Refinement to CSP

For the key scheduling part we suggest two designs. The first design implements the scheduling in a data-parallel fashion. The second design economises the implementation by carefully removing replication from one of the main building blocks. For the encryption part, we suggest three designs. The first design presents a fully pipelined network of rounds. The second design uses only one stage from the pipeline suggested in the first design. In this case inputs and outputs are refined to streams. The third

design leaves a flexible choice for the level of parallelism, allowing control over the number of pipelined stages.

4.2.1. Key Scheduling

At this development stage, we refine each function from the specification of the key scheduling part. In the following section, the two suggested designs are presented and explained.

4.2.1.1. First Design. The types used in the specification of the function *keySchedule* are refined to a 256-bit Integer item for the private key, and a vector of vectors of items of size (33×4) for the output subkeys:

$$keySchedule :: Int256 \rightarrow \lfloor \lfloor Int32 \rfloor_4 \rfloor_{33}$$

The refinement implements the function *keySchedule* as a process *KEYSCHEDULE*. According to the specification, the first event to occur is the segmentation of the input key into eight segments using a predefined process *SEGS*. These eight segments are passed to the process *GENERATEWS*.

$$\begin{gathered} KEYSCHEDULE = ((PRD(32) \triangleright SEGS) \gg_8 \\ STORE_v(ws)); (GENERATEWS(8, ws)) \gg_{132} \\ (VMAP_4(VMAPWITH([S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7])) \parallel S_3) \end{gathered}$$

where,

$$S_0 \sqsubseteq s0; S_1 \sqsubseteq s1; S_2 \sqsubseteq s2; S_3 \sqsubseteq s3; S_4 \sqsubseteq s4; S_5 \sqsubseteq s5; S_6 \sqsubseteq s6; S_7 \sqsubseteq s7;$$

The higher-order process ($VMAP_4$) creates four parallel instances of the process *VMAPWITH*. In turn, 32 parallel instances of the S-boxes processes is now available for parallel computation. These 32 S-boxes process takes 128 items from the 132 generate prekeys in the process *GENERATEWS*. The final four prekeys are passed to a parallel instance of the process *S3*. The output from these parallel S-boxes processes is the desired vector of 132-round subkeys. The process *KEYSCHEDULE* is depicted in Fig. 12.

The function *generateWs* could be refined as follows:

$$generateWs :: Int32 \rightarrow \lfloor Int32 \rfloor_8 \rightarrow \lfloor Int32 \rfloor_{132}$$

$$\begin{gathered} generateWs \sqsubseteq GENERATEWS \\ GENERATEWS(i, ws) = \end{gathered}$$

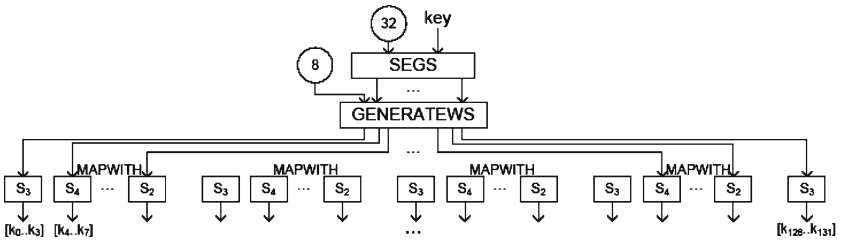

Fig. 12. The process KEYSCHEDULE, first design.

$if(7 < i < 140)$
$then \quad WsD(i, ws) \gg StoreItem(wsd);$
$\quad GENERATEWS(i + 1, ws +\!\!+ [wsd])$
$else \quad PRD(ws)$

Unrolling the above recursive implementation for $GENERATEWS(8, ws)$:

$GENERATEWS(8, ws) =$
$WsD(8, ws) \gg STORE_v(wsd); GENERATEWS(9, ws +\!\!+ [wsd]);$
$WsD(9, ws) \gg STORE_v(wsd); GENERATEWS(10, ws +\!\!+ [wsd]);$
.
.
.
$WsD(139, ws) \gg STORE_v(wsd); GENERATEWS(140, ws +\!\!+ [wsd]);$
$PRD(ws).$

This could be done as:

$GENERATEWS(8, ws) =$
$for(i = 8; i < 140; i++)\{$
$WsD(i, ws) \gg StoreItem(wsd); \},$

where,

$$WsD(i, ws) = \text{out!}(\lll_{11} (ws[i-8] \oplus ws[i-5] \oplus ws[i-3] \oplus ws[i-1] \oplus (9e3779b9_{hex}) \oplus (i-8)))$$

4.2.1.2. Second Design. The second design intends to eliminate the replication in the S-boxes computation processes. This leads to a smaller hardware circuit in the later stage as a trade for the expected speed. The change from the first design is made by refining *map* to its stream setting.

