Programming Real-time Image Processing for Manycores in a High-level Language

Essayas Gebrewahid[1], Zain-ul-Abdin[1], Bertil Svensson[1], Veronica Gaspes[1], Bruno Jego[2], Bruno Lavigueur[2], and Mathieu Robart[3]

1. Center for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
2. STMicroelectronics – Advanced System technology, Grenoble, France
3. STMicroelectronics – Advanced System technology, Bristol, United Kingdom

Abstract. Manycore architectures are gaining attention as a means to meet the performance and power demands of high-performance embedded systems. However, their widespread adoption is sometimes constrained by the need for mastering proprietary programming languages that are low-level and hinder portability.

We propose the use of the concurrent programming language `occam-pi` as a high-level language for programming an emerging class of manycore architectures. We show how to map `occam-pi` programs to the manycore architecture Platform 2012 (P2012). We describe the techniques used to translate the salient features of the language to the native programming model of the P2012. We present the results from a case study on a representative algorithm in the domain of real-time image processing: a complex algorithm for corner detection called Features from Accelerated Segment Test (FAST). Our results show that the `occam-pi` program is much shorter, is easier to adapt and has a competitive performance when compared to versions programmed in the native programming model of P2012 and in OpenCL.

Keywords: Parallel programming; `Occam-pi`; Manycore architectures; Real-time image processing.

1 Introduction

The design of high-performance embedded systems for signal processing applications is facing the challenge of increased computational demands. Moore's Law still gives us more transistors per chip but, since increased processor clock speed is no longer an option, current hardware designs are shifting to manycore architectures to cope with the computational demand of DSP applications. However, developing applications that employ such architectures poses several other challenging tasks. The challenges include learning multiple proprietary low-level languages for describing the communication structure of the application and the computational kernels, as well as partitioning and decomposing the application into several sub-tasks that can execute concurrently. Sequential programming languages (like C, C++, Java …), which were originally designed for sequential computers with unified memory systems and rely

on sequential control flow, procedures, and recursion, are difficult to adapt for many-core architectures with distributed memories. Usually, as a partial solution, these languages provide annotations that the programmer can use to direct the compiler how to adapt the implementation to the target architecture.

We propose to use the concurrent programming model of `occam-pi` [1] that combines Communicating Sequential Processes (CSP) [2] with the pi-calculus [3]. This model allows the programmer to express concurrent computations in a productive manner, matching them to the target hardware using high-level constructs. The features of `occam-pi` that make it suitable for mapping applications to a wide class of embedded parallel architectures are: a) constructs for expressing concurrent computations, b) computations that reside in different memory spaces, c) dynamic parallelism, d) dynamic process invocation, and e) support for placement attributes.

The feasibility of using the `occam-pi` language to program an emerging class of massively parallel reconfigurable architectures has been demonstrated in earlier work [4]. The applicability of the approach was also previously demonstrated 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. P2012 is a scalable manycore computing fabric based on multiple processor 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.

The paper describes the different translation steps involved in the code generation phase of the compiler. The paper also presents as a case study the implementation of the FAST (Features from Accelerated Segment Test) algorithm [7] for corner detection. The case study aims at verifying that programming is actually simplified, and at evaluating the competitiveness in performance of our compilation based approach compared to the use of the native programming model of the P2012 architecture. We have used a parameterized approach in the form of replicated parallel processes in the `occam-pi` language to control the degree of parallelism.

In previous papers we have demonstrated the suitability of `occam-pi` for expressing task parallelism in applications like FIR (finite impulse response) filter, DCT (discrete cosine transform) and Autofocus in image forming radar systems [4, 5, 18]; here we show the applicability of the approach also for truly data parallel computations.

In the following three sections we present some related work, review the `occam-pi` language basics, and give an overview of the P2012 architecture and its native programming model. We then describe the compiler framework and the various translation steps involved to generate code for P2012. The approach is experimentally evaluated through a case study implementation of the FAST algorithm, and conclusions are drawn.

