Occam-pi as a High-level Language for Coarse-Grained Reconfigurable Architectures

Zain-ul-Abdin and Bertil Svensson
 Centre for Research on Embedded Systems (CERES),
 Halmstad University, Halmstad, Sweden.
 Email: Zain-ul-Abdin@hh.se, Bertil.Svensson@hh.se

Abstract—Recently we proposed `occam-pi` as a high-level language for programming coarse grained reconfigurable architectures. The constructs of `occam-pi` combine ideas from CSP and pi-calculus to facilitate expressing parallelism, communication, and reconfigurability. The feasibility of this approach was illustrated by developing a compiler framework to compile `occam-pi` implementations to the Ambric architecture.

In this paper, we demonstrate the applicability of `occam-pi` for programing an array of functional units, eXtreme Processing Platform (XPP). This is made possible by extending the compiler framework to target the XPP architecture, including automatic floating to fixed-point conversion. Different implementations of a FIR filter and a DCT algorithm were developed and evaluated on the basis of performance and resource consumption. The reported results reveal that the approach of using `occam-pi` to program the category of coarse grained reconfigurable architectures appears to be promising. The resulting implementations are generally much superior to those programmed in C and comparable to those hand-coded in the low-level native language NML.

I. INTRODUCTION AND MOTIVATION

Embedded signal processing systems are facing the challenges of increased computational demands. Reconfigurable architectures, which can be configured to form application-specific hardware, offer not only high degree of parallelism but also the possibility to dynamically allocate the resources during run-time, which allows the user to implement applications which are otherwise too large to be handled by a particular device. The reconfigurable computing devices have evolved over the years from gate-level arrays to a more coarse-grained composition of highly optimized functional blocks or even program controlled processing elements, which are operated in a coordinated manner to improve performance and energy efficiency [1]. One of the emerging classes of these parallel architectures is the reconfigurable arrays of functional units, which consist of numerous ALUs composed in a reconfigurable interconnection network. These arrays achieve performance growth by exploiting parallelism, instead of scaling the clock frequency of a single powerful processor.

However, the applicability of coarse-grained reconfigurable arrays in embedded and high performance computing is constrained by the need for mastering low-level structural description languages and their lack of support for expressing dynamic reconfiguration. Furthermore, programs intended for these parallel architectures contain several parts that execute

concurrently, which creates dependencies between the processing elements such that one element needs data computed by another element before it can continue with the processing. It is difficult to realize these communication dependencies at compile time, which limits the exploitable parallelism. Another related challenge with the parallel programming is the tools to avoid deadlock, timing and synchronization errors. Clearly, all these challenges need to be taken care of by using an appropriate programming model.

A traditional way is to use sequential languages such as C, and introduce new software tools that can automatically parallelize the existing sequential code and provide some sort of abstraction layer to enhance portability. The semantics of the sequential languages imposes a lot of restrictions on the execution order of the application and most of the parallelism inherent in the mathematical model of the application is lost while transforming the model to sequential code. The efficient mapping of algorithms to massively parallel architectures requires data dependency analysis to expose the inherent parallelism. The data dependency analysis of C programs involves pointer analysis which is undecidable [2]. Threads have been used as an approach to program concurrent systems in sequential languages, but threads as a model of computation have proved to be highly unpredictable and additional mechanisms are built to overcome the non-determinism [3].

We propose to use a concurrent programming model that allows the programmer to express computations in a productive manner by matching it to the target hardware using high-level constructs, and the language is further supported by a compiler for providing portability across different hardware resources. `Occam` is a programming language based on the Communicating Sequential Processes (CSP) [4] concurrent model of computation and was developed by Inmos for their microprocessor chip Transputer [5]. However, CSP can only represent a static model of the application, where processes synchronize communication over fixed channels. In contrast, the pi-calculus [6] allows modeling of dynamic constructions of channels and processes, which enables the dynamic connectivity of networks of processes. Thus, `occam-pi`, combining CSP with pi-calculus, seems to be an interesting approach to programming of run-time reconfigurable systems. `Occam-pi` [7], is a concurrent programming language that combines the ideas of `occam` with pi-calculus. A program in `occam-pi` is a composition of processes in which communication between

the processes is managed by unbuffered message passing channels. The language features of `occam-pi` allow the programmer to structure highly concurrent programs in such a way that they can be easily understood and effectively use the concurrency features of the underlying hardware.

