Mapping Occam-pi programs to a Manycore Architecture

Essayas Gebrewahid, Zain-ul-Abdin, and Bertil Svensson
Center for Research on Embedded Systems, Halmstad University,
Halmstad, Sweden

{Essayas.Gebrewahid, Zain-ul-Abdin, Bertil.Svensson}@hh.se

ABSTRACT

Efficient utilization of available resources is a key concept in embedded systems. This paper is focused on providing the support for managing dynamic reconfiguration of computing resources in the programming model. We present an approach to map `occam-pi` programs to a manycore architecture, Platform 2012 (P2012). We describe the techniques used to translate the salient features of the `occam-pi` language to the native programing model of the P2012 architecture. We present the initial results from a case study of matrix multiplication. Our results show the simplicity of `occam-pi` program by 6 times reduction in lines-of-code.

1. INTRODUCTION

Embedded systems are facing the challenges of increased computational power together with the need to handle many different modes of operation such as switching between different signal processing algorithms in radar systems and handling different types of services in radio base stations. Dynamic allocation of computing resources is important in embedded systems not only for efficient utilization of resources, but also for having a power efficient solution. Therefore it is important to give the support for implementing dynamically reconfigurable designs in the programming language.

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. 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 more fine-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. The paper describes the different translation steps involved in the code generation phase of the compiler. The resulting code has been tested at the functional level on the transaction level simulator of the P2012 and further evaluation of the approach involving case studies is part of our future work.

P2012 [6] is a scalable manycore computing fabric based on multiple clusters with independent power and clock domains. Clusters are connected via a high-performance fully asynchronous network-on-chip (NoC). 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.

2. 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.

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 (`?`), 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 feature of creating replicated parallel processes helps in managing the amount of parallel resources used in the given hardware architecture.

A process in `occam` contains both the data and the operations it is required to perform on the data. The data in a process is strictly private and can be observed and modified by the owner process only. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems. In contrast, in `occam-pi` the data can be declared as `MOBILE`, which means that the ownership of the data can be passed between different processes. 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.

2.2 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 in the programming model to express the different configurations of hardware resources, whose reconfiguration at run-time can be controlled by using dynamic process invocation and process placement attributes.

Mobile Data and Channels: 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 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 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 communication 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

3. P2012 Architecture and Development Tools

P2012 is a manycore architecture which aims to replace existing specialized hardware and software subsystems by a single, modular, scalable, and programmable computing fabric. The P2012 fabric can support up to 32 P2012 clusters. The current P2012 cluster is composed of a cluster controller, 1-16 ENcore processors and Hardware Processing Elements (HWPEs). 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 a wide range of platform programing models which can be classified into three main classes. **Native programming layer** is a low-level C-based API which provides the highest productivity 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. P2012 SDK provides GePOP platform for simulation. The **Native programming model (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 programing model. Figure 1 shows the translation of `occam-pi` to NPM, i.e. a C file for the host-side program and ADL, C and optionally an IDL file for the P2012 fabric.

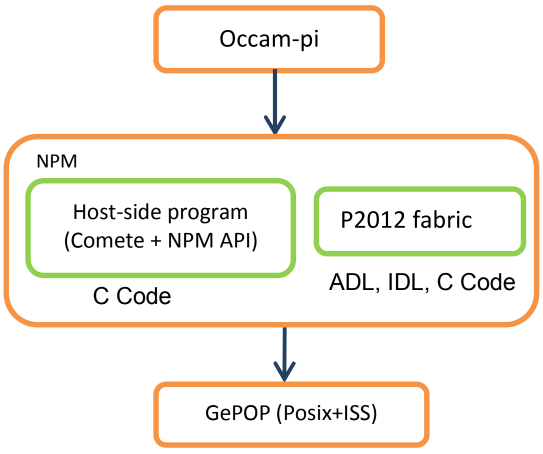


Figure 1. Compilation of Occam-pi to P2012 Platform.

