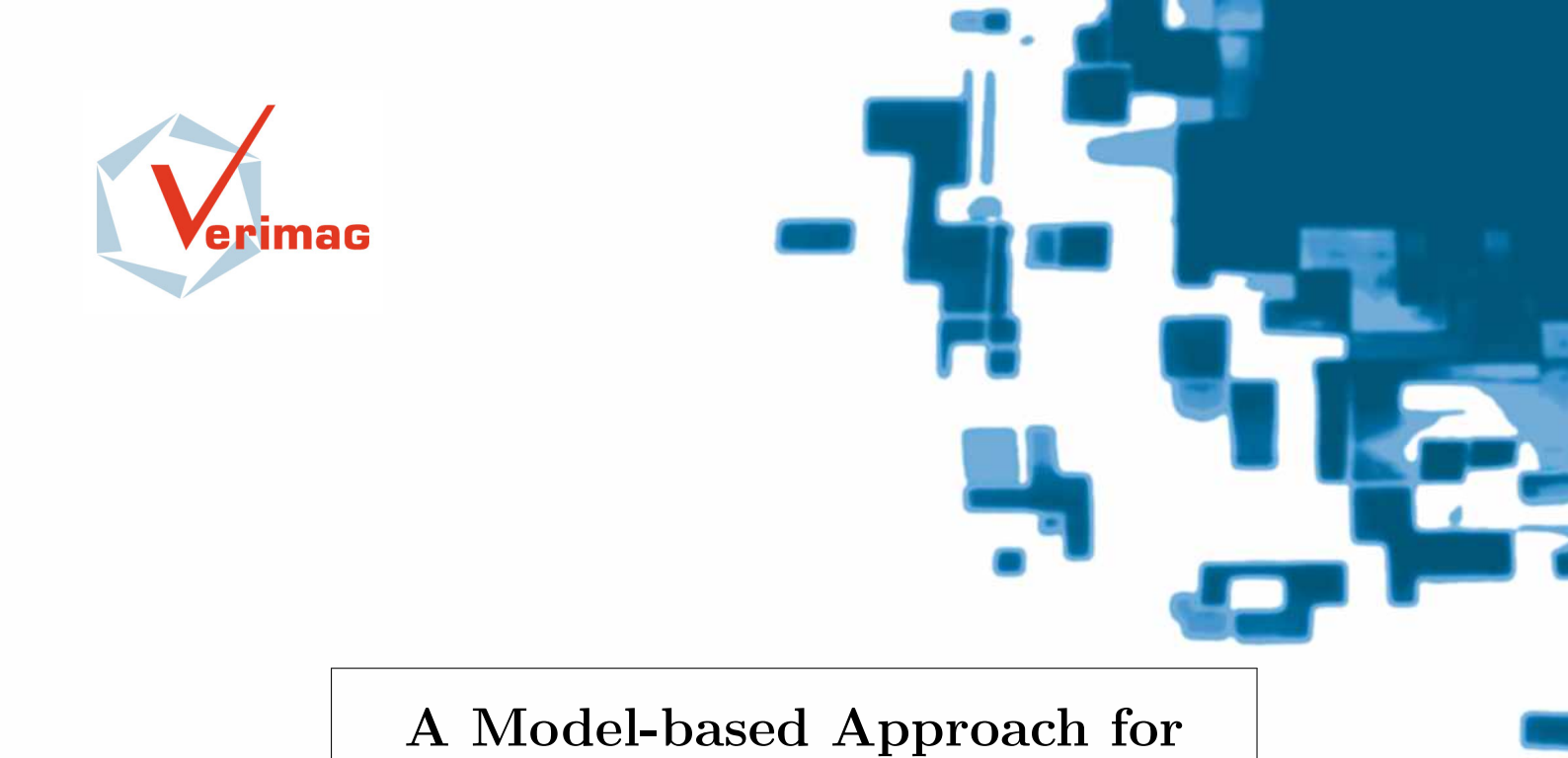

A Model-based Approach for Rapid Prototyping of Parallel Applications on Manycore

Ayoub Nouri[1], *Anca Molnos*[2], *Julien Mottin*[2], *Marius Bozga*[1], *Saddek Bensalem*[1], *Arnaud Tonda*[2], *Francois Pacull*[2]

Verimag Research Report n^o TR-2014-9

June 30, 2014

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

A Model-based Approach for Rapid Prototyping of Parallel Applications on Manycore

Ayoub Nouri[1], *Anca Molnos*[2], *Julien Mottin*[2], *Marius Bozga*[1], *Saddek Bensalem*[1], *Arnaud Tonda*[2], *Francois Pacull*[2]

June 30, 2014

Abstract

Rapid prototyping of highly parallel applications on manycore platforms is extremely challenging. This paper presents an automated analysis and code generation flow for implementing high-level KPN models on STHORM, an embedded 64-core computing fabric developed by STMicroelectronics. The flow is model-based with sound semantical basis and enables formal verification and performance analysis at different stages. The target for code generation is the MCAPI programming standard which has been ported on STHORM. Currently, the flow is fully operational. We report concrete results obtained on image-processing algorithms and illustrate the potential benefits of the flow for exploring implementation trade-offs.

Keywords: Code generation flow, Manycore architecture, Model-based design

Reviewers: Marius Bozga, Anca Molnos

Notes: [1] VERIMAG, Grenoble, France, [2] CEA/LETI Minatec, Grenoble, France

How to cite this report:

@techreport {TR-2014-9,
title = {A Model-based Approach for Rapid Prototyping
of Parallel Applications on Manycore},
author = {Ayoub Nouri[1], Anca Molnos[2], Julien Mottin[2], Marius Bozga[1], Saddek
Bensalem[1], Arnaud Tonda[2], Francois Pacull[2]},
institution = {{Verimag} Research Report},
number = {TR-2014-9},
year = {}
}

1 Introduction

Exploiting the increasing computational capabilities of manycore platforms requires applications to be designed in a very thought way. Finding the best parallelization strategy, identifying an optimal mapping, reducing communication overhead, optimizing extra functional requirements such as power consumption, while guaranteeing correct behavior are among important design parameters that need to be considered. Exploring and evaluating implementation alternatives is notoriously hard, and time consuming, even for experts.

This paper presents a model-based design flow encompassing system-level analysis and automatic code generation for STHORM [15], an embedded low-power many-core platform developed by STMicroelectronics. This flow is rigorous, automated and allows fine-grain analysis of final hardware/software system dynamics. It is rigorous because it is based on formal models described in BIP component framework [3], with precise semantics that can be analyzed by using formal techniques. A system model in BIP is derived by progressively integrating constraints induced on application software by the underlying hardware platform. The system construction method has been previously introduced in [4]. The application software and the abstract model of the platform are initially defined using Kahn Process Network (KPN) [12]. In contrast to ad-hoc modeling approaches, the system model is obtained, in a compositional and incremental manner, from BIP models of the application software and the hardware architecture. This is done by application of automated transformations that are proven correct-by-construction in addition to calibration steps. The system model describes the behavior of the mixed hardware/software system and can be simulated and formally verified using the BIP toolset [1].