In earlier work, we have demonstrated the effectiveness of generated HDL code from a CSP based high-level language [8]. In this paper, we present a methodology to program coarse-grained arrays of functional units using `occam-pi` language. The method is based on implementing a compiler backend for generating native code of the target architecture. We will also be focusing on expressing the reconfigurability of the underlying hardware in a programming model by relying on the concepts of mobility introduced in the `pi-calculus`. Previously we have reported the use of `occam-pi` for programming the Ambric array of processors [9]; in this paper the target architecture for our proof of concept implementations is the eXtreme Processing Platform (XPP) [10]. We present the results of streaming FIR and DCT algorithm implementations.

The rest of the paper is organized as follows: Section II provides an overview of the XPP reconfigurable architecture and its programming model. Section III presents the related work. Section IV describes the `occam-pi` language basics and in particular extensions for supporting reconfigurability. Section V describes the compiler framework. Section VI presents the implementation and results of different algorithms, and the paper is concluded with some remarks and future work in Section VII.

II. PACT XPP

The eXtreme Processing Platform (XPP) [10] from PACT XPP Technologies has a packet-oriented communication mechanism for both data and events, which makes it very suitable for compute intensive streaming applications. The PACT XPP64-A1 processor core consists of an 8×8 array of ALU-PAEs (Processing Array Elements) in the middle, along with 16 RAM-PAEs divided into two columns at the edges, as shown in Figure 1. Each ALU-PAE consists of an ALU, forward register (FREG), and backward register (BREG) objects. The ALU object performs 24-bit arithmetic, shift and logical operations. Both FREG and BREG objects are mainly used for routing data and even packets, in addition they also provide support for addition, subtraction, and shift operations. The interconnection network consists of horizontal and vertical buses, and data and event packets are communicated on every clock cycle over these buses. A configuration manager with integrated cache memory configures the array and controls the configuration sequencing. It also has a facility for partial run-time reconfiguration.

The XPP array can be programmed in either Native Mapping Language (NML) or XPP-VC language. NML is a low-level language based on the streaming model of computation which allows full access to all the resources of the array, whereas XPP-VC is a subset of C language, which is compiled by the XPP vectorizing compiler to generate NML code. The vectorization capability of XPP-VC allows loop

iterations to be executed in a pipelined fashion. The XMAP tool available from XPP generates binaries from the NML code.

III. RELATED WORK

There has been a number of initiatives in both the industry and the academia to address the requirement of high-level languages for reconfigurable silicon devices. A few of them based on the CSP model are described here.

Handel-C is a high-level language with ANSI-C like syntax used to program gate level reconfigurable hardware [11]. It supports behavioral descriptions with parallel processing statements (`par`) and constructs to offer communication between parallel elements. Handel-C is being used for compilation to synchronous hardware and inherits sequential behaviors. The communication between the different parallel blocks is handled by channels, which also manage the synchronization.

Streams-C [12], a project initiated by Los Alamos National Laboratory, is based on the CSP model for communication between processes and used for stream-oriented FPGA applications. The Streams-C implementation consists of annotations and library function calls for stream modules. The annotations define the process, stream, and signal. The Abstract Syntax Tree (AST), consisting of sequences of the basic and pipelined data path blocks, is generated by the compiler, and the compiler analyzes the AST for partitioning of control and data flow blocks. Streams-C, which is a subset of ANSI-C, lacks the support for two-dimensional arrays.