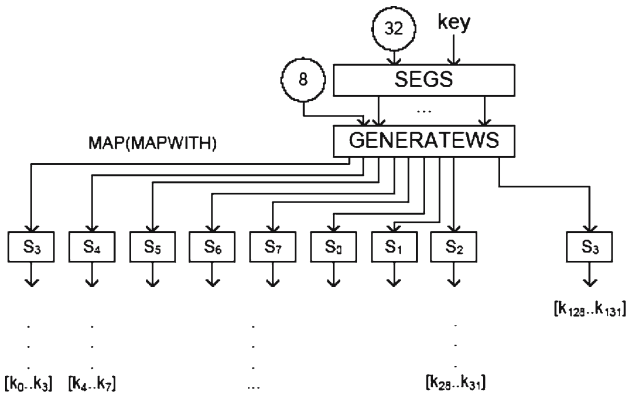

Fig. 13. The process KEYSCHEDULE, second design with replication reduced.

This implementation is depicted in Fig. 13 and described in the following CSP network:

$$KEYSCHEDULE = ((32 \triangleright SEGS) \gg_8 STORE_v(ws)); (GENERATEWS(8, ws)) \gg (SMAP(VMAPWITH([S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7])) \parallel S_3)$$

4.2.2. Serpent Block Cipher

The current refinement is done in three different designs. The process responsible for a single block ciphering is *SERPENTESEG*, the refinement of the function *serpentEncryptSeg*. The input data items, for instance, could be passed as a stream of vectors of four 32-bit data items to the encrypting block *SERPENTESEG*. The output is refined also to a stream of items as follows:

$$serpentEncrypt(key) :: \langle \lfloor Int32 \rfloor_4 \rangle \rightarrow \langle \lfloor Int32 \rfloor_4 \rangle$$

Consequently, we suggest the following refinement employing the higher-order process *SMAP*. The *key*, in this case, is passed as an argument to the process *SERPENTENCRYPT*.

$$SERPENTENCRYPT(key) = KEYSCHEDULE(Key) \gg SMAP(SERPENTESEG)$$

A multi-way *Serpent* encryption version is implemented as follows:

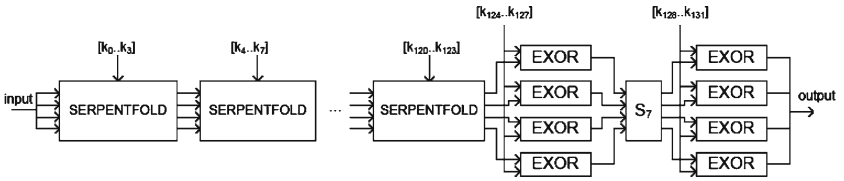

Fig. 14. The process SERPENTESEG, first pipelined design.

$$serpentEncrypt(key) :: \langle \lfloor \langle \lfloor Int32 \rfloor_4 \rangle \rfloor_n \rangle \rightarrow \langle \lfloor \langle \lfloor Int32 \rfloor_4 \rangle \rfloor_n \rangle$$

$$SERPENTENCRYPT(key) = KEYSCHEDULE(key) \gg SMAP(VMAP_n(SMAP(SERPENTESEG)))$$

where the value of n is limited by the ability to realise this network on the available hardware in the following stage. The following three designs are suggested for the implementation of the process *SERPENTESEG*.

4.2.2.1. First Design. This design suggests a fully pipelined implementation of the *Serpent* encryption specification. The pipeline is constructed by replicating the single round specified as the function *serpentFold*. The replication is done using the vector setting refinement of the higher-order function *foldl*, where the input is a vector of items. The input 132 subkeys are distributed to the pipelined folds as shown in Fig. 14. Also, the number of the round in use is distributed to the pipelined folds. The output from the pipeline is the input to the higher-order process *VZIPWITH(EXOR)*, zipping it with a set of four subkeys. The result of zipping is passed to an S_7 S-box process, whose output vector is zipped again using another *VZIPWITH(EXOR)* with the last generated set of four subkeys. The CSP description is as follows:

$$serpentEncryptSeg = \lfloor Int32 \rfloor_{132} \rightarrow \langle \lfloor Int32 \rfloor_4 \rangle \rightarrow \langle \lfloor Int32 \rfloor_4 \rangle$$
$$serpentEncryptSeg \sqsubseteq SERPENTESEG$$

$$SERPENTESEG = (BROADCAST_3([0..7]) \parallel (PRD([0..6]))) \triangleright (VVFOLDL(SERPENTFOLD) \parallel VZIPWITH_4(EXOR) \gg_4 S_7 \parallel VZIPWITH_4(EXOR))$$

where,

$$serpentFold \sqsubseteq SERPENTFOLD.$$

The serpent fold is implemented as in the following:

$$serpentFold :: \lfloor Int32 \rfloor_4 \rightarrow (Int3, \lfloor Int32 \rfloor_4) \rightarrow \lfloor Int32 \rfloor_4$$

$$SERPENTFOLD = (in?i \rightarrow SKIP); VZIPWITH_4(EXOR) \gg S_i \gg LTRANSFORM,$$

