Simulating Run-Time Task Migration in Many-Core Systems

Koosha Ahmadi

A thesis in fulfilment of the requirements for the degree of

Masters by Research

THE UNIVERSITY OF NEW SOUTH WALES

SYDNEY · AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering

September 2014

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Ahmadi

First name: Koosha Other name/s:

Abbreviation for degree as given in the University calendar: Msc

School: Computer Science and Engineering Faculty: Engineering

Title: Simulating run-time task migration in many-core systems

Abstract 350 words maximum: (PLEASE TYPE)

With the ever increasing number of processing elements on Network-on-Chip multiprocessors, it is crucial to utilise the resources efficiently. Allocation of tasks to processing elements is one of the most important problems to be solved for these systems. This requirement is further complicated as embedded systems become evermore open ended.

In addition, as the requirements and the volume of applications are hard to predict at design time, not only do we need to allocate resources at runtime, but we may want to reallocate them as well. Having frameworks for testing different allocation and reallocation policies is therefore of great importance. However, simulators that are readily available for performing task migration and dynamic mapping in the context of open-ended embedded systems are scarce or hard to configure.

We propose a framework for testing different allocation and reallocation policies for 2D-mesh many-core systems, where requirements of applications are not known a priori. Our simulator is based on the Booksim2 network simulator, and we have used Synchronous Data Flow Graphs provided by the SDF^3 tool as our application model. The presented simulator is extensible and provides useful insights into on-chip communications and runtime behaviour of applications. It is therefore to be hoped that it could be used to explore and develop task migration policies.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

Koosha Ahmadi	Oliver Diessel	22/09/2014
Signature	Witness	Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ……Koosha Ahmadi…….....

Date ……22/09/2014…….....

COPYRIGHT STATEMENT

‘I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.
I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed ……Koosha Ahmadi…….....

Date ……20/02/2015…….....

AUTHENTICITY STATEMENT

‘I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.’

Signed ……Koosha Ahmadi…….....

Date ……20/02/2015…….....

Abstract

With the ever increasing number of processing elements on Network-on-Chip multiprocessors, it is crucial to utilise the resources efficiently. Allocation of tasks to processing elements is one of the most important problems to be solved for these systems. This requirement is further complicated as embedded systems become evermore open ended.

In addition, as the requirements and the volume of applications are hard to predict at design time, not only do we need to allocate resources at runtime, but we may want to reallocate them as well. Having frameworks for testing different allocation and reallocation policies is therefore of great importance. However, simulators that are readily available for performing task migration and dynamic mapping in the context of open-ended embedded systems are scarce or hard to configure.

We propose a framework for testing different allocation and reallocation policies for 2D-mesh many-core systems, where requirements of applications are not known a priori. Our simulator is based on the Booksim2 network simulator, and we have used Synchronous Data Flow Graphs provided by the SDF^3 tool as our application model. The presented simulator is extensible and provides useful insights into on-chip communications and runtime behaviour of applications. It is therefore to be hoped that it could be used to explore and develop task migration policies.

Acknowledgements

First and foremost, I would like to thank my supervisor Oliver Diessel for his guidance and patience throughout my Master's.

I would like to acknowledge Dr. Muhammad Shafique and Prof. Jörg Henkel from Karlsruhe Institute of Technology (KIT), Germany as well as Prof. Akash Kumar and Amit Kumar Singh from National University of Singapore (NUS) for providing me with the SDFs for applications JPEG decoder, MPEG4 decoder, and H.264 decoder and encoder.

I express my warm thanks to Nan Jiang and Sander Stuijk for their help with Booksim2 and SDF^3.

Lastly, a special thanks to my family and friends for their support and kindness throughout my studies.

Contents

List of Acronyms

BDF Boolean Data Flow

CSDF Cyclo-Static Data Flow

IQ Input Queued

KPN Kahn Process Network

MoC Model of Computation

NoC Network-on-Chip

OS Operating System

PE Processing Element

SDF Synchronous Data Flow

SDFG Synchronous Data Flow Graph

VC Virtual Channel

List of Figures

List of Tables

Chapter 1

Introduction

The architectural complexity, excessive power consumption, effective utilisation of available silicon, and stagnation in raising the maximum clock frequency of processors has led to the emergence of architectures with numerous cores integrated onto a single chip. These architectures improve performance by offering parallel processing solutions. With the growing number of transistors on a single chip, having thousands of cores on a single chip is feasible [13]. Many-core systems usually consist of a regular grid or a mesh of cores with an underlying Network-on-Chip (NoC) for communication and a range of memory hierarchies. They can be homogeneous, meaning that all the cores share the same architecture, or heterogeneous, meaning that the cores' instruction set architectures are different.

A number of many-core processors are already available. Intel's Single Chip Cloud computer [34], consists of 4 clusters with 6 tiles each. Each tile contains two processors and each cluster is connected to a memory controller. Tilera's Tile-GX family [2] is a series of many-core processors with a mesh of 16 to 64 cores. Kalray's MPPA many-core chips [1] integrate up to 1024 cores onto a single chip. The chips are divided into clusters of 16 cores with shared memory. All the chips mentioned above use NoCs for communication.

The vast resources offered by these systems require effective and scalable resource management approaches. Application mapping has been identified as one of the most important problems for designing NoCs and on-chip multi-processors [46, 48]. Application mapping is the process of allocating the set of concurrently active tasks

of an application to a suitable set of processors in a mesh with a specific optimisation goal e.g. execution time, network throughput, power consumption, thermal profile, etc. The task to core mapping problem is a special case of the Quadratic Assignment Problem[1], which is a well known NP-hard problem [56, 39]. Hence, there is no fast optimal solution for this problem in general, and heuristics for runtime solutions or exhaustive algorithms for design-time solutions are used for solving the mapping problem.

Mapping methods could target homogeneous or heterogeneous architectures. A resource manager carries out mapping and scheduling, i.e policies for sharing resources between applications in order to satisfy the temporal and resource requirements of each application. These managers are either centralised or distributed. Centralised managers are a class of controllers where the manager has a global view of the system which leads to better mapping decisions. However, as the number of processing elements integrated onto a chip grows, scalable and fault-tolerant managers are required. Another drawback of using centralised managers is that they introduce a single point of failure, thus if the central module stops working the whole system comes to a halt. On the other hand, distributed resource managers spread the resource management task among the processors and localise decision making. Consequently, they are capable of being more scalable and they can reduce the traffic that is caused by control messages. However, due to the lack of a global view of the system, mapping decisions are more likely to be suboptimal.

Depending on the time when the mapping of task to cores is performed and optimised, mapping methodologies are classified into three categories namely: static, runtime and hybrid.

Static mapping methodologies target small to medium-sized embedded systems where the application set is known a priori and optimal mappings can be realised offline. However, with the growing number of cores and applications it is not feasible to use these methods. For example, A typical digital TV may run 60 applications simultaneously which means that 2^{60} different scenarios can occur [9]. That is a pro-

[1] For N locations and N weighted facilities, the Quadratic Assignment Problem is the problem of deploying facilities at chosen locations such that the sum of the product of the distances and the weights of the facilities is minimised.

hibitively large design space for even off-line approaches . Moreover, these methods cannot cope with dynamism in the system or the applications e.g faults occurring at runtime, applications with dynamic workloads, addition of new applications at runtime, etc.

On the other hand, runtime methods realise task to core mappings at runtime. Contrary to static methods, where the application set is known a priori and exhaustive algorithms can run for a long time, with these methods the mapping time is a design factor and they rely on lightweight heuristics to speed up the mapping process. The temporal and computational constraints on these methods render them suboptimal. However, they can cope with dynamism at runtime by attempting to make adequate mapping decisions according to the latest state of the system.

Another class of mapping method couples the off-line profiling of applications and study of the design space with the dynamism of runtime mapping methods. In these so-called hybrid methods, different scenarios for applications are studied at design time and various mappings are devised for each scenario. At runtime, based on the latest state of the system the mapping that is most suitable for the current situation is chosen.

Task migration can be used with both runtime and hybrid methods to improve the initial mapping by observing the current state of the system and changing the mapping in order to optimise a certain criterion. However, migrating tasks introduces new costs and requires additional hardware and software support. Consequently, task migration should be used judiciously.

Studying effective mapping and task migration policies requires simulators that simulate many-core architectures that take into account realistic costs of implementing task allocation and migration. Simulators that are readily available for the study of task migration policies are scarce. We therefore introduce a many-core simulator for the study of task placement and migration methods that employs a dataflow application model and a cycle-accurate NoC simulator. An overview of the thesis is provided below.

Chapter 2 reviews the literature on mapping methods that employ task migration, on task migration implementations and application models, as well as surveying

the state-of-the-art regarding NoC-based many-core simulators. Chapter 3 describes the design of our simulator, while Chapter 4 verifies the usefulness of our simulator via a sequence of application mapping, execution and migration experiments. Finally, Chapter 5 concludes the thesis with a summary of the main accomplishments and directions for future work.

Chapter 2

Background

2.1 Mapping and task migration

Traditionally, embedded systems were small-scale systems in which the application set was known before operation and the system designer could optimise the system for the known application set. Hence, the majority of the work in the mapping literature focused on the design of custom architectures and static mappings in which optimal solutions are realised for small-scale systems. However, with the emergence of many-core processors and the explosion in the variety and complexity of embedded applications, it has become necessary to employ runtime methods to cope with the dynamism of these systems.

Numerous runtime mappings methods have been described in the literature. These methods map the application on the fly, meaning that they use heuristics for placing the application sub-tasks as they are needed at runtime. Works reported in [19, 22, 57, 64, 24, 5] use on-the-fly mapping methods. These methods consist of an initial phase during which the initial tasks of an application are mapped. These methods in some cases divide the system into multiple clusters [57, 24, 64, 5] and carry out the initial mapping by choosing a suitable cluster for each application according to size and resource requirements of the application. The work in [22] does not use a clustering method, instead it tries to form convex regions where applications are mapped contiguously. After the initial phase, the remaining tasks of an application are mapped to the system according to a heuristic with a certain

optimisation criterion. The works described in [57, 19] map the communicating tasks closer to one another to reduce the traffic congestion. The work reported in [22] aims to reduce energy consumption by mapping the remaining tasks of the application to the convex region occupied by the initial task. The works reported in [64, 5] aim at reducing the traffic load and the work in [24] employs a greedy algorithm in order to reduce the power consumption.

However, our focus is primarily on runtime mapping methods that employ task migration to improve the mapping at runtime. Task migration adds adaptive capabilities to the mapping approach that allow task mappings to change according to the latest state of the system. Table 2.1 lists state-of-the-art runtime mapping methods that use task migration. A small number of works reported in the runtime mapping literature employ task migration as well, and some of these works do not assign a cost to task migration. This means that it is more difficult to assess their benefits. All of the works listed in Table 2.1 use custom simulators. A general purpose simulator, that is readily available for experimenting with task migration and that allows the cost of task migration to be considered, could help researchers focus on this aspect of runtime mapping.

System throughput, i.e. the number of messages delivered over an interval, is an important optimisation criterion for multimedia and network applications. However, it is only considered in a few runtime mapping algorithms that incorporate task migration [67, 37]. Our proposed simulator offers a tool for studying the effect of runtime mapping methods and task migration by focusing on throughput and execution time for many-core system on a chip.

2.2 Task migration implementation

Task migration has been traditionally used in distributed systems in which a networked set of machines communicate through message passing. Task migration is primarily used in these systems to distribute the workload between different processors (load balancing), moving the task away from overloaded or faulty cores, or easing access to data [50, 58]. Task migration involves interrupting the task on a pro-

Reference	Optimisation criteria	Cost considered
Anagnostopoulos et al. [6]	Execution time, hop distance	No
Brião et al. [15]	Execution time, energy	Yes
Kobbe et al. [40]	Execution time, traffic	No
Jahn et al. [37]	Throughput	Yes
Zipf et al. [67]	Throughput	No
Nollet et al. [53]	Mapping time	Yes
Goodarzi et al. [32]	Communication latency, power consumption	Yes
Goh et al. [31]	Execution time, migration latency	Yes

Table 2.1: State-of-the-art runtime mappings with task migration

cessor, saving the program context, moving the context to the destination processor and restarting the program at the new location. The amount and type of program context that needs to be migrated is closely related to the processor architecture, the Operating System (OS) and the memory hierarchy. It consists of the instruction code, the stack memory, the heap memory (for architectures with dynamic memory support), the processor registers and the OS context e.g. I/O requests. However, implementing task migration in on-chip multi-processors and many-core systems has different implications, since these systems have more constraints on scheduling and power consumption, and have a lower latency between the cores, but a more restricted communication bandwidth. Therefore, we cannot directly apply the task migration methods used in the distributed computing domain to on-chip, many-core systems.

In shared memory systems, where cores share the same physical memory, there is no need to physically move instruction code and data memory, whereas in distributed memory systems, where each core has its own local memory, the instruction code and data memory need to be sent over the network. Furthermore, moving tasks in heterogeneous systems where different cores have different instruction sets, requires a step during which the instruction code is transcoded for the new processor [36].

A task migration method for bus-based, distributed memory, on-chip multi-

processor architectures was proposed in [10]. This work utilises user checkpointing, where the programmer explicitly defines safe points for task migration and the user also defines the necessary context that needs to be moved. The OS context is flushed and the task migration process is coordinated by a central controller. The work in [7] keeps replicas of all the tasks on each processor and targets deeply embedded operating systems i.e. reliable, low cost systems with pre-determined sets of tasks that are usually incapable of dynamically loading new tasks.

The work in [36] offers a task migration mechanism for systems with fully distributed memory, where each core runs an OS kernel in its local memory. They claim that heap memory constitutes the bulk of the the program context and they aim at mitigating the task migration cost by transferring the heap memory in a manner that reduces page faults on the destination processor.

The work in [33] offers a task migration methodology for a distributed kernel OS for many-core systems similar to OSs like Barrelfish [8], Corey [14], fos [65] and Helios [52]. In these OSs, system calls are directed at the local kernel for each core rather than at a single, large kernel. Hence they are scalable to many-core systems. Shared memory is used in this work to store the program code, stack memory, heap memory and inter-core messages. The architecture they use is based on Intel's Single Chip Cloud computer. For migrating the tasks, the memory space pointers of the task need to be translated for the new core and the context itself is not physically moved.

Two task migration approaches with message consistency are reported in [53]. The work in [62] offers a task migration checkpointing method, similar to the one in [53], that is based on the principle that pipeline applications (synchronous data flows in this case) continuously visit points in execution where they are processing data that is not dependant on previous data and hence the state of the task is minimal at these points. Consequently these points are more efficiently used as checkpoints for task migration.

Many-core processors are more likely to have distributed memory architectures, as shared-memory architectures are not scalable. Consequently, to migrate a task, the state of the task needs to be migrated via the network. Hence, we follow the

same trend and assume that the state of each task, as described by our benchmarks, needs to be explicitly migrated via the network. Our task migration method is described in Chapter 3.

2.3 Application models

Network-on-chip (NoC) many-core systems consist of multiple processing elements and communication modules. Hence, it is desirable to use application models that model concurrency at the task level and separate communication from computation. Furthermore, to gain meaningful insights at system level, the enormous complexity of applications needs to be abstracted during modelling. At system level, it is desirable to study the impact of different architectural choices at a faster pace. A suitable application model should be able to model a variety of applications, it should also be easy to use, be amenable to analysis, and be compatible with the target architecture.

Numerous concurrent application models exist in the literature. These models either have precise underlying Models of Computation (MoC), or they are extensions of sequential programming languages or hardware descriptive languages without any specific underlying MoC (SystemC and Bluespec System Verilog) [23]. Different concurrent computation models are studied in [44] and an overview of general models of computation can be found in [23]. Among the models studied in the mentioned works, the dataflow MoC is widely used for modelling streaming and multimedia applications for multi-processors. The dataflow MoC separates computation from communication. Tasks communicate by sending asynchronous data messages called tokens. In the following we introduce the dataflow MoC followed by a review of application models used in the runtime mapping literature and finally our choice of application model.

2.3.1 Dataflow MoCs

Dataflow is a term used for a concurrent model of application specification that consists of concurrent processes or tasks communicating through FIFOs and corresponds to the concurrent nature of many-core systems on a chip. These models

express concurrency explicitly. DSP and multimedia applications have long been associated with dataflow MoCs. It is common to use graphical illustrations for these models. Applications are depicted as graphs where nodes represent processes while edges correspond to FIFOs (communication channels). In the following, three major dataflow concurrent models, namely Kahn process network, Dennis dataflow and synchronous dataflow are presented:

Kahn Process Network (KPN) [30] consists of a series of sequential processes that communicate through unbounded FIFOs. Processes produce tokens that are sent through the FIFOs, which in turn are consumed by the receiver process. Writing on FIFOs is non-blocking but reading is blocking. KPN is a deterministic model, meaning that for a given set of inputs, outputs are independent from process scheduling. Deadlocks may happen in these models and they terminate in case of a global deadlock. Furthermore, the behaviour of processes is data-dependent and as a result, a self-timed FIFO-based dynamic scheduling is usually used for these models. Using FIFOs means that there is no need for a shared memory and hence the model is suited to a distributed memory architecture. Issues such as developing runtime scheduling policies for execution of the tasks and allocating the buffer spaces for FIFOs at runtime, add to the complexity of implementing KPNs.

Dennis Dataflow [26] or dataflow process networks execute a series of firings. Instead of having a blocking read like KPN, Dennis dataflow actors have a set of firing rules, that once they are satisfied, cause the act of firing (execution of process) to occur. Firing rules determine the number of tokens that should be present on an edge before they can be consumed. The firing may also change the internal state of the actor which affects the future firing of that actor. It is undecidable whether a Dennis dataflow application can be statically scheduled. Consequently a central runtime scheduler is required, and this adds to the complexity of implementing this model [44, 16].