Mobius is a tiny, domain specific, concurrent, recently emerging programming language with CSP-based interprocess communication and synchronization methodologies using handshaking [13]. It has a Pascal like syntax with bit specific control and `occam` like extensions, suitable for fine-grained architectures. The hierarchical modules in Mobius are composed of procedures and functions. The processes execute concurrently and communicate with each other through message passing unidirectional channels. A channel consists of *req*, *ack* and *data* signals and provides unbuffered, point-to-point communication.

Although all of the discussed languages are based on the CSP computation model, they differ from each other in the way they expose parallelism. For instance, while Handel-C and Streams-C both have C-like syntax, Streams-C relies on the compiler to expose parallelism whereas Handel-C has extensions of statement level parallel constructs to identify collection of instructions to be executed in parallel. The latter is similar to the approach taken in Mobius. All of the above mentioned languages are intended for fine-grained architectures, whereas we are interested in targeting coarse-grained architectures. Another important feature lacking in all of these languages is the ability to express run-time reconfiguration, which brings us to introduce the `occam-pi` language.

IV. Occam-pi LANGUAGE

The `occam` language [5] is based on the CSP process algebra with well-defined semantics and is a suitable source language because of its simplicity, minimal run-time overhead

Fig. 1. Block diagram of XPP64-A1.

and power to express parallelism. *Occam* has built in semantics for concurrency and interprocess communication. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems.

Occam-pi [7] can be regarded as an extension of classical *occam* to include the mobility feature of the pi-calculus [6]. The mobility feature is provided by the dynamic asynchronous communication capability of the pi-calculus. It is this property of *occam-pi* that is useful when creating a network of processes, where the functionality of processes and their communication network changes at run-time.

A. Basic Constructs

The hierarchical modules in *occam* are composed of processes and functions. The primitive processes provided by *occam* include assignment, input process (?), and output process (!). In addition to these there are also structural processes such as sequential processes (SEQ), parallel processes (PAR), WHILE, IF/ELSE, and replicated processes [5].

A process in *occam* contains both the data and the operations it is required to perform on the data. The data in a process is strictly private and can be observed and modified by the owner process only. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems. In contrast, in *occam-pi* the data can be declared as *MOBILE*, which means that the ownership of the data can be passed between different processes. *occam-pi* also supports *REAL* data type to express floating-point computations. Compared to the channel definition in classical *occam*, the channel type definition in *occam-pi* has been extended to include the direction specifiers, Input (?) and Output (!). Thus a variable of channel type refers to only one end of a channel. The channel types added to *occam* are considered as first class citizens in the type system, allowing the channel ends of that type to be declared and communicated to other processes. A channel direction specifier is added to the type of a channel definition and not to its name. Based on the direction specification, the

compiler performs its usage checking both outside and within the body of the process. Channel direction specifiers are also used when referring to channel variables as parameters of a process call.

Let us now take a look at an *occam-pi* program that computes raise to the power 8 of integers. The main process invokes three instantiations of a process *square*, which are executed in parallel, as shown in Figure 2. The inputs to the main process are passed through input channel-end *in* and the results are retrieved from output channel-end *out*. The *square* process contains a sequential block that takes one input value, computes its square and passes the resulting value at its output channel.

```

PROC main(CHAN INT in?, out!)
  CHAN INT a,b:
  PAR
    square(in?, a!)
    square(a?, b!)
    square(b?, out!)
  :

PROC square(CHAN INT c?, d!)
  INT x,y:
  SEQ
    c ? x
    y = x * x
    d ! y
  :

```

Fig. 2. An *Occam-pi* program.

B. Language Extensions to Support Reconfigurability

In this section, we will describe the semantics of the extensions in the *occam-pi* language, such as mobile data, dynamic process invocation, and process placement attributes. These extensions are used to express the different configurations of hardware resources in the programming model, whose reconfiguration at run-time can be controlled by using dynamic process invocation and process placement attributes.

1) *Mobile Data*: The assignment and communication in classical *occam* follows the copy semantics, i.e., for transferring data from the sender process to the receiver both the sender and the receiver maintain separate copies of the communicated data. The mobility concept of the pi-calculus enables the movement semantics during assignment and communication, which means that the respective data has moved

from the source to the target and afterwards the source loses the possession of the data. In case the source and the target reside in the same memory space, then the movement is realized by swapping of pointers, which is secure and no aliasing is introduced.