3.1 P2012 Native Programming Model

The Native Programing 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]. NPM application is designed by using Architecture Description Language (ADL), Interface Description Language (IDL), and an extended C code. ADL is used to define the structure of each component, IDL specify component interface and extended C language will be used for the implementation of the code that run on ENcore processors and cluster controller. After the application is designed, to deploy, manage and run the application a host-side program must also be developed. NPM and Comete define host-side APIs for deploying and controlling an application running on the P2012 fabric. Comete is a middleware for loading, deploying and managing software components.

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.

4. 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]. Tock is developed by Haskell language based on heavy use of monads and generics. Our compiler can be divided into three main phases as shown in Figure 2. 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) backendes were developed in previous works [4][5].

In this project we have developed a new backend for P2012. To achieve an efficient implementation of *occam-pi* properties, we propose to generate NPM. Our P2012 backend targets the whole platform and its integration with the host system.

The frontend of the Tock compiler consists of several modules which perform operations like lexical analysis, parsing and semantic analysis. The lexical analyzer is generated using Alex [10], a tool to generate lexical analysers in Haskell, and the parser stage uses a combinator-based parsing library called Parsec [11] that greatly simplifies writing monadic programs. 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 [4][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.

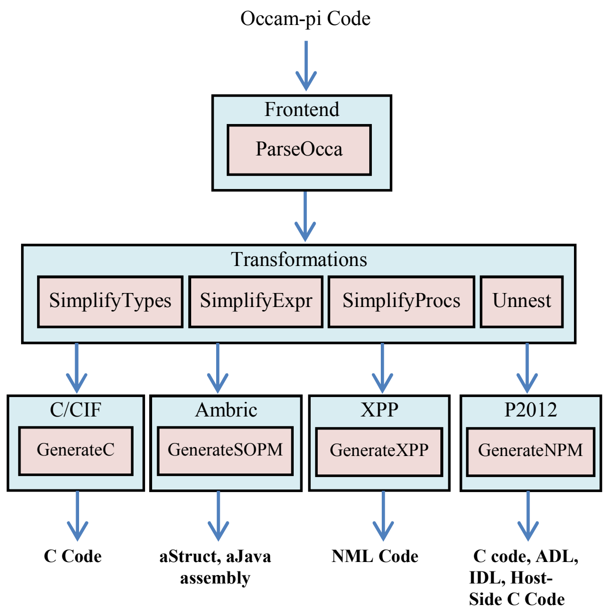


Figure 2. Tock compiler block diagram

The P2012 backend generates the complete structure of application components in NPM as well as 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 **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 3b shows the ADL file generated for a component called 'prod', which corresponds to a process call in `occam-pi` (Figure 3a). PullBuffer and PushBuffer are services provided by the NPM communication component. The two source files, 'prod_cc.c, and 'so_prod.c', will be generated in the next pass.

```
PROC SimpleEx()
  CHAN INT e:
  CHAN INT f:
  PAR
    prod(f?,e!)
    con(e?,f!)
:
```

(a)

```
primitive SimpleEx.prod {
   requires PullBuffer as f;
   requires PushBuffer as e;
   @CC
   source prod_cc.c;
   source so_prod.c;
}
```

(b)

Figure 3. Translation of Occam-pi process (a) to ADL file (b).

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 NPM_instantiateAppComponent API, then the cluster controller will execute this binary code on one of the ENCore processors. NPM_instantiateFIFOBuffer API is used to bind the push buffer (*`output process (!)`*) with the corresponding pull buffer (*`input process (?)`*). Figure 4 shows an overview of the host-side program with bare deployment of a procedure 'prod' (without using the two execution engines (RTM & MTE)).