2 Related Work

There have been a number of initiatives in both industry and academia to address the requirement of raising the abstraction level in the form of high-level parallel programming languages. Recently developed parallel programming languages include `Chapel` [8], `Fortress` [9], and `X10` [10]. These mainly rely on implicit parallelism based on data-parallel operations on parallel collections and are primarily targeting high-performance large-scale computers.

Apart from the above-mentioned parallel programming languages, there are some recently introduced domain specific languages (DSLs) intended for the domain of digital signal processing (DSP). The `Feldspar` language [11], being developed at Chalmers University of Technology, is one such DSL where the domain expert expresses the DSP algorithms by using constructs like filters, vectors, and bit manipulation operations. The functional basis of the `Feldspar` core language facilitates performing different source code transformations such as fusion techniques and graph transformations. `CAL` [12] is another domain-specific language, developed at UC Berkeley, for dataflow programming and is based on the actor's model of computation. By describing the application as a dataflow network of actors, the available parallelism is explicitly exposed. `CAL` has been chosen as a specification language for the ISO/IEC 23001-4 MPEG standard. The `Spiral` project at CMU [13] deals with the domain of linear signal transforms in the broad field of DSP algorithms. `Spiral` makes use of the mathematical knowledge expressed in a particular algorithm in order to transform it into a concise declarative framework that is suitable for computer representation, exploration, and optimization. These high-level domain-specific languages are best suited for application programming because of their productivity and expressiveness. However, they are not well suited for compiling directly down to the manycore architectures; rather, they require transformations via a parallel intermediate representation.

Since we are interested in both the signal processing domain and mapping to the manycore architectures, we have proposed the use of `occam-pi` because it provides explicit control of concurrency in terms of processes that communicate by message passing (however, this is not demonstrated in the present paper) [22]. This closely matches the underlying architecture and it supports both task and data-level parallelism, thereby allowing the programmer to exploit the available parallelism more effectively. Based on this property, `occam-pi` is a candidate for the parallel intermediate representation mentioned above.

3 `Occam-pi` Language Overview

`Occam-pi` [1] is a programming language based on the concurrency model of CSP [2] and the pi-calculus [3]. It offers a minimal run-time overhead and comes with constructs for expressing parallelism and reconfigurations. It has a built in semantics for concurrency and inter-process communication. `Occam-pi` can be regarded as an extension of classical `occam` [14] to include the mobility feature 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.

The primitive processes of occam include assignment, input (?) and output (!). In addition to these there are constructs for sequential processes (SEQ), parallel processes (PAR), iteration (WHILE) selection (IF/ELSE, CASE) and replication [2]. In occam-pi the SEQ and PAR constructs can be replicated. A replicated SEQ is similar to a for-loop. A replicated PAR can be used to instantiate a number of processes in parallel and helps managing the multitude of parallel resources in a given hardware architecture.

PAR *i* = *start* FOR *Number of Replications*
<process i>

Finally, a procedure is a named process that can take parameters. In occam the data a process can access is strictly local and can be observed and modified by the owner process only. The communication between processes uses channels and message passing, which helps to avoid interference problems. In occam-pi data can be declared to be MOBILE, which means that the ownership of the data, including communication channels, can be passed between different processes. Moreover, channel type definitions have been extended to include the direction specifiers input (?) and output (!). Thus, a variable of a channel type refers only to one end of the channel. Channels in occam-pi are first-class citizens. Channel direction specifiers are added to the type of a channel definition and not to its name. Based on the direction specification, the compiler can do static checks of the usage of the channel both in the body of the process and the processes that communicate with it. Channel direction specifiers are also used when referring to channel variables as parameters of a process call.

Mobile data and channels, together with dynamic process invocation and the process placement attributes of occam-pi, are used to express the different configurations of hardware resources as well as run-time reconfiguration.

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.

4 P2012 Architecture and Development Tools

P2012 [6] is a manycore architecture, which is aimed at replacing existing specialized hardware and software subsystems by using 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). 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 execution. 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 that can be classified into three main classes. The native programming layer is a low-level C-based API providing 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.

