Research Article

Occam- π for Programming of Massively Parallel Reconfigurable Architectures

Zain-ul-Abdin and Bertil Svensson

Centre for Research on Embedded Systems (CERES), Halmstad University, 301 18 Halmstad, Sweden

Correspondence should be addressed to Zain-ul-Abdin, zain-ul-abdin@hh.se

Received 29 April 2011; Revised 7 August 2011; Accepted 31 August 2011

Academic Editor: João Cardoso

Copyright © 2012 Zain-ul-Abdin and B. Svensson. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Massively parallel reconfigurable architectures, which offer massive parallelism coupled with the capability of undergoing run-time reconfiguration, are gaining attention in order to meet the increased computational demands of high-performance embedded systems. We propose that the *occam- π* language is used for programming of the category of massively parallel reconfigurable architectures. The salient properties of the *occam- π* language are explicit concurrency with built-in mechanisms for interprocessor communication, provision for expressing dynamic parallelism, support for the expression of dynamic reconfigurations, and placement attributes. To evaluate the programming approach, a compiler framework was extended to support the language extensions in the *occam- π* language and a backend was developed to target the Ambric array of processors. We present two case-studies; DCT implementation exploiting the reconfigurability feature of *occam- π* and a significantly large autofocus criterion calculation based on the dynamic parallelism capability of the *occam- π* language. The results of the implemented case studies suggest that the *occam- π* -language-based approach simplifies the development of applications employing run-time reconfigurable devices without compromising the performance benefits.

1. Introduction and Motivation

The computational requirements of high-performance embedded applications, such as video processing in HDTV, baseband processing in telecommunication systems, and radar signal processing, have reached a level where they cannot be met with traditional computing systems based on general-purpose digital signal processors. Massively parallel reconfigurable processor arrays are made up of highly optimized functional blocks or even program-controlled processing elements composed in a reconfigurable interconnect. The coarse-grained composition leads to less reconfiguration data than in their more fine-grained counterparts, which improves the reconfiguration time, while the communication overhead is also decreased. The ability of coarse-grained reconfigurable architectures to undergo partial and run-time reconfiguration makes them suitable for implementing hardware acceleration of streaming applications.

However, developing applications that employ such architectures poses several other challenging tasks. The pro-

cedural models of high-level programming languages, such as C, rely on sequential control flow, procedures, and recursion, which are difficult to adapt for reconfigurable arrays. The focus of these sequential languages is to provide abstractions for algorithm specification, but the abstractions, intentionally, do not say much about how they are mapped to underlying hardware. Furthermore, because these languages were originally designed for sequential computers with unified memory system, applying them for arrays of reconfigurable processing units with distributed memories results in inefficient use of available hardware, leading to increased power consumption and increased communication delays. The development challenges include the need to learn multiple low-level languages, the requirement of partitioning and decomposing the application into several independent sub-tasks that can execute concurrently, and the need for expressing reconfigurations in order to cope with the adaptability requirements. Clearly, all these challenges need to be taken care of by using an appropriate programming model.

We propose to use the concurrent programming model of *occam-pi* [1], combining Communicating Sequential Processes (CSP) [2] with *pi-calculus* [3]. This model allows the programmer to express computations in a productive manner by matching them to the target hardware using high-level constructs. *Occam-pi*, with its minimal run-time overhead, has built in semantics for concurrency and interprocess communication. The explicit expression of concurrency in *occam-pi*, with its ability to describe computations that reside in different memory spaces, together with the facility of expressing dynamic parallelism, dynamic process invocation mechanisms, and the language support for placement attributes, makes it suitable for mapping applications to a wide class of coarse-grained reconfigurable architectures. These are based on tiles of processing units which have nearest neighbour links, no shared memory, and which are reconfigurable. The compiler that we have developed provides portability across different hardware architectures.

In earlier work, we have demonstrated the feasibility of using the *occam-pi* language to program an emerging massively parallel reconfigurable architecture by implementing a 1D-DCT algorithm [4]. We have also previously demonstrated the applicability of the approach on another reconfigurable architecture, namely, PACT XPP [5]. The contributions of this paper are as follows.

- (i) Identification of a CSP-based programming model and language extensions to express reconfigurability.
- (ii) Implementation of a compiler framework to support language extensions of the *occam-pi* language, such as channel direction specifiers, mobile data and channels, dynamic process invocation, and process placement attributes. These can be used to express run-time reconfiguration in the underlying hardware and for development of the Ambric backend.
- (iii) Evaluation of the approach by implementing a reconfigurable version of the 1D-DCT algorithm and by programming compute-intensive parts of Synthetic Aperture Radar (SAR) systems [6]. In particular, we have used the dynamic process invocation mechanism of *occam-pi* to implement the reconfigurable version of the DCT algorithm and the dynamic parallelism feature of *occam-pi* in the form of replicated parallel processes to implement autofocus criterion calculations on the Ambric array of processors.

The rest of the paper is organized as follows. Section 2 presents some related work, and Section 3 presents the Ambric architecture and its programming environment. Section 4 describes the *occam-pi* language basics, in particular extensions for supporting reconfigurability. Section 5 provides an overview of the compiler framework. Section 6 describes a component framework that is used to implement the dynamic reconfigurability features of *occam-pi*. Section 7 presents the 1D-DCT case study. Section 8 describes the SAR system and the significance of the autofocus algorithm. Section 9 presents the autofocus criterion case study and the two design approaches. Section 10 discusses

the implementation results of the two case studies, and the paper is concluded with some remarks and future work in Section 11.

2. Related Work

There has been a number of initiatives in both industry and academia to address the requirement of high-level languages for reconfigurable silicon devices. The related work presented here covers a range of prominent programming languages and compilers based on their relevance to the field of reconfigurable computing.

Handel-C is a high-level language with ANSI-C like syntax used to program gate level reconfigurable hardware [7]. 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 inherit sequential behaviors.

Streams-C [8], a project initiated by Los Alamos National Laboratory, is based on the CSP model for communication between processes and is 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.