In order to incorporate mobile semantics into the `occam` language, the keyword `MOBILE` has been introduced as a qualifier for data types [14]. The definition of the `MOBILE` types is consistent with the ordinary types when considered in the context of defining expressions, procedures and functions. However the mobility concept of `MOBILE` types is applied in assignment and communication. The syntax of mobile data variables and channels of mobile data is given as:

```
MOBILE INT x:
CHAN OF MOBILE INT c:
```

2) *Dynamic Process Invocation*: For run-time reconfiguration, dynamic invocation of processes is necessary. In `occam-pi` concurrency can be introduced by not only using the classical `PAR` construct but also by dynamic parallel process creation using forking. Forking is used whenever there is any requirement of dynamically invoking a new process which can either execute concurrently with the dispatching process or replace the previously executing processes. In order to implement dynamic process creation in `occam-pi`, two new keywords, `FORK` and `FORKING`, are introduced [15]. The scope of the forked process is controlled by the `FORKING` block in which it is being invoked.

The parameters that are allowed for a forked process are:

- `VAL` data type: whose value is copied to the forked process.
- `MOBILE` data type and channels of `MOBILE` data type: which are moved to the forked process.

The parameters of a forked process follow the communication semantics instead of the renaming semantics adopted by parameters of ordinary processes.

3) *Process Placement Attribute*: Having presented the extensions in the `occam-pi` language, we now introduce the placement attribute, which is inspired by the *placed parallel* concept of `occam`. The placement attribute is essential in order to identify the location of the components that will be reconfigured in the reconfiguration process. The qualifier `PLACED` is introduced in the language followed by two integers to identify the location of the hardware resource where the associated process will be mapped. The identifying integers are logical numbers which are translated by the compiler to the physical address of the resource.

V. COMPILATION METHODOLOGY

In this section we will give a brief overview of a method for compiling `occam-pi` programs to reconfigurable arrays of functional units. The method is based on implementing a compiler backend for generating native code for the target architecture. The generated NML code together with the library of NML modules is used by the XMAP tool to generate binaries

Fig. 3. Block diagram of Occam-pi to XPP Compiler.

for configuring the XPP array. These binaries can also be used for simulation and debugging purposes.

A. Compiler for XPP

When developing a compiler for XPP, we have made use of the frontend of an existing *Translator from Occam to C from Kent (Tock)* [16]. The compiler is divided into front end, as shown in Figure 3, which consists of phases up to machine independent optimization, and back end, which includes the remaining phases that are dependent upon the target machine architecture. In this case, we have extended the frontend for supporting `occam-pi` and developed a new backend, targeting XPP, thus generating native code in the proprietary language NML. The Ambric backend targeting Ambric array of processors was developed in an earlier work [9].

In the following, we give a brief description of the modifications that are incorporated in the compiler to support the language extensions of `occam-pi`, introduced to express reconfigurability and the backend to support the XPP.

1) *Frontend*: The frontend of the compiler, which analyzes the `occam-pi` source code, consists of several modules for parsing and syntax and semantic analysis. We have extended the parser and the lexical analyzer to take into account the additional constructs for introducing mobile data types, dynamic process invocation and process placement attributes. We have also introduced new grammar rules corresponding to these additional constructs to create Abstract Syntax Trees (AST) from tokens generated at the lexical analysis stage. Steps for resolving names and type checking are performed at this stage. The frontend also tests the scope of the forking block and whether the data passed to a forked process is of `MOBILE` data type, thus fulfilling the requirement for communication semantics.

In order to support the channel end definition, we have extended the definition of channel type to include the direction whenever a channel name is found followed by a direction token, i.e., ‘?’ for input and ‘!’ for output. In order to implement the channel end definition for a procedure call, we have used the *DirectedVariable* constructor to be passed to the AST whenever a channel end definition is found in the procedure call.

