Managing Dynamic Reconfiguration for Fault-tolerance on a Manycore Architecture

Zain-ul-Abdin, Essayas Gebrewahid, and Bertil Svensson
Center for Research on Embedded Systems, Halmstad University,
Halmstad, Sweden
{Zain-ul-Abdin, Essayas.Gebrewahid, Bertil.Svensson}@hh.se

Abstract— With the advent of manycore architectures comprising hundreds of processing elements, fault management has become a major challenge. We present an approach that uses the occam-pi language to manage the fault recovery mechanism on a new manycore architecture, the Platform 2012 (P2012). The approach is made possible by extending our previously developed compiler framework to compile occam-pi implementations to the P2012 architecture. We describe the techniques used to translate the salient features of the occam-pi language to the native programming model of the P2012 architecture. We demonstrate the applicability of the approach by an experimental case study, in which the DCT algorithm is implemented on a set of four processing elements. During runtime, some of the tasks are then relocated from assumed faulty processing elements to the faultless ones by means of dynamic reconfiguration of the hardware. The working of the demonstrator and the simulation results illustrate not only the feasibility of the approach but also how the use of higher-level abstractions simplifies the fault handling.

I. INTRODUCTION

Embedded systems are facing challenges such as the need for increased computational power and the need for concurrently handling many different modes of operation. Examples include simultaneously handling different types of services in radio base stations and rapidly switching between different signal processing algorithms in radar systems. Programmable manycore architectures provide us a possible solution to meet the challenges of increased performance and adaptivity. However, the fault handling techniques for such architectures is a major challenge. The occurrence of faults that result in permanent damage to the processing units can be dealt with by providing the possibility for task relocation. This means that the tasks executed by the processing units encountering permanent faults are relocated to other processing units available on the chip. The relocation of tasks is performed by dynamically reconfiguring the underlying hardware. The management of task relocation is a very cumbersome process and requires low-level knowledge of the hardware. The process of configuration management can be facilitated by developing higher-level abstractions to assist the developer. Therefore it is important to, in the programming language, give the necessary support for implementing dynamically reconfigurable designs.

We propose to use the concurrent programming model of occam-pi [1], combining Communicating Sequential Processes (CSP) [2] with pi-calculus [3] to model the dynamic reconfigurability of the underlying hardware in order to enable fault tolerance. This model allows the programmer to express computations in a productive manner by matching them to the target hardware using high-level constructs. 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 embedded computing architectures.

In earlier work we have demonstrated the feasibility of using the occam-pi language to program an emerging class of massively parallel reconfigurable architectures [4]. We have also previously demonstrated the applicability of the approach on a medium-grained reconfigurable architecture viz., PACT XPP [5]. This paper is focused on using occam-pi to map applications to an embedded manycore architecture, the Platform 2012 (P2012) [6] which is currently under joint development by STMicroelectronics and CEA. In an experimental case study, we demonstrate the use of abstractions provided in the occam-pi language to model the fault tolerance aspects of the application. The abstractions enable the programmer to implement the application tasks in the form of processes with well-defined interfaces. This facilitates the management of dynamic task relocation from the faulty processing elements to the faultless ones. In particular, we have used the extensions in the occam-pi language, such as channel direction specifiers, mobile data and channel types, dynamic process invocation, and process placement attribute, to express dynamic reconfiguration of hardware resources for the P2012 platform. Further evaluation of the approach involving more complex case studies is part of our future work.

The rest of the paper is organized as follows: Section II describes the occam-pi language basics and in particular
extensions for supporting reconfigurability. Section III provides an overview of the P2012 architecture and its native programming model. Section IV describes the compiler framework and the various translation steps involved to generate code for P2012. Section V describes the experimental demonstration, and the paper is concluded with some remarks and future work in Section VI.

II. OCCAM-PI LANGUAGE OVERVIEW

The occam-pi [1] language is known for its simplicity, minimal run-time overhead and power to express parallelism. Occam-pi has built in semantics for concurrency and interprocess communication. Occam-pi can be regarded as an extension of classical occam [7] 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 changes at runtime.

A. Basic Constructs

The hierarchical modules in occam are composed of procedures 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, CASE, and replicated processes [2]. The possibility of creating replicated parallel processes helps in managing the multitude of parallel resources used in the given hardware architecture.