Mobius is a tiny, domain specific, concurrent, recently emerging programming language with CSP-based interprocess communication and synchronization methodologies using handshaking [9]. 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.

Pebble [10] has been developed at Imperial College London to facilitate development of hardware circuits and support modeling of run-time reconfigurations. The language has a block-structured syntax, with the primitive block describing bit-level logic gates. The reconfigurability is supported by introducing control blocks consisting of either a multiplexer or a demultiplexer around the logic block that needs to be reconfigured.

Apart from the above-mentioned languages, there have been attempts to exploit the dynamic reconfiguration capabilities of reconfigurable architectures by implementing a library of custom hardware modules each supporting a specific instruction. These custom-instruction modules are reconfigured onto the FPGA under the control of a global controller which resides on the FPGA [11]. Burns et. al. [12] have proposed a similar approach in which the reconfiguration process is controlled by a run-time system that is executed on a host processor.

To summarize, although most 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 entirely on the compiler to expose parallelism, whereas Handel-C offers extensions. These extensions allow 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 have been implemented for fine-grained architectures, whereas we are interested in targeting coarse-grained architectures. Another important feature lacking in these languages, except Pebble, is the ability to express run-time reconfiguration. With Pebble, the reconfigurability support is provided at a very low level, describing individual logic blocks mainly intended for fine-grained architectures, whereas we are interested in exploring the abstractions to support reconfigurability at task and process level, which is more suitable for coarse-grained architectures. These limitations of the above-mentioned languages have motivated us to suggest using the *occam-pi* language, which provides platform-independent abstractions that enable the programmer to target a variety of coarse-grained architectures. *Occam-pi* can also be adopted for fine-grained architectures, and, in that case, it will closely resemble the Mobius language.

In addition, there are also compiler frameworks such as Riverside Optimizing Compiler for Configurable Computing (ROCCC) [13] for fine-grained architectures and Dynamically Reconfigurable Embedded System Compiler (DRESC) [14] for coarse-grained architectures. Both of these frameworks use C language for application description and perform aggressive program analysis to identify loops in the source code that can then be transformed into pipelines. The loop-based transformations are limited to innermost loops that do not involve function calls. Furthermore, since the C language was originally designed for sequential computers with unified memory system, applying it for arrays of reconfigurable processing units with distributed memories results in inefficient use of available hardware. In contrast to the C language approach, the *occam-pi* language allows the programmer to explicitly describe the statements to be executed in parallel by using the PAR construct. Thus, our compiler framework does not require the loop-level transformations that both ROCCC and DRESC rely on for extracting parallelism, but we do incorporate other optimizations similar to ROCCC such as function inlining, floating-point to fixed-point conversion, and division and multiplication elimination.

3. Ambric Architecture and Programming Model

Ambric, being an example of a massively parallel processor array, is an asynchronous array of so called bricks based on the globally asynchronous locally synchronous (GALS) principle. Each brick is composed of two pairs of Compute Unit (CU) and RAM Unit (RU) [15]. The CU consists of two 32-bit Streaming RISC (SR) processors, two 32-bit Streaming RISC processors with DSP extensions (SRD), and a 32-bit reconfigurable channel interconnect for interprocessor and inter-CU communications. The RU consists of four banks of RAM along with a dynamic channel interconnect to facilitate communication with these memories. The Am2045 device has a total of 336 processors in 42 bricks, as shown in Figure 1.

The Ambric architecture supports a structured object programming model, as shown in Figure 2. The individual objects are programmed in a sequential manner in a subset of the java language, called aJava, or in assembly language [16]. The individual software objects are then linked together using a proprietary language called aStruct. The primitive objects contain the functionality of the component and can be combined together to form a composite software object. Each primitive software object is mapped to an individual processor, and objects communicate with each other using hardware channels without using any shared memory. Each channel is unidirectional, point-to-point, and has a data path width of a single word. The channels are used for both data and control traffic.

Thus, when designing an application in the Ambric environment, the programmer needs to partition the application into a structured graph of objects and define the functions of the individual objects. It is then up to the proprietary tools to compile or assemble the source code and to generate the final configuration after completing placement and routing.

4. Occam-pi Language Overview

Occam [17] is a programming language based on the Communicating Sequential Processes (CSP) concurrent model of computation and was developed by Inmos for their micro-processor chip Transputer. However, CSP can only express a static model of the application, where processes synchronize communication over fixed channels. In contrast, the pi-calculus allows modeling of dynamic constructions of channels and processes, which enables the dynamic connectivity of networks of processes. *Occam-pi* [1] can be regarded as an extension of classical *occam* to include the mobility feature of the pi-calculus. 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 in which the functionality of processes and their communication network change at run time. The *occam-pi* language is based on well-defined semantics and is suitable because of its simplicity, static compilation properties, minimal run-time overhead, and its power to express parallelism and reconfigurability. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems. The dynamic parallelism features of the language make it possible for the compiler to perform resource-aware compilation in accordance with the application requirements.

4.1. 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 [17].

A process in *occam* contains both the data and the operations required to be performed on the data. The data

FIGURE 1: Ambric architecture.

FIGURE 2: Ambric structured object programming model.

in a process is strictly private and can be observed and modified by the owner process only. 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 the *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 Code Example 1. The inputs to the main process are passed through the input channel-end in, and the results are retrieved from the output channel-end out. The *square* process contains a sequential block that takes one input value, computes its square, and passes the resulting value to its output channel.

4.2. 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 and channels, dynamic process invocation, and process placement attributes. These extensions are used to express the different configurations of hardware resources in the programming model. The reconfiguration of the hardware resources at run-time can be controlled by using dynamic process invocation and process placement attributes.