Synchronous Data Flow (SDF) [43] is similar to Dennis dataflow but the firing rules are more restricted. The number of tokens that is required by each process to fire is predefined and fixed. Consequently, it is possible to statically schedule this model and in some cases buffer requirements can be set statically as well (in runtime

scenarios it is not always possible to guarantee buffer requirements, as resources are limited and system has no a priori knowledge of the incoming applications' requirements). On the other hand, the predefined rates limit the expressiveness of these models as they are not able to model dynamic workloads. In addition, data access patterns and conditional execution cannot be modelled using SDFs. The **Cyclo-Static Data Flow (CSDF)** [11] allows for a finite sequence of rates rather than fixed rates and the **Boolean Data Flow (BDF)** [42] supports data-dependant conditional behaviour.

KPN, Dennis dataflow and SDF are all widely used to represent DSP and multimedia applications. They define applications at a high level of abstraction, they are explicitly parallel, they separate computation from communication, and they are suited to distributed memory architectures. All of which makes these models suitable application model candidates for on-chip multi-processors with distributed memories. However, these models have their shortcomings. There is no mechanism to express iterative behaviour and it is difficult to model shared resource access conflicts e.g. off-chip memory access. These MoCs are data driven and as a result they are not able to manifest the control flow (contrary to finite state machines), that is desirable for a range of embedded applications.

2.3.2 Application models used in the mapping literature

A considerable proportion of the studies on mapping and task migration use acyclic task graphs as their application models where the nodes represent processes and the arcs model communication. Each node is usually associated with a deadline. Works reported in [57, 5, 24, 64, 67, 19, 22, 35] use acyclic task graphs. TGFF [27] and TGG [3] are the main tools used for generating random acyclic task graphs.

Task graphs with SDF and KPN as the underlying MoC are used in some hybrid mapping works [25, 51, 55].

The task migration methods in [37, 40, 6] use the so-called malleable application model, where the application expands or retracts its resource usage according to the availability of resources.

Despite their limitations, dataflow task graphs are widely used in the mapping and scheduling studies. SDF and KPN task graphs are more expressive than the acyclic graphs that are generated by the TGFF tool. While KPNs are better for capturing runtime dynamic behaviour of applications, the unbounded buffer models and the need for complex runtime schedulers and lack of benchmarks, make them infeasible for our work. Therefore, we have chosen the more restricted SDF model as our application model, as they are widely used in the literature, they can be run with bounded memory and benchmarks are available for this model.

2.4 Simulators

A variety of multi-processor system-on-chip and many-core simulators are available in the literature. These simulators range from NoC simulators to full-system simulators. An ideal simulator for studying dynamic mapping in the context of many-core processors should be fast and provide interfaces for runtime mapping and task migration.

Table 2.2 lists state-of-the-art, on-chip multi-processor simulators found in the literature. gem5 [12] and GEMS [47] are cycle-accurate, full-system simulators. These simulators, model the hardware and the complete software stack (application and system software) in great detail, and as a result the simulations are slow. Moreover, the effort of adding runtime mapping and task migration support to the operating systems of these simulators is non-trivial. These systems are capable of simulating many-core processors, however as the number of cores rises, the already slow speed of these simulators becomes an even bigger obstacle.

Graphite [49], Sniper [18] and Hornet [45], use multi-threaded simulations to speed up the simulation time. They execute at the application abstraction level, meaning that the complete software stack is not modelled. Task migration and runtime mapping is not supported in these simulators. MAPS [20] simulates dataflow applications and supports dynamic scheduling and task migration for small-scale bus-based multi-processor system on a chip. MCoreSim [41] is a many-core simulator built on top of OMNeT++ [63] that focuses on the communication overheads

Simulator	**Abstraction level**	**Runtime mapping**	**Task migration**	**Many-core**	**MoCs/programming languages supported**	**Communication model**	**Processing/processor model**
gem5 [12]	Full System	No	No	Yes	SLICC	Based on GEMS	Alpha, ARM, MIPS, Power, SPARC, and x86
GEMS [47]	Full System	No	No	Yes	SLICC	Any point to point connection topology	SPARC v9
Hornet [45]	Application	No	No	Yes	Programs compiled for MIPS or x86 executables for Pin based execution	NoC with custom architecture/topology	MIPS
Hemps [17]	Application	Yes	No	No	KPN	2D mesh NoC (up to 64 cores)	Plasma
Graphite [49]	Application	No	No	Yes	pthread	Analytical NoC	Pin supported processors
Sniper	Application	No	No	Yes	pthread	Analytical NoC (based on Graphite)	IntelXeon X7460
MAPS [20]	Application	Yes	Yes	No	KPNs described with CPN	Bus based	Executes KPN traces and simulates execution delays
MCoreSim [41]	Application	No	No	Yes	Custom application model	Based on OMNet++, supports mesh and torus topology	Simulates the delay of processing instructions
Booksim2 [38]	NoC	N/A	N/A	N/A	N/A	NoC with custom architecture/topology	N/A
Garnet [4]	NoC	N/A	N/A	N/A	N/A	NoC with custom architecture/topology	N/A

Table 2.2: State-of-the-art simulators

of heterogeneous many cores; runtime mapping and task migration are not implemented. Hemps [17] is a multi-processor system-on-chip simulator, implemented using VHDL and SystemC. Hemps can simulate up to 64 cores and uses KPN MoC for its application model. Task migration and many-core architectures are not supported in Hemps.

Booksim2 [38] and Garnet [4] are NoC simulators. They use synthetic traffic patterns or communication traces to simulate custom NoCs.

As none of these simulators present mechanisms for task migration for many-core systems, there is a need for a simulator that offers the option to integrate task migration with realistic costs into runtime mapping methods. Such a facility would allow migration strategies to be developed and would support the testing of alternative task migration algorithms.

2.5 Summary

Task migration can be used to improve task-to-core mappings at runtime. However, most of the runtime mapping methods reported in the literature do not employ task migration. Hence, there is room for more focus and study on task migration methods for many-core processors. Furthermore, a majority of the task migration works use custom-built simulators to test their algorithms . While state-of-the-art simulators are detailed and accurate in their own respective fields, there are no simulators that are readily available for mapping dataflow graphs at runtime and there is no support for task migration. Use of task graphs in the mapping literature is prevalent and they model the application at an acceptable level of abstraction for studying the result of task migration on system throughput and application execution time. The SDF application model is already used for hybrid mapping methods that use task migration as well in [25, 62] and is adequate for our work. We therefore, propose developing a simulator that maps SDFGs at runtime and supports task migration.

Chapter 3

Simulator

3.1 Overview

We aim to study task migration algorithms for many-core chips with **Network-on-Chips** (NoCs) as the underlying communication infrastructure, assuming the application set is not known a-priori.

Runtime mappings are suboptimal as they rely on fast heuristics. Consequently, issues such as fragmentation and contention for resources arise. It is known that re-mapping tasks (migrating tasks) to other processors at runtime reduces fragmentation [31, 32] and can help improve the application throughput [37, 67]. Our simulator is intended to provide a platform for studying the impact of this technique, e.g. alternative defragmentation algorithms, on flit throughput and application execution time.

We need to find a proper level of abstraction for modelling different components of the system so we can study the benefits and cost of different mappings and remapping strategies with fast simulations. We have chosen an application model that explicitly models concurrency and that also captures inter-task communication at sufficient detail so as to provide an accurate estimation of traffic in the system. In addition, our NoC model is detailed enough to be able to model contention for shared architectural resources. In the following, we introduce our application model and also the architecture of our target system.

3.2 Application model

3.2.1 Synchronous Data Flow Graphs

Synchronous Data Flow Graphs (SDFGs) are the schematic representations of the **Synchronous Data Flow** (SDF) application model. SDFGs explicitly model concurrency and separate computation and communication. Assuming that an application consists of a set of communicating tasks, the nodes of an SDFG represent the computation of tasks, while the directed edges of the graph model inter-task communication. Nodes are called actors and arcs are called channels. An SDFG is formally defined below:

Definition 1 *(SDFG) SDFGs consist of a set of actors and dependency channels* (A, D) *where* A *denotes the finite set of actors and* $D \subseteq A^2 \times \mathbb{N}^2$ *is the finite set of dependency edges.* $d = (a, b, p, q)$ *represents the dependency of actor* b *on actor* $\boldsymbol{a}$*. When actor* a *fires* p *tokens are put on* d *and when* b *fires,* q *tokens are consumed by* b*. Edges may contain initial tokens*

Each actor $a \in A$ may have multiple input and output ports. Channels connect different ports of different actors and actors communicate via tokens. Each port has a predetermined fixed rate of firing, which is defined as the number of tokens a port consumes or produces when it is executing. Execution of actors is called firing. In order for an actor to fire, it must gather enough tokens on its input ports (as specified by the input rates). When sufficient input tokens are present, the actor fires and produces tokens on its output ports. Firings are atomic and cannot be interrupted in the SDF application model. However, we allow the firings to be delayed in our use of the model in case there is not enough space for writing the output tokens on the local memory of the processor.

SDFGs may be used to model pipelined streams or cyclic dependencies between actors (tasks). Furthermore, as there are fixed rates on the ports, SDFGs can be scheduled statically to be executed with time constraints and bounded memory. SDFGs are assumed to be constantly active and repeat the same function over an infinite stream of inputs.

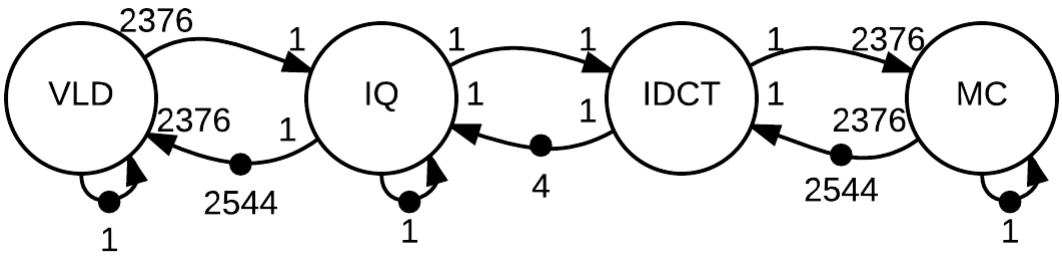


Figure 3.1: H.263 Decoder [28].

SDFG actors represent blocks of code in an application. Methods to extract SDFGs from application codes are explained in [60].

Each actor is assigned an internal state. The internal state of a task refers to the sum of the worst-case stack size, instruction size, and data size [60] .

An SDFG models the global variables used during the execution of an actor, i.e. the data required for subsequent firings that is not stored as internal state, by self-loops. Self-loops have one initial token with the size of the global data required by the actor. In addition to modelling global data usage, SDFGs use self-loops to limit auto-concurrency. In this case, initial tokens indicate the number of instances of an actor that can be active during the execution of an application.

Figure 3.1 depicts the SDFG model of an H.263 decoder as described by [28]. An H.263 decoder is a video decoder used in video conferencing. It consists of four actors (tasks) : Variable Length Decoding (VLD), Inverse Quantiser (IQ), Inverse Discrete Cosine Transformer ($IDCT$) and Motion Compensation (MC). VLD is the initial decoding task. A video frame is decoded once MC consumes all its input tokens. The numbers on the ends of the edges denote rates e.g. VLD produces 2376 tokens each time it fires. Each token in this example represents a block of 64 pixels worth of data. The self loops represent data that need to be conserved for the next round of firing. The black dots denote the availability of initial tokens on each channel.

At the start of execution, VLD is the only actor that has enough input tokens (one initial token on one input and 2544 on the other one). VLD fires and produces 2376 tokens on the channel to IQ and one token on its self loop channel. Now IQ has enough tokens to fire; as a result of one firing of VLD, IQ can fire 2376 times,

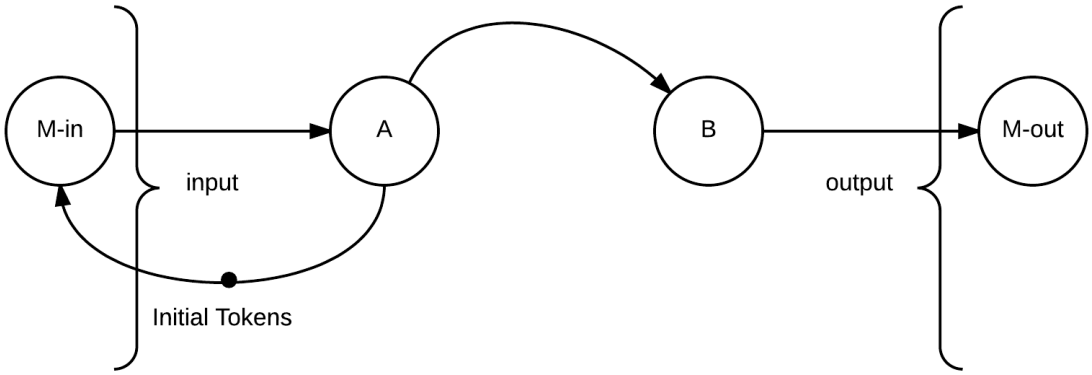


Figure 3.2: Modelling memory access for the SDFGs

consuming the tokens on the *VLD* to *IQ* channel and producing tokens on its output edges. Each time *IQ* fires, *IDCT* gets enough tokens on its inputs to fire as well (one token from *IQ* and 2544 initial tokens from *MC*). After 2376 firings of *IQ* and *IDCT*, *MC* fires once. This cycle of firings will go on indefinitely.

In order to put a bound on the buffer requirements of edges for an SDFG (A, D), for each edge $(a, b, p, q) \in D$, we add a back edge $(b, a, q_\delta, p_\delta)$ from actor b to actor a where $Rate(p) = Rate(p_\delta)$ and $Rate(q) = Rate(q_\delta)$ [60]. The buffer space is set by the number of initial tokens on the $(b, a, q_\delta, p_\delta)$ edge. For example, in Figure 3.1 the edge from *IQ* to *VLD*, represents a buffer space of 2544 tokens as indicated by initial tokens on the link.

3.2.2 Modelling memory access

As SDFGs do not conventionally model input and output, we have extended the graph model to include I/O access by adding dummy actors.

Figure 3.2 depicts an SDFG with memory actors. The *M-in* actor models off-chip memory reads, and the *M-out* actor models off-chip memory writes. In order to read from memory, the actor sends a request to the off-chip memory through the memory controller. This request is modelled by the edge from the actor A to the actor *M-in*. For writing to memory, there is no need to send a request, hence there is only a need for a channel to send tokens to off-chip memory.

3.2.3 Modelling transient applications

There are two categories of application graphs, namely, periodic applications with infinite input streams and transient applications with bounded amount of input data, which bounds the number of execution iterations. In order to model transient applications we use two properties of SDFGs, namely the so-called repetition vector and its iteration:

Definition 2 *(Repetition Vector) A repetition vector rpt of an SDFG* (A, D) *is a function* $A \rightarrow \mathbb{N}$ *such that for each edge* $(a, b, p, q) \in D$ *,* $Rate(p) \cdot rpt(a) = Rate(q) \cdot rpt(b)$*. A repetition vector a is called non-trivial if and only if* $rpt(a) > 0$ *for all* $a \in A$ *[60].*

Definition 3 *(Iteration) Assume SDFG (A,D) has repetition vector rpt. An iteration is a set of actor firings such that for each* $a \in A$*, the set contains* $rpt(a)$ *firings of a [29].*

We model the bound on the input data by setting a limit on the number of iterations of an application. Once an application reaches the iteration limit, it stops executing. Since an SDF processes a meaningful block of data during each transition, stopping the execution before the end of a transition leaves a portion of the input data unprocessed.

3.2.4 SDF^3

The SDF^3 tool [59] generates random, timed SDFGs in the shape of chains, acyclic graphs, and cyclic graphs. In timed SDFGs, each actor has a fixed execution time that defines how many processor cycles it takes for an actor to complete its firing on a specific processor type. A number of SDFG representations of real streaming applications like H.263 encoder and decoder, MP3 decoder, MP3 playback, etc. can be found in [61]. The applications we have used in our study that are not on the SDF^3 website are taken from [56].[1] SDF^3 graphs are in XML format. Actors and

[1] I would like to acknowledge Dr. Muhammad Shafique and Prof. Jörg Henkel from Karlsruhe Institute of Technology (KIT), Germany as well as Prof. Akash Kumar and Amit Kumar Singh

channels in SDF^3 are described in these models. Token sizes are defined for each channel in bits. For each actor, the actor's context is given in bits.

SDF^3 offers a range of tools for offline profiling of SDFGs. We used SDF^3 to compute repetition vectors, set buffer sizes and also for creating synthetic applications.

3.3 Architecture

Our goal is to simulate and measure network metrics and application execution times running on a many-core system while tasks are being allocated and reallocated at runtime. As our application model abstracts away the details of instruction execution and provides estimates of the execution time of an actor per firing, modelling communication requirements and the latency of executing applications as a result of communication delays is our main concern. We therefore decided to modify a cycle-accurate simulator that provides sufficient accuracy to model network congestion and contention.

Our system assumes a 2-D mesh on a chip multiprocessor with an NoC as the underlying communication infrastructure. Figure 3.3 depicts a 2×2 model of our architecture. Tiles can be **Processing Element** (PE) tiles or memory controller tiles. Each tile is connected to a router. A memory controller is connected to the off-chip memory via off-chip links. We have assumed a distributed memory architecture and that PEs communicate via message passing. Our architecture is homogeneous, meaning that all the processing elements have the same architecture and use the same instruction set. In addition, we assume that there is a global software entity called the Central Controller in charge of resource management.