In this paper, we are focusing on the backend part of the design flow, namely the automatic code generation from BIP system models to the STHORM platform and trade-off analysis at system-level using simulation. Writing good applications for STHORM is extremely challenging. The platform delivers an impressive computing power thanks to the 64 tightly interconnected cores, a NUMA memory architecture (with the L1 level accessible in a single cycle), and many data communication and control mechanisms. All these resources are available to the programmer and offer many opportunities for implementation. Yet, their naive utilization may easily lead to poor performance as interferences can occur and alter the expected behavior.

To overcome such issues which are frequent for low-level programming, as a first step, we ported the MCAPI standard [2] on STHORM and provided a minimal MCAPI-aware runtime on top of it. The runtime is meant to be deployed both on the manycore fabric and its associated host processor and provide a low-level software stack for thread management, memory allocation, inter-thread communication and synchronization. As a second step, we developed an analysis method and an automatic code generation process from BIP system models into C code to run on top of this runtime. We consider two distinct objectives namely (1) implement the functionality of the application with good design parameter, and (2) ensure its correct and automatic deployment on the manycore fabric and the host. The functional code is generated from the BIP software model. It consists of a set of parallel tasks, each corresponding to a process from the KPN representation of the application. Communication between tasks is performed using MCAPI primitives. The deployment code is fully generated from the description of the static mapping. This code implements the allocation of various application objects onto the platform, i.e., allocation of task threads to specific MCAPI domains (host or fabric) and/or cores, allocation of FIFO buffers and/or other shared objects to memories, and performs all the initialization/synchronizations needed to bring the application to an initially valid and functional state i.e., opening and connecting queues, etc.

We experimented the design flow above on the parallel software model derived from a sequential version of the HMAX Models application [17]. We were able to quickly investigate several configurations of the application software (pipelining parameters, FIFOs sizes, etc.) and different mappings on the physical platform. As the flow enables automatic code generation, any change within the application and/or its mapping is rapidly propagated down to the concrete implementation which considerably reduces the development time and avoid intricate deployment bugs with respect to a manual or semi-automated approach. Moreover, the system-level analysis allows to investigate several design alternatives and to find good trade-offs.

The rest of the paper is organized as follow. Section 2 shows an overview of the design flow. Section 3 introduces the STHORM platform and its corresponding MCAPI implementation. The code generation and analyis steps are illustrated in section 4. In section 5, we present the HMAX Models case study and the exploration results obtained by applying our approach. In section 6, we present some related works, and finally, the end tail portion concludes the paper and gives future directions.

2 Design Flow for Manycore

As shown in Figure 1, the entry point in the proposed design flow are parallel applications described as KPN and statically mapped on the target platform. Applications are concretely represented using the Distributed Operation Layer (DOL) [20] syntax. In a first step, applications are translated into BIP *software models* using the *dol2bip* tool. These models capture solely the functionality of the application software and are totally independent of the platform. At this level, it is possible to use the *D-finder* tool [7] to perform functional verification (deadlock freedom, etc.) of the application software. In a second step, the models are progressively refined trough specific transformations (see [4]) using *BIP/weaver* into BIP *system models*. They represent the behavior of the application mapped to the platform. This model does not yet contain hardware extra-functional information. In a third step, software models are used to generate a concrete implementation, that is, platform-dependent code. In our flow, the generated code is targeting the Multicore Communication API (MCAPI) implementation and its associated runtime for STHORM platform. The next step consists to profile the generated code, execute it on the platform, and use the execution information (latency, communication time, power consumption, etc.) to calibrate the system model, that is, to include all relevant non-functional constraints induced by the platform into the application. Finally, the obtained model is used to explore possible design parameters, to find good trade-offs, and to decide which ones to use. This is performed using simulation and debugging back-end BIP tools [1].

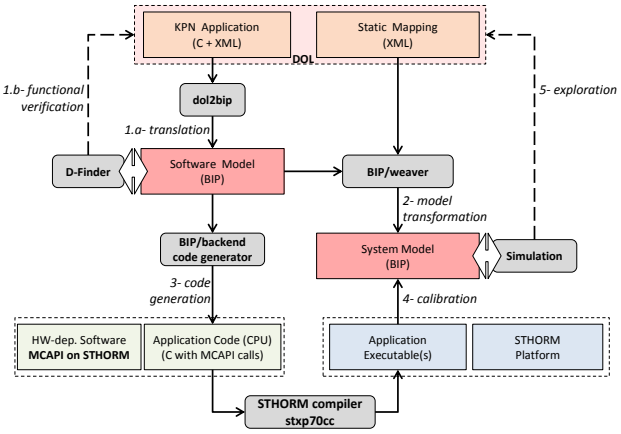

Figure 1: Overview of the Design Flow.

KPN/DOL: A High-Level Programming Model DOL is a framework devoted to the specification and analysis of mixed hardware and software systems. It provides languages for the representation of particular classes of applications software, multi-processor architectures and their mappings. In DOL, application software is defined using a variant of KPN model. It consists of a set of deterministic, sequential processes (in C) communicating asynchronously through FIFO channels. The hardware architecture is described as interconnections of computational and communication resources such as processors, buses and memories. The mapping associates application software components to resources of the hardware architecture, that is, processes to processors and FIFO channels to memories.

BIP: An Intermediate Model for Analysis, Validation and Code Generation BIP (Behavior-Interaction-Priority) [3] is a formal framework for building complex systems by coordinating the behavior of a set of atomic components. Behavior is defined as automata or Petri nets extended with data and functions described in C/C++. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between interactions and is used to express scheduling policies. BIP has clean operational semantics that describes the behavior of a composite component as the composition of the behaviors of its atomic ones. This allows a direct relation between the underlying semantic model (transition systems) and its implementation. In BIP, atomic components are finite-state automata extended with variables and ports. Variables are used to store local data. Ports are action names, and may be associated with variables. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is a step, labeled by a port, from a control location to another. It has associated a guard and an action, that are respectively a Boolean condition and a computation defined on local variables. The BIP toolset [1] offers a rich set of tools for modeling, simulation, analysis (both static and on-the-fly) and transformations of BIP models. It provides a dedicated modeling language for describing BIP models. The front-end tools allow editing and parsing of BIP descriptions, followed by a backend code generation step (in C/C++). The generated code can be used either for execution or for performance analysis using backend simulation tools.

MCAPI: A Low Level Programming Model MCAPI is a standard proposed by the Multicore Association to define an API and a semantic for communication and synchronization between processing cores in embedded systems. The MCAPI specification is based on few basic concepts such as *Node, Endpoint*, and *Domain*. A *Node* is an independent thread of control that can communicate with other nodes. The exact nature of a node is defined by the MCAPI implementation, but it can basically be a process, a core, a thread, or a HW IP. An *Endpoint* is a communication termination point and is therefore connected to a Node. One node can have multiple endpoints, but one endpoint belongs to a single node. Finally, a *Domain* is a set of MCAPI nodes that are grouped together for identification or routing purpose. The semantics attached to a domain is given by the implementation, but it can basically be a cluster, a chip, a terminal, a mobile, etc. Each node can only belong to a single domain. Compared to the Massage Passing Interface (MPI)[1], MCAPI offers inter-core communication with potential low-latency and minimal footprint. MCAPI communication nodes are all statically predefined by the implementation. MCAPI offers three fundamental communication mechanisms. First, *Messages* which are streams sent from one endpoint to another. No connection is established between the two endpoints to send a message. This is the easiest way of communicating between two nodes. Second, *Packet Channel* which is a FIFO unidirectional stream of data packets of variable size, sent from one endpoint to another. And finally, *Scalar Channel* that is similar to a packet channel, except that only fixed-length word of data can be sent through the channel. A word may be 8, 16, 32 or 64 bit of data. Figure 2 presents the main concepts of MCAPI.