4.1 P2012 Native Programming Model

The Native Programming Model (NPM) is a component-based development framework. Application components are developed based on the MIND framework [16]. A component may provide services to other components by its provided interfaces and get service from its environment by using required interfaces. The communication between two components is hidden by binding their provided and required interface [16]. 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 for the implementation of the code that runs on the ENcore processors and the cluster controller. After the application is designed, a host-side program also has to be developed to deploy, manage and run the application. For this purpose, the middleware Comete 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 means for dynamic resource allocation, we propose to translate `occam-pi` to the P2012 native programming model. Fig. 1 shows our approach to map `occam-pi` programs to the Platform 2012 Software Development Kit stack.

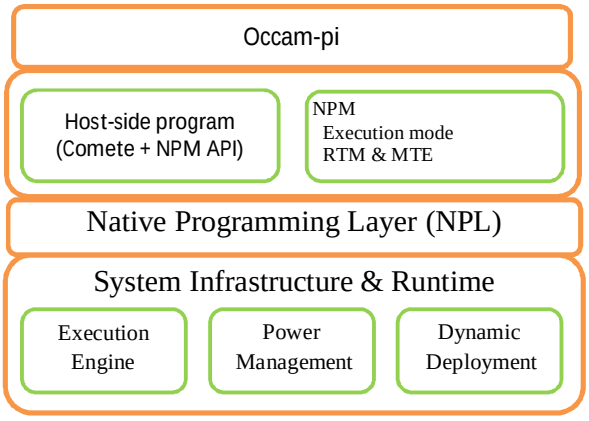

Fig 1. Mapping of `occam-pi` to P2012 SDK

The main implementation of an NPM application will run on the ENcore processors, and the cluster controller will execute code for resource allocation and configuration. Interaction between the cluster controller and the 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 any of the two execution engines.

5 `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) [17]. Our compiler can be divided into three main phases as shown in Fig. 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) backends were developed and describer earlier [18] [5]. We have also earlier described the P2012 backend, focusing on fault recovery mechanisms using dynamic reconfiguration [22].

In the current paper we have extended the P2012 backend to support data intensive computations. Our P2012 backend targets the whole platform including its integration with the host system.

Frontend: The frontend of the Tock compiler consists of several modules, which perform operations like lexical analysis, parsing and semantic analysis. The frontend of the compiler has been extended to support mobile data and channel types, dynamic process invocation, and process placement attributes [18][5]. We have also introduced new grammar rules corresponding to the additional constructs to create Abstract Syntax Trees (AST) from tokens generated at the lexical analysis stage. In the current

work, we have revised the frontend in order to provide support for channels that communicate an entire array of data in a single transfer.

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.

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. In addition to parameters the list also includes two integer values which store the first value and the count of replicated PAR.