2) *XPP backend*: The XPP backend is further divided into three main passes. The first pass makes use of the *introduceProcSpec* and *genProcess* functions to generate the NML module code corresponding to processes which do not have the PAR and FORK constructs. The pass uses the structured composition of the *occam-pi* constructs, such as SEQ, PAR, and IF/ELSE, which allows intermingling processes and replication of the constructs like (SEQ, PAR). The processes invoked in a PAR construct are mapped to separate hardware resources, which operate concurrently. On the other hand the statements enclosed within a SEQ construct are mapped to hardware resources which operate in a pipeline manner according to the dataflow patterns of the application. The *genProcess* function traverses the AST to generate the corresponding NML code for different primitive processes such as assignment, input process (?), output process (!), and constructs like IF/ELSE, WHILE, and replicated SEQ. The backend generates a separate ALU object for each arithmetic operation, whereas the replicated SEQ construct translates into a COUNT object. In order to reduce the usage of ALU-PAEs, the backend can also use FREG/BREG ALUs instead of standard ALUs for implementing addition/subtraction operations, depending on the optimization flags the we have built into the compiler. The array declarations are mapped to internal RAM blocks within the XPP array and the subsequent use of array variables results in generated code for writing to and reading from the used internal RAM blocks. Reading multiple data values from the input port and writing multiple values on the output port results in more complex control logic as shown in Figure 4.

Fig. 4. Source code and resulting logic diagram illustrating (a) Multiple Inputs, (b) Multiple Outputs.

The *occam-pi* language supports floating-point representation (in the form of REAL data types); however, it is

not supported by XPP. Thus the floating-point numbers are translated into fixed-point numbers during this pass of the XPP backend. The supported fixed-point operations are explained as follows:

- The assignment operation converts the constant value on the right side of the operator to the selected fixed-point format. If the selected format of the left side variable does not have enough precision for representing the constant value, then attributes such as saturation, overflow and rounding are performed on the constant.
- The add and subtract operation are applied directly without any loss of accuracy during the operation.
- The fixed-point multiply operation is implemented as an NML module and this module is instantiated to replace the multiply operator. The word length of the product is equal to the sum of word lengths of the two operands. The multiply NML module consists of shift operations to align the decimal point and throw away the superfluous sign bit.
- The fixed-point division operation is also implemented as an NML module. The divider NML module consists of shift operations to align the decimal part of the result.

The next pass generates the module definitions for the top-level process, including the interface specifications for external IOs and the interface specifications for the instantiated processes specified in the *occam-pi* source code. Before generating the module interface code, the backend traverses the AST to collect a list of all the parameters passed in procedure calls specified for processes to be executed in parallel. This list of parameters, along with the list of names of procedures called, is used to generate the structural interconnection code for each of the parallel modules.

The final pass of the backend generates the *Application* section of the NML code, which is used to specify the configuration management. The *Application* section is generated only if the *occam-pi* source code of the implementation contains FORK statements. The generated *Application* section consists of a number of configurations corresponding to the processes, which are to be reconfigured by a forked process. The *Application* section also contains the configurations containing the RECONF statements for each object defined in the generated NML code. In case of the FORK construct, the backend generates the background NML code for managing the storing of the MOBILE data variables into the internal RAM, whose references are then passed to the forked process. The NML code for loading the MOBILE data variables from internal RAM is also generated by the backend.

VI. IMPLEMENTATION CASE STUDIES

In this section, we present an overview of two quite common signal processing kernels, viz., Finite Impulse Response (FIR) filter and One-Dimensional Discrete Cosine Transform (1D-DCT). We also discuss different implementations of the two kernels, which are developed in NML, XPP-VC, and *occam-pi* languages and then ported to XPP.

Fig. 5. Block diagram of an N -tap FIR filter.

A. FIR Filter

The FIR filter is mainly characterized by the finite impulse response of the filter. An N -tap FIR filter consists of N multiplications and $N-1$ additions as shown in Figure 5.