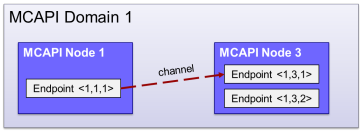

Figure 2: MCAPI main concepts.

[1] http://www.mcs.anl.gov/research/projects/mpi/

3 MCAPI for STHORM

This section first introduces the STHORM platform general characteristics and presents the test-board used in the case study. It then presents our MCAPI implementation.

3.1 STHORM Platform

STHORM [15] is a power efficient many-core system consisting of a host processor and a many-core fabric. The host processor is a dual-core ARM cortex A9. The STHORM fabric comprises computing clusters, inter-connected via a high-performance fully-asynchronous (2D mesh structure) network-on-chip (NoC), which provides communication with high, scalable bandwidth. Each cluster aggregates a multi-core computing engine, called ENCore, and a cluster controller (CC). The ENCore embeds a set of tightly-coupled processors elements (PE) which are customizable 32-bit STxP70-v4 RISC processors from ST Microelectronics. On the STHORM test-board used in our experiments, the fabric comprises 4 clusters, with 16 PEs each. The PEs in one cluster share a multi-banked level-1 (L1) data memory of 256 KBytes. The banks of the L1 memory can be accessed in parallel in one processor cycle. Each PE has its private instruction cache with a size of 16 KBytes.

The CC consists of a cluster core (STxP70-v4), a multi-channel advanced DMA engine, and specialized hardware for synchronization. The latter two are accessible also by the PEs. The CC interconnects with two interfaces: one to the ENCore and one to the asynchronous NoC. All clusters share 1 MByte of level-2 (L2) memory, accessible via the NoC. The access time is several tens of cycles. A DDR3 level-3 (L3) memory is available off-chip (1 GByte); this memory has a large size, however its access time and bandwidth are much slower than the ones of on-chip memory.

In summary, due to area and power constraints, the fast memory available on the chip is scarce. Furthermore, the host processor and the cores on the fabric have a different instruction-set-architecture. These two aspects make efficient programming on STHORM a challenging task.

3.2 MCAPI Implementation on STHORM

The main objective of the MCAPI implementation on STHORM is to offer a homogeneous programming interface for the entire platform (fabric and host). This uniform representation covering the entire platform can perfectly integrate the full design flow targeting STHORM, which significantly eases code generation and analysis. The current MCAPI implementation features five domains: one for each cluster on the fabric, and one for the host as shown in Figure 3.

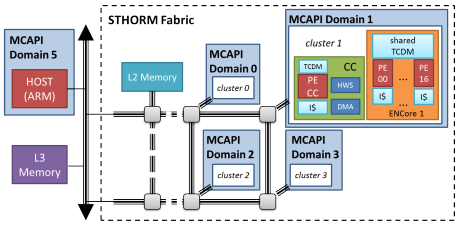

Figure 3: MCAPI Domains on STHORM.

The five domains naturally reflect the STHORM hardware organization. Furthermore, in each cluster-domain each MCAPI node is mapped on a PE. The host-domain has only a single node corresponding to the ARM dual-core processor. The host node is responsible for the deployment of the entire execution, as it is the main entry point of an application. The MCAPI initialization of the host node automatically loads the fabric binary code into the L2 memory, and starts the fabric MCAPI nodes. The MCAPI implementation totally hides from application the complexity of binary and symbols

dynamic load and dependencies solving. A single semantic for the entire STHORM platform is exposed for homogeneous programming.

The channels are implemented using FIFO buffers allocated in any of the memory levels available on STHORM. The size of the allocated buffer and its memory placement can be set by using endpoint attributes, as specified in the MCAPI standard. The sending endpoint and the receiving endpoint must have consistent attributes definition for the creation of a channel. No MCAPI node is mapped on the CCs, meaning that they are not directly visible to the programmer. Instead, they are used internally in the MCAPI implementation to support the various communication mechanisms and synchronizations. DMA engines are also hidden from the programmer, but used by the implementation to transfer data from one level to the another.

4 Code Generation and Analysis

This section describes the transformation and exploration steps performed to obtain low-level running code with appropriate design parameters. It consists mainly to produce an intermediate BIP model enabling analysis and code generation. The process is shown on an illustrative example.

4.1 Generating the BIP Software Model

Figure 4 shows a KPN model composed by six processes, namely *Config, Splitter, Joiner*, and three *Worker* instances. The communication between these processes is based on blocking *Read/Write* primitives on FIFO channels following the DOL semantics. In this example, the initial step consists to configure (data size, data type, etc.) the *Worker* processes. It is performed by the *Config* process. Generic processes are considered to enable the processing of different data types/sizes. After configuration, the data to be processed is split and sent by the *Splitter* to the workers that run in parallel. Finally, the results is collected by the *Joiner*. The code sample below shows the corresponding DOL implementation of the *Worker* process.

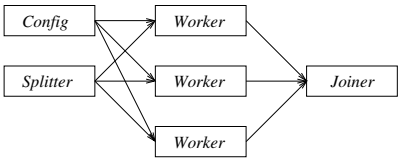

Figure 4: A KPN model example.

```
/** DOL Worker Process **/
void worker_init(){status = CONFIG;}
void worker_fire()
{ if(status == CONFIG){
    // Read configuration parameters
    // from input configuration FIFO
    read(CONF_FIFO, &config);
    // Update number of steps
    step = config.step;
    // Update status
    status = EXEC;}
  if(status == EXEC){
    // Read data from input data FIFO;
    read(DATA_FIFO_IN, &data);
    // Execute computation function
    compute(&data);
    // Write data to output FIFO;
    write(DATA_FIFO_OUT, data);
    // Update number of steps
```

```
step--;
// Update status
if(step == 0) status = CONFIG;}}
```

Given the KPN model in Figure 4, the generation of the corresponding BIP representation is straightforward. As shown in Figure 5, each process is transformed to an atomic BIP component modeled as an extended automaton following the BIP formalism. An example of the *Worker* BIP component (which corresponds to the code sample above) is depicted in Figure 6. In this automaton, the DOL *Read/Write* calls are transformed to equivalent BIP interactions (ports) synchronized with BIP FIFO components providing *send/receive* primitives. Parametrized FIFO components (with a maximum capacity) are explicitly inserted for each inter-process communication. An example of a FIFO component is shown is Figure 7. The connection of the processing components (*Splitter, Gabors, Joiner*) with FIFOs is made through BIP render-vous connectors that models strong synchronization.

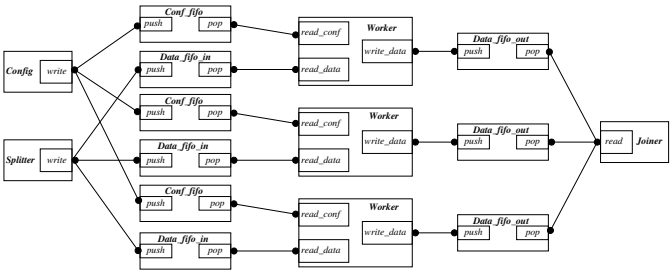

Figure 5: The BIP Model of the *Worker* example.