4.2.1. Mobile Data and Channels. The assignment and communication in classical *occam* follow the copy semantics, that is, 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-pi* language, the keyword *MOBILE* has been introduced as a qualifier for data types [18]. The definition of the *MOBILE* types is consistent with how ordinary types

<pre> PROC main (CHAN INT in?, out!) CHAN INT a, b: PAR square (in?, a!) square (a?, b!) square (b?, out!) : </pre>	<pre> PROC square (CHAN INT c?, d!) INT x, y: SEQ c ? x y = x * x d ! y : </pre>
---	--

CODE EXAMPLE 1: An occam-pi program.

are defined 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:

```

The modeling of mobile channels is independent of the data types and structures of the messages that they carry.

Mobile Assignment. Having defined the syntax of mobile types, we now illustrate the movement semantics as applied in the case of the assignment operation. Let us consider the assignment of a variable y to x , where x initially has a value $v0$ and y has an initial value of $v1$. According to the copy semantics of occam, x will acquire the value $v1$ after the assignment has taken place and y will retain its copy of the value $v1$. Instead, applying the movement semantics for mobile assignment, x will acquire the value $v1$ after the assignment has taken place but the value of y will become undefined.

Mobile Communication. Mobile communication is introduced in the form of mobile channel types, and the data communicated on mobile channels has to be of the mobile data type. Channel type variables behave similarly to the other mobile variables. Once they are allocated, communicating them means moving the channel-ends around the network. In terms of pi-calculus, it has the same effect as if passing the channel-end names as messages. Let us explain the mobility concept of pi-calculus by considering a composition of three processes, A, B, and C, such that all of them are executing concurrently as shown in Figure 3, where u and o are the names for input channel-ends and $\bar{u}$ and $\bar{o}$ represent output channel-ends.

Now, in order to undergo a dynamic change of communication topology between processes, process A acquires a channel-end, whereas process B loses its channel-end. The realization of the transfer of channel-end is performed by transmitting the name of the channel-end as the value of the communication between the two processes. Thus, the transmitting process loses the possession of the communicated channel-end. In the case of Figure 3, a mobile channel-end named $\bar{o}$ is sent along the channel u from process B to process A, where $\bar{o}$ becomes undefined in the sending process B

FIGURE 3: Mobile channel-end communication flow graph.

afterwards. The receiving process A receives the channel-end named $\bar{o}$ and later uses it for communicating with process C.

MOBILE Parameter. Passing parameters in an ordinary PROC call consisting of mobile types do not introduce any new semantics implications and are treated as renaming when mobile variables are passed to either functions or processes.

Dynamic Process Invocation. For run-time reconfiguration, dynamic invocation of processes is necessary. In occam-pi, concurrency can be introduced not only by 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 [19]. The scope of the forked process is controlled by the FORKING block in which it is being invoked, as shown in Code Example 2.

The parameters that are allowed for a forked process are

- (i) VAL data type: whose value is copied to the forked process;
- (ii) 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.

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