We have used a streaming approach to implement the 16-tap FIR filter in NML, XPP-VC, and *occam-pi* languages. The *occam-pi* implementation is compiled using two optimization flags of our compiler. One of the optimization flags is used to instruct the compiler to minimize the latency and the other is used to minimize the usage of ALUs in the XPP.

B. 1D-DCT

DCT is a lossy compression technique used in video compression encoders to transform an $N \times N$ image block from the spatial domain to the DCT domain [17].

The DCT algorithm can be expressed either as a matrix multiplication operation between an $N \times N$ input matrix and the cosine coefficients matrix [17] or in a streaming approach based on a set of filters which operate in four stages in a pipelined manner. The dataflow diagram of an 8-point streaming 1D-DCT algorithm is shown in Figure 6. When computing the forward DCT, a 12-bit signed vector is applied to the input on the left, and the forward DCT vector is received at the output on the right.

Both the NML and *occam-pi* implementations are based on the streaming approach to implement the 1D-DCT algorithm, whereas the XPP-VC version is based on the matrix multiplication. The streaming version described in XPP-VC language could not produce correct results.

C. Reconfigurable 1D-DCT

The reconfigurable version of the 1D-DCT algorithm uses the streaming approach, where the four stages of computational tasks are implemented as four configurations which can then be reconfigured on the XPP array successively. Each stage takes input values from its previous stage, performs the computation and supplies the result to the next stage. The first and the last stages are connected to the external I/Os, whereas the intermediate stages take their inputs from the internal RAMs and also write the results into the internal RAMs. The implementation uses FORKING and FORK constructs of *occam-pi* to express which tasks are to be reconfigured, and the MOBILE data type is used for the variables corresponding to the input and output values as shown in Figure 7.

D. Implementation Results and Discussion

The implementation results are obtained by realizing the above-mentioned implementations using the XSIM simulator provided by PACT XPP. We have assumed XPP64-A1 device with 24-bit data and configuration buses width and four ports on each FREG/BREG objects in the simulator. The XPP64-A1 device can operate at a clock frequency of 64 MHz. Table I presents the resources consumed in terms of number of used ALUs, FREGs/BREGs, IRAMs, and I/Os, and the performance results in the form of the number of cycles taken to configure the device added with the initial latency, and the throughput of the implementation. The throughput of the application, measured in number of output samples per second, is calculated according to the following formula:

$$\text{Throughput} = \text{System Frequency} / \text{Cycles per output sample} \quad (1)$$

The throughput results of 16-tap FIR implementations depict that all the implementations can provide one output sample per cycle. The results for utilized resources show that the NML implementation consumes the least resources, followed by the second version of the *occam-pi* implementation which uses special flags to instruct the compiler to minimize the usage of ALUs by using some of the ALUs available in the FREG/BREG objects for add operations. The NML version also requires the least number of cycles for configuration and latency, because of the optimum specifications of placement of each object in the source code, whereas all the other implementations rely on the mapping tool to automatically do the placement. The first *occam-pi* implementation requires slightly more ALUs but has better configuration and latency cycles result, as compared to the second *occam-pi* and XPP-VC implementations.

The results for the NML implementation of the 1D-DCT algorithm are taken from PACT's application note [18], which does not include the configuration and latency cycles. Again the NML implementation produces the best throughput results of producing one output sample every two clock cycles. The *occam-pi* implementation produces one output sample every three cycles, and uses more ALUs as compared to the NML version, but does not require IRAM blocks. The greater use of IRAM blocks by the NML implementation results in the greater usage of FREG/BREG objects for implementing control logic for reading and writing to the internal RAM. The XPP-VC implementation uses the least resources, since it uses the coefficient and input matrices, operating in a sequential manner. This is witnessed in the significantly higher latency of the implementation and much lower throughput as compared to the other two versions.