The architecture of a PE tile is depicted in Figure 3.4. The tile consists of a processor, local memory, and a network interface. The network interface consists of a Packetiser-Depacketiser unit and two buffers, one for injecting packets to the network and one for ejecting packets from the network. PEs model execution of SDFG actors. Since PEs have limited capacity and we have sufficient PEs to distribute the tasks

from National University of Singapore (NUS) for providing me with the SDFGs for the applications JPEG decoder, MPEG4 decoder, and H.264 decoder and encoder.

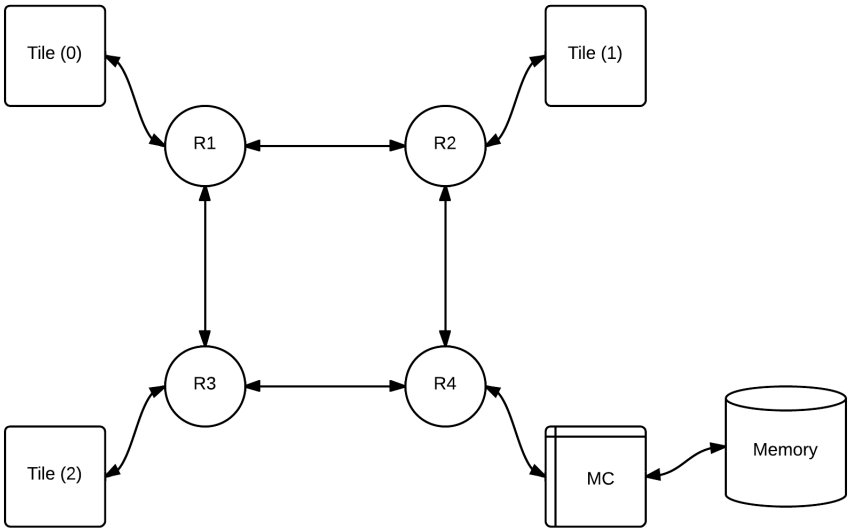


Figure 3.3: NoC Architecture

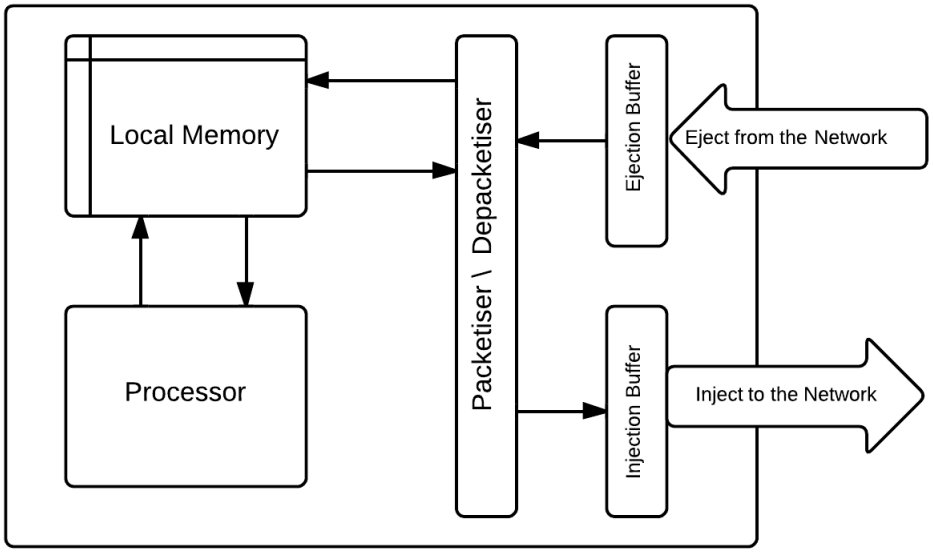


Figure 3.4: Tile architecture

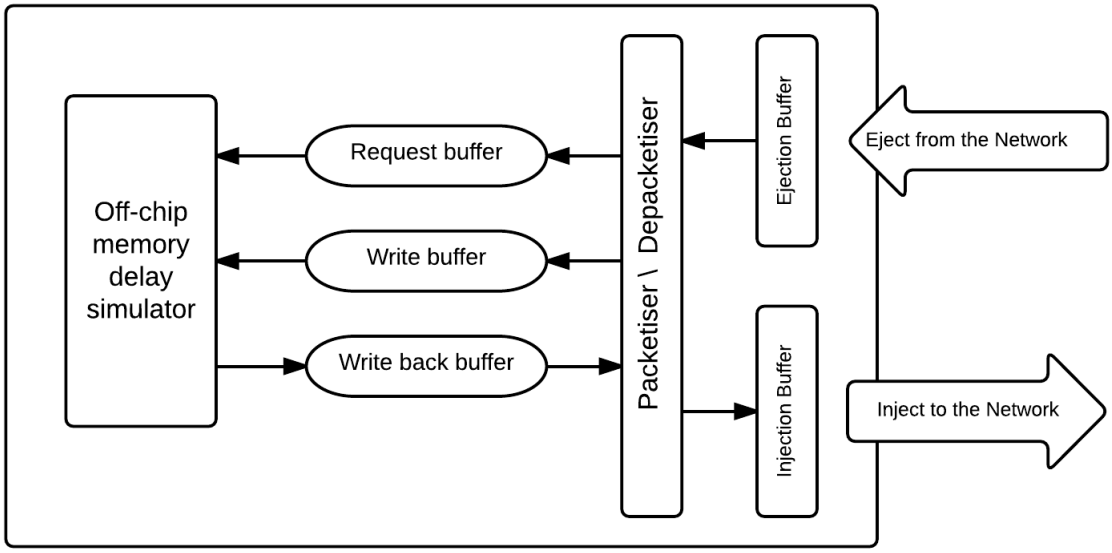


Figure 3.5: Memory controller architecture

in a many core system, we assume that they run a single task at a time. Each PE is equipped with a local memory of size M bits. The Ejection Buffer's size is $V \times P$. Where V denotes the number of **Virtual Channels** (VCs) and P the maximum packet size. The Injection Buffer has to capacity to store one packet.

Routers have input and output ports to the north, south, east and west, plus a local link to the connected PE. We assume X/Y routing and wormhole switching are used for flow control.

There are two pairs of links connecting each component to its neighbours, a pair of data and credit links in one direction, and a complementary pair in the opposite direction. Credit links carry credits for flow control while data links carry data flits. Data link capacity is determined by the flit size. Each data link can carry a single flit during each cycle.

Flits have constant sizes. Packets range in size from three flits to a maximum that is set by the user. For our simulations we set the flit size to be 32 bits and the maximum packet size was set to 10 flits (8 flits for data and 2 for head and tail flits).

All off-chip memory requests are directed at the memory controller. The memory controller in turn sends the requests to the off-chip memory. Figure 3.5 depicts the architecture of the memory controller. The request buffer of the memory controller

is $N \times RP$, where N is the number of nodes and RP is the memory request packet size. Each memory request packet consists of three flits. One flit for the request plus the head and tail flits. We set the burst rate of the off-chip memory to be a packet size. Therefore, the maximum size of a memory request per request message is the maximum packet size. With 32-bit flits and packets sizes of 10 flits, 8 bits of the request message is used for identifying how much data we intend to fetch from the memory and 24 bits are used to specify the memory address.

Assuming that the latency for accessing a memory block is M cycles and the size of the request is f flits, then the latency for receiving a block of memory from the off-chip memory by the memory controller is $M + f$ cycles. Furthermore, the memory controller has a buffer of size $N \times P$, for writing to the off-chip memory, where N is again the number of nodes and P is the maximum packet size. In addition, a write back buffer with the size of one packet stores the replies from the off-chip memory.

The memory controller serves the read/write requests in a FIFO manner. Read and write channels are separate, consequently reads and writes can be carried out simultaneously. The memory controller only serves a new read request once the off-chip memory replies to the previous request and the data leaves the Injection Buffer of the memory controller module.

We do not store data on the off-chip memory (we do not save actual data inside a data structure), but we account for the latency of the off-chip memory access as discussed above, in order to mimic communication delay and contention for the memory controller.

3.4 The network simulator

We have based our simulator on Booksim2 [38]. Booksim2 is a cycle-accurate network simulator, which operates at the granularity of flits and clock cycles. Booksim2 has a parametrised modular design that makes it easy to extend. Booksim2 modules are designed based on actual hardware implementations. All the modules are synchronised with a clock and configurable latencies are associated with links, router stages, and credit flows. The comparison of Booksim2 with an RTL imple-

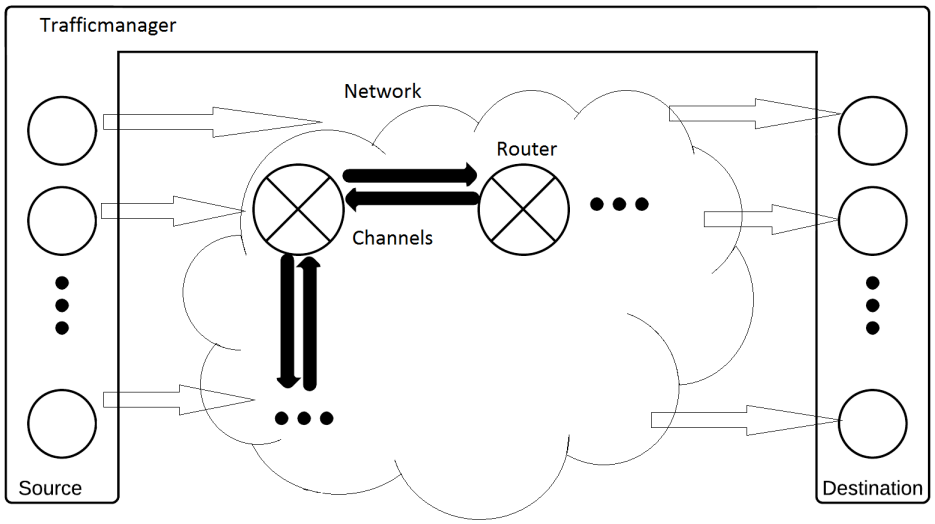


Figure 3.6: Top level view of Booksim2 [38]

mentation in [38] indicates that Booksim2's behaviour is nearly identical to the RTL implementation. Consequently, Booksim provides the accuracy needed for modelling contention for shared resources like VCs, switch allocates, flit buffers, etc.

Figure 3.6 depicts a top-level view of the simulator. The Trafficmanager is a controller wrapped around the network that sweeps the endpoints (i.e., source and destination nodes in the figure) and network components in each cycle.

Figure 3.7 depicts the module hierarchy of Booksim2. Several modules of Booksim2 have multiple implementations. As a result researchers can use a range of different network configurations to simulate NoCs. The Trafficmanager instantiates all the network modules and then runs the simulation. The Network module creates the topology and instantiates the routers and the channels.

Multiple router implementations are available in Booksim2. We have mainly used the **Input Queued** (IQ) router in our experiments. The IQ router has four stages of pipeline as depicted in Figure 3.8. In the first stage, flits are read into their respective VC buffers, and routing is done for each packet using the specified routing function. In the second and third stages, computations for VC allocation and the switch allocation are carried out. Once an output VC is allocated, the flit traverses the crossbar and is ready to be put on the link. With this architecture,

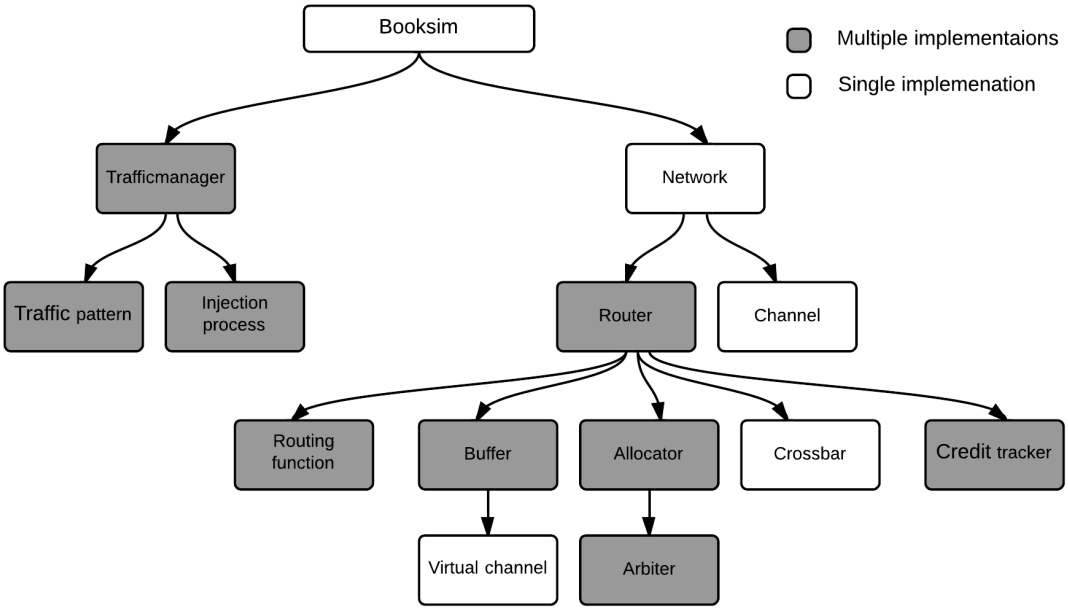
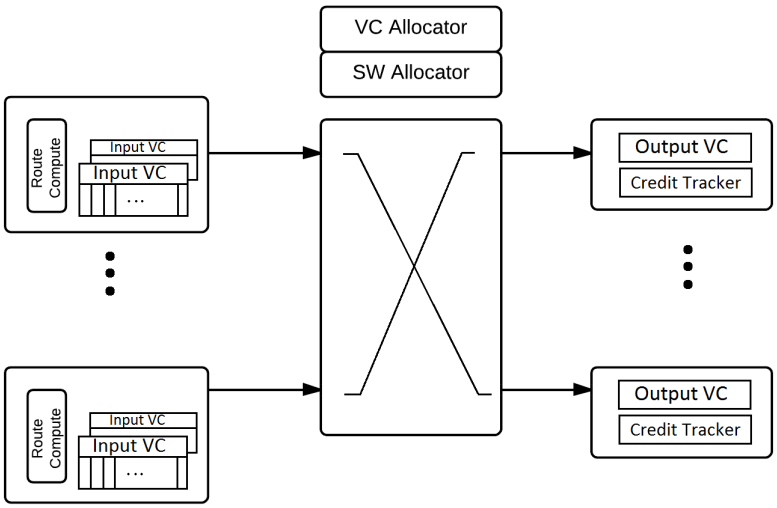


Figure 3.7: Module hierarchy of Booksim2 [38]

the minimum latency for sending a flit from the Injection Buffer of a source node to the Ejection Buffer of an adjacent endpoint in the mesh topology is twelve flit cycles (the flit has to traverse three links with one cycle per link, and two routers with four stages of pipeline each and a final cycle to latch the flit into the Ejection Buffer of the destination node).

As depicted in Figure 3.9, our model treats the network as a black box and mainly extends the Trafficmanager module. We extended the Trafficmanager module to instantiate processors and memory controllers according to the topology. Our Central Controller is responsible for mapping the incoming SDFGs to the processors that are connected to endpoints of Booksim (Mapping Manager) and also carrying out task migration (Task Migration Manager). Our code is not dependant on the network configuration and as a result our processor model can be tested with different network architectures that use different parameters. However, to implement multiple memory controllers or topologies other than 2D meshes, the memory controller addresses would need to be updated manually by the user.

The endpoints in Booksim are in charge of creating the traffic and also ejecting packets. Booksim uses traffic generators (using traffic pattern and injection process



(a) IQ router

(b) Pipeline Stages of the IQ router

Figure 3.8: (a) Mirco architecture of the IQ router [38], (b) Pipeline stages of the IQ router [38].

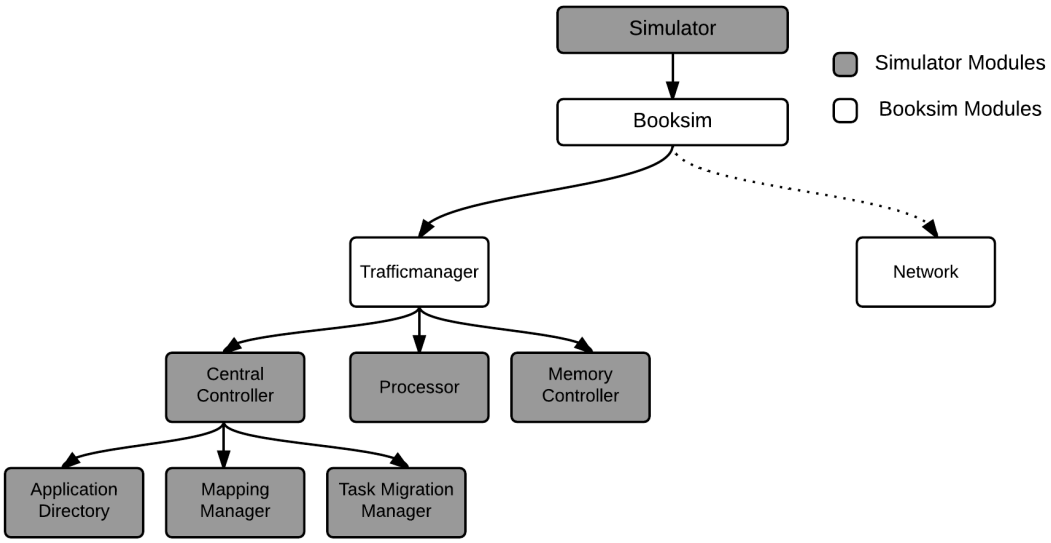


Figure 3.9: Module hierarchy of our simulator

modules) in order to create a variety of synthetic traffic patterns. However, in our simulator, rather than using synthetic traffic generators, we have extended the endpoints to simulate processors that produce and consume packets according to the semantics of the SDFGs.