A process in occam contains both the data and the operations the process 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. As an extension to classical occam, occam-pi allows the data to be declared as MOBILE, which means that the ownership of the data can be passed between different processes. Compared to the channel definition in classical occam, 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 a channel. The channel types added to occam-pi are considered as first class citizens in the type system. 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.

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

Mobile Data and Channels: Assignment and communication in classical occam follow 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 has lost the possession of the data. In case the source and the target reside in the same memory space, the movement is realized by swapping of pointers, which is secure and does not introduce aliasing.

In order to incorporate mobile semantics into the language, the keyword MOBILE has been introduced as a qualifier for data types [5]. 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 modeling of mobile channels is independent of the data types and the structures of the messages that they carry.

Mobile Assignment: Having defined the syntax of mobile types, we are now going to 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 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.

MOBILE parameter: Passing parameters in an ordinary PROC call consisting of mobile types does not introduce any new semantics implications and is 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 a requirement of dynamically invoking a new process that can execute concurrently with the dispatching process. In order to implement dynamic process creation in occam-pi, two new keywords, FORK and FORKING, are introduced [6]. 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 can be of VAL type or MOBILE type. The parameters of a forked process follow the movement semantics.

That is:
- VAL data type: Its value is copied to the forked process.
- MOBILE data type and channels of MOBILE data type: These are moved to the forked process.

**Process Placement Attribute:** Having presented the extensions in the occam-pi language, we now introduce one extra extension, the placement attribute. It 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.

### III. P2012 Architecture and Development Tools

P2012 [6] is a manycore architecture which aims to replace existing specialized hardware and software subsystems by a single, modular, scalable, and programmable computing fabric. The architecture is based on multiple clusters with independent power and clock domains. Clusters are connected via a high-performance fully asynchronous network-on-chip (NoC), as shown in Figure 1. The independent power domain for each cluster allows switching-off power to a cluster and the independent clock domain enables frequency/voltage scaling in order to achieve energy-efficient solutions. The P2012 fabric can support up to 32 clusters [6]. The current P2012 cluster is composed of a cluster controller, one to sixteen ENcore processors and Hardware Processing Elements (HWPEs) [6]. The cluster controller is responsible for starting/stopping the execution of ENcore processors and notifying the host system. The processing elements share an advanced DMA engine, a hardware synchronizer, level-1 shared data memories and an individual program cache [6].

The P2012 Software Development Kit (SDK) supports several programming models which can be classified into three main classes. The native programming layer is a low-level C-based API which provides the most efficient use of P2012 resources at the expense of a lack of abstraction. Standards-based programming models target effective implementations of industry standards, such as OpenCL and OpenMP, on the P2012 platform. The SDK provides the GePOP platform for simulation.

#### A. P2012 Native Programming Model

The Native Programming Model (NPM) is a component-based development framework. Application components are developed based on the MIND framework [8]. A component may provide services to other components by its provided interfaces and get service from its environment by using required interfaces. The communication of two components will be hidden by binding their provided and required interface [8]. An NPM application is designed by using the Architecture Description Language (ADL), Interface Description Language (IDL), and an extended C code. ADL is used to define the structure of each component, IDL to specify component interface, and the extended C language will be used for the implementation of the code that runs on the ENcore processors and the cluster controller. After the application is designed, to deploy, manage and run the application a host-side program must also be developed. For this, NPM and Comete, which is a middleware for loading, deploying and managing software components, is used.

NPM is designed to have direct access to specific features of the P2012 hardware platform, while still providing a high level of abstraction. Since the current standards-based programming models of P2012 don’t have explicit ways of dynamic resource allocation, we propose to translate occam-pi to the P2012 native programming model. Figure 2 shows the translation of occam-pi to NPM, i.e. a C file for the host-side program and an ADL
The main implementation of an NPM application will run on ENcore processors, and the cluster controller will execute code for resource allocation and configuration. Interaction of cluster controller and ENCore processors can be handled by two execution engines: Reactive Task Manager (RTM) and/or Multi-Threaded Engine (MTE). RTM expresses parallelism based on forking and duplication of tasks, and MTE allows execution of synchronized parallel threads. Currently, our compilation directly uses the APIs provided by the base runtime and hardware abstraction layer (HAL), instead of using the two execution engines.