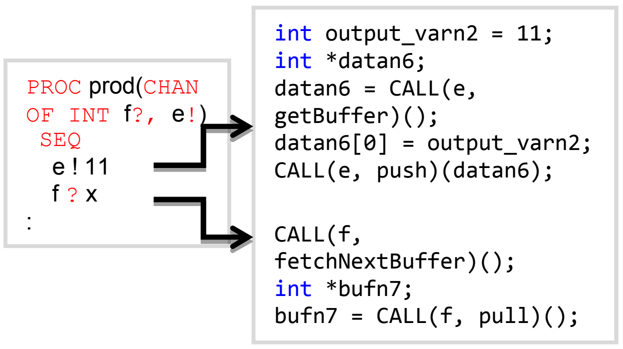
```
. .  . .  . .
static int deploy prodBare(int id,
prod_processor_bare_t *processor_Inst)
{
 . .  . .  . .
err =
   NPM_instantiateAppComponent(
        &processor_Inst-> appComp,
        getFullName(fabricBinaryPath,
             "SimpleEx/prod.so", compName),
        CM_P2012_CLUSTER0, prodRunEnded,
        NULL,prodEngineStopped, NULL);

CM_CHECK(err);
err = CM_start(processor_Inst->appComp.comp);
CM_CHECK(err);
. .  . .  . .
}
int main(int argc, char **argv)
{
err = NPM_init();
. .  . .  . .
  prod_processor_bare_t prod_inst_0_100;

   err = deployprodBare(0,&prod_inst_0_100);
   CM_CHECK(err);
   . .  . .  . .
  NPM_fifoBuffer_t channel_n44;
  err = NPM_instantiateFIFOBuffer(&channel_n44,
          prod_inst_0_100.appComp.comp,
          "e", con_inst_1_100.appComp.comp,
          "e", sizeof(int), 2);
  CM_CHECK(err);
 . .  . .  . .
NPM_run(&prod_inst_0_100.appComp.runItf);
 . .  . .  . .//wait till the processors stop
CM_stop(prod_inst_0_100.appComp.comp);
 . .  . .  . .
NPM_deinit();
}
```

Figure 4. Overview of the host-side program with common NPM and Comet API.

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 use runtime APIs to execute, control and configure the application component. Cluster controller code is differentiated from component code by inserting the '*@CC*' annotation; in Figure 3 the 'prod_cc.c, will be executed on the cluster controller and 'so_prod.c' will run on the ENCore processors. Figure 5 shows the translation of input and output primitives of `occam-pi` to the corresponding NPM code.



```
PROC prod(CHAN
OF INT f?, e!)
 SEQ
  e ! 11
  f ? x
:
```

```
int output_varn2 = 11;
int *datan6;
datan6 = CALL(e,
getBuffer)();
datan6[0] = output_varn2;
CALL(e, push)(datan6);

CALL(f,
fetchNextBuffer)();
int *bufn7;
bufn7 = CALL(f, pull)();
```

Figure 5. Translation of Occam-pi primitives

NPM components are deployed and controlled dynamically by a program running on the host system. 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. The Comete middleware will deploy and manage the component dynamically by just adding the '*@controller.StdControllers*' annotation at the start of the ADL file. If the forked process makes use of `MOBILE` data as shown in Figure 6a, a communication 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 6b) and implement the interface in the cluster controller (Figure 6c). 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 6d.

```
FORKING
  MOBILE INT x:
  SEQ
    x := 42
    FORK ProcName(x)
:
```
(a)

```
interface example.ProcNameConfig {
config(int mobDataSize,  intptr_t x);
}
```
(b)

```
data {{
struct {
int mobDataSize;
void *x;
} DATA;
}}
@CC
 source {{
   #include "p2012/debug.h"
   void METH(config, config)
     (int mobDataSize, intptr_t x)
   {
   DATA.mobDataSize = mobDataSize;
   DATA.outputImage = (void *) x;
   }
}};
```
(c)

```
 //generated by Mind compiler
#include "example/ProcNameConfig.h"
 ...
 void *clientItf;
 // bind the host-side program to a
component
 CM_bindFromUser(&clientItf, ...);
 example_ProcNameConfig configItf =
 (example_ProcNameConfig) clientItf;
 // Call the config interface of the
component
 CALL(configItf, config)(dataSize, x);
```
(d)