In Booksim2, the traffic pattern module determines the packet destination, while in our simulator the application model determines the packet destinations (i.e., the processor to which the destination actor is mapped). The Central Controller therefore tracks the processor address of each task for each application using the Application Directory Module. The Central Controller updates the information stored in this module each time a new application is mapped or migrated. Processors are only aware of application layer addresses, i.e. the destination actor and port for each token. Once the output tokens are packetised, the Network Interface module of the processor fetches the network layer address from the application directory.

The Application Directory also stores the time when tasks are mapped and the performance metrics such as the execution time of a task and its load time. Network performance metrics such as packet latency, injection/ejection rates, and average hop distances can thus be calculated by Booksim2.

Booksim2 uses credit-based flow control, where downstream buffers send credits to upstream routers to specify how much buffer space they have. As we discussed before, each tile uses its Injection Buffer that can store one whole packet for injecting packets to the network. Once the processor fills the buffer, the flits of the packet are injected to the network one-by-one. The PE cannot inject a new packet until the tail flit of the previous packet is injected to the network. This helps modelling source queuing and head-of-line blocking on the endpoints, which means that the back pressure from the downstream nodes blocks the upstream nodes from injecting more packets to the network if there is congestion in the network.

As Booksim2 endpoints act mainly as traffic generators, Booksim2 does not explicitly model ejection back pressure. Ejection back pressure occurs when a processor runs out of memory and cannot eject any more packets. In order to simulate this behaviour, once the local memory of a processor becomes full, the PE stops the credit flow to its local router until it has enough space to eject packets again.

3.5 Mapping and scheduling SDFGs

The basis for the scheduling of our SDFGs is self-timed execution scheduling [29]. In self-timed execution scheduling, each actor should fire as soon as it is able to, i.e. has enough tokens on its input ports.

Mapping the communication channels of the SDFGs to the NoC does not guarantee the bandwidth requirements of the SDFG channels. In other words, communications between actors could be delayed due to congestion in the network. As a result, we extended the constraints for the firing of an actor to having enough tokens on input ports plus having enough space in the local memory for storing the output tokens. We further assumed that the processors have enough local memory space for one firing of all the actors in our benchmark (for both input and output tokens). Local memories are large enough to store both the tokens and the SDFGs' internal state.

Our simulator receives SDFG application descriptions as input and after parsing the XML description of SDFGs, it places them into a global queue called the application queue. Figure 3.10 depicts the stages for mapping and executing applications in our simulator.

The Mapping Manager determines where tasks should be mapped according to a user-defined mapping algorithm. The **default mapping algorithm** fetches an application from the application queue and checks the system to determine whether there are enough processors to map all of the actors of the application. If enough resources are available, the mapper scans the mesh from the top-left node to the bottom-right node and maps each task of the application to the first free PE. Otherwise, it blocks the application queue and waits until an application leaves the system before reattempting to map the application at the head of the queue. Since our architecture is homogeneous, any task of any application can be mapped to any free processor.

The Mapping Manager is implemented as a sub-module of the Trafficmanager module. As mentioned in the previous paragraph, the manager receives an application and chooses a processor for each actor. The final allocation of a task to

a processor is performed by a subroutine of the Mapping Manager based on task and processor IDs. Hence, to develop new algorithms, the user simply changes the section of the code where the Mapping Manager chooses the processor ID.

After the mapping is done, processors load their allocated tasks from off-chip memory. To load a task to a PE, the processor sends requests for the initial tokens and the state (program instruction memory and data as specified by the stateSize property of actors in the XML descriptions of our SDFGs) of the actor to the off-chip memory and stores the data it receives on its local memory. Note that while packets are transferred to emulate the necessary communication, the actual data is irrelevant since program execution is merely simulated to account for the execution delay.

Once all the tasks of an application are loaded, processors start executing their tasks according to the semantics of the SDFGs. If task migration is triggered during execution, the task's context is moved over the network from the source processor to the destination processor and once the move has been completed, the new processor continues to execute the task until the task is done.

The Central Controller is also in charge of removing any transient applications from the system once they finish executing. Once a processor finishes executing a task of a transient application, the Central Controller frees the processor and updates the Application Directory to indicate that the processor is now free and that the task has finished its execution.

3.6 Execution of SDFG actors on the processors

Figure 3.11 depicts the Processor module hierarchy. The network interface connects the processor to the network. The SDFG execution module simulates execution of SDFG actors and the Control Module coordinates the start and end of task execution as well as task migration. Each module is explained in detail below:

1. **Network Interface:** As mentioned before, the Network Interface consists of Injection and Ejection buffers, plus a Packetiser-Depacketiser unit. Once a packet is received on the Ejection Buffer, if the local memory has enough

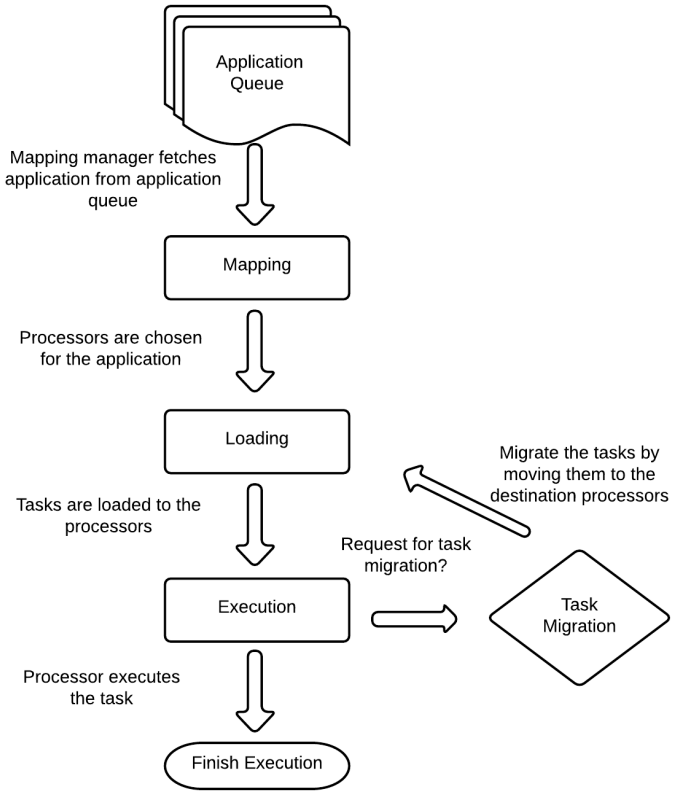


Figure 3.10: Mapping and execution flow of applications in our simulator

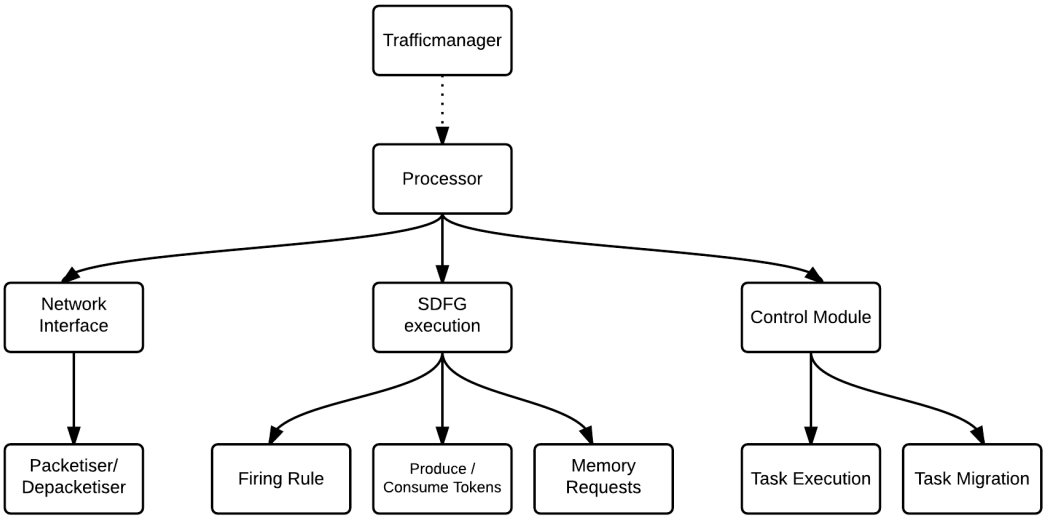


Figure 3.11: Processor module hierarchy

space, the Depacketiser module extracts the data from the packet and writes it to the local memory.

The PEs generate three types of messages (packets) during load time and execution (a fourth kind of packet carrying the program context is created during task migration): memory requests, memory writes, and packets carrying SDFG tokens. Round-Robin arbitration is used to settle the priority between messages when the PE needs to send more than one of any type in a cycle. Furthermore, when sending SDFG tokens, if the actor has more than one output port, Round-Robin arbitration is used again to choose the output port which is permitted to send a packet. The Packetiser module extracts data from the local memory, organises the data into flits, adds the head and tail flits and saves the packet to the Injection Buffer.

The size of the packet is determined by the product of the rate of an output port and the token size of the SDFG channel connected to that port. If the product is larger than the maximum packet size, the data has to be sent using multiple packets.

2. **SDFG Execution:** The SDFG execution module consists of the three submodules listed below:

   (a) **Memory Requests:** Although we modelled memory accesses in our application model as actors, memory actors are implemented as source and sink ports on the actors connected to *M-in* and *M-out* actors. Actors with source ports, have to send requests to the memory for the number of tokens given by their rate. In case of sink ports, the processor simply sends the tokens to the memory controllers.

   As stated in the previous section, the maximum memory request size is the maximum packet size. If the rate of a source port is larger than the maximum packet size then it has to issue multiple memory request messages.

   (b) **Firing Rule:** This module checks the local memory and issues a firing command once the necessary number of input tokens are present and there is enough space for the output tokens in local memory.

(c) **Produce/Consume Tokens:** Once the Firing Rule module issues a firing command, the Produce/Consume module waits for as many cycles as specified by the actor execution time. The time unit for the SDFG actor execution is one simulation cycle. The execution time is further delayed if the local memory does not have enough space for output tokens. Once the execution time is up, the module removes the input tokens required for firing from the local memory and produces output tokens and writes them to the memory. The reading and wiring of tokens to the local memory is carried out in one cycle.

3. **Control Module:** This module consists of Task Execution and Task Migration modules:

   (a) **Task Execution:** Task Execution module coordinates loading and execution of tasks. Once a task is mapped to a processor the Task Execution module starts sending requests to the off-chip memory to fetch the task's initial tokens and the actor's state and after the loading is done, it issues a command to start execution. This module also determines when a task of a transient application is done executing (when an application finishes its final iteration).

   (b) **Task Migration** When the Mapping Manager issues a Task Migration command (see Section 3.7), this module determines whether the task has reached the migration checkpoint and if it has, it starts the task migration. This module is also in charge of determining when a processor is done migrating a task or receiving a task from another processor.

3.7 Task migration

In order to migrate a task from one tile to another tile, the application needs to be halted. Then we migrate the task state (a property of each actor, as given by the application model) and the tokens stored on the local memory of the tile to the migration destination tile.

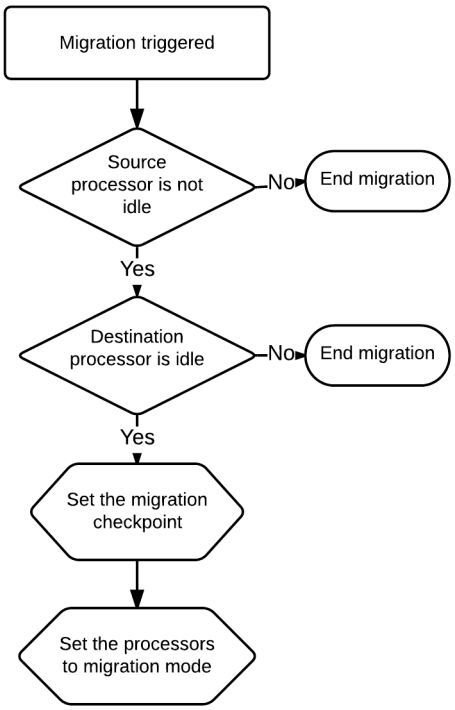


Figure 3.12: Triggering task migration flow chart

We use the same approach as the one in [62] to choose a migration point. This approach uses a special property of SDFG graphs, which is, that after a certain number of firings (according to the repetition vector) the graph returns to its original state (iteration). In this approach, the migration is only carried out once all the actors of the application are at the beginning of the same iteration. This is because at the beginning of an iteration there are no flits in flight between the tasks and the only tokens that we need to migrate are the initial tokens.

As we use self-timed execution scheduling of actors, iterations may interleave. Each processor keeps track of the iteration at which their task is. When the controller issues a request to migrate a task of an application, it checks the iteration of each task to determine the highest iteration number amongst all the tasks and sets that iteration number as the task migration checkpoint. After all the tasks of the application are at the start of the same iteration number, migration is performed. During the migration no tasks of the application fires or make requests for off-chip memory access. Migration destinations can be any PE tile. In case that it is desirable to swap the tasks of two PEs, the respective applications of each task are

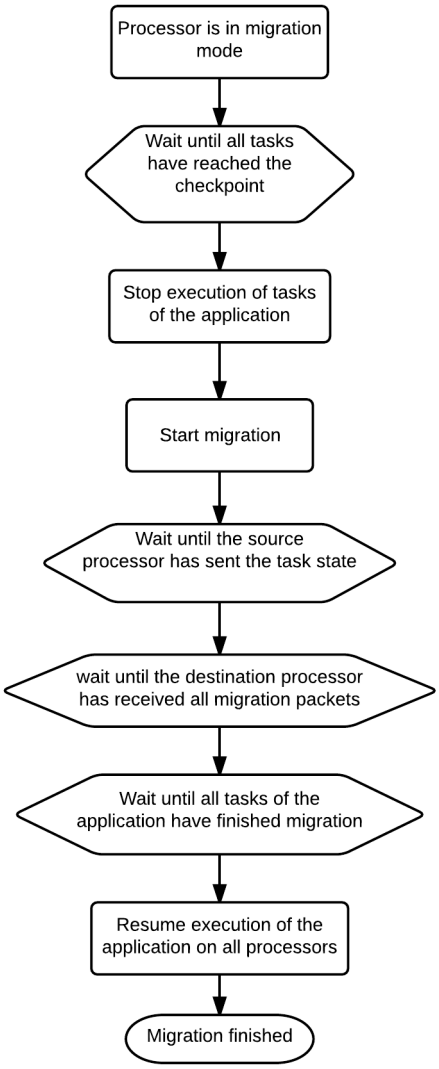


Figure 3.13: Task migration process flow chart

halted and the migration process is carried out for both tasks.

As mentioned above, task migration in our simulator consists of two stages, namely, triggering task migration and carrying out the process of migration. Figure 3.12 depicts the stages involved in triggering migration. First, it is checked that the source processor has a task to be migrated to begin with. Second, the destination should be an idle processor as a processor in our simulator can only run a single task at a time. If these conditions are met, then using the method mentioned above the checkpoint iteration is set and the source and destination processors are set to migration mode. The trigger is invoked using a method called "Migrate" in our code, which accepts three arguments: the application ID from the Application Directory, the source processor ID and the destination processor ID (each processor has a unique ID).

Figure 3.13 depicts the stages involved in carrying out the migration of a task. The Task Migration Manager (Figure 3.9) coordinates task migration. The Central Controller issues migration requests according to the task migration algorithm (in our simulations we triggered migration manually). Once the Task Migration Manager receives migration requests for actors of a particular application, it checks the iteration of each actor of the application and sets the migration checkpoint according to the procedure outlined above. After choosing the checkpoint, it notifies all the processors running the tasks of the application to halt once they reach the checkpoint. Once all the processors reach the checkpoint, the Task Migration Manager notifies the Task Migration module of the affected processors that they need to migrate their task or that they are receiving a task from another processor. Once a PE starts migrating a task, the Packetiser module of the processor obtains the migration destination from the Task Migration Manager and sends the task's entire state to the destination PE. Once all the processors finish migrating or receiving new tasks, the Task Migration Manager updates addresses of tasks inside the Application Directory and notifies the processors that they can resume executing their tasks.

During the migration of a task, the controller can issue additional migration requests to the tasks of the same application. Multiple migration events can be carried

out simultaneously. The application only resumes execution once all the tasks are done with migration. Other applications that are not being migrated continue to execute as normal. However, traffic loads may be affected during migration events.

3.8 Performance metrics

Listed below are the metrics for measuring the performance of applications when they are mapped to our system:

- **Application load time:** The time expressed in cycles, from when an application is mapped, until all the actors of the application are loaded.
- **Application execution time:** The time expressed in cycles, from when the application is loaded until the last actor(s) of the application finish(es) execution.
- **Actor execution time:** The time expressed in cycles, from when the application is loaded until the actor finishes execution.
- **Actor flit throughput:** The number of flits sent by the actor divided by the execution time of the actor. The flits of packets that are used for task migration and application load are not counted.
- **Average flit throughput:** The average over all the actors of an application of their flit throughputs expressed in flits per cycle.
- **Packet latency:** The time expressed in cycles from when the head flit of a packet is injected to the network in the source tile until the tail flit of the packet is ejected at the destination tile.
- **Average packet latency:** The average over all the packets sent during a simulation of their latency expressed in cycles.
- **Link utilisation:** The number of cycles during which a particular link is transferring flits during the simulation.