In the case of the reconfigurable version of the 1D-DCT algorithm implemented in *occam-pi*, we have listed the results corresponding to the four stages implemented as separate configurations which can be loaded successively. The results depict that the overall resource usage is comparable to the corresponding results of the XPP-VC implementation and much

Fig. 6. Dataflow diagram for 1D-DCT.

```

PROC configmgr()
  MOBILE [100]INT idata, odata:
  SEQ
    FORKING
      FORK task2(idata, odata)
    ...
  :

```

(a)

```

PROC task2(MOBILE [100]INT idata,
  MOBILE [100]INT odata)
  PLACED PAR
    PROCESSOR 1,1
      stage2(idata, odata)
  :

```

(b)

```

APPLICATION dctreconf{
  CONFIG c_task1 {
    CONF_MODULE(task1)
  }
  CONFIG r_task1 {
    RECONF(task1.stage1_inst.iNa18)
    RECONF(task1.stage1_inst.iEa18)
    RECONF(task1.stage1_inst.ADD_n0)
    RECONF(task1.stage1_inst.ADD_n3)
    RECONF(task1.stage1_inst.SUB_n4)
    RECONF(task1.stage1_inst.SUB_n5)
    RECONF(task1.stage1_inst.SUB_n6)
    RECONF(task1.stage1_inst.SUB_n7)
    RECONF(task1.stage1_inst.oM7)
    RECONF(task1.stage1_inst.E10)
  }
  CONFIG c_task2 {
    CONF_MODULE(task2)
  }
}

```

(c)

Fig. 7. Source code illustrating (a) Configuration Manager, (b) Task2 process code, (c) NML Application section code.

less than the non-reconfigured *occam-pi* implementation. The total configuration and latency cycles are even less than those for the *XPP-VC* implementation. From the throughput results, it is evident that the reconfigurable implementation can provide output samples at a similar rate as the ordinary *occam-pi* implementation.

VII. CONCLUSIONS AND FUTURE WORK

We have presented our approach of using a CSP based language for programming the emerging class of coarse-grained array architectures. We have also described the mobility features of the *occam-pi* language and the extensions in language constructs that are used to express run-time reconfiguration of the target architecture. The ideas are demonstrated by a working compiler, which compiles *occam-pi* programs to native code for an array of functional units, *XPP*.

Application case studies of FIR and DCT algorithms have been performed and the results corresponding to implementations developed in NML, *XPP-VC*, and *occam-pi* language are presented.

In terms of performance, all the implementations of the FIR filter are able to achieve the same throughput with minor variations in the latency. The *occam-pi* implementation shows that we can instruct the compiler to generate an optimized configuration so that the resource usage becomes comparable to the NML version. From results of the DCT algorithm, it is deduced that the *occam-pi* implementation can achieve much better throughput with improved latency as compared to the *XPP-VC* implementation. The *occam-pi* language also allows expression of different configurations of the algorithm which can be used successively to implement a particular algorithm within limited resources.

TABLE I
UTILIZED RESOURCES AND PERFORMANCE RESULTS.

Implementations		Resources				Performance	
		ALUs	FREGs/BREGs	IRAMs	IOs	Configuration + Latency (cycles)	Throughput (million samples/sec.)
16-tap Integer FIR	NML	16	34	0	2	455	64
	XPP-VC	22	58	0	2	494	64
	Occam-pi Opt-I	31	45	0	2	469	64
	Occam-pi Opt-II	16	54	0	2	507	64
1D-DCT	NML	28	141	10	2	-	32
	XPP-VC	6	56	3	2	3843	0.019
	Occam-pi	42	84	0	2	989	21.3
Reconfigurable 1D-DCT	Stage1(Occam-pi)	8	53	1	1	533	64
	Stage2(Occam-pi)	7	49	2	0	439	32
	Stage3(Occam-pi)	14	35	2	0	462	21.3
	Stage4(Occam-pi)	12	45	1	1	482	21.3

From the programmability point of view, it is observed that programming in *occam-pi* exposes the communication requirements in an application in a more profound way compared to programming in *XPP-VC*. Knowing the communication dependencies, the programmer can parallelize the design according to the available resources and performance requirements. The turnaround time for implementing various designs in *occam-pi* is significantly less as compared to the low-level NML language. The feature of the REAL data type in *occam-pi* also reduces the overall burden on the programmer by converting the floating-point arithmetics to fixed-point by the compiler backend, as compared to manually implementing the fixed-point arithmetics. Furthermore, the support for expressing reconfigurability enables effective reuse of resources.