```

FORKING
  MOBILE INT x:
  SEQ
    x := 42
    FORK P(x)
  :

```

CODE EXAMPLE 2: An example illustrating a forking process.

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.

5. Compiler Framework

In this section, we will give a brief overview of a method for compiling *occam-pi* programs to reconfigurable processor arrays. The method is based on implementing a compiler backend for generating native code for the target architecture.

5.1. Compiler for Ambric. When developing a compiler targeting coarse-grained reconfigurable arrays, we have made use of the frontend of an existing *Translator from Occam to C from Kent* (Tock) [20]. As shown in Figure 4, the compiler is divided into frontend, which consists of phases up to machine independent optimization, and backend, which includes the remaining phases that are dependent upon the target machine architecture. We have extended the frontend for supporting *occam-pi* and developed two new backends, targeting Ambric and PACT XPP, thus generating native code in the proprietary languages aJava, assembly, aStruct, and Native Mapping Language (NML). In this paper, we are only going to describe the Ambric backend, whereas the details of the XPP backend can be found in [5].

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 two target architectures.

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 and channel 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 or not the data

FIGURE 4: Block diagram of *occam-pi* compiler.

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, that is, “?” 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.

The transformation stage, which follows the front end, consists of a number of passes either to simplify its input to reduce complexity in the AST for subsequent phases or to convert the input program to a form which is accepted by the backend or to implement different optimizations required by some specific backend. As mentioned, Tock relies heavily on the use of monad transformers, and we describe here the monad transformer that is used for implementing the target-specific transformations. The *PassM* monad is used to transform the function definition in *occam-pi* to a method in aJava and to avoid wrap-up of PARs to PROCs during the transformation phase.

Ambric Backend. The Ambric backend is further divided into two main passes. The first pass generates declarations of aStruct code, including the top-level design and the interface and binding declarations for each of the composite as well as primitive objects corresponding to the different processes specified in the *occam-pi* source code. Thus, each process in *occam-pi* is translated to a primitive object, which can then be executed on either an SR or an SRD processor of Ambric. Before generating the aStruct code, the backend traverses the AST to collect a list of all the parameters passed

FIGURE 5: Reconfigurable components mapping.

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 interface and binding code for each of the parallel objects.

The next pass makes use of the structured composition of the *occam-pi* constructs, such as *SEQ*, *PAR*, and *CASE*, which allows intermingling processes as well as declarations and replication of the constructs like (*SEQ*, *PAR*, *IF*). The backend uses the *genStructured* function from the generate C module to generate the aJava class code corresponding to processes which do not have the *PAR* construction. In case of the *FORK* construct, the backend generates the background code for managing the loading of the successive configuration from the local storage and communicating it to the concerned processing elements.

Floating-point representation is supported in the *occam-pi* language (in the form of *REAL* data types); however, it is not supported by the Ambric architecture. Thus, a transformation from floating-point numbers to fixed-point numbers has been developed and added to this pass of the Ambric backend. The supported arithmetic operations are explained as follows.

- (i) 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 functions such as saturation, overflow, and rounding are performed on the constant.
- (ii) The add and subtract operations are applied directly without any loss of accuracy during the operation.
- (iii) The multiply operation is implemented as an assembly module, and each instance of the multiply operator is replaced by a function call to the assembly module.
- (iv) The division operation is also implemented as an assembly module. The divider module consists of shift operations to align the decimal part of the result.

6. Implementing the Reconfiguration Framework

Let us explain how the *occam-pi* language can be applied for the realization of dynamic reconfiguration of hardware

resources. The reconfiguration process based on its specification in the *occam-pi* language can be performed by taking into account a work farm design approach [21, 22].

A worker is a particular function mapped onto a specific processor or group of processors. The functionality of an individual worker is described either by one process, or it can be a composition of a number of processes which are interconnected according to their communication requirements. A worker can either occupy one processing element or it can be mapped to a collection of processing elements together performing a particular function as shown in Figure 5. Each worker (indicated as *W1* and *W2*) can have multiple inputs and outputs. The reconfiguration process for the whole application consisting of multiple functions is controlled by a configuration controller (*CC*), which is composed of a configuration loader (*CL*) and a configuration monitor (*CM*). In Ambric, both the loader and the monitor processes are mapped to some of the processors in the array, but, in other cases, the reconfiguration management processes can instead be mapped to dedicated hardware. The configuration loader has a local storage of all the configurations in the form of precompiled object codes, which can then be loaded successively. The order of the reconfigurations is explicitly defined in the configuration loader. The communication channels within each worker are established by taking into account the communication requirements of all the configurations to be mapped on a given set of resources.

Two types of packets are communicated between the configuration loader and different workers, that is, work packets and configuration packets, as shown in Figure 6. The work packets can also be communicated directly to the workers from the external stimuli in case they have multiple inputs. The work packets consist of the data to be processed, and the configuration packets contain the configuration data. Both types of packets are routed to different workers based on either the worker ID or some other identifier. Each worker executes a small kernel to differentiate between the incoming packets based on their header information. Whenever a worker finishes its function, it returns control to its internal kernel after sending a reconfiguration request packet indicating that the particular worker has completed its function and the corresponding hardware resources are ready to be reconfigured to a new configuration. The configuration monitor keeps track of the current state of each worker

FIGURE 6: Communication within the framework of reconfigurable components.

and receives the reconfiguration request from the particular worker once it has completed its specific task and issues it to the configuration loader, which forks a new worker process to be reconfigured in place of the existing worker. The location of the worker is specified by the placement attribute, which consists of two integers. The first integer relates to the identification of the worker, and the second integer identifies the individual processing element within the worker. The placement attributes are logical integers, and they are translated to the physical address of the target architecture by the compiler. The configuration data is communicated in the form of a configuration packet that includes the instruction code for the individual processing elements. The configuration packet is passed around all the processing elements within the worker, where each processing element extracts its own configuration data and passes the rest to its adjacent neighbors.

7. 1D-DCT Case Study

In this section, we present and discuss the reconfigurable implementation of the one-dimensional discrete cosine Transform (1D-DCT), which is developed in *occam-pi* and then ported to Ambric using our compilation platform. DCT is a compression technique used in video compression encoders, such as MPEG encoding, to transform an $N \times N$ image block from the spatial domain to the DCT domain [23]. Since DCT is one part of the overall MPEG encoders, it becomes feasible to implement a reconfigurable version of DCT in order to conserve the hardware resources, so that these resources can be used for implementing other parts of the compression encoders. The mathematics for the 1D-DCT algorithm is described by the following equations:

$$X(k) = C(k) \sum_{n=0}^{N-1} \cos \frac{(2n+1)k\pi}{2N}, \quad 1 \leq k \leq N-1, \quad (1)$$

$$C(0) = \sqrt{\frac{1}{N}}, \quad C(k) = \sqrt{\frac{2}{N}}.$$

We have used a streaming approach to implement the 1D-DCT algorithm, and the dataflow diagram of an 8-point 1D-DCT algorithm is shown in Figure 7. When computing the forward DCT, an 8×8 samples block is input on the left and the forward DCT vector is received as output on the right. The implementation is based on a set of filters which operate in four stages, and two of these stages are reconfigured at run-time based on the framework presented in Section 5. The reconfiguration process is applied between these stages in such a way that when the first two stages are completed, the next two stages of the pipeline are configured on the same physical resources, thus reusing the same processors. The function of “*worker1*,” which consists of the first two stages of the DCT algorithm and which are mapped to two individual SRD processors of “*compute-unit 1*,” as they are invoked in a parallel block. The implementation of the configuration loader as expressed in the *occam-pi* program is shown in Code Example 3(a), which has one output channel-end “*cnf*” of mobile type because it is used to communicate the configuration data. (Note that Code Example 3 only shows the code related to configuration management, not the complete code.) The implementation of the configuration monitor is shown in Code Example 3(b). The configuration monitor will wait until it receives a “RECONFIG” message from the worker, which indicates that the worker has finished performing its functions and the corresponding hardware resource is ready to be reconfigured. The monitor will generate a reconfiguration request message along with the logical address of the resource to be reconfigured to the configuration loader. The configuration loader, upon receipt of a reconfiguration request, will issue a FORK statement, as shown in Code Example 3(a), which includes the name of the process to be configured in place of “*worker1*,” its corresponding configuration data, and its associated channels. The configuration data is defined as a mobile data type, meaning that the configuration loader loses the possession of the configuration data after it has been passed to the forked process. The new forked “*worker2*” process has

FIGURE 7: Dataflow diagram for 1D-DCT.