where

$$lTransform \sqsubseteq LTRANSFORM.$$

The linear transformation function *lTransform* is refined to the process *LTRANSFORM*. The input and output are refined as a vector of items as follows:

$$lTransform :: \lfloor Int32 \rfloor_4 \rightarrow \lfloor Int32 \rfloor_4$$

The process *LTRANSFORM* is implemented as follows:

$$\begin{aligned} &LTRANSFORM = (|||_{i=0}^{i=3} in[i]?x[i] \rightarrow SKIP); \\ &LSHIFT(3) \parallel LSHIFT(7) \parallel \\ &(VZIPWITH_4(EXOR) \gg_4 VZIPWITH_4(EXOR)) \parallel \\ &VMAPWITH([SHIFT(1), SHIFT(13), SHIFT(3), \\ &SHIFT(5), SHIFT(1), SHIFT(22), SHIFT(7)]) \end{aligned}$$

4.2.2.2. Second Design. In this design, the network component processes are still the same, as shown in the first design, with a modification to the way they communicate. The stream communication with the main process *SERPENTFOLD*, allows the elimination of copies of this process using *SVFOLDL* the stream refinement of *foldl*, where the input is a vector of items. The subkeys distribution, at this point, are passed sequentially to the process *SERPENTFOLD*. Only the last two sets of subkeys are produced as vectors to be used in the two similar parallel processes $VZIPWITH_4(EXOR)$. This network is shown in Fig. 15. The CSP description is as follows:

$$serpentEncryptSeg = \langle Int32 \rangle \rightarrow \langle \lfloor Int32 \rfloor_4 \rangle \rightarrow \langle \lfloor Int32 \rfloor_4 \rangle$$

$$\begin{aligned} SERPENTESEG = &(BROADCAST_3([0..7]) \parallel \\ &(PRD([0..6]))) \triangleright (SVFOLDL(SERPENTFOLD) \parallel \\ &VZIPWITH_4(EXOR) \gg_4 S_7 \parallel VZIPWITH_4(EXOR)) \end{aligned}$$

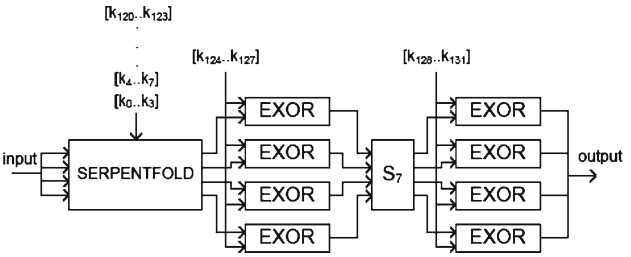

Fig. 15. The process SERPENTESEG, second design with stream of subkeys.

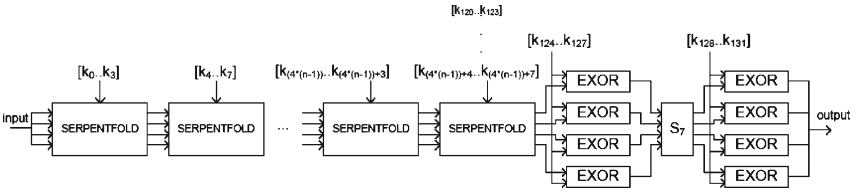

Fig. 16. The process SERPENTESEG, third partially pipelined design.

4.2.2.3. Third Design. Based on the above suggested implementations, this design composes both a pipelined part and a stream-based part to build the final desired *Serpent* network. This implementation is shown in Fig. 16 and done as follows:

$$serpentEncryptSeg = \langle Int32\rangle \rightarrow \lfloor Int32\rfloor_{132} \rightarrow \langle \lfloor Int32\rfloor_4\rangle \rightarrow \langle \lfloor Int32\rfloor_4\rangle$$

$$\begin{gathered} SERPENTESEG = (BROADCAST_3([0..7]) \parallel (PRD([0..6]))) \\ \triangleright(((PRD(n) \triangleright VVFOLDL(SERPENTFOLD)) \parallel \\ SVFOLDL(SERPENTFOLD)) \parallel VZIPWITH_4(EXOR) \gg_4 \\ S_7 \parallel VZIPWITH_4(EXOR)) \end{gathered}$$

4.3. Reconfigurable Hardware Implementations

The part of the hardware implementation included in this section is aimed to show samples of the implemented code. We put some emphasis on some code segments, where we could not base the implementation from the previous stage in a straightforward manner. Remember that the main reason behind the faced coding difficulties resides in the level of generality of the constructs to be implemented.