Raising the abstraction level for the programmer, while not compromising the performance benefits, will be the key to success for the adoption of the emerging reconfigurable architectures in the mainstream computing industry. In short, we can conclude that *occam-pi* raises the abstraction level to a similar level as that of the *XPP-VC* language, thus making it easier for the programmer (measured in lines of code metric and development effort), while the implementation results (measured in terms of throughput) achieved from *occam-pi* programs are comparable to those of NML programs.

Future work will focus on porting more complex applications developed in the *occam-pi* language to XPP, such as autofocus criterion calculation developed for Ambric in previous work [19], and to exploit the run-time reconfiguration capability of XPP for such complex applications. We would also like to extend the compiler framework to target other reconfigurable architectures such as picoarray or Element CXI.

ACKNOWLEDGMENT

The authors would like to thank PACT XPP Technologies for giving access to their software development suite. This research is part of the CERES research program funded by the Knowledge Foundation and the ELLIIT strategic research initiative funded by the Swedish government. The support from European Union Artemis Project SMECY is also acknowledged.

REFERENCES

- [1] Zain-ul-Abdin and Svensson, B.: Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors and Microsystems*, Vol. 33(3), 161-178. (2009)
- [2] Hind, M.: Pointer analysis: Haven't we solved this problem yet? *In Proceedings of PASTE'01*. (2001)
- [3] Lee, E. A.: The problem with threads. *Computer*, Vol. 39(5), 33-42. (2006)
- [4] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall. (1985)
- [5] Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited. (1995)
- [6] Milner, R., Parrow, J., and Walker, D.: *A calculus of mobile processes, Part I*. Information and Computation, 100. (1989)
- [7] Welch, P.H., and Barnes, F.R.M.: *Communicating mobile processes: Introducing occam-pi*. Lecture Notes in Computer Science, Springer Verlag. 175-210. (2005)
- [8] Zain-ul-Abdin and Svensson, B.: A study of design efficiency with a high-level language for FPGAs. *Proceedings of 14'th Reconfigurable Architectures Workshop (RAW) held in conjunction with International Parallel and Distributed Processing Symposium (IPDPS)*. (2007)
- [9] Zain-ul-Abdin, and Svensson, B.: Using a CSP based programming model for reconfigurable processor arrays. *Proceedings of International Conference on Reconfigurable Computing and FPGAs*. (2008)
- [10] Baumgarte, V., Ehlers, G., May, F., Nuckel, A., Vorbach, M., and Weinhardt, M.: PACT XPP: A self-reconfigurable data processing architecture. *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. (2001)
- [11] Handel-C Language Reference Manual, Version 3.1, Celoxica Inc. (2002)
- [12] Frigo, J., Gokhale, M., and Lavenier, D.: Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. 134-140. (2001)
- [13] Mobius Language Manual. Codetronix LLC. (2006)
- [14] Welch, P.H., and Barnes, F.R.M.: *Prioritised dynamic communicating processes: Part I*. Communicating Process Architectures, IOS Press. 321-352. (2002)
- [15] Welch, P.H., and Barnes, F.R.M.: *Prioritised dynamic communicating processes: Part II*. Communicating Process Architectures, IOS Press. 353-370. (2002)
- [16] Tock: Translator from Occam to C by Kent. "https://www.cs.kent.ac.uk/research/groups/sys/wiki/Tock", [Online; accessed 8th July, 2008]
- [17] XAPP610: Video compression using DCT. "http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf", [Online; accessed 16th September, 2006]
- [18] Single Configuration DCT32. PACT XPP Technologies AG. (2003)
- [19] Zain-ul-Abdin and Svensson, B.: Programming real-time autofocus on a massively parallel reconfigurable architecture using Occam-pi. Technical Report IDE1102, Halmstad University, 2011.