```

PROC loader (CHAN INT inp?, CHAN MOBILE INT cnf!,
             CHAN INT ack?)
INT cstatus, value, id:
MOBILE [100] INT config:
CHAN MOBILE INT cnf:
CHAN INT res:
VAL RECONFIG IS 255:
SEQ
  FORKING
    WHILE TRUE
      SEQ
        inp ? value
        cnf ! value
        ack ? cstatus
        IF
          cstatus = RECONFIG
            SEQ
              ack ? id
              IF
                id = 1
                  FORK worker2 (config, cnf?, res!)
                id = 2
                ...
  :
(a)

```

```

PROC monitor (CHAN INT res?, CHAN INT ack!,
              CHAN INT outp!)
INT status:
VAL RECONFIG IS 255:
WHILE TRUE
  SEQ
    res ? status
    IF
      status = RECONFIG
        ack ! RECONFIG
      status <> RECONFIG
        outp ! status
:
(b)

```

```

PROC worker2 (MOBILE [100] INT config,
              CHAN MOBILE INT cnf?, CHAN INT res!)
CHAN INT ch:
PLACED PAR
  PROCESSOR 1,1
    stage3 (config, cnf?, ch!)
  PROCESSOR 1,2
    stage4 (config, ch?, res!)
:
(c)

```

CODE EXAMPLE 3: Source code illustrating (a) Configuration loader, (b) configuration monitor, and (c) worker process.

the same placement attributes as those of “worker1,” as shown in Code Example 3(c), meaning that the “worker2” process will be mapped to the same processing element as that of the “worker1” process. The newly configured “worker2” process consists of the last two stages of the DCT algorithm. The computed results of “worker1” are also passed

from the monitor to the configuration loader and are fed into the “worker2” process along the same channel that is used for communicating configuration data. The computations of different stages of the DCT algorithm are described in the form of expressions in separate processes that are invoked in a parallel block in the individual worker processes.

8. SAR and Autofocus

In this section, we illustrate our approach on a larger application example, part of a synthetic aperture radar (SAR). SAR systems can be used to create high-resolution radar images from low-resolution aperture data. A SAR system produces a map of the ground while the platform is flying past it. The radar transmits a relatively wide beam to the ground, illuminating each resolution cell over a long period of time. The effect of this movement is that the distance between a point on the ground and the antenna varies over the data collection interval. This variation in distance is unique for each point in the area of interest. This is illustrated in Figure 8 where the area to be mapped is represented by $M \times N$ resolution cells and L represents the number of pulses. The cells correspond to paths in the collected radar data. The task for the signal processor is to integrate, for each resolution cell in the output image, the responses along the corresponding path. The flight path is assumed to be linear.

8.1. Image Forming. A computationally efficient method for creating the image is the Fast Factorized Back-Projection (FFBP) [24]. In FFBP, the whole image initially consists of a large number of small subimages with low angular resolution. These subimages are iteratively merged into larger ones with higher and higher angular resolution, until the final image with full angular resolution is obtained. The autofocus method used here assumes a merge base of two subimages.

8.2. Autofocus. In reality, the flight path is not perfectly linear. This can, however, be compensated for in the processing. In the FFBP, the compensations typically are based on positioning information from GPS. If this information is insufficient or even missing, autofocus can instead be used. The autofocus calculations use the image data itself and are done before each subaperture merge. One autofocus method, which assumes a merge base of two, relies on finding the flight path compensation that results in the best possible match between the images of the contributing subapertures in a merge. Several flight path compensations are thus tested before a merge. The image match is evaluated according to a selected focus criterion, as shown in Figure 9. The criterion assumed in this study is maximization of correlation of image data. As the criterion calculations are carried out many times for each merge, it is important that these are done efficiently. Here, the effect of a path error is approximated to a linear shift in the data set. Thus, a number of correlations between subimages that are slightly shifted in data are to be carried out. Interpolation is performed in order to compute the value from samples in the contributing data set of the subimages. More details about the calculations are given in the next section. Autofocus in FFBP for SAR is further discussed in [25].

8.3. Performance Requirements. The integration time may be several minutes. The computational performance demands are tens or hundreds of GFLOPS. The large data sets themselves represent a challenge but also the complicated

FIGURE 8: Simplified illustration of stripmap SAR.

memory addressing scheme due to, for example, changing geometric proportions during the processing. The exact computational requirements are dependent on the chosen detailed algorithms and radar system parameters.

9. Autofocus Criterion Case Study

In order to realize the autofocus algorithm on the Ambric platform, the first step in the development process is to determine the dataflow patterns of the algorithm and estimate an approximate amount of resources to be used. The next step is to write the *occam-pi* application code in terms of processes based on the dataflow diagram and compose these processes to be executed either in sequence or in parallel to each other. The *occam-pi* application is tested for functional correctness by using the Kent Retargetable *occam* Compiler (KRoC) [26] run-time system, and finally the *occam-pi* application code is compiled by our compiler to the native languages of Ambric. The generated code can then be compiled to generate binaries for the Ambric platform using its proprietary design environment.

We have implemented two versions of the same algorithm, with a different degree of parallelism exploited by the two approaches. We have used a parameterized approach for both of the designs, so the amount of parallelism can be varied easily by using the construct of replicated PAR of *occam-pi* based on parameters such as area of interest (A) and number of pixels processed per interpolation kernel (P). In addition, there are the other parameters of degree of shift and degree of tilt which are to be passed to the algorithm. Both design approaches take as an input two 6×6 blocks of image pixels from the area of interest of the contributing images. Cubic interpolation based on Neville's algorithm [27] is performed in the range direction followed by the beam direction to estimate the value of the contributing pixels along the tilted lines, and the resulting subimages are to be correlated according to the autofocus criterion. Figure 10(a) illustrates how an interpolated value is computed from samples in the contributing data set, and Figure 10(b) indicates how the intermediate interpolated results in the range direction are reused in order to calculate several

FIGURE 9: Illustration of autofocus and focus criterion.

interpolated values in the beam direction. Each pixel data comprises two 32-bit floating-point numbers corresponding to the real and imaginary components. These floating-point numbers are represented by the REAL data type in *occam-pi*, and the REAL data values are translated to Q16 format fixed-point representation. For fixed-point arithmetics, specialized assembly language code is inserted in place of arithmetic operations in the generated code by the compiler backend. Following is a description of the two design approaches.

9.1. Design-I. In the first design, we have used six-range interpolators to calculate the cubic interpolation along the six rows of pixel data in one of the two input pixel blocks, as shown in Figure 11. The input pixel data is fed to the range interpolators through two splitters, which route the pixel data values received from the source distributor block. Since there are no arithmetic computations performed by the source and splitter blocks, when mapped on the Ambric array, these blocks are implemented on SR processors.