The following macros are for the two designs of key scheduling. The first macro *KeySchfedule1st* outputs the subkeys as a vector of vectors of vectors of items from the macros *GenerateWsVOVOV* and *S3*.

```
macro proc KeySchedule1st (keyIn, KssOutVOVOV,
lastksV){
.
.
.
par{
 Segs(keyIn, segmentsOut);

 GenerateWsVOVOV
 (segmentsOut, WsOutVOVOV, lastwsV);

 VMap
 (WsOutVOVOV, 4, KssOutVOVOV, VMapWithSs);
 S3(lastwsV, lastksV);}
```

The macro for the second design with its stream implementation is as follows:

```
macro proc KeySchedule2nd
   (keyIn, KssOutSOVOV, lastksV) {
.
.
.
par{
 Segs(keyIn, segmentsOut);

 GenerateWsSOVOV
   (segmentsOut, WsOutSOVOV, lastwsV);

 Map(WsOutSOVOV, KssOutSOVOV, VMapWithSs);
 S3(lastwsV, lastksV);}}
```

The stream-version macro implementing the process *GenerateWs* is shown in the following code section. In this macro a 140 (32-bit Integer) elements array *ws* is used to store the generated prekeys. This means occupying a large area from the targeted FPGA. An alternative implementation is to use the available internal RAM, so, this would dramatically save the needed space. The RAM property of allowing only one access to it at once (read or write at a time) imposes some restrictions. For instance,

the production of the final calculated prekeys should be done as stream of items instead of a stream of vectors of vectors of items. Both cases are shown in the following code sections:

```
macro proc GenerateWsSOVOV
(wsIn, wsOutSOVOV, lastwss) {
.
.
.
Int32 ws[140];

par(j = 0; j < 8; j++){
 jTemp[j] = 0@j;
 wsIn.elements[j].channel ?
      ws[jTemp[j]];}

PHI = 0x9e3779b9;

for(i = 8; i < 140; i++){
   iTemp = 0@i;
   wTemp = ws[i-3]^ws[i-5]^
           ws[i-8]^ws[i-1]^
           PHI^(iTemp-8);

   par{
      ProduceItem(wItem, wTemp);
      Shift(wItem, 11, sOut);
      StoreItem(sOut, ws[i]);}
   if (i == 139){
   break;}}

ProduceSOVOVOItemsFromArrayWithOffset
   (wssOutSOVOV, 4, 8, 4, ws, 8);
par{
 lastwss.elements[0].channel ! ws[136];
 lastwss.elements[1].channel ! ws[137];
 lastwss.elements[2].channel ! ws[138];
 lastwss.elements[3].channel ! ws[139];}}
```

The second version is as follows:

```
macro proc GenerateWsRam
   (wsIn, wssOut) {
.
.
.
ram Int32 ws[140];
```

```
par(j = 0; j < 8; j++){
jTemp[j] = 0@j;
wsIn.elements[j].channel ?
  ws[jTemp[j]];}

PHI = 0x9e3779b9;

for(i = 8; i < 140; i++){
  iTemp = 0@i;
  wTemp = ws[i-1]^PHI^(iTemp-8);
  wTemp1 = wTemp ^ ws[i-8];
  wTemp2= wTemp1^ws[i-5];
  wTemp3 = wTemp2^ws[i-3];
  par{
     ProduceItem(wItem, wTemp3);
     Shift(wItem, 11, sOut);
     StoreItem(sOut, ws[i]);}
  if (i == 139){
  break;}}
ProduceStreamOfItems(wsOut, 140, ws);}
```

The use of an FPGA's on-chip memory is constrained with its supported memory capabilities and corresponding *Handel-C* compilation options. The available sophisticated *SelectRAM* memory hierarchy available on the used *Virtix-E* FPGA supports *True Dual-Port BlockRAMs* and Distributed RAMs. However, *Handel-C* declaration of an array is equivalent to declaring a number of variables. Each entry in an array may be used exactly like an individual variable, with as many reads, and as many writes to a different element in the array as required within a clock cycle. Arrays are more efficient to implement in terms of concurrent access required by fast pleasantly parallel designs. Arrays are implemented using the available logic blocks in an FPGA (Slices in the case of *Xilinx* devices). RAMs, are normally more efficient to implement in terms of hardware resources than arrays since they use the on-chip RAM blocks. RAMs, would allow one location to be accessed in any one clock cycle.

To take the advantage of an available multi-port memory blocks, one can use the *mpram* declaration in *Handel-C* instead of *ram*. A design that uses an *mpram* with two ports would outperform the sequential design in terms of speed, but still replications of some processes would be necessary to cope with the doubled amount of information retrieved. A design that uses a dual-ported memory to store a list should have refined the list as a stream of vectors of two elements in the description stage.

