Specifying Run-time Reconfiguration in Processor Arrays using High-level language

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

Abstract. The adoption of run-time reconfigurable parallel architectures for high-performance embedded systems is constrained by the lack of a unified programming model which can express both parallelism and reconfigurability. We propose to program an emerging class of reconfigurable processor arrays by using the programming model of ocaml-pi and describe how the extensions of channel direction specifiers, mobile data, dynamic process invocation, and process placement attributes can be used to express run-time reconfiguration in ocaml-pi. We present implementations of DCT algorithm to demonstrate the applicability of ocaml-pi to express reconfigurability. We concluded that ocaml-pi appears to be a suitable programming model for programming run-time reconfigurable processor arrays.

1 Introduction and Motivation

The design of high-performance embedded systems for signal processing applications is facing the challenges of not only increased computational demands but also increased demands for adaptability to future functional requirements for these applications. Reconfigurable parallel architectures offer the possibility to dynamically allocate the resources during run-time, which allows the user to implement applications which adapt according to changing demands and workloads. 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. These reconfigurable processor arrays are well suited for streaming applications that have highly regular computational patterns.

However, developing applications that employ massively parallel reconfigurable architectures poses several challenges. Traditionally, system developers have either used low-level proprietary languages or relied on programming in C and the use of advanced synthesis tools and automatic parallelization techniques; however, the latter techniques lag in terms of achieved run-time performance. Moreover, existing tools mainly support reconfiguration of the complete device, thus allowing changes in the hardware only at a relatively slow rate. The procedural models of imperative languages, such as C and Pascal, rely on sequential
control flow because these languages were originally designed for sequential computers with unified memory system. Applying them for arrays of reconfigurable processing units result in limited extraction of instruction level parallelism, leading to inefficient use of available hardware and increased power consumption.

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. Portability across different hardware resources is provided by means of a compiler. **Occam** is a programming language based on the Communicating Sequential Processes (CSP) [1] concurrent model of computation. However, CSP can only represent a static model of the application, where processes synchronize communication over fixed channels. In contrast, the pi-calculus [2] allows modeling of dynamic constructions of channels and processes, which enables the dynamic connectivity of networks of processes. Thus, **occam-pi** [3], combining CSP with pi-calculus, seems to be an interesting approach to programming of run-time reconfigurable systems.

In earlier work, we have demonstrated the effectiveness of generated code from the **occam-pi** language for the Ambric [4] array of processors [5]. In this paper, 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. The target architecture for our first proof of concept implementations is the Ambric fabric of processors and we believe that the use of **occam-pi** as a unified programming language is suitable for other reconfigurable architectures such as PACT XPP [6] and ElementCXI programmable device [7]. We present the results of streaming DCT algorithm implementation.

## 2 Occam-pi language

The **occam** language [8] 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 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** [3] can be regarded as an extension of **occam** to include the mobility features of the pi-calculus [2]. The mobility feature is provided by the dynamic asynchronous communication capability of the pi-calculus, which is useful when creating a network of processes that changes its configuration at run-time.

### 2.1 Basic Constructs

The hierarchical modules in **occam** are composed of procedures and functions. The primitive processes provided by **occam** include assignment, input process (\(?\)), output process (\(!\)), skip process (\(\text{SKIP}\)), and stop process (\(\text{STOP}\)). In addition to these there are also structural processes such as sequential processes (\(\text{SEQ}\)), parallel processes (\(\text{PAR}\)), \(\text{WHILE}\), \(\text{IF/ELSE}\), \(\text{CASE}\), and replicated processes [8].
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. 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.

2.2 Language Extensions to Support Reconfigurability

In the following section, we will describe the semantics of the extensions in the occam-pi language such as channel direction specifiers, mobile data, dynamic process invocation, and process placement attributes. These extensions are used to express the reconfiguration of hardware resources in the programming model.

Channel Direction Specifier: The channel type definition has been extended to include the direction specifiers, Input (?) and Output (!). Thus a variable of channel type refers to only one end of channel. 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.

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 [9]. The definition of the MOBILE types is consistent with the ordinary types when considered in the context of defining expressions, procedures and functions.

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 execute concurrently with the dispatching process. In order to implement dynamic process creation in occam-pi, two new keywords FORK and FORキング, are introduced [10]. 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.

**Process Placement Attribute**: The placement attribute is essential in order to identify the location of the components that will be reconfigured in the reconfiguration process, and it is inspired by the *placed parallel* concept of **occam**. 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.

### 3 Compilation Methodology

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

#### 3.1 Ambric Architecture and Programming Model

Ambric is an asynchronous array of so called brics, each composed of two pairs of Compute Unit (CU) and RAM Unit (RU) [4]. 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 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 45 brics.

The architecture was designed to support a structured object programming model. Using the proprietary tools the individual objects are programmed in a sequential manner in a subset of the java language, called **aJava** or in assembly language [11]. 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 individual software objects are then linked together using the proprietary language called **aStruct**.

#### 3.2 Compiler for Ambric

When developing a compiler for Ambric, we have made use of the frontend of an existing **Translator from Occam to C from Kent** (Tock) [12]. The compiler is divided into front end, which consists of phases up to machine independent
optimization, and back end, which includes the remaining phases that are de-
pendent upon the target machine architecture. In this case, we have extended
the frontend for supporting occam-pi and developed a new backend, targeting
Ambric, thus generating native code in aJava and aStruct.