The range interpolators perform the same operation on different rows of pixel data. During the first iteration, each range interpolator takes data values corresponding to four pixels from their inputs and performs the cubic interpolation; then the resultant interpolated pixel data values are passed to the beam interpolation stage. Each range interpolator is implemented on a set of three SRD processors which are connected in a pipeline manner. Since the computed results of the different range interpolators are to be used by multiple beam interpolators, some of the range interpolator blocks have multiple outputs and the resulting interpolated data is copied to these multiple outputs.

The next stage in the dataflow diagram is to perform the cubic interpolation in the beam direction. Three beam interpolators are implemented to perform the beam interpolation on the resulting output of the range interpolation stage. Similar to the range interpolator, each beam interpolator block is also composed of three SRD processors connected in a pipeline. Each beam interpolator takes four inputs from four different range interpolators, and it receives its input data values corresponding to four-range interpolated pixels on these input ports. The resulting data values of the beam interpolation stage is passed to the three correlators. Each correlator is implemented on one SRD processor and takes

pixel data values from each of the pixel data blocks, calculates their correlation, and passes the result to the summation block to calculate the final autofocus criterion. The summation block is implemented on a single SRD processor. Three iterations of the range interpolation, beam interpolation, correlation, and summation stages are performed in order to compute the autofocus criterion for the entire 6×6 image block.

The parameters of area of interest (A) and number of pixels processed per interpolation kernel (P) have been used in the construct of replicated PAR of *occam-pi* to control the resource usage as shown in Code Example 4. The values of these parameters determine how many instances of processes invoked in the replicated PAR block will be generated, that is, how many split, rangeinterp, beaminterp, and so forth processes will be instantiated. The parameters of an invoked process define the input and output channel-ends used by the said process. Based on the definition of the input and output channel-ends required by each process, the compiler generates the static interconnections between different processes.

9.2. Design-II. The second design uses three times as many range and beam interpolators as the first design, as shown in Figure 12, so that only one iteration of execution of each of the stages will result in computation of the autofocus criterion for the complete 6×6 pixel block. However, due to the limitation of the maximum number of SRD processors available on the Am2045 chip that we are using as a target for realization, we have to reduce the number of pipelined processors within each of the range and beam interpolation blocks to two.

The increase in the number of range interpolators is also reflected in the increase in the number of splitters, so there are six splitters used to feed the 18-range interpolators performing the cubic interpolation in the range direction of one of the input 6×6 pixel blocks. The six splitters are fed by a single source through two source distributors, because the number of output ports on an SR processor cannot exceed five. As in the first design, the source and source distributors are executed on SR processors. The dataflow patterns from the range interpolators to the beam interpolators, from the beam interpolators to the correlators, and further on to

FIGURE 10: Usage of cubic interpolation kernel to calculate contributing pixel data points.

the summation stage are similar to those in the previous design except that we now have separate resources for each iteration of interpolation and correlation stages.

10. Implementation Results and Discussion

10.1. ID-DCT Case Study. We now present the results of the reconfigurable 1D-DCT which is implemented by using the framework presented in Section 4. Our aim in this case study is to demonstrate the applicability of the programming model of *occam-pi*, together with the proposed framework for expressing reconfigurability, thus we do not claim to achieve efficient implementations with respect to performance. The application case studies to prove the performance benefit of carrying out reconfigurations using our proposed methodology is part of our future work.

The coarse-grained parallelized DCT is implemented in a four stage pipeline. Earlier results reveal that an implementation using four SRD processors takes 1340 cycles to compute 64 samples of 1D-DCT [4]. This time includes the time consumed during communication stalls between different stages. We compared this implementation with a reconfigurable one that uses only two SRD processors, which are reconfigured to perform the different stages successively. The computation of the same amount of samples now takes 2612 cycles, which includes the cycle count for the reconfiguration process, which is 550 cycles. The number of instruction words to be stored in the local memory of each individual processor is 97. The SRD processor takes 2 cycles to write one memory word in its local memory, thus the memory writing time is a significant part of the overall reconfiguration time. The reconfiguration process is controlled in such a way that the time taken by the two processors to update their instruction memories is partially overlapped, meaning that the first processor will be performing computations while the second one is being reconfigured.

In the two-processor reconfigurable implementation, most of the communication stalls that appeared in the four processor implementation are eliminated, and time is instead used for the reconfiguration management. The results also show that the reconfiguration time is one fifth of the overall time of computation, indicating reasonable feasibility of the approach.

10.2. Autofocus Criterion Case Study. For the autofocus application, the implementation results are achieved by realizing both the designs on the Ambric Am2045 architecture and executing them on a GT board containing one Am2045 chip being operated at 300 MHz clock. We have used the performance harness library provided by Nethra Imaging Inc. to obtain cycle accurate performance measurements. We have also obtained results of a sequential version of the same algorithm by executing it as a single threaded application on an Intel i7-M620 CPU operating at 2.67 GHz. Table 1 presents the resources consumed in terms of number of used SRD processors, SR processors, and RU banks, alongwith the percentage of total amount of available resources.

The greater number of SRD processors used as compared to the SR processors is due to the fact that most of the blocks involve complex arithmetics which cannot be performed on SR processors, and also due to the limited instruction memories of the SR processors, which in this case makes them useful only for data distribution. A significant number of RU banks are used to store the additional instructions for SRD processors that exceed the internal memory of 256 words. Some of the RU bank memory is also used in implementing FIFO buffers on the channels between different processors to reduce the effect of communication stalls. When going from the first design to the second one, the number of SRD processors should be three times the number of SRD processors used in Design-I, but, due to the limited number of available SRD processors, we have to reduce

FIGURE 11: Dataflow diagram of Design-I.

TABLE 1: Resources consumed for autofocus criterion calculation.

	SRDs	SRs	RU Banks
Am2045 full capacity	168	168	336
Design-I on Ambric	70 (42%)	24 (14%)	113 (34%)
Design-II on Ambric	141 (84%)	28 (17%)	208 (62%)

the pipelined processors within each interpolator to two. The use of the performance harness library results in the use of one additional SR and one additional SRD processor, as well as six additional RU banks.

Table 2 shows performance and power results: the latency, in cycle count, for producing first correlation output,

the throughput, in terms of number of pixels per second on which the given autofocus criterion is computed, and the speedup figures for the design realized on Ambric compared to a sequential implementation executed on Intel i7-M620 CPU. It also shows the estimated power consumed by the two parallel and one sequential implementation based on the figures obtained from Am2045 [28] and Intel i7-M620 processor [29] data sheets.

The latency results of the design-II depict an improvement in terms of 30% less cycles as compared to design-I. The throughput of the second design is 2.1x times the throughput of the first design, and the throughput speedup with respect to the sequential implementation is 11x and 23x,