Before we present parts of the realization of the encryption designs, we note the solution we suggest for implementing the higher-order process *VMAP-WITH* with a list of different processes. The macro *VMapWith* needs to map a list of macros to a list of items. The problem we faced is for how to pass a

list of macros as an argument to the macro *VMapWith*. A best case scenario is having the following code implementation:

```
macro proc VMapWith
   (vIn,, vProcesses, vOut, n){

par(i = 0, i < n, i++){
  vProcesses[i]
   (vIn.elements[i],
     vOut.elements[i]);}}
```

The vector of macros *vProcesses* passing to the macro *VMapWith* is not supported in the current version of *Handel-C*. A second possible form for a possible implementation in *Handel-C* is as follows:

```
macro proc VMapWith
(vIn, P1, P2,..., Pn, vOut, n){

par{
 P1(vIn.elements[i], vOut.elements[i]);
 P2(vIn.elements[i], vOut.elements[i]);
 .
 .
 .
 Pn(vIn.elements[i], vOut.elements[i]);}}
```

A step forward in the code generation leads to the third possible form of implementation. This form would fit the calling of the process *VMAPWITH* from another higher-order macro as had been done in:

```
VMap(WsOutVOVOV, 4,
    KssOutVOVOV, VMapWithSs);.
```

This suggests the removing of the zipped-with macro names from the arguments lists in the macro procedure definition as follows:

```
macro proc VMapWithPs(vIn, vOut, n){

par{
 P1(vIn.elements[i], vOut.elements[i]);
 P2(vIn.elements[i], vOut.elements[i]);
 .
 .
 .
 Pn(vIn.elements[i], vOut.elements[i]);}}.
```

A possible solution to such a limitation is, again, the availability of a preprocessor automatically generating the allowed implementation from the best case scenario presented. For the case of mapping with the list of S-boxes macros; the code is as follows:

```
macro proc VMapWithSs(vIn, vOut){
par{
 S3(vIn.elements[0], vOut.elements[0]);
 S2(vIn.elements[1], vOut.elements[1]);
 S1(vIn.elements[2], vOut.elements[2]);
 S0(vIn.elements[3], vOut.elements[3]);
 S7(vIn.elements[4], vOut.elements[4]);
 S6(vIn.elements[5], vOut.elements[5]);
 S5(vIn.elements[6], vOut.elements[6]);
 S4(vIn.elements[7], vOut.elements[7]);}}
```

For the encryption part, we include the implementation done for the third design. Whereby, a combination of parallel and sequential fold are employed with vector of items as input. Based on the CSP implementation, the macro *EncryptSegsVVandSV* is implemented as follows:

```
macro proc EncryptSegsVVandSV
    (input, sKeysVOV, VRnds,
   sKeysSOV, finalKeys, output) {

par{
 VVFoldL(sKeysVOV, output1, 4,
   NParalRnds, SerpentFold, input);

 SVFoldL(sKeysSOV, 4, output2,
   SerpentFold, output1, 4, NParalRnds);

 VZipWith(4, output2,
   finalKeys.elements[0], output3, EXOR);

 S7(output3, output4);

 VZipWith(4, output4,
   finalKeys.elements[1], output, EXOR);}}
```

The macro *SerpentFold* implements its corresponding process as follows:

```
macro proc SerpentFold
    (input, i, sKeys, output) {

VectorOfItems (vOut, 4, Int32);
VectorOfItems (output1, 4, Int32);
```

```
par{
  par{
  VZipWith(input, sKeys, vOut, EXOR);}

  if(i==0)
     S0(vOut, output1);
  else if(i==1)
     S1(vOut, output1);
  else if(i==2)
     S2(vOut, output1);
  else if(i==3)
     S3(vOut, output1);
  else if(i==4)
     S4(vOut, output1);
  else if(i==5)
     S5(vOut, output1);
  else if(i==6)
     S6(vOut, output1);
  else if(i==7)
     S7(vOut, output1);

  LinearTransformation
     (output1, output);}}
```

5. GENERAL EVALUATION

In this paper, the contribution of the presented work could be found in many aspects. Some additions were crucial to the realization step of the method so that it can cope with real-life complex areas of applications. A famous algorithm from cryptography has been targeted as a test case that has given a clear idea about the practical use of the methodology. Reusable libraries are created at all levels of development. The availability of such libraries supports and facilitates the development in general. The created libraries for the different studies from cryptography are highly reusable for developing other cryptographic algorithm. This might include the introduction of new components to the libraries, or slightly modifying the available ones. According to these points, we stress the following aspects:

The development is originated from a specification stage, whose main key feature is its powerful **higher-level of abstraction**. During the specification, the isolation from parallel hardware implementation issues allowed for deep concentration on the specification details. Whereby, for the most part, the style of specification comes out in favor of using higher-order functions. Two other inherent advantages for using the functional paradigm are **clarity** and **concise-**

ness of the specification. This was reflected throughout all the presented studies. At this level of development, the **correctness** of the specification is insured by construction from the used correct building blocks. The implementation of the formalized specification is tested under *Haskell* by performing random tests for every level of the specification.

The correctness will be carried forward to the next stage of development by applying the provably correct rules of refinement. The available pool of refinement formal rules enables a high degree of **flexibility** in creating parallel designs. This includes the capacity to divide a problem into completely independent parts that can be executed simultaneously (pleasantly parallel). Conversely, in a nearly pleasantly parallel manner, the computations might require results to be distributed, collected and combined in some way. Remember at this point, that the refinement steps are done by combining off-the-shelf **reusable** instances of basic building blocks.