Figure 6(a). Forking process, (b). IDL for `ProcName`, (c). Section of ADL file for `ProcName`, (d). Host side program for the forked processor (`ProcName`)

5. Experiment description

To substantiate the simplicity of `occam-pi` languages, we compare matrix multiplication implementation written in `occam-pi` with hand written NPM version, Table 1. To quantify implementation complexity we use lines-of-code. The lines-of-code metric clearly shows the simplicity of our occam-pi based programming approach. The `occam-pi` version uses two splitters, four multipliers and one process to combine the result. The hand written NPM version uses the host-CPU to split and combine the result, and four multipliers, as show in Figure 7. In Table 1, we also show the Accumulated Time for both versions. The Accumulated Time includes the configuration, deployment and computation time. Total simulation time of `occam-pi` version is greater than the NPM version because the `occam-pi` version uses three more ENcore processors.

Table 1. Comparison of 4x4 Matrix Multiplication implementation

	`Occam-pi`	Hand Written NPM
Lines-of-code	95	572
No. of ENcore Processors	7	4
Accumulated Time (µs)	67 471	43 432

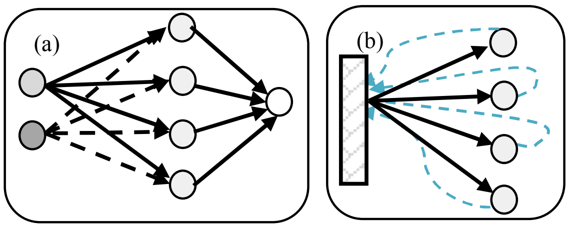


Figure 7 Matrix multiplication (a) Occam-pi version (b) Hand written version

6. Summary and Future Work

We have presented our approach to map a CSP based language to manycore architecture. We have extended `occam-pi` compiler framework to generate native programing model of Platform 2012. We have shown the simplicity and expressive power of `occam-pi` by using matrix multiplication. Future work will focus on supporting the different NPM execution engines such as RTM. We will also make further evaluation of the approach using 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

1. Welch, P.H., and Barnes, F.R.M. "Communicating mobile processes" Introducing occam-pi. Lecture Notes in Computer Science, Springer Verlag. 175-210 (2005).
2. Hoare, C.A.R. "Communicating Sequential Processes" Prentice-Hall. (1985).
3. Milner, R., Parrow, J., and Walker, D. "A Calculus of Mobile Processes" Part I. Information and Computation, 100, (1989).
4. Zain-ul-Abdin, and Svensson, B. "Using a CSP based programming model for reconfigurable processor arrays" Proceedings of International Conference on Reconfigurable Computing and FPGAs. (2008)
5. Zain-ul-Abdin, and B. Svensson, "Occam-pi as a high-level language for coarse-grained reconfigurable architectures", *RAW'11*, May 2011.
6. STMicroelectronics and CEA, "Platform 2012: A manycore programmable accelerator for ultra-efficient embedded computing in nanometer technology", November 2010.
7. Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited. (1995)
8. The MIND Project, http://m ind.ow2.org.
9. Tock: Translator from Occam to C by Kent. http://projects.cs.kent.ac.uk/projects/tock/trac/
10. "Alex: Lexer generator for Haskell", 8th July, 2008. http://pages.cpsc.ucalgary.ca/~gilesb/alex/alex.html
11. D. Leijen, and E. Meijer, "Parsec: A practical parser library", Electronic Notes in Theoretical Computer Science, Vol. 41(1), 2001.
12. Zain-ul-Abdin, A. Ahlander, and B. Svensson, "Programming Real-time Autofocus on a Massively Parallel Reconfigurable Architecture using Occam-pi", *FCCM'11*, May 2011.