IV. OCCAM-PI COMPILATION TO P2012

The compiler that we have developed is based on the frontend of an existing Translator from occam to C from Kent (Tock) [9]. Our compiler can be divided into three main phases as shown in Figure 3. The front end consists of phases up to machine independent optimization and the backend includes the remaining phases that are dependent upon the target machine architecture. The Ambric and the eXtreme Processing Platform (XPP) backends were developed in previous works [10][5].

In this work, we have developed a new backend for P2012. Our P2012 backend targets the whole platform and its integration with the host system.

A. Frontend

The frontend of the Tock compiler consists of several modules which perform operations like lexical analysis, parsing and semantic analysis. In earlier works, the frontend of the compiler has been extended to support mobile data and channel types, dynamic process invocation, and process placement attributes [10][5].

The transformation stage consists of a number of passes either to reduce complexity in the Abstract Syntax Tree (AST) for subsequent phases or to convert the input program to a form which is suitable for the backend or to implement different optimizations required by some specific backend.

B. P2012 Backend:

The P2012 backend generates the complete structure of application components in NPM as well as the host-side program to deploy, control and run the application components on the P2012 fabric. The generated code can then be executed on the GePOP simulation environment. The P2012 backend is divided into two main passes. The first pass traverses the AST to create a list of parameters passed in procedure calls specified for processes to be executed in parallel. Since a procedure can be called more than once, besides name of the procedure, a counter is also added on the parameter list to indicate parameters of this particular procedure call. This list of parameters of procedure calls is used to generate the required and provided interface of each component along with its specific binding codes, i.e. the architectural description of the application using ADL and IDL. Figure 4b shows the ADL file generated for a component called 'prod', which corresponds to a process call in occam-pi (Figure 4a). PullBuffer and PushBuffer are services provided by the NPM communication components. The two source files, 'prod_cc.c, and 'so_prod.c', will be generated in the next pass.
The NPM instantiateFIFOBuffer API is used to bind the push buffer with the corresponding pull buffer. Figure 5 shows an overview of the host-side program with bare deployment of a procedure ‘prod’ (without using the two execution engines (RTM & MTE)).

The list of parameters of procedure calls is also used to generate deployment, instantiation and control code of an application component from the host-side. For each procedure call, binary code of the procedure is deployed on the intended cluster using the NPM_instantiateAppComponent API, then the cluster controller will execute this binary code on one of the ENCore processors.

The second pass generates implementation code of the application components and the cluster controller. The genProcess function traverses the AST to generate the corresponding extended C code for different occam-pi primitive processes such as assignment, input process (?), output process (!), WHILE, IF/ELSE, and replicated SEQ. Since we are not using the execution engines, the cluster controller code uses runtime APIs to execute, control and configure the application component. Cluster controller code is differentiated from the component code by inserting the ‘@CC’ annotation; in Figure 4 the ‘prod_cc.c’ will be executed on the cluster controller and ‘so_prod.c’ will run on the ENCore processors. Figure 6 shows the translation of output and input primitives of occam-pi to the corresponding NPM code using CALL method with the push and pull parameters respectively.

NPM components are deployed and controlled dynamically by a program running on the host system. In occam-pi the dynamic process invocation can be used for managing the dynamic deployment of software components by the use of FORKING and FORK constructs. Thus, the backend will generate host-side code for a process with FORKING block. An IDL will also be generated for the interaction of an application component with the host system and the ‘@controller.StdControllers’ annotation is generated at the start of the ADL file. The Comete middleware will deploy and manage the component dynamically. If the forked process makes use of MOBILE data as shown in Figure 7a, a communication instance has to be established between host-side program and components. The Comete API (CM_bindFromUser) will be used to bind the host-side program with the application components. The component will provide an interface in the IDL file (Figure 7b) and implement the interface in the cluster controller (Figure 7d). Then the host-side program will include a header file that is generated by the MIND compiler from the IDL, to pass the MOBILE data by invoking the component method as shown in Figure 7c.
V. EXPERIMENTAL CASE STUDY

We have implemented an application case study making use of the constructs available in the occam-pi language to manage the fault recovery on the P2012 platform. The application illustrates the use of occam-pi to manage the relocation of tasks from faulty to faultless processing elements within a P2012 cluster. The application is the One-Dimensional Discrete Cosine Transform (1D-DCT) algorithm which is implemented in four stages in a pipelined manner executing on four ENCore processing elements of Cluster 0 of the P2012 fabric as shown in Figure 8.