- **Average hop distance:** The average number of routers that packets traverse during a simulation.
- **Task migration response time:** The number of cycles from when the task migration command is triggered by the Task Migration Manager until the affected application reaches the checkpoint iteration and starts the migration process.

Apart from the above-mentioned metrics, the simulator prints the logs for the cycle in which each application is mapped, the tile which each actor is mapped, the cycle during which the application is loaded and the cycle during which each actor of a transient applications finishes execution. Moreover, the cycles during which any migration commands are triggered, the migration checkpoint iteration, the migration response time, and the duration of migration of each task is also recorded. Furthermore, Booksim2 provides mechanisms for tracing individual packets which are useful for tracing communication flows. The number of packets and flits sent and received by each node is logged by Booksim2 as well.

3.9 Limitations

In this section we list the limitations of our platform. We discuss how they can affect our results and what the solutions for ameliorating them are.

3.9.1 Processors

Our processors can only run a single task while in the thousands of core era, we have enough resources to run single tasks on cores, this could lead to underutilisation of resources.

Furthermore, our processors should have large enough local memories for the firing of any actor and they cannot read/write excess tokens from/to off-chip memory. This limitation could be overcome by a more sophisticated scheduling scheme plus a more detailed modelling of the off-chip memory.

3.9.2 Memory Controller

Memory controllers usually have a higher bandwidth than other tiles, since they have to serve multiple requests at the same time. In our model all the links have the same capacity. In order to support heterogeneous link capacities, a new router port and link architecture should be developed for Booksim2.

In addition, in order to reduce the network traffic that the memory request packets cause, we could allow memory request packets with a larger size than the maximum packet size (which is our off-chip memory's burst length) and have the memory controller breaking each request from the processor into multiple requests.

Moreover, the memory controller's address is hard-coded at the moment (we only used one memory controller per simulation and only used the 2D mesh topology for our simulations). However, the memory controller(s)' address could be automatically updated when other topologies are used or when more than one memory controller is desirable without much effort.

3.9.3 Central Controller

For simplicity, the Central Controller module is a software entity with global access to all the modules inside the simulator. Hence, we are not simulating the overheads and the delay in sending control messages from the controller to the processors or the traffic these messages create while, for instance, the Mapping Manager or the Task Migration Manager are carrying out mapping or migration. In a more realistic scenario, the controller should either be a special tile or it should be a service running on one or more of the processors.

3.9.4 Task Migration

The accuracy of measuring the cost of task migration depends on the level of abstraction we use. Costs such as translating memory pointers, saving an operating system's context, etc. requires more accurate processor and operating system modelling.

Our checkpointing mechanism requires all actors of the application to reach the same iteration. While this makes the behaviour of applications more predictable, and hence easier to schedule, it slows down the migration response time, which is the time between requesting a migration until it is granted. we could improve the response time by using more complex scheduling algorithms and forwarding the packets that arrive at the previous position of a task.

3.10 Summary

We introduced our many-core simulator based on Booksim2. We have extended the Trafficmanager module of the Booksim2 to execute SDFG applications and perform runtime mapping and task migration. Moreover, We have extended the SDFG structure to model off-chip memory access and we modified the firing rules of SDFGs to account for local memory occupancy before firing.

Table 3.1 lists the design parameters used in our simulations:

Design Parameter	**Description**
Maximum packet size	10 Flits with 8 body flits and 2 flits for head and tail
Flit size	32 bits
Memory request packet size	3 flits, one body and 2 for head and tail. The body flit reserves 8 bits for data and 24 bits for memory address.
Topology	2D Mesh
Routing algorithm	X/Y
Router buffer size	10 flits
Router architecture	Input Queued
Number of VCs	2
PEs' Ejection Buffer size	2×10 flits
PE's Injection Buffer size	10 flits
Memory controller Request buffer size	$N \times 32$ bits (N: number of nodes)
Memory controller Write buffer size	$N \times 8 \times 32$ bits (N: number of nodes, 8: maximum number of data flits in a packet)
Arbitration	Round-Robin

Table 3.1: Design parameters for our simulations

Chapter 4

Task placement and migration

Task placement can create or eliminate fragmentation and contention in many-core systems that use network-on-chips. In this Chapter, through various simulation scenarios we demonstrate how our simulator can simulate the different task mappings that cause fragmentation and contention. Furthermore, we devise task migration scenarios and investigate how our simulator takes the cost of task migration into account as well as showing the merits of using the task migration technique.

In all our simulations we used the design parameters outlined in Section 3.10. However it should be noted that, we scaled and changed the execution times of our SDFGs to stress the traffic in the system and to be able to demonstrate how our simulator captures the effect of contention in the system.

4.1 Mapping and contention

Contention for architectural resources like links, router ports and virtual channels can greatly affect the packet latency and execution time of tasks running on a multi-processor system. Effective switching and routing algorithms as well as careful adaptive mapping schemes can resolve contention. Our focus is on simulating the effect of task placement and migration on network contention.

From the network layer perspective, contention can be classified into three different classes, namely, source-based, destination-based and path-based contention

[21]. Source-based contention occurs when multiple flows originating from the same source, contend for the local router's injection link/port. Destination-based contention occurs when multiple flows with the same destination contend for the router's ejection link/port. In case of path-based contention, data flows may or may not share the same source or destination, but they contend for shared network resources along their paths. Results in [21] show that mapping cannot affect the source-based or destination-based contention as these types of contentions have direct relation to architectural characteristics of the processors and the network (i.e., maximum injection or ejection rate of the processors). However, task mapping can greatly reduce the occurrence of path-based contention.

In our model, source-based contention occurs if a Synchronous Data Flow Graph (SDFG) actor has multiple output ports and destination-based contention occurs when an SDFG actor has multiple input ports.

From an application's perspective, contention can be either internal or external. Internal contention refers to the contention that occurs between the data flows of tasks of the same application, while external contention occurs when data flows of different applications contend for architectural resources.

In the experiment below, we demonstrate a path-based external contention scenario that occurs as a result of a task mapping. We demonstrate internal contention scenarios in the next section.

Let us assume that we have a 4×4 mesh labelled in row-major order and that node 0 is the memory controller tile while the H.263 decoder depicted in Figure 4.1 is our application. Figure 4.2 depicts our mesh with each tile representing a processing tile and its local router. In this scenario (Scenario 1), the H.263 decoder is the only application in the system. In Scenario 2, depicted in Figure 4.4, we added a synthetic application (depicted in Figure 4.3) mapped to tile 2 and tile 13.

Table 4.1 compares the results of these two scenarios. In Scenario 1, in which only one application is mapped to the mesh, there is no contention for network resources. However, once the synthetic application is mapped to the mesh as well, path-based contention on the link from node 5 to node 9 occurs. Figure 4.5 depicts the router connected to Tile 5. The data flows that came from the northern and local

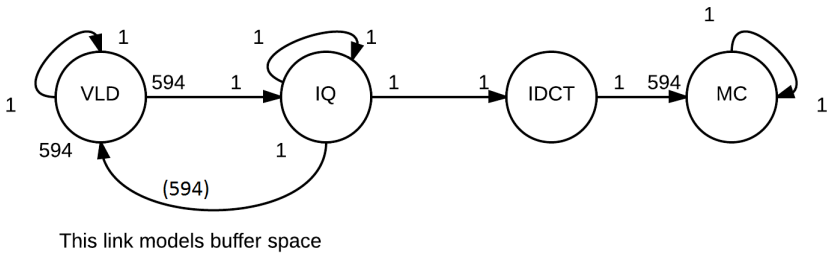


Figure 4.1: Schematic diagram of the H.263 decoder SDFG [61].

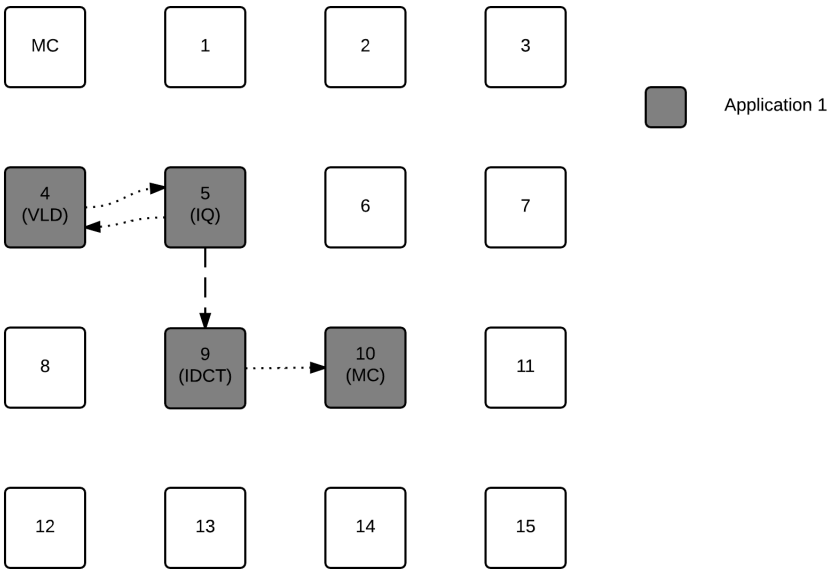


Figure 4.2: Scenario 1: Contention example single application

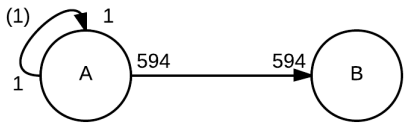


Figure 4.3: Schematic diagram of the Synthetic Application

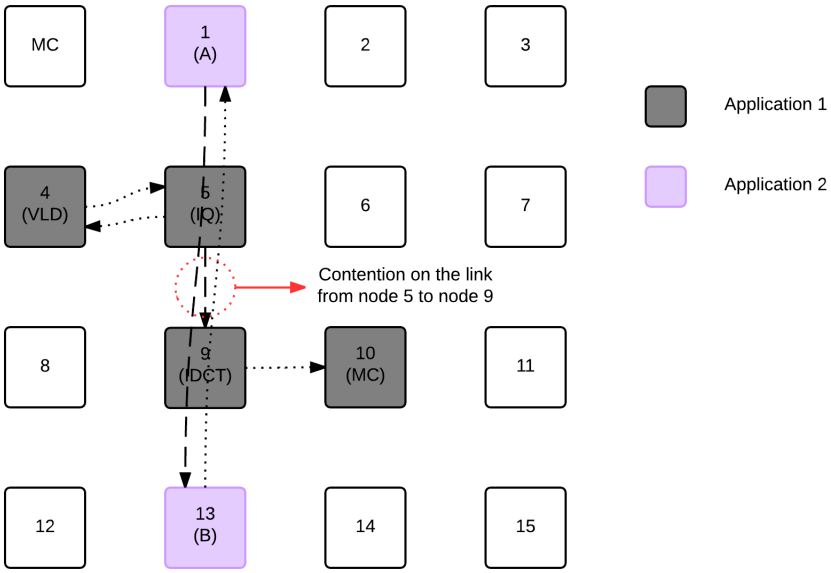


Figure 4.4: Scenario 2: Contention example two applications

input ports of the router contended for the southern output port. As a result of this contention, the execution times of all the actors for the H.263 Decoder increased. Furthermore, looking at the utilisation of the link between tiles 5 and 9, we notice the dramatic increase in Scenario 2, as both flows have to use this link to get to their destinations. Contention also degraded the average flit throughput of the H.263 decoder.

As we demonstrated in this experiment, our simulator is capable of modelling path-based contention and its effects on execution time and throughput of the applications.

	Flit Throughput For Actor IQ (Flits/Cycle)	Link 5-9 Utilisation (Cycles)	Actor VLD Execution Time (Cycles)	Actor IQ Execution Time (Cycles)	Actor IDCT Execution Time (Cycles)	Actor MC Execution Time (Cycles)
Scenario 1 (Single application)	0.798	237,619	464,272	470,237	470,272	470,287
Scenario 2 (Two applications)	0.659	820,225	554,297	568,075	568,145	568,160

Table 4.1: Contention as a result of mapping

4.2 Fragmentation

Fragmentation occurs when there is no contiguous block of processors that can fit all the tasks of an application and consequently, the tasks of an application are

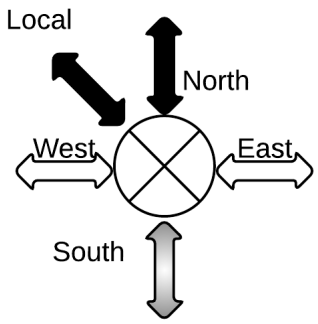


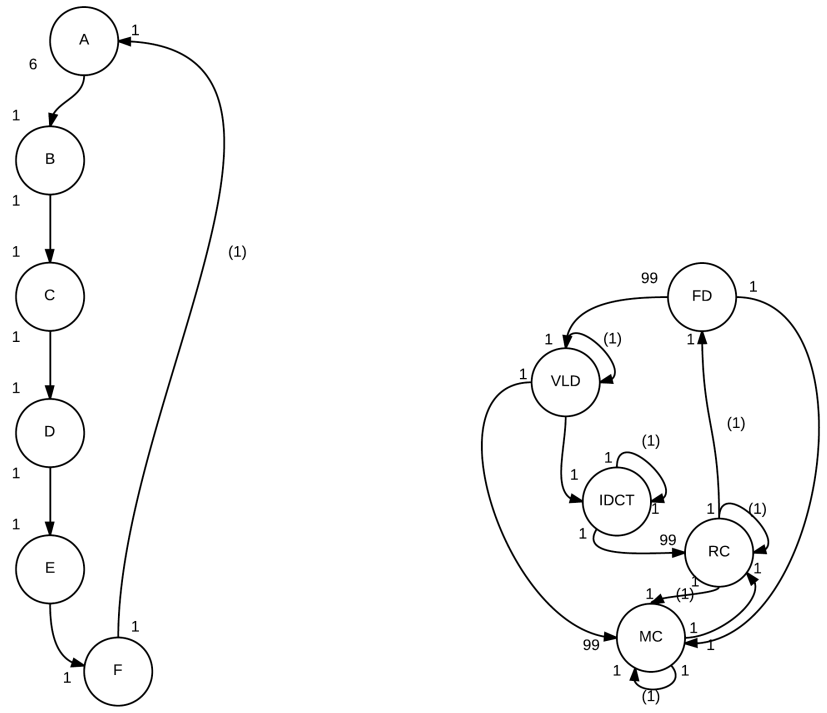
Figure 4.5: Contending flows inside the router attached to tile 5 for the mapping scenario depicted in Figure 4.4.

scattered across the network. Fragmentation increases the hop distance between the communicating tasks of an application, which in turn increases both the packet latency and power consumption of the system. Moreover, when communicating tasks are mapped non-contiguously, external contention as a result of interference between the data flows of unrelated applications is more likely to occur.

We studied the effects of fragmentation on the execution time and communication metrics for a JPEG decoder and an MPEG4 decoder [56]. In each experiment only one application is mapped to the system. For each application, we compared the performance of a contiguous mapping against that of a fragmented mapping. Figures 4.6(a) and 4.6(b) depict SDFGs for these two applications. The JPEG decoder's structure is a simple cycle while the MPEG4 decoder has a more complex communication pattern with multiple internal cycles. In both experiments, applications are mapped to a 4×4 mesh. We ran the JPEG decoder for one hundred iterations and the MPEG4 decoder for fifty iterations.

Figure 4.8(a) depicts the contiguous mapping scenario while Figure 4.8(b) depicts the fragmented mapping scenario for the JPEG decoder. The dotted arrows, depict the communication flows between processors. As we can observe there are no contending flows in either scenario.

Table 4.2 summarises the results of the experiments with the JPEG decoder. The fragmented scenario (Scenario 2) performed worse than the contiguous scenario (Scenario 1). Execution time of the application in Scenario 2 increased by 16%,



(a) (b)

Figure 4.6: (a) The JPEG decoder SDFG [56] (b) The MPEG4 decoder SDFG [56]. The numbers in parentheses indicate the number of initial tokens.

while the average packet latency increased by 60%. The average hop distance of Scenario 2 increased by 132% compared to Scenario 1. The reason why the average hop distance of the contiguous mapping of Scenario 1 is 2.002 and not 2, which is the average hop distance between the mapped actors in this scenario, is due to the additional hops spanned by packets sent and received for loading the application. In the absence of external or internal contention in either scenario, the fragmented mapping of Scenario 2 caused a higher hop distance between the communicating tasks. The higher hop distance, in turn, led to a greater packet latency, higher execution time, and lower throughput.

Figures 4.9(a) and 4.9(b), depict a contiguous mapping scenario and a fragmented mapping scenario for the MPEG4 decoder application respectively. Apparent points of contention are pointed out with red circles. Table 4.3 summarises the results of the simulations with the MPEG4 decoder mappings. This time, contrary to what we observed with the JPEG decoder application, despite the greater hop distance and higher packet latency in the fragmented mapping of Scenario 2, the

contiguous mapping of Scenario 1 performed worse.

The order of execution of the MPEG4 decoder's actors is depicted in Figure 4.7 assuming that the tokens are delivered without delay. In our simulations, with the mappings in both scenarios, the order of execution is the same as in Figure 4.7. With the mentioned order of execution, the flows from the actors *FD* and *VLD* to the actor *MC* are not active at the same time. Hence, there is no contention between these flows. Another apparent point of contention is where multiple flows originate from the same tile and share a number of links along their paths. This scenario is pointed out in Figure 4.9(a) where two flows originate from tile number 4, and in Figure 4.9(b) where multiple flows originate from tiles $3, 4$ and 8. Irrespective of the mapping we use, source-based contention exists in both scenarios. However, as Round-Robin arbitration is used to settle the priority between the output ports of actors, and given that the Injection Buffer can only ever be occupied by one packet at a time, in the absence of external contention, the packets from different output ports do not contend with each other and path-based contention does not occur for the apparent points of contention mentioned above.