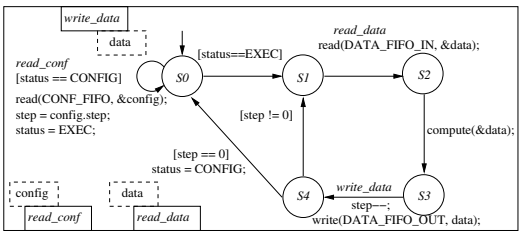

Figure 6: The BIP model of the *Worker* process.

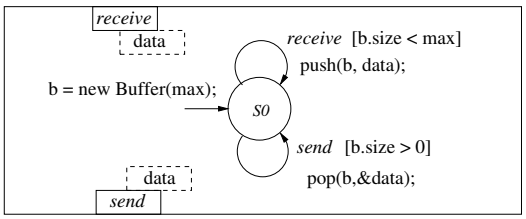

Figure 7: The BIP model of the FIFO component.

4.2 Exploring the BIP System Model

In this phase we aim to analyze the generated model in order to find good design parameters (fifo sizes, etc.). Note that the BIP software model generated in the previous step is limited to the ap-

plication behavior which is not sufficient for parameter exploration. Those are strongly correlated to the hardware platform. We propose a method to enrich the BIP software model with relevant hardware extra-functional information while keeping it small enough for simulation/exploration.

The idea is to use the BIP software model to generate running code on the target platform, that is, in this context, STHORM. Initial parameters values can be given by expert or chosen arbitrarily, similarly for the mapping of the processes on the platform. This code will be profiled and executed on the real platform to retrieve concrete information about execution time, latency, energy consumption, etc. This information is then injected in the BIP model in form of probability distributions (calibration phase). Note that the retrieved information is subject to random variability because of inherent hardware characteristics. Figure 8 shows the *Worker* BIP component of Figure 6 annotated with time information trough the tick transitions on states *S2, S3, S4*. The component uses three different probability distributions f_read(),f_exec(), and f_write() (obtained from a concrete run on the platform) to model respectively the read, the compute, and the write time. The distributions are used to update the *time* variable in a stochastic manner. This amount of time is consumed on the *tick* transition. When it becomes null, the next transition is enabled. The obtained model is a stochastic BIP model [6].

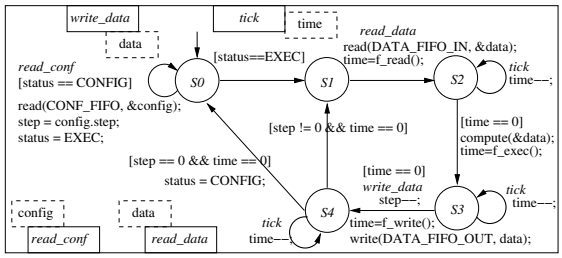

Figure 8: Time annotated BIP Worker component.

The obtained BIP system model is more faithful than the BIP software model since it takes into account, in addition, the hardawre characteristics. Using the BIP back-end simulation and debugging tools, it is possible to quickly explore several parameters. This exploration phase is iterative, that is, it may lead, given the simulation results, to change the high level KPN model and/or the associated mapping, regenerate and re-explore the new model, and so on. When good parameters/mapping are found, they are used together with the software BIP model to generate a final running version of the application.

4.3 Generating C/MCAPI Executable Code

Given the software BIP model, a set of design parameters, and a mapping, this step consists to generate the low-level running C/MCAPI code for STHORM. This phase is divided into two main steps: (1) Object generation, that is processes, FIFO buffers, and shared objects and (2) Deployment/Glue code generation that consists on mapping the generated objects on the physical platform.

Process Generation During this step, each BIP component is transformed to a C/MCAPI process. A challenging point at this level is to generate small set of process local variables to fit the small amount of available memory per process stack on STHORM. The generated processes are basically composed of two parts:

- **Initialization:** in this part, BIP components interfaces (ports) are transformed into equivalent MCAPI endpoints (Appendix A shows a sample of initialization code). The generated process endpoints are initialized, opened, and connected to other processes in this same block. Moreover, endpoints memory attributes are generated to allocate FIFO buffers and shared objects (given their sizes) and to map them into specific memory locations. MCAPI

implementation for STHORM provides several possibility to map buffers into specific target memory as shown in Table 1. Buffers could be mapped either in $L3$, $L2$ or $L1$ memory using the attributes shown in the table. For $L1$ memory, it depends on the sender/receiver processes location. If these run in the same cluster, the buffer is allocated in the $L1$ memory of that cluster. Otherwise, it is mapped in the $L1$ of the cluster where the sender (respectively the receiver) runs.

Target Memory	*Sender Attribute*	*Receiver Attribute*
$L3$	*Remote*	*Remote*
$L2$	*Shared*	*Shared*
Receiver $L1$	*Shared*	*Local*
Sender $L1$	*Local*	*Shared*

Table 1: MCAPI endpoints memory attributes.

- **Behavior:** in this part, BIP component behavior is transformed to equivalent C code with MCAPI primitives calls. It consists essentially on an infinite while loop with several steps (using Switch/Case statements) reproducing the BIP automata behavior as shown in the code sample below that corresponds to the BIP component in Figure 6. In the generated code, all BIP synchronizations are transformed to C/MCAPI *Send/Receive* primitives.

```
void worker_ins_execute(void* arg) {
/* Initialization */
...
/* Behavior */
while(Wlcontinue){
 switch(BIP_CTRL_LOC){
  case S0 : {
   if (status == CONFIG) {
    ...
    status = EXEC; BIP_CTRL_LOC = S0;}
   if (status == EXEC) BIP_CTRL_LOC = S1;
   break;}
  case S1 : {
   mcapi_pktchan_recv(h_WORKER_read_data,
   (void**)&mcapi_buffer, &mcapi_received, &mcapi_status);
   if((mcapi_received != size) ||
   (mcapi_status != MCAPI_SUCCESS))
    ERR_RAISE("FAIL TO READ DATA");
   memcpy(data, mcapi_buffer, mcapi_received);
   size = mcapi_received;
   mcapi_pktchan_release(mcapi_buffer, &mcapi_status);
   if(mcapi_status != MCAPI_SUCCESS)
    ERR_RAISE("FAIL TO RELEASE CHANNEL");
   BIP_CTRL_LOC=S2 ;
   break;}
  case S2 : {
   compute(&data); BIP_CTRL_LOC=S3;
   break;}
  case S3 : {
   mcapi_pktchan_send(h_WORKER_write_data, data,
   size, &mcapi_status);
   if(mcapi_status != MCAPI_SUCCESS)
    ERR_RAISE("FAIL TO SEND DATA");
   step--; BIP_CTRL_LOC=S4;
   break;}
  ... }}}
```

Deployment/Glue Code Generation This code is composed by two parts as well: a generic and a generated part. The generic part is parametrized by the amount of memory to be allocated per processes stack. It is meant to allocate the generated processes memory and to launch them on the specified hardware location (given the mapping). The generated part is the one that specifies the mapping of the generated objects on the physical platform. This is obtained based on the input mapping. In addition, parametric Makefiles are automatically generated to compile and run the generated C/MCAPI code on the STHORM test-board.