In the first configuration the 1D-DCT algorithm is mapped to four processing elements denoted 1 through 4. During the execution of the DCT kernel, the run-time system detects a fault in one of the processing elements and passes the error code to the application running on the processing elements. The run-time system makes use of a lightweight detection technique based on thread duplication that is similar to the fault-tolerant simultaneous multi-threading (FT-SMT) [11] technique. In this case the redundant threads will be executed on the vacant cores available in the same cluster or in the neighboring cluster. By comparing the results of the replicated application threads, the technique allows detection of errors that may occur in processors. The thread duplication technique that we have relied on, does not result in loss of performance because the computing processing elements will be providing the same throughput as without the use of this technique. The downside is that the duplication of computations will result in consumption of extra energy.

The run-time system communicates the error-message together with its corresponding processing element ID to the host processor by making use of the generated NPM communication components. The host processor, after receiving an error-message, will first request the cluster controller to terminate the current execution on the concerned processing elements. Afterwards the host will issue a new configuration deployment by making use of the
FORK construct of occam-pi, in which the error code is passed as a parameter to the forked process. The host-side program will consist of separate instances of the NPM_instantiateAppComponent API corresponding to each FORK construct used in the occam-pi source, and the communication network for the new configuration is described by the use of the NPM_instantiateFIFOBuffer API. This new forked process will result in the invocation of the next configuration in which the identification information about the faulty processing element is passed in the instantiation of the NPM_instantiateAppComponent API to avoid its use in the deployment of next configuration by the runtime system. On completion of the deployment, the newly configured processing elements will begin executing the DCT kernel to compute the DCT transform of input data values provided by the host processor.

In this case study, we have not used the MOBILE data. The MOBILE data is useful when considering partial reconfiguration of a set of processing elements, where the intermediate results are declared as MOBILE data so that their ownership can be passed to the new allocated processing elements.

Let us now look at the simulated results of the 1D-DCT application to investigate the extra overhead incurred for communicating the error code to the host-CPU and to deploy a new configuration in the presence of faults. Table 1 shows the accumulated time for computing 1000 samples of 1D-DCT without fault-recovery and the 1D-DCT implementation with fault-recovery mechanism. The accumulated time includes the configuration deployment, fault detection, and computation time. The fault tolerance model performs reconfiguration only in the occurrence of a failure. In the case where there is no error (second row of Table 1), there will be 23% overhead incurred due to the fault detection mechanism. When a fault is detected in one of the processing elements (third row of Table 1) and a reconfiguration is performed then there is about 33% overhead incurred in communicating the error code and deploying the new configuration. The reconfiguration adds only 7% to process and deploy the new configuration. Our analysis shows that the process of detecting a faulty processor and communicating the error code adds more overhead than deploying the new configuration.

TABLE 1. SIMULATION RESULTS FOR 1D-DCT CASE-STUDY.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Fault Recovery</th>
<th>Faulty Processor</th>
<th>Accumulated Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D-DCT</td>
<td>No</td>
<td>Nil</td>
<td>48.65</td>
</tr>
<tr>
<td>Fault Tolerant 1D-DCT</td>
<td>Yes</td>
<td>Nil</td>
<td>60.07</td>
</tr>
<tr>
<td>Fault Tolerant 1D-DCT</td>
<td>Yes</td>
<td>One</td>
<td>64.61</td>
</tr>
</tbody>
</table>

VI. CONCLUSIONS AND FUTURE WORK

We have presented our approach of using a CSP based language for programming manycore architectures. We have described how the occam-pi compiler framework has been extended to generate native programming model code for the new Platform 2012 architecture. We have shown the simplicity and expressiveness of the fault recovery mechanisms. The abstractions in occam-pi to describe computations that reside in different memory spaces, together with the facility of expressing dynamic process invocation mechanisms, and the language support for placement attributes enables the expression of dynamic reconfigurability. These high-level language abstractions hide the superfluous details and expose the structure of a system in terms of components, which allows the developers to reason about the run-time changes independent of the application logic. From the simulation results we can conclude that the total overhead incurred in enabling the fault-tolerance mechanism is reasonable to justify the usefulness of the approach.

Future work will focus on supporting the different NPM execution engines such as RTM and MTE. We will also make further evaluation of the approach using more complex examples.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 and from the national programmes / funding authorities.

REFERENCES