6. PERFORMANCE ANALYSIS

In this section we show the testing results of mapping the designs, analyzing their timing, and showing the speeds as measured for testing the *RC-1000* board from the used *P4* machine. The fully pipelined design was over-mapped, thus, the following presented speeds are for the remaining designs. Note that in the suggested *Serpent* implementations, the finest grains of basic building blocks are refined as processes rather than using *Handel-C* operators. Thus, an increase in communications cost between processes is found.

In Table I, we show the testing results of the encryption subkeys generation. The keys generation (second design) runs with a throughput of 96 Mbps occupying 13097 Slices, i.e., 68% of the FPGA area.

As shown in Table II, the testing results of the *Serpent* second and third designs are included, while the first design failed to compile with its large gates count. The maximum achieved parallelism was in running the third design with two parallel folds and a third performing the remaining 29 sequential folds. This implementation has a throughput of 12.21 Mbps occupying an area of 19198 Slices (99% of the available FPGA area). The second design with its sequential single fold implementation achieved throughput of 12.15 Mbps with an area of 12291 Slices.

In Table III, we include some results from literature mapping the same algorithms onto *FPGAs*. The high-speeds achieved for the suggested optimised implementations is very clear, as compared to our high-level (un-optimised) implementation (yet) — from performance perspective. The shown results include a high-speed implementations for the *Serpent* (333 Mbps) presented by Elbirt *et al.*[42] Gaj *et al.* in Ref. 45 presented another high-speed implementation for the *Serpent* (431.4 Mbps).

Table I. Testing Results of *Serpent* Encryption Subkeys Generation

Design Metrics	First Parallel-Output Design	Second Streamed-Output Design
Number of Gates	NA—Simulators failed to produce results due to the design size and simulation complexity.	137256 NANDs
Number of Occupied Slices		13097 Slices (68%)
Number of Slice Flip Flops		9137 (23%)
Number 4 input LUTs		9773 (25%)
Number used for 32×1 RAMs		256
Number used as 16×1 RAMs:		32
Total equivalent gate count		180963 Gates
Number of Cycles		NA
Maximum Frequency of Design		67.45 MHz
Throughput		NA
Measured Execution Time		44 Micro sec.
Measured Throughput		96 Mbps

In Table IV[46–48] we compare the number of cycles for different hardware implementations of the *Serpent* including a number of microprocessor-based implementations.

The higher-level development caused high replication in using basic building blocks, and more clearly their communications. Many instances of *PRODUCE* and *STORE* processes caused the high use of intermediate variables. Other processes were used for structuring data in the format corresponding to their functional definitions. For instance, to collect some vectors of subkeys and produce them as a vector of vectors of vectors of items. Such use also plays a big role in occupying larger silicon area after realization.

If we consider the implementation of an algorithm without using our proposed method, we might implement the whole design with a small number of macros and minimum use of communications. Moreover, possible handmade enhancements could be done with the aid of shared variables. This would undoubtedly reduce the cost paid for communicating parallel processes implementation and might lead to a more economical realization and less congested design with a higher frequency. This certainly comes as *quid pro quo* for the step-wise development.

Table II. Testing Results of *Serpent* Encryption

Design Metrics	First Fully-Pipelined	Second Stream-Based	Third Partially-Pipelined (2 Parallel and 29 Sequential)
Number of Gates	NA-Simulators failed to produce results due to the design size and simulation complexity.	114885 NANDs	222121 NANDs
Number of Occupied Slices		12291 Slices (64%)	19198 Slices (99%)
Number of Slice Flip Flops		10249 (26%)	18647 (48%)
Number 4 input LUTs		9041 (23%)	16975 (44%)
Total equivalent gate count		139876 Gates	257745 Gates
Number of Cycles		1314	NA
Maximum Frequency of Design		12.15 Mbps	12.21 Mbps
Throughput		56.6 MHz	56.7 MHz
Measured Execution Time		5.51 Mbps	NA
Measured Throughput		10.53 Ms	10.48 Ms

Table III. Comparisons Among Similar FPGA Systems Implementing Optimized Serpent

System	Speed (Mbps)	Clock Frequency (MHz)	Area (Slices)
Xilinx Virtex XCV 1000 FPGA [13]	77–333	10.2–19.4	5890–6467
Xilinx Virtex XCV 1000 FPGA [12]	431.4	NA	2152

Table IV. Comparisons Among Different Hardware Systems, With Respect to Number of Clock Cycles, Implementing The Serpent

System	Number of Clock Cycles
RC-1000 2nd Stream-based Design	1314
Pentium Pro (32-bit processor)	1738
MorphoSys	1792
LA-64 Superscalar	2269
PA-RISC Based workstation	2415
Ada Boolean on 8-bit processors	11000
Ada bit-slice on 8-bit processors	34000
8-bit machines Boolean Implementation	70339