5 Case study

This section introduces the HMAX Models algorithm for image recognition and shows the pertinence of using our flow to perform design exploration on high-level models trough simulation and automatic code generation.

5.1 Application Overview

HMAX [17] is an hierarchical computational model of object recognition which attempts to mimic the rapid object recognition of human brain. Hierarchical approaches to generic object recognition have become increasingly popular over the years, they indeed have been shown to consistently outperform flat single-template (holistic) object recognition systems on a variety of object recognition task. Recognition typically involves the computation of a set of target features at one step, and their combination in the next step. A combination of target features at one step is called a layer, and can be modeled by a 3D array of units which collectively represent the activity of set of features (F) at a given location in a 2D input grid. HMAX starts with an image layer of gray scale pixels (a single feature layer) and successively computes higher layers, alternating "S" and "C" layers: Simple ("S") layers apply local filters that compute higher-order features by combining different types of units in the previous layer. Complex ("C") layers increase invariance by pooling units of the same type in the previous layer over limited ranges. At the same time, the number of units is reduced by sub-sampling.

In the case study, we only focus in the first layer of the HMAX Models algorithm (see Figure 9) as it is the most computationally intensive. In a pre-processing phase, the input raw image is converted to gray scale input (only one input feature: intensity at pixel level) and the image is then sub-sampled at several resolutions (12 scales in our case). For the S1 layer, a battery of three 2D-Gabor filters is applied to the sub-sampled images and then for C1 layer, the spatial max of computed filters across two successive scales is taken. In this application, parallelism can be exploited at several levels. First, at layer level, where independent features can be computed simultaneously. Second, at pixel level, that is, the computation of contribution to a feature may be distributed among computing resources.

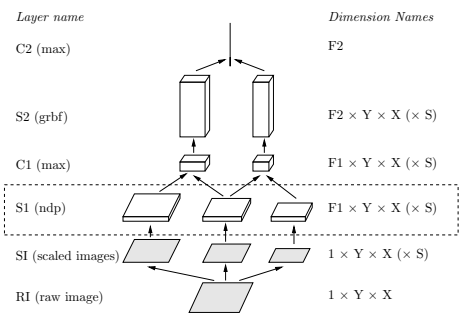

Figure 9: HMAX Models algorithm overview.

5.2 High-Level Reconfigurable KPN Model

We developed a parametric KPN model for the S1 layer of HMAX in DOL. The model is based on the worker pattern presented in Section 4. It uses a certain number of reconfigurable processes for implementing the 2D-Gabor filtering and image splitting/joining. Every image is handled by one "processing group" consisting of a *Splitter*, one or more *Gabor* (*Worker*) processes and a *Joiner* process, connected through blocking FIFO channels as illustrated in Figure 4. This model exploits parallelism both at image level, as different images are processed in parallel by different processing groups and at pixel level, as different stripes of the image are processed in parallel by different *Gabor* processes. Moreover, parallelism is exploited between pure computation on *Gabor* processes and data transfer from/to main memory by *Splitter*/*Joiner* processes.

The computation of the entire S1 layer is coordinated by a single main process. Several image scales are handled concurrently. That is, the twelve scaled images are statically pre-allocated and mapped on different processing groups. For every image scale, the processing is pipelined as follows. Initially, the main process sends the first $10 + P$ lines to the corresponding processing group, where $P \geq 0$ is an integer parameter called *line pressure* that specifies the pipelining rate. In normal regime, one input line is sent and one output line is received, for every filter rotation (that is, actually three output lines). Finally, once all the input image has been sent, the main process receives P more output lines. At this point, the processing group is ready (empty) and can be reconfigured to restart computation for another image scale.

Within the processing group, the *Splitter* receives input images (see Appendix), line by line from the main process. Every line is split into a number of equal length (and overlapping) fragments, one for every *Gabor* process, and sent to these processes. *Gabor* processes implement the computation of the 2D-filter itself. Filter size is fixed to 11×11 in the case study. Hence, *Gabor* processes need to accumulate 11 line fragments in order to perform computation. Henceforth, they maintain and compute the result operating on an internal "sliding" window of 11 line fragments. Finally, the resulting fragments are sent further to the *Joiner*, which packs them into complete output lines and send them to the main process.

5.3 Parameters Exploration at Image Level

Parameter-space exploration is usually required for transforming a parametrized design into a functional and efficient implementation on the target platform. In our case, several questions are naturally raised e.g., What is the best pipelining parameter P that gives a maximum throughput ? What are the optimal FIFOs sizes that reduce the overall execution time and fit the platform available amount of memory ? What is the best images mapping that reduces the processing time and the energy consumption ? All these are important design parameters that may eventually impact the application performance and behavior. Exploring the parameters space to find good trade-offs is essential for a good design. This task is not feasible manually even for experts, especially, at a low-level. Our flow provides a rigorous intermediate representation (BIP) that enables automatic verification of several aspects while taking into account hardware constraints, that is, considering faithful models.

To answer the above questions, we consider one processing group that deals with the biggest scale of the input image (256×256 floating point) as shown in Figure 10, that is, one *Splitter*, 14 *Gabors*, and one *Joiner*. This processing group is mapped on one cluster while the coordinating main process is mapped to the host part of the STHORM platform.

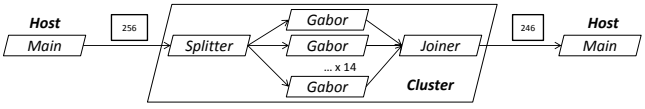

Figure 10: KPN model of one processing group of HMAX.

Using the biggest image on one processing group will help us to correctly scale the FIFOs size without dealing with the whole S1 layer complexity. Similarly the line pressure parameter P can be safely analyzed at this level since it acts independently in each processing group. We follow the steps of the flow illustrated in Figure 1:

First, generate the BIP software model The software model corresponds to the KPN shown in Figure 10 which is similar to the *Worker* pattern illustrated in Section 4. Although, in this model (Figure 10), we don't use a *Config* process (the configuration phase is performed by the *Splitter*) and we have an additional main process. The BIP software model is obtained through the steps depicted in Section 4. The *Gabor* component is similar to the *Worker* model shown in Figure 6. It basically receives an image fragment, applies the the 2D-filter (3 directions), and send them to the *Joiner*. Processing components communicate through generated FIFOs as shown in Figure 5.

Second, generate C/MCAPI code Based on initial parameters values (FIFOs sizes, *line pressure* $P = 0$), we use the back-end code generator as described in Section 4 to produce C/MCAPI code targeting STHORM from the BIP software model. Table 2 shows the initially used FIFOs sizes and their location on the platform. Columns of the table specify the FIFOs positions in the model. For instance *M-S* means the FIFO between the *Main* and the *Splitter* (*G: Gabor and J: Joiner*). These values are approximately computed based on the smallest processed fragment size (28 floating points). For instance, a FIFO with 112 Bytes can handle 1 fragments and the one with 336 Bytes can handle 3 fragments. The FIFOs buffers allocated in $L3$ are bigger. For instance, the FIFO between the *Main* and the *Splitter* can handle 10 lines of 256×256 image and the the FIFO between the *Joiner* and the *Main* can handle 30 lines of the same image.