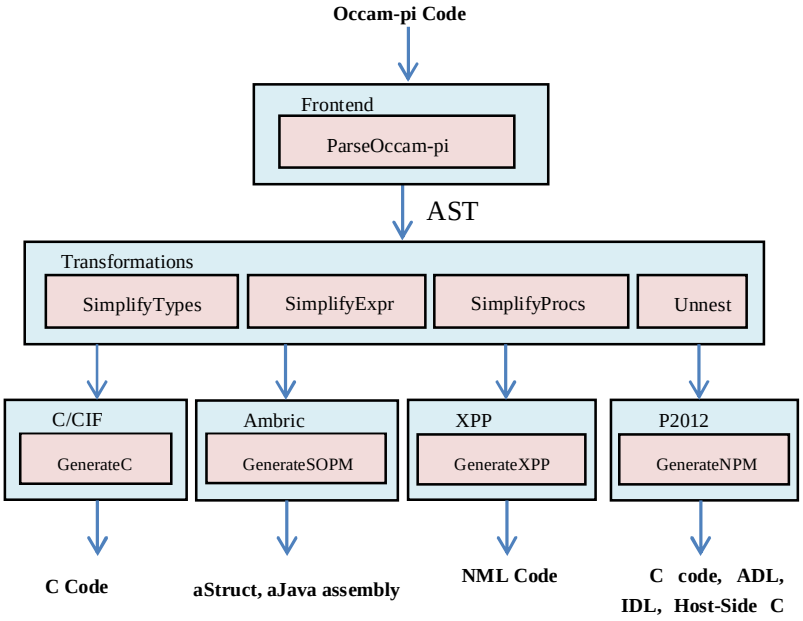

Fig. 2. `Occam-pi` compiler block diagram

Since a procedure can be called more than once in different places, besides name of the procedure, a counter and the name of the procedure that calls the procedure (parent procedure) is also added on the parameter list to indicate parameters of this particular procedure call. To facilitate the code generation, if the list is composed of several parent procedures and simple procedures, it will be transformed to a list of simple procedures and one parent procedure. 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. Listing 1b shows the ADL file generated for a component called 'prod', which corresponds to a process call in occam-pi (Listing 1a). 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.

```
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)

Listing 1. 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 the NPM_instantiateAppComponent API, then the cluster controller will execute this binary code on one of the ENCore processors. The NPM_instantiateFIFOBuffer API is used to bind the push buffer with the corresponding pull buffer. For replicated PAR an array of processes is created and a for-loop is used to deploy, run and stop the processes. The for-loop gets the start and count of the replicator from the information stored in the list of the procedure calls. Listing 2 shows the translation of replicated PAR of occam-pi to the corresponding host code sequences that instantiate, deploy, run and stop each process.

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 Listing 1b prod_cc.c will be executed on the cluster controller and so_prod.c will run on the ENCore processors.

```
PAR pr=0 FOR 16
  fastProc (idT[pr],inIm[pr], offsetX[pr],
            offsetY[pr], inF[pr]?, outF[pr]!)
```

(a)

```
fastProc_processor_bare_t   fastProc_inst_100[16];

for(pr=0;pr<(16+0);pr++)
   err = deployfastProcBare(pr, &fastProc_inst_100[pr]);

for(pr=0;pr<(16+0);pr++)
  NPM_run(&fastProc_inst_100[pr].appComp.runItf);

for(pr=0;pr<(16+0);pr++)
  CM_stop(fastProc_inst_100[pr].appComp.comp);
```

(b)

Listing 2. Translation of Replicated PAR (a) to corresponding C code (b)

6 Experimental Case Study

In this section, we will describe the implementation of a FAST Corner Detection algorithm, which is used to evaluate our compilation methodology. We have compared our implementation in `occam-pi` with a hand written NPM version and with an OpenCL implementation.

6.1 Features from Accelerated Segment Test (FAST) Corner Detection

FAST is an algorithm that is used to spot corners in an image [7]. In image processing, corners are detected and used to derive a lot of information that is important for computer vision systems. The FAST corner detection algorithm is a high performance detector, suitable for real-time visual tracking applications that run on limited computational resources. According to Rosten et al [19], FAST performs better than conventional algorithms in terms of execution time and repeatability (i.e., detecting the same corner in several similar images).

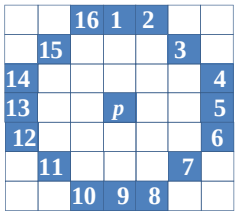

Fig. 3. Bresenham circle of radius three surrounding the pixel of interest