By examining the contiguous mapping depicted in Figure 4.9(a), we observe an internal contention between data flows *VLD-to-MC* and *IDCT-to-RC*. While in Figure 4.9(b), there is no path contention between data flows. This means that shorter hop distance or lower packet latency does not always translate to better execution time. In this case, in Scenario 1, the delay in delivering packets from actors *VLD* and *IDCT* to *RC* and *MC* respectively increased the execution time of actors *MC* and *RC*. Although the volume of the delayed packets of the contending flows is small compared to the total number of packets sent and received by the actors of the application, they cause a great delay. Since an *SDFG* actor needs all input tokens present before firing, the firing time of an actor is determined by the arrival time of the slowest packet that carries input tokens. The longer the actor has to wait for tokens, the longer is the execution time of the actor.

	Average Flit Throughput (flits/cycle)	Execution Time (cycles)	Average Packet Latency (cycles)	Average Hop Distance (Number of Hops)
Scenario 1 (Contiguous mapping)	0.200	47,967	21.002	2.002
Scenario 2 (Fragmented mapping)	0.171	55,947	34.305	4.663

Table 4.2: The JPEG decoder mapped to a 4-by-4 mesh and run for 100 iterations

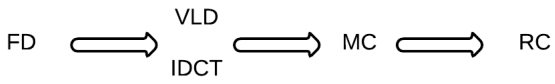
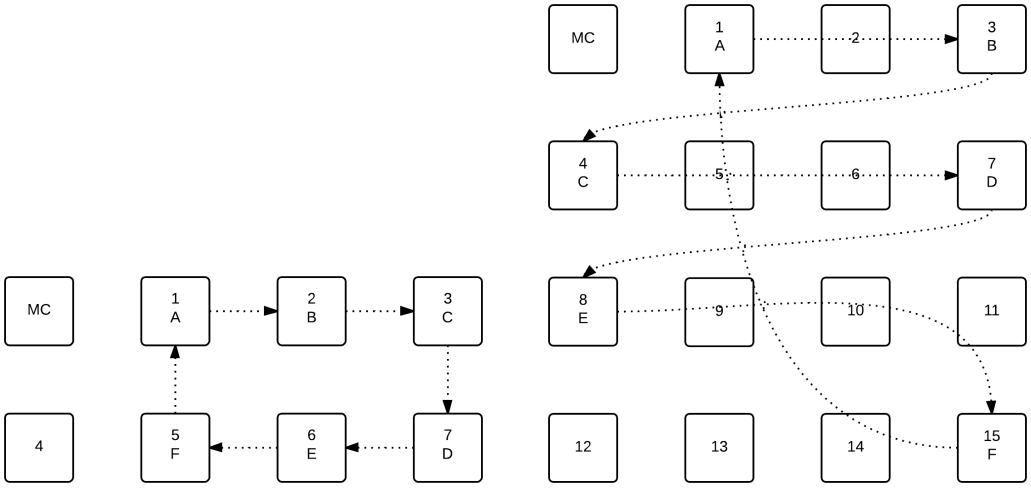
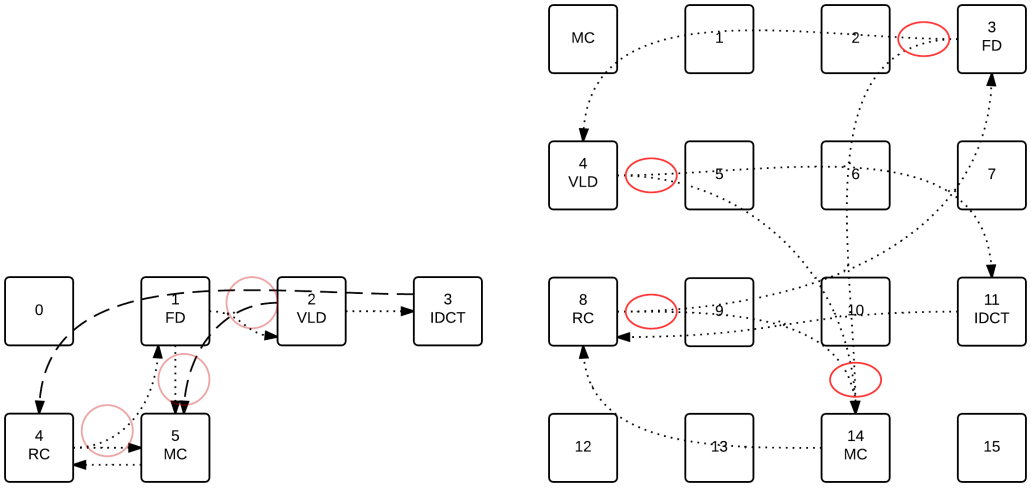


Figure 4.7: The order of the execution of actors of the MPEG4 decoder assuming zero latency communication.



(a) (b)

Figure 4.8: (a) A contiguous mapping (Scenario 1) of the JPEG decoder (b) A fragmented mapping (Scenario 2) of the JPEG decoder



(a) (b)

Figure 4.9: (a) A contiguous mapping (Scenario 1) of the MPEG4 decoder (b) A fragmented mapping (Scenario 2) of the MPEG4 decoder

	Average Flit Throughput (flits/cycle)	Execution Time (cycles)	Average Packet Latency (cycles)	Average Hop Distance (Number of Hops)
Scenario 1 (Contiguous mapping)	0.214	516,422	27.948	2.714
Scenario 2 (Fragmented mapping)	0.219	503,807	34.564	4.712

Table 4.3: The MPEG4 decoder mapped to a 4-by-4 mesh and run for 50 iterations

4.3 Runtime mapping and fragmentation

Runtime mapping may lead to fragmentation of available processors in a system. With careful clustering methods and efficient mappings, we can reduce the possibility of fragmentation occurring. However, as the system does not have a priori knowledge about future application needs, scenarios occur where the size of the available contiguous regions do not match the size of the application being mapped.

We mapped thirty applications to an 8×8 mesh [1], with the top left tile (tile 0) being the memory controller and the remaining being processor tiles (the last being tile 63 on the bottom-right). We used multiple instances of eight multimedia applications as listed in Table 4.4. As our SDFGs are deterministic and they express the same behaviour regardless of the number of iteration they are completing, the number of iterations for transient applications was chosen so the contention in the network would be noticeable. The applications were placed into a global queue and were mapped in FIFO manner when sufficient free tiles became available on the mesh. At time 0, we mapped as many applications as we could and the rest were mapped at runtime. Each time an application finished its execution, the next application at the head of the queue was picked for mapping. Some of the applications (resident applications) were designed to remain allocated and execute periodically, while the rest (transient applications) were executed for a specified number of iterations in a one-off manner. We ran the simulation for four million cycles which took over 297 seconds to finish on a Core-i5 Intel processor running at 3.3 GHz.

We used a very simple mapping heuristic. We scanned the mesh from top left to bottom right, and mapped the tasks of each application to the first free tile found.

At the beginning of the simulation, since all the tiles of the mesh were free, the mapping manager mapped the applications contiguously. However, as different ap-

[1] For the XML representations of the SDFGs see Appendix A. The order in which the applications were mapped is listed in Appendix B

Application	Actors	Instances	Type	Iterations
MPEG4 [56]	5	6	Transient	50
JPEG [56]	6	5	Transient	10
MP3 Dec. [61]	14	6	Transient	10
H.263 Dec. [61]	4	5	Transient	10
Sample-rate [61]	6	3	Transient	4
H.264 Dec. [56]	3	2	Resident	N/A
H.263 Enc. [61]	5	1	Resident	N/A
H.264 Enc [56]	9	2	Resident	N/A

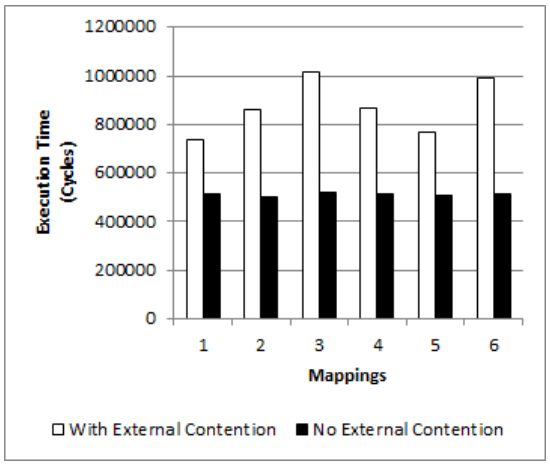
Table 4.4: Applications used for the simulation

plications finished executing, fragmented spaces became available, and the mapping manager mapped the applications in a fragmented way. As we discussed in Section 4.2, in the absence of external contention, fragmentation can either slow down applications by increasing the average hop distance and the average packet latency or it may speed up applications by resolving internal contention. However, once multiple applications exist in the system, fragmentation causes contention between flows that share the same path.

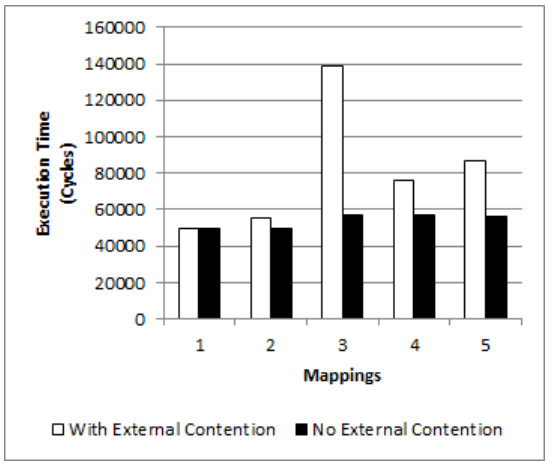
Figures 4.10 to 4.12 illustrate the execution time, average flit throughput, and application load time for the various mappings of the individual instances of the MPEG4, JPEG and H.263 decoders applications in the runtime mapping scenario, compared with identical mappings but without external contention (without other applications being present). As the behaviour of SDFGs is deterministic, the number of flits sent by different instances of the same application is the same. Hence, there is a direct inverse relationship between the execution time and the average flit throughput of SDFGs.

As we can observe from the figures, the scenarios without contention outperformed the ones that experienced contention in all cases and for all metrics. In the runtime mapping scenario, with external contention, the load time of the applications increased dramatically for all three applications, which indicates that the off-chip memory access is an important point of contention.

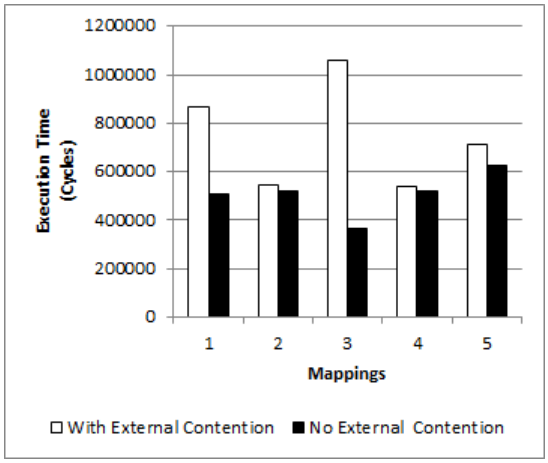
To make matters more clear, we examine one of the instances of the MPEG4 decoder more closely to show how fragmentation as a result of poor mapping causes performance degradation.



(a) MPEG4 Decoder

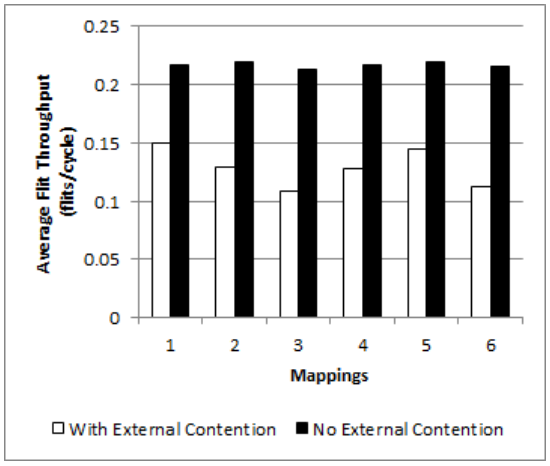


(b) JPEG Decoder

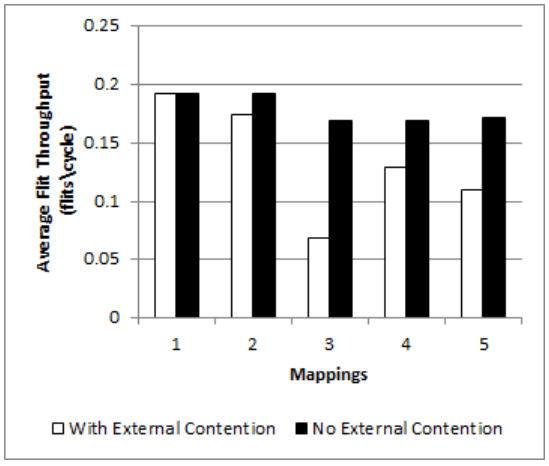


(c) H.263 Decoder

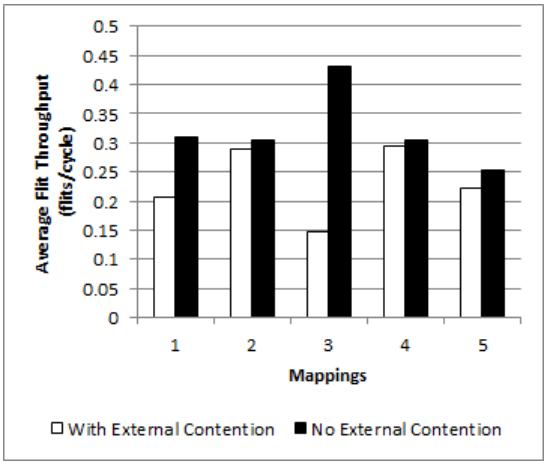
Figure 4.10: Execution time of the individual instances of MPEG4 (a), JPEG (b) and H.263 (c) decoders as recorded in the runtime mapping scenario with external contention compared with identical mapping but without external contention (without competing applications).



(a) MPEG4 Decoder

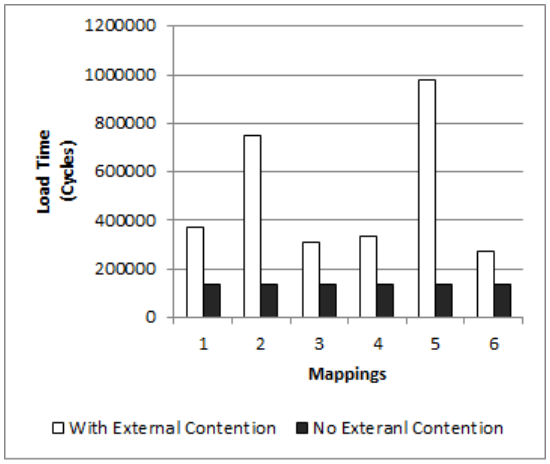


(b) JPEG Decoder

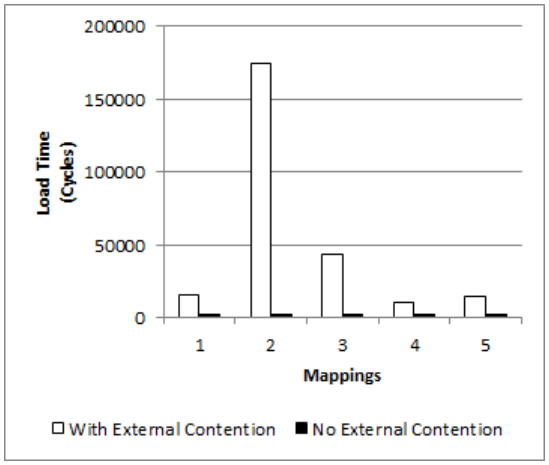


(c) H.263 Decoder

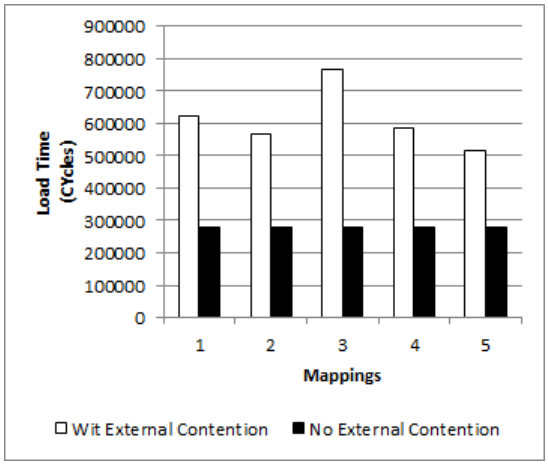
Figure 4.11: Average flit throughput of the individual instances of the MPEG4 (a), JPEG (b), and H.263 (c) decoders as recorded in the runtime mapping scenario with external contention compared with identical mapping but without external contention (without competing applications).