FIFO	*M-S*	*S-G*	*G-J*	*J-M*
Size	10 KB	112 B	336 B	30 KB
Location	*L3*	*L1*	*L1*	*L3*

Table 2: Initial FIFOs sizes and mapping (*Configuration A*).

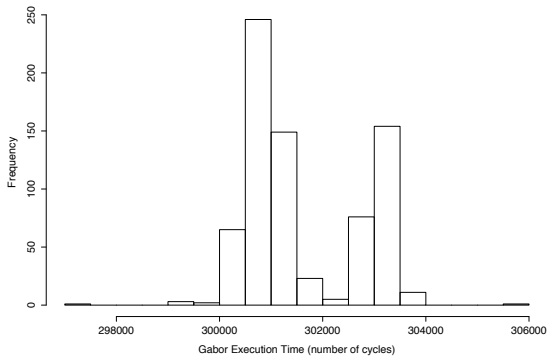

Figure 11: *Gabor* execution time frequency distribution.

Third, profile and run the generated code We mainly focus on execution and communication time of the processes. Our MCAPI implementation offers a set of profiling functions covering STHROM both sides (host and fabric). We execute the generated instrumented code on

a STHORM test-board and collect timing information. Figure 11 shows the distribution of Gabor execution time. This correspond to the time of the *gabor_compute_feature()* function which is the core computation part of the process (2D-filter computation). Note that all the *Gabor* processes use the same function and run on identical cores. Hence, the obtained distribution is used to calibrate all the *Gabor* models. We apply the same procedure for the communication time.

Forth, build and calibrate the BIP system model The system model is constructed flowing the same principles from [4]. Given the distributions of the execution and the communication time, we calibrate the BIP system model as shown for the *Worker* component in Section 4. That is, we build a stochastic BIP model where time is explicitly modeled using *tick* transitions. These model execution and communication time and are sampled from the underlying distributions obtained from the previous step.

And Finally, explore the obtained BIP system model Using the BIP simulator capabilities, we explore the stochastic system model by investigating several FIFOs sizes and P values combinations as depicted in Table 3. In this case study, we focus on the overall execution time. These configurations differ either in the size of the internal FIFOs (*S-G*, *G-J*) sizes or the external (*M-S*, *J-M*) ones. In *Configuration B*, we doubled the internal FIFOs sizes with respect to *Configuration A* and tripled them in *Configuration C*. The latter didn't work on the STHORM test-board due to memory overflow. In *Configuration D*, we kept the same internal FIFOs sizes as *Configuration B* and increased the external ones to see their impact on the execution time.

Configuration	*FIFOs Sizes*	P	*Exec. Time (ms)*
A	*M-S*= 10 KB	0	1098.814811
	S-G= 112 B	2	956.352644
	G-J= 336 B	5	956.739721
	J-M= 30 KB	10	956.573830
B	*M-S*= 10 KB	0	1092.584741
	S-G= 224 B	2	953.108584
	G-J= 672 B	5	956.131459
	J-M= 30 KB	10	955.818112
C	*M-S*= 10 KB	0	Does not work
	S-G= 336 B	2	
	G-J= 1008 B	5	
	J-M= 30 KB	10	
D	*M-S*= 20 KB	0	1076.327576
	S-G= 224 B	2	953.219177
	G-J= 672 B	5	956.371076
	J-M= 50 KB	10	956.315780

Table 3: FIFOs sizes configurations and P values and corresponding execution time on the STHORM test-board.

We construct the system models for all the above-mentioned configurations and simulate them. The overall execution time of the S1 layer (one cluster version) is reported in Figure 12. This shows the execution time evolution when increasing P for the different configurations when using simulation and for concrete run on the test-board. The simulation results do not show any significant differences between the considered FIFO sizes configurations (in Figure 12, a unique curve describes the different configurations when using simulation). Nevertheless, different results are obtained for respectively $P = 0$ and $P > 0$ as shown in the same figure. That is, in the former case (without pipelining), the total execution time is about 684 milli-seconds whereas in the later (with pipelining) is about 560 milli-seconds (regardless the value of P). One can then restrict the exploration space to $0 < P \leq 5$ since greater values of P does not really improve the throughput.

With respect to the FIFOs configurations, since no difference is seen, we decide to limit ourselves to configurations using less memory, that is, *Configuration A* and *B*.

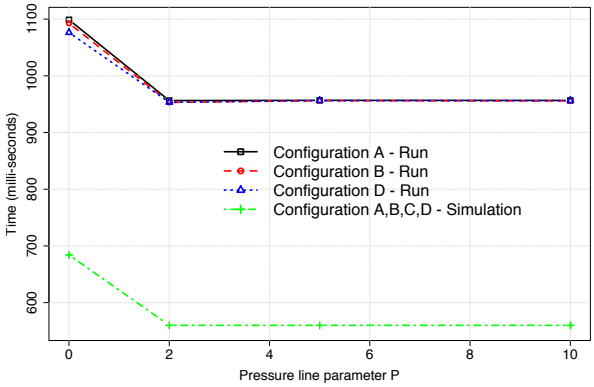

Figure 12: Overall execution time for different P values and FIFOs sizes configurations using simulation and concrete run on the STHORM test-board.

The simulation results are fully inline with the values measured on the test-board (detailed in Table 3) for the same configurations as we can see on Figure 12. Nevertheless, they are more optimistic since the built model still abstracting many hardware details. It is worth mentioning that the simulation time did not exceed 10 seconds for any of the considered configuration.

5.4 Parameter Exploration at the Layer Level

We now consider the full S1 layer which deals with all the image scales. We are going to use the set of candidate parameters obtained in the previous step (*Configuration A* and *B* with $2 \leq P \leq 5$) for further exploration with different images mapping. This version is composed by three distinct processing groups, with different sizes. These are deployed on different STHORM clusters as shown in Figure 13. FIFOs are mapped as earlier, that is, external FIFOs are mapped to $L3$ memory and internal ones are mapped to respective clusters $L1$ memories.

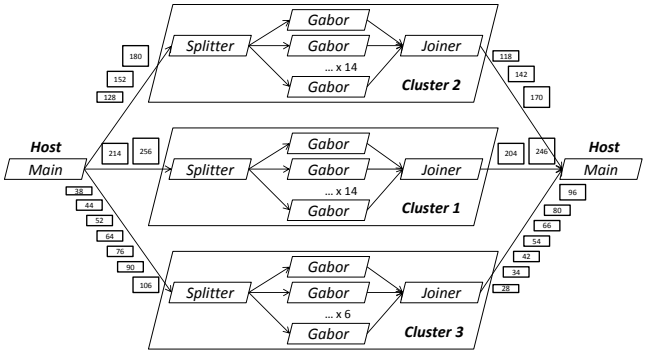

Figure 13: Full S1 layer mapping on the STHORM platform.

Investigating Different Images Mapping We investigate two different images mapping on the three available processing groups as shown in Table 4. These are explored, together with the previous candidate parameters, with respect to the application throughput as depicted in

Table 5. In this case, we only rely on the code generator to produce C/MCAPI code which is executed on the test-board. The reported results in this part are obtained through concrete run of the generated implementation. BIP models are only used to generate the implementations corresponding to the described scenarios. This helps us reducing the prototyping time since done at system-level. Moreover, it reduces bugs introduction in the final executable code with respect to manual approaches.