The FAST algorithm examines a pixel by comparing the intensity value of the pixel with the values of sixteen pixels that surround the pixel in a Bresenham circle of radius three, as shown in Fig. 3 [19]. Among the sixteen pixels, if the intensity of N pixels are either greater than or less than the intensity of the pixel by a threshold T, then that pixel is categorized as a corner. In our implementation, the values of N and T are set to 14 and 35, respectively. This step usually detects multiple neighboring pixels as a corner. To solve this problem, the score of each corner pixel is computed and corner candidates with lower score are discarded by using non-maximal suppression [19].

To speed up the computation, we have implemented a parallel version of the algorithm using `occam-pi` primitives. To control the degree of parallelism, we have used replicated `PAR` statements of `occam-pi`. As shown in Listing 3, the amount of parallelism can be varied by changing just one parameter (noP). In the implementation the host-CPU loads and splits the image vertically for the given number of processes (noP). Listing 3 shows sample `occam-pi` code that starts with converting the RGB image to a grayscale intensity image, and then splits the intensity image according to the given number of processes (noP), which are instantiated by using replicated `PAR`. In our implementation, we create a circle of 16 pixels that surround a pixel p under test and then in order to identify the pixel as a corner the intensity value of 14 neighboring pixels has to be above or below intensity of p by the threshold value of 35.

As mentioned above, to examine a pixel we have to create a Bresenham circle of radius three, which requires a 7x7 block. So, to examine boundary pixels, a process has to share three columns of pixels from both left and right processes. Since an `occam-pi` process cannot share data with any other process, the host-CPU duplicates three columns of pixels on the new borders that are created when the image is split. Therefore, each process examines the pixels of its own portion of the image and computes the scores for detected corners without sharing any data with other processes. If a pixel is not detected as a corner, its score is –1. From its portion a process reads seven lines and examines the pixels in the middle row one by one. When it's done with the middle row, the process pushes the computed score as an output, releases the first input line, moves the remaining six image lines upward, fetches the next input line, and starts to examine the pixels in the new middle row until it has fetched the last input line. In our implementation each process (fastProc from Listing 3) is executed by one ENcore processor and we use the host CPU to select the good corners.

Just like our implementation, the FAST implementation that comes with the P2012 SDK uses ENcore processors to detect corners and to compute scores, and the host CPU for non-maximal suppression. This implementation is not modified. The implementation reads the entire line of the input image and spawns sixteen slave processes using an RTM engine which then works on a specific portion of the input image.