(a) MPEG4 Decoder

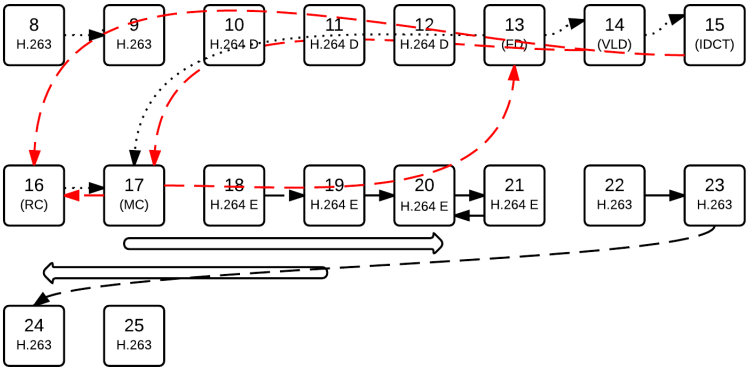


(b) JPEG Decoder

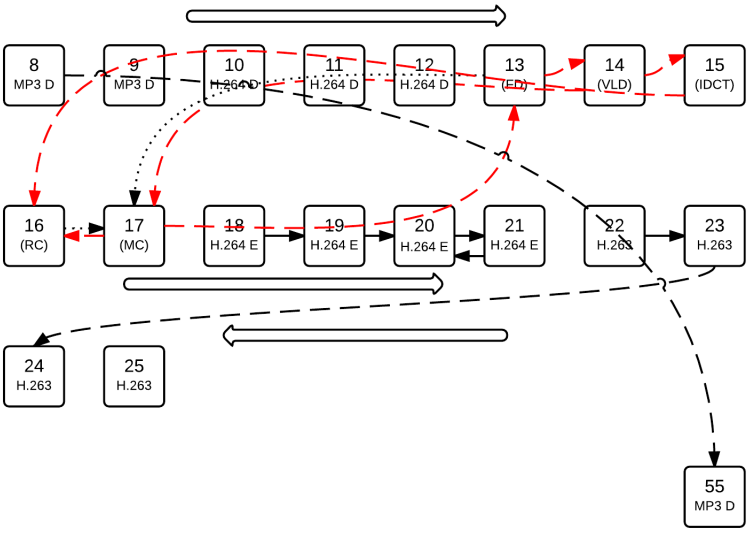


(c) H.263 Decoder

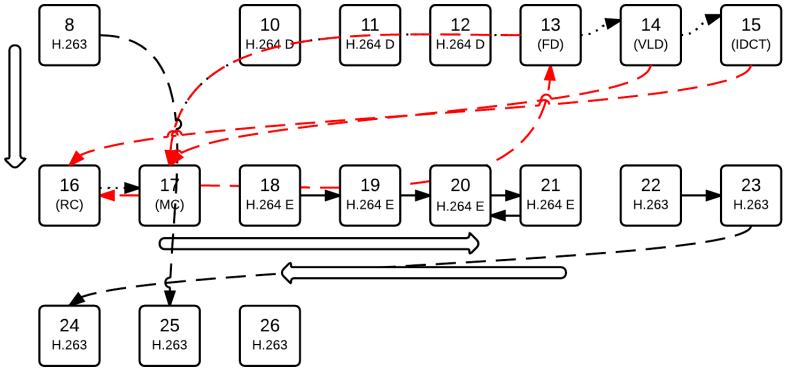
Figure 4.12: Application load time of the individual instances of the MPEG4 (a), JPEG (b), and H.263 (c) decoders as recorded in the runtime mapping scenario with external contention compared with identical mapping but without external contention (without competing applications).



(a)



(b)



(c)

Figure 4.13: Internal and external contention for the third instance of the MPEG4 decoder in the runtime mapping scenario.

Figure 4.13 depicts the mapping of the third instance of the MPEG4 decoder and the applications mapped to the vicinity of it in the runtime mapping scenario as time advanced. We chose this instance of the MPEG4 decoder as it had the longest execution time. The Mapping Manager mapped the application to tiles $13 - 17$. The red dashed arrows indicate the MPEG4 decoder's communication flows that suffer from contention while the dotted arrows depict the communication flows which do not experience contention. The white arrows, depict the presence of external contention.

An instance of a resident H.264 decoder was mapped to tiles $10 - 12$ but did not contend with the MPEG4's flows, as depicted in the figures. However, a fragmented instance of the H.264 encoder occupied tiles $18 - 21$, and the flows from these tiles did contend with the *MC-FD* flow of the MPEG4 decoder. In Figure 4.13(a), tiles 8 and 9 were occupied by an H.263 decoder's actors. The flows from this application did not contend with any of the flows of the MPEG4 decoder. Another instance of the H.263 decoder was mapped to tiles $22 - 25$. Apart from the load memory request packets sent from tiles 22 and 23, the flow from tile 23 to 24 contended with the *MC-RC* flows of the MPEG4 decoder. As time advanced, an instance of an MP3 decoder was mapped to tiles 8 and 9, as depicted in Figure 4.13(b). The flow from tile 8 to tile 55 contended with the *FD-VLD* and *VLD-IDCT* flows of the MPEG4 decoder. Finally, in Figure 4.13(c), two new fragmented instances of the H.263 decoder were mapped to the mesh. Tiles 8 and 25 were occupied by portions of one instance of one of the H.263 decoders and the other instance was mapped to tiles $22 - 24$ and 26. Again, apart from the load memory request packets of tiles 22 and 23, the flow from tile 8 to tile 25 contended with the *FD-MC* and *VLD-MC* flows of the MPEG4 decoder. Moreover, the flow from tile 23 to tile 24, contended with the *MC-RC* flow of the MPEG4 decoder. As a result of both internal and external contentions, this instance of MPEG4 had a longer execution time when compared with other instances when the contention for architectural resources was less.

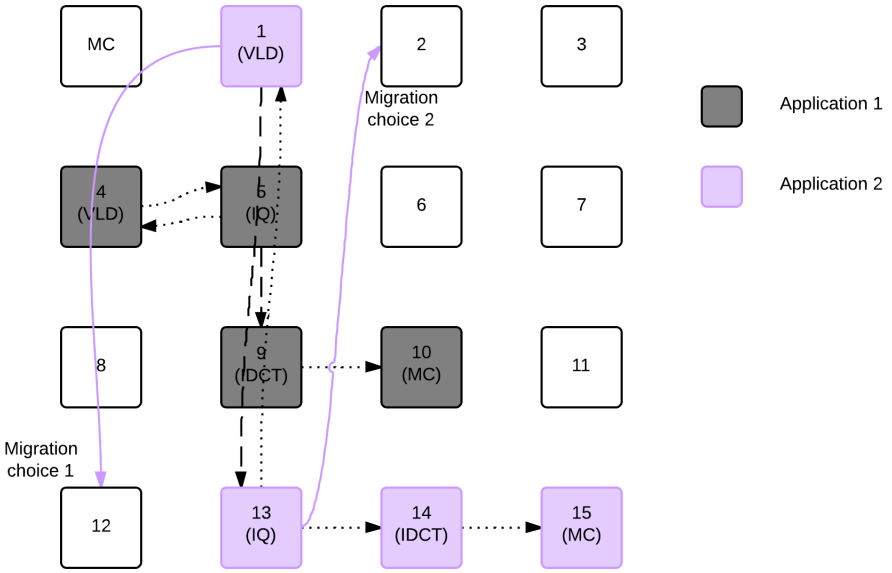


Figure 4.14: a migration scenario with two instances of the H.263 decoder.

4.4 Task migration

As we saw earlier in this Chapter, inefficient mappings lead to path-based contention in the network. Task migration can improve poor initial mappings at the cost of some processing delay and some temporary additional network flow due to the movement of task state. A task migration algorithm needs to address three issues: Which tasks to migrate, when to migrate them and where to migrate them.

Figure 4.14 depicts a scenario where two instances of the H.263 decoder have been mapped to a 4×4 mesh. Both instances were mapped at time 0 and were run for 20 iterations. The dashed arrows indicate the links on which communication flows experienced contention. We experimented with two migration scenarios. In the first migration scenario, we moved the VLD actor of the second application to **Processing Element** (PE) 12 and in the second migration scenario, we migrated the IQ actor of the second application to PE 2 in order to resolve contention.

Tables 4.5 and 4.6 summarise the results of simulations for both migration scenarios. For each scenario we compared the results of running applications with four different mappings. A mapping with external contention depicted in Figure 4.14, a mapping with migration at an early stage of the execution, a mapping with a late

Mapping	App. (1) Execution Time (Cycles)	App. (2) Execution Time (Cycles)	App. (1) Average Flit Throughput (flits/Cycle)	App. (2) Average Flit Throughput (flits/Cycle)	Overall Average Packet Latency (Cycles)
No Migration (Contention)	541,377	475,628	0.466	0.532	34.680
Early Migration	477,398	483,027	0.529	0.523	22.493
Late Migration	537,817	483,568	0.469	0.522	34.131
No Migration (No Contention)	470,287	470,287	0.537	0.537	21.000

Table 4.5: Simulation results for migration choice (1), where we migrated the actor VLD of Application (2) from PE 1 to PE 12.

Mapping	App. (1) Execution Time (Cycles)	App. (2) Execution Time (Cycles)	App. (1) Average Flit Throughput (flits/Cycle)	App. (2) Average Flit Throughput (flits/Cycle)	Overall Average Packet Latency (Cycles)
No Migration (Contention)	541,377	475,628	0.466	0.532	34.680
Early Migration	477,398	470,451	0.529	0.537	23.535
Late Migration	537,817	470,992	0.469	0.536	34.181
No Migration (No Contention)	470,287	470,297	0.537	0.537	22.258

Table 4.6: Simulation results for migration choice (2), where we migrated the actor IQ of Application (2) from PE 13 to PE 2.

migration, and also a mapping in which the task which we migrated in the other scenarios was mapped to the migration destination in the first place, thereby avoiding contention and migration overheads.

In both scenarios, the mapping in which there is no contention in the system outperformed other mappings, as we expected. It should be noted that the contention has a greater impact on Application 1. In both migration cases, task migration improved Application 1's execution time and flit throughput while it degraded the performance of the second application. However, In Scenario 2 (Table 4.6) both applications gained improvements after we migrated the IQ task to PE 2.

The execution time of application 1 was the same in both scenarios, since once migration was triggered, application 1 did not experience further contention in either scenario, and the migration traffic of moving application 2's task for either migration destination choice, did not interfere with Application 1's flows. Furthermore, due to the fragmentation in the mapping of Application 2 in the "no-contention" case of Scenario 2, a one-off additional delay in the execution time incurs. Although, the extra two hops distance for "no-contention" case of Application 2 in Scenario 2 changed the execution time slightly, it still raised the average packet latency, as it

took a longer time for the individual packets from tile 2 to reach tile 14. The same explanation applies to the overall packet latency of task migration simulations of Scenario 2.

Early migrations are better in both cases, compared with late migrations, as the applications got to benefit from the new mappings for a longer periods of time. The difference between the results of the two migration scenarios lies in the task migration penalty of these two scenarios. Tables 4.7 and 4.8 compare the migration cost of the early and late migrations for both scenarios. The actor *VLD* of the H.263 decoder has a much larger state size compared to the actor *IQ*. Hence the time needed to migrate the *VLD* task in Scenario 1 is greater. Consequently, Application 2 has to halt for a longer period in Scenario 1 and that causes a longer execution time for Application 2.

Mapping	Response Time (Cycles)	State (bits)	Packets	Duration (Cycles)	Checkpoint (Iteration)
Migration Choice 1	8,574	323,168	1,263	12,653	2
Migration Choice 2	8,574	912	4	66	2

Table 4.7: Migration cost of early migration of tasks for both migration scenarios.

Mapping	Response Time (Cycles)	State (bits)	Packets	Duration (Cycles)	Checkpoint (Iteration)
Migration Choice 1	3,885	323,168	1,263	12,653	19
Migration Choice 2	3,885	912	4	66	19

Table 4.8: Migration cost of late migration of tasks for both migration scenarios.

As we demonstrated in this example, choosing different tasks of the application to migrate, can dramatically change the performance. Moreover, the right timing to trigger the task migration is another factor that changes the balance between the performance gain and penalty of task migration.

The trade-off threshold between the speed-up of one application and slow-down of another, is a design parameter that should be considered for task migration schemes. For example, in the case of Scenario 1 where task migration made Application 1 faster and Application 2 slower, the migration algorithm needs to decide whether this trade-off is acceptable or not.

4.5 Implementing task migration policies

In this section we provide guidelines for implementing task migration policies using our simulator. For this purpose, we chose two task migration policies published in [31] and [67] as mentioned in our literature review. In the following we explain how the tools in our simulator can be used to implement these policies.

The work in [31] proposes a task migration policy to improve the fragmentation in 2D-mesh NoC multiprocessors that use X-Y routing and wormhole switching. A contiguous first-fit algorithm is used to allocate applications to rectangular submeshes. The task migration policy aims at defragmenting the system by moving the applications to the bottom-left corner of the mesh to open up submeshes in the right and upper sides of the mesh. In order to reduce the cost of the task migration, the tasks of the application are moved in a way that the traffic caused by moving the state of the tasks through the network does not interfere with the traffic inside other submeshes. When migrating an application, the algorithm first moves the application along the X-axis to the left and then along the Y-axis towards the bottom. The application is moved to left as long as all the nodes on the left-hand side of the submesh it is occupying are free, and it is moved towards the bottom as long as all the nodes on the bottom of the submesh are free. This way, further contention is avoided and applications are moved toward the bottom-left corner in order to open up more space for incoming applications. The task migration mechanism used is a checkpoint based method and task migration is triggered once for every two disallocations (when applications finish execution).

In order to implement this policy, we need to know where allocated submeshes are, and it is necessary to move the tasks of the application to a new submesh. The information about allocated applications and tasks are all stored in the Application Directory data structure inside the Central Controller. Hence when moving a submesh, the algorithm can use the Application Directory to check if the nodes along the right-hand or bottom of the submesh are free or not. Once the algorithm decides to migrate tasks, it can use the "Migrate" method of the simulator to move the tasks. The task migration policy in this work uses a diagonal scheme [66] for choosing the order in which tasks of an application are migrated in order to avoid

contention between the migration traffic flows of different tasks. To schedule the individual task migrations of an application, it is possible and necessary to amend the condition for migrating a task inside our Task Migration Manager with the further condition, that it is in fact the task's turn (as indicated by the diagonal scheme) to be migrated. This migration policy would be added as a sub-module inside the Trafficmanager class, and it should be used inside Central Controller method.

The second task migration policy, proposed in [67], uses a hybrid force-directed and tabu search task migration algorithm to improve upon the initial mapping. The algorithm is run with the local view of each processor. A cost function is used to determine weather task migration will improve the local communication cost, overall communication cost and the congestion in the network. The communication cost is determined by the product of the distance between two tasks and the volume of communication between them. In order to formulate the cost function in our simulator, the Application Directory module can be used to calculate the distance between communicating tasks, and also the volume of communication using the rates provided in the SDFG model of the applications.

The task migration policy picks initial tasks of each application, and tries to move the communicating tasks connected to the initial task, based on a spring-connected weights model, where weights represent the communication volume between the tasks and the stiffness of the spring is determined by the communication cost function. The algorithm starts with a one-hop radius of the selected task, and checks whether migrating a task improves the cost function or not. If not, the radius is incremented by one. The tabu search is used to penalise the task each time its migration is unsuccessful. It also sets a threshold for the number of times a task can be migrated in order to avoid excessive migrations. The tabu list should be added to the Application Directory where the task migration algorithm can check to see if a task can be migrated or not. The cost function in conjunction with the algorithm itself would be added as a sub-module inside the Trafficmanager class and it should be used inside the Central Controller module.

As shown above, our simulator provides simple interfaces for acquiring the position and communication costs of the applications using the Application Directory.

The "Migrate" interface can be used to trigger task migrations and policies can be added to the Central Controller with minimal changes to the modules of the simulator.

4.6 Summary

We investigated different mapping scenarios in this chapter. We demonstrated how our simulator captures the effects of contention and fragmentation. With the various statistics and feedback that the simulator provides, it is possible to study in detail various performance metrics for different mapping scenarios. Furthermore, we mapped 30 applications to a system with 63 processing tiles and 1 memory controller tile and ran the simulation for 4 million cycles that took less than 5 minutes to finish. Having a separate module for application mapping allows different mappings to be readily investigated. Moreover, we demonstrated how our simulator could be used to study task migration and how capturing the response time and migration delays of interrupting and moving of tasks can be used to compare the performance of migration policies.

Chapter 5

Conclusion and Future Directions

5.1 Concluding remarks

Task migration is a powerful tool that can be utilised to improve application mappings at runtime. This thesis introduces a simulator that simulates the execution of applications represented by **Synchronous Data Flow (SDF)** model of computation on a 2D-mesh network-on-chip homogeneous many-core system. Our simulator can be utilised to study the effects of different runtime mapping and task migration methodologies on execution time and throughput of applications. We demonstrated how our simulator can simulate the results of contention and fragmentation that is caused by suboptimal mappings and how task migration can improve the execution time and throughput of the applications while realistic costs for task migration are taken into account.

In order to develop and implement our simulator, the following accomplishments were achieved:

- Extended the SDF model to realistically incorporate off-chip memory accesses and account for local memory occupancy before firing;
- Extended and adapted the well-known Booksim2 simulator for the purpose of simulating the execution of applications represented in SDFG form on 2D mesh-of-processor architectures. This simulator allows network traffic to be accurately simulated. The impact of allocation and migration decisions can

thereby be closely studied.

5.2 Future directions

As for future work we intend to extend the simulator by adding more expressive application models. The work reported in [54] offers a model for a more accurate data access pattern for the **SDF** application model where actors consume a fraction of the data for execution rather than waiting for the whole data block to arrive. By incorporating this work into our application model the communication pattern of our application model will be less bursty and more representative of real applications. As mentioned before, the **Cyclo-Static Data Flow** and the **Boolean Data Flow** models, that are extensions of **SDF**, can be used to model a finite sequence of rates, rather than fixed rates, and allow data dependant conditional behaviour respectively.