```

PROC autofocus(VAL INT A, P, VAL REAL xintr, xinti, CHAN REAL dinp0?, dinp1?, CHAN REAL res!)
  [(A/P)*2] CHAN REAL doutp:
  [A*2] CHAN REAL32 soutp:
  [A*4] CHAN REAL routp:
  [A] CHAN REAL boutp:
  [A/2] CHAN REAL coutp:
  PAR
    datadist(A,dinp0?,doutp[0]!,doutp[1]!)
    PAR i=0 FOR ((A/P)-1)
      PAR j=0 FOR ((A/P)-1)
        PAR
          split(A,doutp[(i*2)+j]?,soutp[(i*3)+(j*3)]!,soutp[((i*3)+(j*3))+1]!,
            soutp[((i*3)+(j*3))+2]!)
          rangeintp1(xintr,xinti,soutp[(i*6)+(j*5)]?,routp[(i*12)+(j*11)]!)
          rangeintp2(xintr,xinti,soutp[((i*6)+(j*3))+1]?,routp[((i*12)+(j*8))+1]!,
            routp[((i*12)+(j*8))+2]!)
          rangeintp3(xintr,xinti,soutp[((i*6)+(j*1))+2]?,routp[((i*12)+(j*3))+3]!,
            routp[((i*12)+(j*3))+4]!,routp[((i*12)+(j*3))+5]!)
        PAR j=0 FOR (A/P)
          beamintp(xintr,xinti,routp[(i*12)+(j*4)]?,routp[((i*12)+(j*4))+1]!,
            routp[((i*12)+(j*4))+2]!,boutp[(i*3)+j]!)
      PAR j=0 FOR (A/P)
        corr(boutp[i]?,boutp[i+3]?,coutp[i]!)
    corrsun(coutp[0]?,coutp[1]?,coutp[2]?,res!)
  :

```

CODE EXAMPLE 4: Simplified illustration of Design-I implementation of autofocus criterion calculation in *occam-pi*.

TABLE 2: Performance and estimated power results of autofocus criterion calculation.

Implementations	Latency (cycles)	Throughput (pixels/sec.)	Speedup throughput	Power (watts)
Sequential on Intel i7 @ 2.67 GHz	—	21,600	1	17.5
Design-I on Ambric @ 300 MHz	16,497	236,386	11	6.52
Design-II on Ambric @ 300 MHz	12,793	486,224	23	9.8

respectively, for the two designs. With 94 processors which are clocked 9 times slower, a speedup of 11 shows that the design programmed in *occam-pi* is indeed efficient. Ideally, the throughput of design-II should have been three times that of design-I, but the use of almost twice the number of processors results in some communication stalls in between the data distribution and interpolation stages. Also, the effects of the reduced number of pipelined processors within individual interpolators is reflected in the reduction of the throughput. The two designs realized on Ambric consume much less power than the traditional one, and they provide 29x and 40x, respectively, more throughput per watt as compared to the sequential implementation.

We have experienced that the program code for the different stages of the cubic interpolation kernel to be executed on the pipelined processors have to be optimized to be able to fit into at most two RU banks of memory for each SRD processor. Otherwise, if it exceeds the size of two RU banks, the placement tool cannot make use of the second SRD processor available in the same compute unit of the Ambric architecture. The optimization is achieved by generating the assembly code for the fixed-point arithmetics used in the cubic interpolation kernel by the compiler backend. Other optimizations implemented in the compiler

include scalarization of array variables and exploitation of instruction level parallelism by using the `mac_32_32` instruction in place of successive multiplication and addition instructions.

11. Conclusions and Future Work

We have presented an approach of using a CSP-based language for programming the emerging class of processor 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 reconfigurability. The ideas are demonstrated by a working compiler, which compiles *occam-pi* programs to native code for an array of processors, Ambric. The presented approach is evaluated by implementing one common signal processing algorithm and one more complex case study which is part of a radar signal processing algorithm.

In terms of performance, the two implementations of Autofocus criterion calculation targeted on ambric outperform the CPU implementation by factors of 11–23, despite operating at a clock frequency of 300 MHz as compared to 2.67 GHz. This shows that the designs programmed in *occam-pi* are indeed efficient. The use of a much lower clock

FIGURE 12: Dataflow diagram of Design-II.

frequency together with the switching off of unused cores in the Ambric architecture provides the side advantage of a significant reduction in energy consumption of the two parallel implementations, which is an important factor to consider for embedded systems. The reconfigurable versions of the benchmark algorithms prove that the *occam-pi* language allows expression of different configurations of the

algorithm which can be used successively to implement a particular algorithm within limited resources.

From the programmability point of view, it is observed that the explicit concurrency of *occam-pi* with the ability to describe computations that reside in different memory spaces, together with the dynamic process invocation mechanism, makes it suitable for mapping applications to massively

parallel reconfigurable architectures. The *occam-pi* language is based on well-defined semantics, and its simplicity, static compilation properties, minimal run-time overhead, and power to express parallelism help in the task of parallelization. The existence of the *REAL* data type in *occam-pi* and the introduced conversion of the floating-point arithmetics to fixed-point by the compiler backend also reduces the overall burden on the programmer, compared to manually implementing the fixed-point arithmetics. Furthermore, the support for expressing dynamic parallelism in the form of replicated *PAR* constructs enables the compiler to perform resource-aware compilation in accordance with the application requirements. The reconfigurability support allows effective reuse of resources, and the placement attributes allow processes to be colocated, which gives a potential to avoid unnecessarily expensive communication. In addition to the language features, our proposed methodology of testing the functionality of the application in *occam-pi* before compiling the generated native code using the Ambric design environment reduces the turnaround time for implementing various design alternatives quite significantly.