7. CONCLUSION

Mapping parallel versions of algorithms onto hardware could enormously improve computational efficiency. Recent advances in the area of reconfigurable computing came in the form of FPGAs and their high-level HDLs such as *Handel-C*. In this paper, we build on these recent technological advances by presenting, demonstrating, and examining a high-level hardware development method. The used method creates a functional specification of an algorithm without defining parallelism. Correspondingly, an efficient parallel implementation is derived in the form of *CSP* network of processes. Accordingly, we create diffident parallel implementations in *Handel-C*. The presented work included theory and practices about the suggested methodology. In this paper, we observed a case study from applied cryptography, namely the *Serpent* algorithm. The encryption block ciphers and key expansions were addressed. The correctness, conciseness and clarity of the specification is emphasized. The systematic and flexible refinements of the specification allowed the reasoning about various implementations with different degrees of parallelism for each case. The described designs ranged from fully pipelined, partially pipelined, to

streamed input and output implementations. At this stage, the realization using *Handel-C* is presented, emphasizing some code segments which tackled different noted implementation pitfalls. Future work includes extending the theoretical pool of rules for refinement, the investigation of automating the development processes, and the optimization of the realization for more economical implementations with higher throughput.

ACKNOWLEDGMENTS

I would like to thank Dr. Ali Abdallah, Prof. Mark Josephs, Prof. Wayne Luk, Dr. Sylvia Jennings, and Dr. John Hawkins for their insightful comments on the research which is partly presented in this paper.

REFERENCES

1. Xilinx, Information available from, http://www.xilinx.com
2. Altera, Information available from, http://www.Altera.com
3. Celoxica, Information available from, http://www.celoxica.com
4. S. Stepney, *CSP/FDR2 to Handel-C Translation*, Tech. Rep. YCS-2002-357, Department of Computer Science, University of York (June 2003).
5. D. Edwards, S. Harris, and J. Forge, High performance hardware from java, Xilinx Whitepaper http://www.xilinx.com
6. Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, Hardware-software codesign of embedded reconfigurable architectures, in *Proceedings of the 37th Design Automation Conference*, Los Angeles, USA (2000).
7. N. Technology, Information available from, http://www.nimble.com
8. S. Network, Information available arom, http://www.systemc.org
9. G. Michaelson, N. Scaife, P. Bristow, and P. King, Nested Algorithmic Skeletons from Higher Order Functions, *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, **16**(2–3):181–206 (August 2001).
10. A. E. Abdallah, *Functional Process Modelling, Research Directions in Parallel Functional Programming*, Springer, Berlin (1999), pp. 339–360.
11. A. E. Abdallah, Derivation of Parallel Algorithms: From Functional Specifications to csp Processes, in B. Moller (ed.), *Proceedings of Mathematics of Program Construction*, Vol. 947 of Lecture Notes in Computer Science, Springer, Berlin (1994), pp. 67–96.
12. A. E. Abdallah and J. Hawkins, Calculational Design of Special Purpose Parallel Algorithms, in *Proceedings of 7th IEEE International Conference on Electronics, Circuits and Systems* (*IEEE/ICECS*), IEEE Computer Society Press, Silver Spring, MD (2000), pp. 261–267.
13. A. E. Abdallah and J. Hawkins, Formal Behavioural Synthesis of Handel-c Parallel Hardware Implementation for Functional Specifications, in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Silver Spring, MD (2003), pp. 278–288.
14. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ (1985).