Mapping	*PG* 1	*PG* 2	*PG* 3
1	256, 214	180, 152, 128	106, ..., 38
2	256, 180	214, 152, 106	128, 90, ..., 38

Table 4: Image scales mapping on processing groups (PG).

The execution time evolution corresponding to each scenario of Table 5 is presented in Figure 14. One can conclude out of this figure that the scenario that ensures the lower execution time is the one of *mapping* 2 combined with configuration B and specifically when $P = 3$. In Figure 15 we show in detail the execution time of each image scale for this specific scenario compared to *mapping* 1 combined with configuration B for $P = 2$. We can note a clear change in the execution time of the interchanged image scale (from *mapping* 1 to *mapping* 2). For instance, image scale 106 was moved from $PG3$ in *mapping* 1 to $PG2$ in *mapping* 2. The latter is composed by 14 *Gabor* processes while the latter uses only 6. This explains the fall in *mapping* 2 curve for that scale in Figure 15. In contrast, scale 128 was moved from $PG2$ to $PG3$. This increases its processing time for the same reasons. One can note the same behavior for scales 180 and 214. Apart these scales, the other ones have more or less the same execution time in both scheduling. It seems that the reduction due to moving certain images from $PG3$ to $PG2$ is more important than the moves in the other way which reduces the global execution time of the S1 layer. Note that *mapping* 2 with *Configuration A* did not work on the STHROM test-board. This is due to the small size of the internal FIFOs (exactly one fragment) which is not sufficient for the image scale 128 on $PG3$ that uses only 6 *Gabor*.

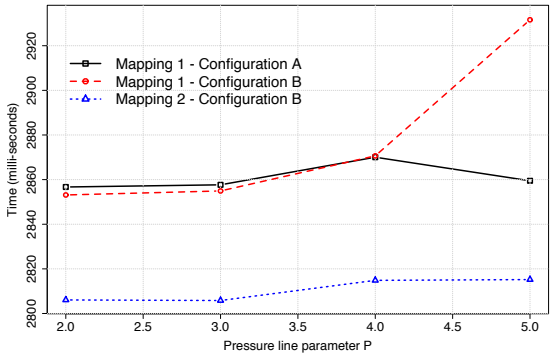

Figure 14: Overall execution time for different image scales mapping, configurations, and P values.

Discussion Exploring all these alternatives would have been very difficult manually and at a low-level. Using the automatic flow, it took as each time less than one seconds to regenerate a configuration and run it. Modifying these parameters at a high-level is indeed much more easier. It is worth mentioning that the BIP model of the S1 layer of HMAX is composed by 115 components with a lot of synchronization. The size of the generated C/MCAPI code is about 15000 loc over

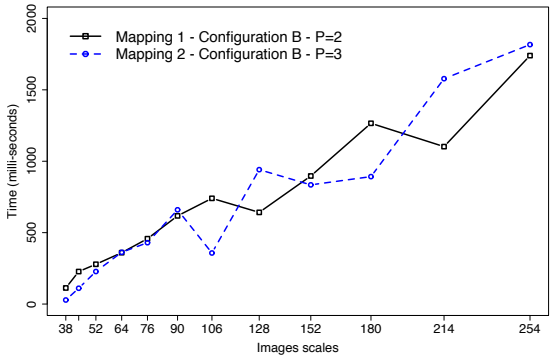

Figure 15: Execution time for each image scale with different mapping, configurations, and P values.

Mapping	*Configuration*	P	*Time (ms)*
1	A	2	2856.689953
		3	2857.666856
		4	2870.034837
		5	2859.473209
	B	2	2853.114112
		3	2854.920466
		4	2870.643097
		5	2931.616685
2	A	2	Does not work
		3	
		4	
		5	
	B	2	2806.075242
		3	2805.817190
		4	2814.848947
		5	2815.199160

Table 5: Investigated scheduling, configuration, and P values with respect to the application execution time.

47 files (1 file per process plus 6 files for the deployment). The MCAPI library size is about 7000 loc. Its footprint on the fabric is 190 KBytes and 66 KBytes on the host side.

6 Related Work

Many frameworks encompass code generation, analysis, and simulation tools like our flow. VISTA [16] is a tool enabling modeling and performance analysis of embedded applications on virtual architecture and exploration at system level. It relies on SystemC/TLM models to provide a cycle-accurate functional representations for simulation which could be seen as a drawback because of long simulation time and lack of formal semantics. The Artemis workbench [18] begins with Simulink representations that are transformed later to KPN models. These could be used to generate VHDL code for FPGA-based prototyping. The framework enables co-simulation to avoid long simulation when estimating performance numbers. It performs, like our approach, system-

level model calibration. Similarly, the Sesame environment [9] aims at early system performance analysis and design space exploration. This tool is more specific to multimedia applications and also uses co-simulation and model calibration techniques.

Other tools use formal methods for system level performance analysis like SymTA/S [10] or Real Time Calculus [19]. They basically rely on analytical techniques to determine latencies, worst-case scheduling scenario, buffer sizes, etc. This require adequate abstract models of the application software. Moreover, they allow only estimation of pessimistic quantities. MetaMoc [8] is another tool that uses formal methods but is more specific to Worst Case Execution Time (WCET) and schedulability analysis for hard real-time embedded software. It is based on UPPAAL [5] modeling and model checking, in addition to static analysis techniques. Our flow proposes an alternative based on rigorous, formal semantics model (BIP) plus scalable analysis techniques combining simulation and SMC.

Prior work has been done to generate code for Native Programming Layer (NPL) from BIP models in the context of the MPARM virtual platform [4]. Compared to MCAPI, the NPL library provides a minimal set of low-level communication mechanisms that only covers the clusters side (no host part). Moreover, it lacks standard definition and clear semantics likely to make it portable on different platforms. Among the most related examples covering code generation, Leung et al. [13] also consider KPN as a high-level specification model. In this work, the low-level target is the Message Passing Interface (MPI), a more sophisticated API with over 300 functions which make it heavier than MCAPI. Like in our case, the flow is based on correct-by-construction transformations. A code generation process based on graph transformations has been proposed in [14]. Here, the authors consider UML/OCL description as specification model. Despite their wide use in industry, this models lacks formal semantics which could be a hindrance towards formal analysis. This flow targets specifically CUDA/NPP primitives for image processing applications.

7 Conclusion

We presented a framework for rapid exploration of parallel applications on manycore platforms. The flow is centered on the BIP formal semantics which enables performance analysis using formal techniques and simulation, in addition to automatic code generation for different hardware platforms. In this paper we focus on STHORM as target which is a manycore and power efficient platform designed by STMicroelectronics. We implemented a runtime for STHORM following the MCAPI standard specification. The goal is to reduce STHORM programming complexity by providing homogeneous view of the whole platform and hide burden details for the designer.

Moreover, we presented a method to build faithful system models, that is, BIP models encompassing application software and hardware behavior. It consists of instrumenting the application software model and using the flow code generator to perform rapid execution on a concrete hardware platform to obtain performance measures. These are characterized as probability distributions and are used to build a stochastic BIP model of the whole software/hardware system. The latter is then used for exploration purposes. This paper mainly focuses on the simulation capabilities of the the flow. It is worth mentioning that the obtained stochastic BIP model enables also formal analysis techniques such as Statistical Model Checking (SMC) [21, 11]. As future work, we are planning to use this technique for quantitative analysis of performance. Indeed, BIP is equipped with an SMC engine called SBIP [6].