In conclusion, using the *occam-pi* language is a practical and flexible approach to enable mapping of applications to massively parallel reconfigurable architectures that are based on globally asynchronous locally synchronous (GALS) principle and distributed memory model. The success of the approach stems from the well-defined semantics of the language that allows the expression of concurrent computations, interprocess communication, and reconfigurations with a formal basis. By simplifying these tasks, the problem of efficiently mapping applications to the massively parallel reconfigurable architectures is more readily addressed, as demonstrated in this work.

Future work will focus on developing more complex applications in the *occam-pi* language to exploit the run-time reconfiguration capability of the target hardware and on extending the compiler framework to target other reconfigurable architectures such as *picoArray* and *Element CXI*.

Acknowledgments

The authors would like to thank Nethra Imaging Inc. for giving access to their software development suite and hardware board. They would also like to acknowledge the support from Saab AB for the SAR application. 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.

References

- [1] P. H. Welch and F. R. M. Barnes, "Communicating mobile processes introducing *occam-pi*," in *Proceedings of the Symposium on the Occasion of 25 Years of CSP—Communicating Sequential Processes*, Lecture Notes in Computer Science, pp. 175–210, Springer, July 2004.
- [2] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
- [3] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.
- [4] Zain-ul-Abdin and B. Svensson, "Using a CSP based programming model for reconfigurable processor arrays," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 343–348, December 2008.
- [5] Zain-ul-Abdin and B. Svensson, "Occam-pi as a high-level language for coarsegrained reconfigurable architectures," in *Proceedings of the 18th International Reconfigurable Architectures Workshop held in Conjunction with International Parallel and Distributed Processing Symposium (IPDPS/RAW '11)*, Anchorage, Alaska, USA, May 2011.
- [6] A. Åhlander, H. Hellsten, K. Lind, J. Lindgren, and B. Svensson, "Architectural challenges in memory-intensive, real-time image forming," in *Proceedings of the 36th International Conference on Parallel Processing in Xi'an (ICPP '07)*, September 2007.
- [7] "Handel-C language reference manual," Version 3.1, Celoxica, 2002.
- [8] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: an applications perspective," in *Proceedings of the ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays (FPGA '01)*, pp. 134–140, February 2001.
- [9] "Mobius Language Manual," Codetronix LLC, 2006.
- [10] W. Luk and S. McKeever, "Pebble: a language for parametrised and reconfigurable hardware design," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, vol. 1482 of *Lecture Notes in Computer Science*, pp. 9–18, Springer, 1998.
- [11] M. J. Wirthlin and B. L. Hutchings, "Dynamic instruction set computer," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99–107, April 1995.
- [12] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. D. Wit, "A dynamic reconfiguration run-time system," in *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [13] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 127–134, May 2010.
- [14] M. Bingfeng, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures," in *Proceedings of the IEEE International Conference on Field-Programmable Technology*, 2002.
- [15] A. M. Jones and M. Butts, "TeraOPS hardware: a new massively-parallel MIMD computing fabric IC," in *Proceedings of IEEE Hot Chips Symposium*, 2006.
- [16] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pp. 55–64, April 2007.
- [17] "Occam® 2.1 reference manual, SGS-Thomson Microelectronics Limited," 1995.
- [18] P. H. Welch and F. R. M. Barnes, "Prioritised dynamic communicating processes: part I," in *Communicating Process Architectures*, pp. 321–352, IOS Press, 2002.

- [19] P. H. Welch and F. R. M. Barnes, “Prioritised dynamic communicating processes—part II,” in *Communicating Process Architectures*, pp. 353–370, IOS Press, 2002.
- [20] Tock, “Translator from Occam to C by Kent,” <http://projects.cs.kent.ac.uk/projects/tock/trac>, 2011.
- [21] M. Butts, B. Budlong, P. Wasson, and E. White, “Reconfigurable work farms on a massively parallel processor array,” in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 206–215, April 2008.
- [22] N. Shirazi, W. Luk, and P. Cheung, “Run-time management of dynamically reconfigurable designs,” in *Proceedings of the 8th International Workshop on Field Programmable Logic and Applications (FPL '98)*, vol. 1482 of *Lecture Notes in Computer Science*, Springer, 1998.
- [23] Xilinx, “Video compression using DCT,” <http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf>, 2006.
- [24] L. M. H. Ulander, H. Hellsten, and G. Stenström, “Synthetic-aperture radar processing using fast factorized back-projection,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 3, pp. 760–776, 2003.
- [25] H. Hellsten, P. Dammert, and A. Åhlander, “Autofocus in fast factorized backprojection for processing of SAR images when geometry parameters are unknown,” in *Proceedings of the IEEE International Radar Conference (RADAR '10)*, pp. 603–608, May 2010.
- [26] “KRoc: Kent Retargetable Occam Compiler,” <http://www.cs.kent.ac.uk/projects/ofa/kroc/>, 2010.
- [27] E. H. Neville, “Iterative interpolation,” *Journal of Indian Mathematical Society*, vol. 20, pp. 87–120, 1934.
- [28] *Am2045 Data Book*, Ambric, 2007.
- [29] Intel® Core™ i7-600, i5-500, i5-400, and i3-300 Mobile processor series datasheet, Intel Corporation, 2010.