In the following we give a brief description of the modifications that are incor-
porated in the compiler to support the language extensions of occam-pi,
introduced to express reconfigurability.

**Frontend:** The frontend of the compiler, which analyzes the source code in
occam-pi, consists of several modules for parsing and syntax and semantic anal-
ysis. We have extended the parser and the lexical analyzer to take into account
the additional constructs for introducing mobile data types, dynamic process in-
vocation and process placement attributes. We have also introduced new gram-
mar rules corresponding to these additional constructs to create Abstract Syntax
Trees (AST) from tokens generated by the lexical analysis. 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 defini-
tion 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.

**Ambric backend:** The Ambric backend is further divided into two main passes.
The first pass generates declarations of aStruct code including the top-level de-
sign, 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. Before generating the aStruct code, the backend tra-
verses 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
interface and binding code for each of the parallel objects.

The next pass makes use of the structured composition of the occam con-
structs, such as SEQ, PAR, and CASE, which allows intermingling processes and
declarations and replication of the constructs like (SEQ, PAR, IF). The backend
uses the genStructured function from the generateC module of the C backend
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.
4 Implementing the Reconfigurable Framework

Let us explain how the \texttt{occam-pi} language can be applied for the realization of dynamic reconfiguration of hardware resources. The reconfiguration process based on its specification in the \texttt{occam-pi} language can be performed by taking into account a work farm design approach [13] as shown in Figure 1.

![Fig. 1. Framework of Reconfigurable Components.](image)

A worker is a specific area of hardware executing a particular task. The task can either consist of one process, or it can be composed of a number of processes which are interconnected according to their communication requirements. A worker can either occupy one processing element or be mapped to a collection of processing elements. Each worker can have multiple inputs and outputs, but in Figure 1, we show only the connections used during the reconfiguration.
process. The reconfiguration process is controlled by a configuration loader and a configuration monitor. 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 pre-compiled object codes. Two types of packets are communicated from the loader to the workers: work packets and configuration packets. The former consist of the data to be processed and the latter 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 has a small kernel to differentiate between the incoming packets based on their header information. Whenever a worker finishes its task, it returns control to its input kernel after sending a reconfiguration request packet indicating that the particular worker has completed its task and is ready to be reconfigured to a new configuration. The configuration monitor observes the reconfiguration request 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 worker and the second integer identifies the individual processing element within the worker, as shown in Figure 2. 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 neighbor.

5 1D-DCT Case Study

In this section, we present and discuss the 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 lossy compression technique used in video compression encoders to transform an $N \times N$ image block from the spatial domain to the DCT domain [14].

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 3. When computing the forward DCT, an $8 \times 8$ 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 4. 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’ is described by a process named ‘task1’, which consists of the first two stages of the DCT algorithm that 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 pro-
gram is shown in Figure 4a, which has one output channel-end ‘cnf’ of mobile type because it is used to communicate the configuration data (Note that Figure 4 only shows the code related to configuration management, not the complete code). The implementation of the configuration monitor is shown in Figure 4b. The configuration monitor will wait until it receives a ‘RECONFIG’ message from the worker, which indicates that the worker has finished performing its task and 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 reception of a reconfiguration request, will issue a FORK statement as shown in Figure 4a, which includes the name of the process to be configured in place of ‘worker1’, its corresponding configuration data, and its associated channels. The new forked ‘task2’ process has the same placement attributes as those of ‘task1’ as shown in Figure 4c, to determine the mapping locations. The newly configured ‘task2’ process consists of the last two stages of the DCT algorithm.

![Dataflow diagram for 1D-DCT](image)

Fig. 3. Dataflow diagram for 1D-DCT.

5.1 Implementation Results and Discussion

We now present the results of the reconfigurable 1D-DCT which is implemented by using the framework presented in Section 4. Our aim here 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 coarse-grained parallelized DCT is implemented in a four stage pipeline and earlier results reveal that the 4-stage DCT implementation that uses four SRD processors, takes 1340 cycles to compute 64 samples of 1D-DCT. This time includes the time consumed during communication stalls between different stages. The computation of the same amount of samples performed by two SRD processors, which are reconfigured to perform the different stages successively 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 memories of individual processors are 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. The above-mentioned stalls can be eliminated in the reconfigurable two-processor implementation. This 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, which depict the feasibility of the approach.

6 Conclusions and Future Work

We have presented our concept about using the mobility features of the *occam-pi* language and the extensions in language constructs to express run-time reconfig-
urablity in processor arrays. The ideas are demonstrated by a working compiler, which compiles occam-pi programs to native code for an array of processors, Ambric. A reconfigurable component framework is presented, which is adopted to control the reconfiguration of dynamic processes with minimal disruption of the rest of the system. An application study is also performed and the results show two different ways to implement the 1D-DCT algorithm, which are compared on the basis of performance versus resource requirements.

We believe that the compositional nature of process-oriented parallel programming enhances the programmer’s understanding when developing multimedia signal processing systems. The properties of exposing parallelism and separating communication from computation help in the task of parallelization, and the support for expressing reconfigurability enables effective use of resources as demonstrated by the cycle-count results of 1D-DCT algorithm.

In the future we plan to perform more application studies using the compiler platform and demonstrate the usefulness of the approach in implementing runtime reconfiguration of radar signal processing applications.

References