For the time being, the flow has been used for design space exploration for complex dataflow applications in the context of different research projects. As illustrated for HMAX Models, we used it for finding trade-offs between FIFOs sizes (especially with the platform memory scarceness), pipelining rates, and task deployment on multi-cluster platforms. Our analysis focus on timing constraints. We plan to extend them towards other classes of extra-functional constraints, such as energy or thermal constraints.

References

[1] http://www-verimag.imag.fr/BIP-Tools,93.html. 1, 2, 2

[2] www.multicore-association.org/workgroup/mcapi.php. 1

[3] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, May 2011. 1, 2

[4] Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, Mayur Maheshwari, and Joseph Sifakis. Component assemblies in the context of manycore. In *FMCO*, 2011. 1, 2, 5.3, 6

[5] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004. 6

[6] Saddek Bensalem, Marius Bozga, Benoît Delahaye, Cyrille Jégourel, Axel Legay, and Ayoub Nouri. Statistical model checking qos properties of systems with sbip. In *ISoLA (1)*, pages 327–341, 2012. 4.2, 7

[7] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009. 2

[8] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In *WCET*, pages 113–123, 2010. 6

[9] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP J. Emb. Sys.*, 2007, 2007. 6

[10] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. In *IEE Proceedings Computers and Digital Techniques*, 2005. 6

[11] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate Probabilistic Model Checking. In *VMCAI*, pages 73–84, January 2004. 7

[12] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. 1

[13] Man-Kit Leung, Isaac Liu, and Jia Zou. Code generation for process network models onto parallel architectures. (UCB/EECS-2008-139), 2008. 6

[14] Bo Li, János Sallai, Péter Völgyesi, and Ákos Lédeczi. Rapid prototyping of image processing workflows on massively parallel architectures. In *WISES*, 2012. 6

[15] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. DAC '12, New York, NY, USA, 2012. ACM. 1, 3.1

[16] Imed Moussa, Thierry Grellier, and Giang Nguyen. Exploring sw performance using soc transaction-level modeling. DATE, pages 20120–, 2003. 6

[17] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision*, 2008. 1, 5.1

[18] Andy D. Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *Int. J. Embedded Systems*, page 2005. 6

[19] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCA*, volume 4, pages 101 –104 vol.4, 2000. 6

[20] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD*, 2007. 2

[21] H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005. 7

8 Appendix

A Code Generation Steps

The code sample below illustrates the initialization phase of a generated process. Initially, MCAPI related variables are declared and initialized. Then, local and distant endpoints are created, connected and opened.

```
...
/*auxilliary MCAPI variables*/
size_t mcapi_received;
size_t mcapi_pipo;
void* mcapi_buffer;
mcapi_node_attributes_t mcapi_attribute;
mcapi_status_t mcapi_status;
mcapi_request_t mcapi_req;
mcapi_info_t mcapi_info;

/*Node and Domain id*/
process_map_t p_map = process_map[WORKER];
mcapi_domain_t domain_id = p_map.cluster_id;
mcapi_node_t node_id = p_map.core_id;

/*Get default attributes*/
mcapi_node_init_attributes(&mcapi_attribute,
&mcapi_status);
if(mcapi_status != MCAPI_SUCCESS)
ERR_RAISE("FAIL TO GET DEFAULT NODE ATTRIBUTES");

/*Initialize MCAPI for this node*/
mcapi_initialize(domain_id, node_id,
&mcapi_attribute, NULL, &mcapi_info, &mcapi_status);
if(mcapi_status != MCAPI_SUCCESS)
  ERR_RAISE("FAIL TO INIT MCAPI");
else MCAPI_TRACE_C("STARTING NODE");

/*Create local endpoints and set attributes*/
mcapi_endpoint_t endp_WORKER_read_data =
mcapi_endpoint_create(WORKER_read_data,
&mcapi_status);
if(mcapi_status != MCAPI_SUCCESS)
  ERR_RAISE("FAIL TO CREATE PORT");
else MCAPI_TRACE_C("PORT CREATED");
setEndPAttributes(endp_WORKER_read_data, 152,
MCAPI_ENDP_ATTR_LOCAL_MEMORY, MCAPI_TIMEOUT_INFINITE);

mcapi_endpoint_t endp_WORKER_write_data =
mcapi_endpoint_create(WORKER_write_data,
&mcapi_status);
if(mcapi_status != MCAPI_SUCCESS)
  ERR_RAISE("FAIL TO CREATE PORT");
else MCAPI_TRACE_C("PORT CREATED");
setEndPAttributes(endp_WORKER_write_data, 152,
MCAPI_ENDP_ATTR_LOCAL_MEMORY, MCAPI_TIMEOUT_INFINITE);


/*Create Distant endpoints if any
(only for send ports)*/
```

```
p_map = process_map[JOINER];
mcapi_endpoint_t endp_JOINER_read_data =
mcapi_endpoint_get(p_map.cluster_id, p_map.core_id,
JOINER_read_data, MCAPI_TIMEOUT_INFINITE,
&mcapi_status);
if(mcapi_status != MCAPI_SUCCESS)
  ERR_RAISE("FAIL TO GET DISTANT PORT");
...

/*Connect local endpoints with distant one if any*/
do {
  mcapi_pktchan_connect_i(endp_WORKER_write_data,
  endp_JOINER_read_data, &mcapi_req, &mcapi_status);
} while(mcapi_status == MCAPI_ERR_ATTR_INCOMPATIBLE);
if(mcapi_status != MCAPI_SUCCESS)
  ERR_RAISE("FAIL TO CONNECT PORTS");
if(!mcapi_wait(&mcapi_req, &mcapi_pipo,
MCAPI_TIMEOUT_INFINITE, &mcapi_status))
  ERR_RAISE("FAIL TO CONNECT PORTS");
...

/*Open local endpoints*/
mcapi_pktchan_recv_hndl_t h_WORKER_read_data;
mcapi_pktchan_recv_open_i(&h_WORKER_read_data,
endp_WORKER_read_data, &mcapi_req, &mcapi_status);
if(!mcapi_wait(&mcapi_req, &mcapi_pipo,
MCAPI_TIMEOUT_INFINITE, &mcapi_status))
  ERR_RAISE("FAIL TO OPEN CONNECTION");
mcapi_pktchan_send_hndl_t h_WORKER_write_data;
mcapi_pktchan_send_open_i(&h_WORKER_write_data,
endp_WORKER_write_data, &mcapi_req, &mcapi_status);
if(!mcapi_wait(&mcapi_req, &mcapi_pipo,
MCAPI_TIMEOUT_INFINITE, &mcapi_status))
  ERR_RAISE("FAIL TO OPEN CONNECTION");
...
```

B HMAX Models Case study

The figures below show examples of input and output images of HMAX. Figure 16 is an input image of 118×118 resolution. The corresponding output images are illustrated in Figure 17 which shows the three computed directions.

Figure 16: Grayscale input image of resolution 118×118.

(a) Direction 1 (b) Direction 2 (c) Direction 3

Figure 17: A sample of output images (3 directions).