```
inImT[i][j]:=((im[i][j][1]+im[i][j][2])+
im[i][j][0])/3

SEQ k=0 FOR noP
SEQ jy=0 FOR (procWT+6)
input[k][jy] := inImT[i][jy+(procWT*k)]

PAR pi=0 FOR noP
inF[pi] ! input[pi]

PAR pr=0 FOR noP
fastProc (idT[pr], inIm[pr], offsetX[pr],
offsetY[pr], inF[pr]?,outF[pr]!)

  SEQ op=0 FOR noP
  outF[op] ? output[op]
```

Listing 3. Occam-pi code that splits an intensity image for the given number of processes

7 Implementation Results and Discussion

In this section, we will analyze our compilation methodology using the FAST corner detection case study. Our aim is to demonstrate the applicability of the programming model of `occam-pi`, to verify that programming is simplified when using the `occam-pi` language, and to assess the competitiveness in terms of performance. We compare our compilation based implementation with one implementation that was hand written in NPM, as well as with an other compiled implementation based on OpenCL. We implement a computation intensive application, which can benefit from the parallel compute resources of P2012 and show the simplicity of using `occam-pi` to express parallelism in an algorithm where communicating processes are natural elements of abstraction.

In Table 1 we have compared our implementation with the hand written NPM version. As a measure of implementation complexity we use the number of lines-of-code. The `occam-pi` program shows significant reduction in lines-of-code, 2x in the implementation of FAST. In Table 1, we also show the set up times and execution times for both versions. The set up time includes the configuration and deployment, and the execution time includes computation and communication time.

Both versions of the FAST implementation use 16 processes to detect corners and to compute corner scores, and they both use the host CPU to execute non-maximal suppression. As seen by the measured performance times, the `occam-pi` version outperforms the hand written version not only in simplicity in terms of lines-of-code but also in speed. The difference in time is the result of three main reasons:

1. The FAST implementation in `occam-pi` transfers data in the form of arrays. After the image is split, each process reads the entire line of its portion in a single step.
2. The hand written version has an overhead of protecting shared memory accesses. The `occam-pi` version solves this problem by duplicating the boundary pixels when splitting the image.
3. The third reason is the overhead due to dynamic allocation of resources when using RTM engine.

Both versions of FAST implantations have been tested on the same image (VGA sized input image) and they have detected 3146 corners. With non-max suppression, 772 have been selected as good corners. The (identical) outputs of the `occam-pi` version and the hand written NPM implementation are shown in Fig. 4.

Image analysis applications are usually data intensive and are suitable for programming models that can expose a high degree of data-level parallelism like OpenCL [20]. Our `occam-pi` implementation has utilized data-level parallelism by duplicating critical sections and by using channels that transfer an entire array of data. By this it has achieved better performance than the NPM version, which uses RTM engines. An implementation of FAST on P2012 using OpenCL was reported in [21], where 777 corners were detected as good corners on the same image that was used in the case of `occam-pi` and NPM implementations, resulting in an execution time of 30 milliseconds. In the case of OpenCL implementation, the threshold value is varied from 20 to 35 to get the best value of detected corners.

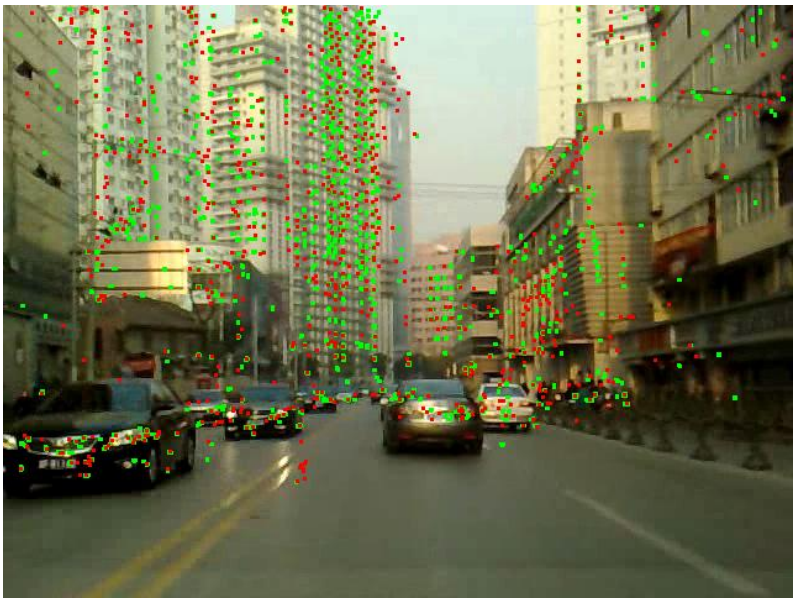

Fig. 4. An image with detected corners (red dots) and suppressed corners (green dots)

Table 1. Simulation results for the FAST Corner Detection implementations

Simulation Results/Languages	NPM	OpenCL	Occam-pi
Lines of Code	453	450	190
No of ENcore processors	16	16	16
Setup time (µs)	53,383	-	47,515
Execution time (µs)	67,115	30,000	32,549

The implementation results reveal that both the OpenCL and `occam-pi` implementations outperform the NPM version in terms of execution time. The `occam-pi` implementation is much simpler when compared to the OpenCL and hand-coded NPM versions, which is evident from the lines of code counts. From the implementation results we can see that the cost of configuration and deployment (the setup time) is significant as compared to the actual execution time. Especially, dynamic loading of tasks, as done in the RTM engine is very costly. But, if the processors are deployed at start up and used for long time, which is often the case in streaming applications, the time spent on configuration and deployment could be compensated. The OpenCL implementation was developed under that assumption, therefore the setup time was not measured. On the other hand, knowing the setup time may be important when scheduling reconfigurations.