15. A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (eds.), Proceedings of the European Conference on Parallel Processing, EuroPar'96, LNCS 1024, Springer, Berlin (1996), pp. 911–920.
16. J. Hawkins and A. Abdallah, Synthesis of a Highly Parallel JPEG Decoder Implementation from its Functional Specification, in *Proceeding of IFIP Working Conference on Distributed and Parallel Embedded Systems*, Kluwer, Dordrecht (2004).
17. A. E. Abdallah, G. Simiakakis, and T. Theoharis, Formal Development of a Reconfigurable Tool for Parallel dna Matching, in *Proceedings of 7th IEEE International Conference on Electronics, Circuits and Systems* (IEEE/ICECS), IEEE Computer Society Press, Silver Spring, MD (2000), pp. 268–272.
18. I. Damaj, Higher-level Hardware Synthesis of the Kasumi Cryptographic Algorithm, *J. Comput. Sci. Technol*. **22**(1):60–70 (2007).
19. I. Damaj, Parallel Algorithms Development for Programmable Logic Devices, *Adv. Eng. Softw*. **37**(9):561–582 (2006).
20. S. Thompson and Haskell, *The Craft of Functional Programming*, 2nd Ed. Addison-Wesley, Reading, MA (1999).
21. D. J. Russel, *Fad: A Functional Analysis and Design Methadology*, Ph.D. thesis, The University of Kent at Canterbury, United Kingdom (August 2000).
22. I. Ltd., OCCAM 2 Reference Manual, Prentice-Hall International, Englewood Cliffs, NJ (1988).
23. J. Peng, S. Abdi, and D. Gajski, Automatic Model Refinement for Fast Architecture Exploration, in *Proceedings of the The Asia-Pacific Design Automation Conference*, Bangalore, India (2002), pp. 332–337.
24. J. Bowen, M. Fränzle, E. Olderog, and A. Ravn, Developing Correct Systems, in *Proc. 5th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, Silver Spring, MD (1993), pp. 176–187.
25. J. Bowen, C. A. R. Hoare, H. Langmaack, E. Olderog, and A. Ravn, A ProCoS II project final report: ESPRIT Basic Research Project 7071, *Bull. Eur. Assoc. Theor. Compu. Sci*. (*EATCS*), **59**:76–99 (1996).
26. S. Abdi and D. Gajski, *Provably Correct Architecture Refinement*, Technical Report CECS0329, Center for Embedded Computer Systems at University of California Irvine, Irvine-USA (September 2003).
27. K. Claessen, *Embedded Languages for Describing and Verifying Hardware*, Ph.D. thesis, Chalmers Univesity of Technology and Göteborg University, Sweden (April 2001).
28. J. Launchbury, J. Lewis, and B. Cook, On Embedding a Microarchitectural Design Language within Haskell, in *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, ACM Press, New york (1999), pp. 60–69.
29. J. Matthews, J. Launchbury, and B. Cook, Specifying Microprocessors in Hawk, in *Proceedings of the International Conference on Computer Languages*, IEEE, 1998, pp. 90–101.
30. J. O'Donnell, Hydra: Hardware Description in a Functional Language using Recursion Equations and High Order Combining Forms, in G. J. Milne (ed.), *The Fusion of Hardware Design and Verification*, North-Holland, Amsterdam (1988), pp. 309–328.
31. Y. Li and M. Leeser, HML: An Innovative Hardware Design Language and its Translation to VHDL, in *Proceedings of the Conference on Hardware Design Languages,* Bangalore, India (1995).

32. D. Barton, Advanced Modeling Features of MHDL, in *In International Conference on Electronic Hardware Description Languages*, Las Vegas, Nevada (1995).
33. S. Johnson and B. Bose, *DDD: A System for Mechanized Digital Design Derivation*, Tech. Rep. 323, Indiana University, Indiana (1990).
34. R. Sharp, *Higher-Level Hardware Synthesis*, Ph.D. thesis, Robinson College University of Cambridge, Cambridge (November 2002).
35. M. Sheeran, muFP: A Language for VLSI Design, in *Proc. ACM Symposium on LISP and Functional Programming*, ACM Press, New york (1984), pp. 104–112.
36. G. Jones and M. Sheeran, Circuit Design in Ruby, in *Proceedings of the Formal Methods for VLSI Design,* North-Holland (1990), pp. 13–70.
37. T. Cheung and G. Hellestrand, Multi-level equivalence in design transformation, in *Proceedings of International Conference on Computer Hardware Description Languages*, Chiba Japan (1996), pp. 559–566.
38. I. Page and W. Luk, Compiling Occam into Field-programmable Gate Arrays, in W. Moore, W. Luk (eds.), *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991, pp. 271–283.
39. H. Jifeng, I. Page, and J. Bowen, Towards a Provably Correct hardware implementation of Occam, in G. Milne, L. Pierre (eds.), *Correct Hardware Design and Verification Methods* (*CHARME'93*), Vol. 683 of Lecture Notes in Computer Science, Springer, Berlin (1993), pp. 214–225.
40. C. T. Library, CSP/FDR2 to Handel-C translation, http://www.celoxica.com/techlib/files/CEL-W0309221A18-133.htm
41. R. Anderson, E. Biham, and L. Knudsen, Serpent: A Proposal for the Advanced Encryption Standard, in *Proceedings of the First Advanced Encryption Standard* (*AES*) *Conference*, Ventura, CA (1998).
42. A. Elbirt and C. Paar, An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher, in *Proceedings of the 2000 ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*, ACM Press, New York, USA (2000), pp. 33–40.
43. P. Bora and T. Czajka, Implementation of the SERPENT Algorithm using ALTERA FPGA Devices, Public Comments on AES Candidate Algorithms, Round 2 (October 2000).
44. A. Yip, W. Chetwynd, and B. Paar, An FPGA-based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists, *IEEE Trans. Very Large Scale Integr*. (*VLSI*) Syst. **9**(4):545–557 (2001)
45. K. Gaj and P. Chodowiec, Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard using field Programmable Gate Arrays, *Lect. Notes Compu. Sci*. **2020**:84–100 (2001).
46. B. Gladman, Implementation Experience with Aes Candidate Algorithms, in *Proceedings of the 2nd AES Candidate Conference,* Rome, Italy (1999).
47. V. Journot, Evaluation of Serpent, one of the Aes Finalists on 8-bit Microcontrollers, in *Proceedings of the 3rd AES Candidate Conference* (2000).
48. R. Anderson, E. Biham, and L. Knudsen, Information available from, http://csrc.nist.gov/encryption/aes