The implementation using `occam-pi` is more concise than the two other versions. This is a consequence of the high level constructs of the language, which is also a feature that leads to fewer opportunities to introduce errors and to a higher likelihood of finding errors. Also, using a high-level approach like `occam-pi` and OpenCL makes the program easier to *scale*, in the sense that changing the number of processing elements involved in the computation is determined in one place in the program (the bound for the replication of processes). We gain also in portability given that a change of platform requires a change in one program: the compiler, instead of changes to all applications.

OpenCL implementations are based on the single instruction stream, multiple threads (SIMT) execution model, meaning that each processing element is executing the same instruction flow. On the other hand the `occam-pi` implementations are based on the multiple instruction streams, multiple data streams (MIMD) approach, where each processing core can execute its own instruction stream. This closely resembles the underlying manycore architecture.

8 Conclusions and Future Work

We have presented our approach to map programs in a CSP based language to a manycore architecture. We have extended our `occam-pi` compiler framework to generate native programming language code for Platform 2012. We have shown the simplicity of programming in `occam-pi` and the performance competitiveness of

our compilation based approach through a case study using FAST corner detection implementations. The result of the case study demonstrates the practicality of our approach for an algorithm that is both communication intensive and compute-data intensive. It has been concluded from the results that the `occam-pi` implementation achieves much better execution time results with respect to the hand-coded NPM version with a relatively less development effort. The `occam-pi` implementation execution time results are also comparable to those of an OpenCL version, again at a reduced development effort as evident from the lines of code counts. Future work will focus on making further evaluations of the approach using complex examples.

Acknowledgment

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., Barnes, F.R.M.: Communicating mobile processes Introducing `occam-pi`. Lecture Notes in Computer Science. Springer Verlag, pp. 175-210, 2005.
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, 1985.
3. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes Part I. Information and Computation, vol. 100(1), 1989.
4. Zain-ul-Abdin, Svensson, B.: Using a CSP based programming model for reconfigurable processor arrays. In: International Conference on Reconfigurable Computing and FPGAs, 2008.
5. Zain-ul-Abdin, Svensson, B.: `Occam-pi` as a high-level language for coarse-grained reconfigurable architectures. In: 18th International Reconfigurable Architectures Workshop (RAW'11) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'11) , May 2011.
6. STMicroelectronics and CEA.: Platform 2012: A manycore programmable accelerator for ultra-efficient embedded computing in nanometer technology. November 2010.
7. Rosten, E., Drummond, T.: Fusing points and lines for high performance tracking", Proceedings of 10th IEEE International Conference on Computer Vision, vol. 2, pp. 1508-1515, 2005.
8. Chamberlain, B., Callahan, D., Zima, H.: Parallel Programmability and the Chapel Language. In: International Journal of High Performance Computing Applications, vol. 21(3), pp. 291–312, 2007.
9. Steele. G. L. Jr.: Parallel programming and parallel abstractions in fortress. IEEE PACT, 2005, pp. 157.
10. Charles, P.C., Grothoff, Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not., vol. 40(10), pp. 519–538, 2005.
11. Axelsson, E., Claessen, K., Devai, G., Horvath, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for

digital signal processing algorithms. In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp 169-178, July 2010.
12. Eker, J., Janneck, J. W.: CAL language report. Technical Report, (UCB/ERL M03/48), 2003.
13. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W. Rizzolo, N.: Spiral: Code generation for DSP transforms", In: EEE Special Issue on Program Generation, Optimization, and Platform Adaptation, 2005.
14. Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited, 1995.
15. Welch, P.H., and Barnes, F.R.M.: Prioritised dynamic communicating processes: Part II. Communicating Process Architectures, IOS Press. pp. 353- 370 (2002).
16. The MIND Project, 15th December, 2011. http://mind.ow2.org
17. Kent.:Tock: Translator from Occam to C. 15th December, 2011. http://projects.cs.kent.ac.uk/projects/tock/trac/
18. Zain-ul-Abdin, Svensson, B.: Occam-pi for programming of massively parallel reconfigurable architectures. In: International Journal of Reconfigurable Computing, vol. 2012, Article ID 504815, 2012.
19. Rosten, E., Porter, R., Drummond, T.: FASTER and better: A machine learning approach to corner detection. In: IEEE Transactions. on Pattern Analysis and Machine Intelligence, vol. 32, pp. 105-119, 2010
20. The Khronos Group, OpenCL 1.0, 21st December 2012. http://www.khronos.org/opencl
21. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In: 49th Annual Design Automation Conference, 2012.
22. Zain-ul-Abdin, Gebrewahid, E., Svensson, B.: Managing Dynamic Reconfiguration for Fault-tolerance on a Manycore Architecture. In: 19th International Reconfigurable Architectures Workshop (RAW'11) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'11), May 2012 P. 312- 319.