On the architecture level, we intend to implement a more detailed memory controller and off-chip memory modules in order to simulate the memory accesses more accurately. Furthermore, adding multi-tasking functionality to our cores and integrating mechanisms for the study of other optimisation criteria like power and thermal profiling are other possible future directions for this work. The work in [25] offers a model for energy consumption and task migration overhead of **SDF** applications that can be applied to our work. Moreover, we intend to investigate distributed control methods in order to make our controller scalable.

Finally, the response time of the task migration command in our simulator depends on the iteration latency of the application. By implementing a more sophisticated task migration mechanism we could improve our response time to task migration requests.

Bibliography

[1] Kalray MPPA MANYCORE. `http://www.kalray.eu/products/mppa-manycore/`. [Online; accessed September, 2014].

[2] Tilera Tile-Gx Processor Family. `http://www.tilera.com/products/processors/TILE-Gx_Family`. [Online; accessed September, 2014].

[3] TGG: Task Graph Generator. `http://sourceforge.net/projects/taskgraphgen/`, 2010. [Online; accessed September, 2014].

[4] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[5] Iraklis Anagnostopoulos, Alexandros Bartzas, Georgios Kathareios, and Dimitrios Soudris. A divide and conquer based distributed run-time mapping methodology for many-core platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 111–116. IEEE, 2012.

[6] Iraklis Anagnostopoulos, Vasileios Tsoutsouras, Alexandros Bartzas, and Dimitrios Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *Proceedings of the 50th Annual Design Automation Conference*, page 168. ACM, 2013.

[7] Acquaviva Andrea, Alimonda Andrea, Carta Salvatore, and Pittau Michele. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP Journal on Embedded Systems*, 2008, 2008.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.

[9] Luca Benini, Davide Bertozzi, and Michela Milano. Resource management policy handling multiple use-cases in MPSoC platforms using constraint programming. In *Logic Programming*, pages 470–484. Springer, 2008.

[10] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, Automation and Test in Europe: Proceedings*, pages 15–20. European Design and Automation Association, 2006.

[11] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[13] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[15] Eduardo Wenzel Brião, Daniel Barcelos, and Flávio Rech Wagner. Dynamic task allocation strategies in MPSoC for soft real-time applications. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 1386–1389. ACM, 2008.

[16] Joseph Tobin Buck and Edward A Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432. IEEE, 1993.

[17] Everton A Carara, Roberto P de Oliveira, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Hemps- A framework for NoC-based MPSoC generation. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1345–1348. IEEE, 2009.

[18] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[19] Ewerson Carvalho and Fernando Moraes. Congestion-aware task mapping in heterogeneous MPSoCs. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4. IEEE, 2008.

[20] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs. *Industrial Informatics, IEEE Transactions on*, 9(1):527–545, 2013.

[21] Chen-Ling Chou and Radu Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 164–169. IEEE, 2008.

[22] Chen-Ling Chou, Umit Y Ogras, and Radu Marculescu. Energy and performance-aware incremental mapping for networks on chip with multiple voltage levels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1866–1879, 2008.

[23] Jason Cong, Karthik Gururaj, Peng Zhang, and Yi Zou. Task-level data model for hardware synthesis based on concurrent collections. *Journal of Electrical and Computer Engineering*, 2012:6, 2012.

[24] Yingnan Cui, Wei Zhang, and Hao Yu. Decentralized agent based re-clustering for task mapping of tera-scale network-on-chip system. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 2437–2440. IEEE, 2012.

[25] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Energy-aware task mapping and scheduling for reliable embedded computing systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2s):72, 2014.

[26] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[27] Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pages 97–101. IEEE Computer Society, 1998.

[28] A. Ghamarian. *Timing Analysis of Synchronous Data Flow Graphs*. PhD thesis, Eindhoven University of Technology, 2008.

[29] Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, AJM Moonen, Marco JG Bekooij, Bart D Theelen, and MohammadReza Mousavi. Throughput analysis of synchronous data flow graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36. IEEE, 2006.

[30] Kahn Gilles. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[31] Lee Kee Goh and Bharadwaj Veeravalli. Design and performance evaluation of combined first-fit task allocation and migration strategies in mesh multiprocessor systems. *Parallel Computing*, 34(9):508–520, 2008.

[32] B Goodarzi and H Sarbazi-Azad. Task migration in mesh NoCs over virtual point-to-point connections. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 463–469. IEEE, 2011.

[33] Simon Holmbacka, Mohammad Fattah, Wictor Lund, Amir-Mohammad Rahmani, Sébastien Lafond, and Johan Lilius. A task migration mechanism for distributed many-core operating systems. *The Journal of Supercomputing*, pages 1–22, 2014.

[34] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Ape, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[35] Jia Huang, Andreas Raabe, Christian Buckl, and Alois Knoll. A workflow for runtime adaptive task allocation on heterogeneous MPSoCs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[36] Janmartin Jahn, MAA Faruque, and Jörg Henkel. CARAT: Context-aware runtime adaptive task migration for multi core architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[37] Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, and Jörg Henkel. Optimizations for configuring and mapping software pipelines in many core systems. In *Proceedings of the 50th Annual Design Automation Conference*, page 130. ACM, 2013.

[38] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, DE Shaw, John Kim, and WJ Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and*

Software (ISPASS), 2013 IEEE International Symposium on, pages 86–96. IEEE, 2013.

[39] David S Johnson and M Garey. Computers and Intractability: A guide to the theory of NP-completeness. *Freeman&Co, San Francisco*, 1979.

[40] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. Distrm: Distributed Resource Management for On-Chip Many-Core Systems. In *Proceedings of the seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 119–128. ACM, 2011.

[41] Sunil Kumar, Tommaso Cucinotta, and Giuseppe Lipari. A latency simulator for many-core systems. In *Proceedings of the 44th Annual Simulation Symposium*, pages 151–158. Society for Computer Simulation International, 2011.

[42] Edward Lee. Consistency in dataflow graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):223–235, 1991.

[43] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[44] Edward A Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. *IEE Proceedings-Computers and Digital Techniques*, 152(2):239–250, 2005.

[45] Mieszko Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, Christopher W Fletcher, Omer Khan, and Srinivas Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 175–185. IEEE, 2011.

[46] Radu Marculescu, Umit Y Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):3–21, 2009.

[47] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[48] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Soonhoi Ha, Chanhee Lee, Qiang Xu, and Lin Huang. Mapping of applications to MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 109–118. ACM, 2011.

[49] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[50] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[51] Davit Mirzoyan, Benny Akesson, and Kees Goossens. Process-variation-aware mapping of best-effort and real-time streaming applications to MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2s):61, 2014.

[52] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[53] Vincent Nollet, Théodore Marescaux, Prabhat Avasare, D Verkest, and J-Y Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234–239. IEEE, 2005.

[54] Hyunok Oh and Soonhoi Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI signal processing systems for signal, image and video technology*, 37(1):41–51, 2004.

[55] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 71–80. ACM, 2012.

[56] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[57] Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. *Journal of Systems Architecture*, 56(7):242–255, 2010.

[58] Jonathan M Smith. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[59] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[60] Sander Stuijk. *Predictable mapping of streaming applications on multiprocessors*. PhD thesis, TU Eindhoven, 2007.

[61] Sander Stuijk. SDFG of real applications . `http://www.es.ele.tue.nl/sdf3/download/examples.php`, 2014. [Online; accessed September, 2014].

[62] Pranav Tendulkar and Sander Stuijk. A Case Study into Predictable and Composable MPSoC Reconfiguration. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 293–300. IEEE, 2013.

[63] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[64] Andreas Weichslgartner, Stefan Wildermann, and Jürgen Teich. Dynamic decentralized mapping of tree-structured applications on NoC architectures. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 201–208. IEEE, 2011.

[65] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[66] Gwo-Jong Yu, Chih-Yung Chang, and Tzung-Shi Chen. Task migration in n-dimensional wormhole-routed mesh multicomputers. *Journal of Systems Architecture*, 50(4):177–192, 2004.

[67] Peter Zipf, Gilles Sassatelli, Nurten Utlu, Nicolas Saint-Jean, Pascal Benoit, and Manfred Glesner. A decentralised task mapping approach for homogeneous multiprocessor network-on-chips. *International Journal of Reconfigurable Computing*, 2009:3, 2009.

Appendix A

SDFGs XML descriptions

The XML descriptions of our SDFGs from [56] and [61] are given below:

- H.263 decoder used in Section 4.1 and Section 4.4:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='h263decoder'>
    <sdf name="h263decoder" type="H263decoder">
      <actor name="vld" type="A0">
        <port name="p0" type="out" rate="594"/>
        <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="1"/>
        <port name="_p4" type="in" rate="594"/>
      </actor>
      <actor name="iq" type="A1">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="_p5" type="out" rate="1"/>
      </actor>
      <actor name="idct" type="A2">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
      </actor>
      <actor name="mc" type="A3">
        <port name="p0" type="in" rate="594"/>
        <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="1"/>
      </actor>
      <channel name="vld2iq" srcActor="vld" srcPort="p0" dstActor="iq"
          dstPort="p0"/>
      <channel name="iq2idct" srcActor="iq" srcPort="p1" dstActor="idct"
          dstPort="p0"/>
      <channel name="idct2mc" srcActor="idct" srcPort="p1" dstActor="mc"
          dstPort="p0"/>
      <channel name="vld2vld" srcActor="vld" srcPort="p2" dstActor="vld"
          dstPort="p1" initialTokens='1'/>
      <channel name="iq2iq" srcActor="iq" srcPort="p3" dstActor="iq" dstPort=
          "p2" initialTokens='1'/>
```

```
    <channel name="mc2mc" srcActor="mc" srcPort="p2" dstActor="mc" dstPort=
        "p1" initialTokens='1'/>
    <channel name="_vld2iqb" srcActor="iq" srcPort="_p5" dstActor="vld"
        dstPort="_p4" initialTokens="623"/>
  </sdf>
  <sdfProperties>
    <actorProperties actor="vld">
      <processor type="proc_0" default="true">
        <executionTime time="26018"/>
        <memory>
          <stateSize max="10848"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="1"/>
        <memory>
          <stateSize max="10848"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="iq">
      <processor type="proc_0" default="true">
        <executionTime time="559"/>
        <memory>
          <stateSize max="1"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="1"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="idct">
      <processor type="proc_0" default="true">
        <executionTime time="1"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="1"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="mc">
      <processor type="proc_0" default="true">
        <executionTime time="10958"/>
        <memory>
          <stateSize max="8000"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="5479"/>
        <memory>
          <stateSize max="8000"/>
        </memory>
      </processor>
    </actorProperties>
    <channelProperties channel="vld2iq">
      <tokenSize sz="512"/>
    </channelProperties>
    <channelProperties channel="iq2idct">
```

```
      <tokenSize sz="512"/>
    </channelProperties>
    <channelProperties channel="idct2mc">
      <tokenSize sz="512"/>
    </channelProperties>
    <channelProperties channel="vld2vld">
      <tokenSize sz="8192"/>
    </channelProperties>
    <channelProperties channel="iq2iq">
      <tokenSize sz="512"/>
    </channelProperties>
    <channelProperties channel="mc2mc">
      <tokenSize sz="304128"/>
    </channelProperties>
    <channelProperties channel="_vld2iqb"/>
      <tokenSize sz="512"/>
    </channelProperties>
    <graphProperties>
    <timeConstraints>
        <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
            MHz clock -->
    </timeConstraints>
    </graphProperties>
  </sdfProperties>
 </applicationGraph>
</sdf3>
```

- H.263 decoder used in Section 4.3:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='h263decoder'>
    <sdf name="h263decoder" type="H263decoder">
      <actor name="vld" type="A0">
        <port name="p0" type="out" rate="594"/>
        <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="1"/>
        <port name="_p4" type="in" rate="594"/>
      </actor>
      <actor name="iq" type="A1">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="_p5" type="out" rate="1"/>
        <port name="_p6" type="in" rate="1"/>
      </actor>
      <actor name="idct" type="A2">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="_p3" type="out" rate="1"/>
      </actor>
      <actor name="mc" type="A3">
        <port name="p0" type="in" rate="594"/>
        <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="1"/>
      </actor>
      <channel name="vld2iq" srcActor="vld" srcPort="p0" dstActor="iq"
          dstPort="p0"/>
      <channel name="iq2idct" srcActor="iq" srcPort="p1" dstActor="idct"
          dstPort="p0"/>
      <channel name="idct2mc" srcActor="idct" srcPort="p1" dstActor="mc"
          dstPort="p0"/>
```

```
    <channel name="vld2vld" srcActor="vld" srcPort="p2" dstActor="vld"
        dstPort="p1" initialTokens='1'/>
    <channel name="iq2iq" srcActor="iq" srcPort="p3" dstActor="iq" dstPort=
        "p2" initialTokens='1'/>
    <channel name="mc2mc" srcActor="mc" srcPort="p2" dstActor="mc" dstPort=
        "p1" initialTokens='1'/>
    <channel name="_vld2iqb" srcActor="iq" srcPort="_p5" dstActor="vld"
        dstPort="_p4" initialTokens="623"/>
    <channel name="_iq2idctb" srcActor="idct" srcPort="_p3" dstActor="iq"
        dstPort="_p6" initialTokens="2"/>
  </sdf>
  <sdfProperties>
    <actorProperties actor="vld">
      <processor type="proc_0" default="true">
        <executionTime time="26018"/>
        <memory>
          <stateSize max="10848"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="13009"/>
        <memory>
          <stateSize max="10848"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="iq">
      <processor type="proc_0" default="true">
        <executionTime time="559"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="450"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="idct">
      <processor type="proc_0" default="true">
        <executionTime time="486"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="355"/>
        <memory>
          <stateSize max="400"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="mc">
      <processor type="proc_0" default="true">
        <executionTime time="10958"/>
        <memory>
          <stateSize max="8000"/>
        </memory>
      </processor>
      <processor type="proc_1" default="true">
        <executionTime time="5479"/>
        <memory>
          <stateSize max="8000"/>
        </memory>
```

```
        </processor>
      </actorProperties>
      <channelProperties channel="vld2iq">
        <tokenSize sz="512"/>
      </channelProperties>
      <channelProperties channel="iq2idct">
        <tokenSize sz="512"/>
      </channelProperties>
      <channelProperties channel="idct2mc">
        <tokenSize sz="512"/>
      </channelProperties>
      <channelProperties channel="vld2vld">
        <tokenSize sz="8192"/>
      </channelProperties>
      <channelProperties channel="iq2iq">
        <tokenSize sz="512"/>
      </channelProperties>
      <channelProperties channel="mc2mc">
        <tokenSize sz="304128"/>
      </channelProperties>
      <channelProperties channel="_vld2iqb"/>
        <tokenSize sz="512"/>
      </channelProperties>
      <channelProperties channel="_iq2idctb"/>
        <tokenSize sz="512"/>
      </channelProperties>
      <graphProperties>
      <timeConstraints>
          <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
              MHz clock -->
      </timeConstraints>
      </graphProperties>
    </sdfProperties>
  </applicationGraph>
</sdf3>
```

- H.263 encoder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
          .xsd">
  <applicationGraph name='h263encoder'>
    <sdf name="h263encoder" type="H263encoder">
        <actor name='me' type='a'>
            <port type='in' name='p0' rate='1'/>
            <port type='out' name='p1' rate='99'/>
        </actor>
        <actor name='mbc' type='a'>
            <port type='in' name='p0' rate='1'/>
            <port type='out' name='p1' rate='1'/>
            <port type='out' name='p2' rate='1'/>
        </actor>
        <actor name='vlc' type='a'>
            <port type='in' name='p0' rate='99'/>
            <port type='in' name='p1' rate='1'/>
            <port type='out' name='p2' rate='1'/>
        </actor>
        <actor name='mbd' type='a'>
            <port type='in' name='p0' rate='1'/>
            <port type='out' name='p1' rate='1'/>
        </actor>
        <actor name='mc' type='a'>
            <port type='in' name='p0' rate='99'/>
            <port type='out' name='p1' rate='1'/>
```

```
            <port type='in' name='p2' rate='1'/>
            <port type='out' name='p3' rate='1'/>
        </actor>
        <channel name='mc2me' srcActor='mc' srcPort='p1' dstActor='me'
             dstPort='p0' initialTokens='1'/>
        <channel name='me2mbc' srcActor='me' srcPort='p1' dstActor='mbc'
             dstPort='p0'/>
        <channel name='mbc2vlc' srcActor='mbc' srcPort='p1' dstActor='vlc'
             dstPort='p0'/>
        <channel name='mbc2mbd' srcActor='mbc' srcPort='p2' dstActor='mbd'
             dstPort='p0' />
        <channel name='mbd2mc' srcActor='mbd' srcPort='p1' dstActor='mc'
             dstPort='p0'/>
        <channel name='vlc2vlc' srcActor='vlc' srcPort='p2' dstActor='vlc'
             dstPort='p1' initialTokens='1'/>
        <channel name='mc2mc' srcActor='mc' srcPort='p3' dstActor='mc'
             dstPort='p2' initialTokens='1'/>
    </sdf>
    <sdfProperties>
        <actorProperties actor='me'>
            <processor type='RH'>
                <executionTime time='191'/>
                <memory>
                    <stateSize max="316352"/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='mbc'>
            <processor type='RH' default='true'>
                <executionTime time='8'/>
                <memory>
                    <stateSize max="17728"/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='vlc'>
            <processor type='RH' default='true'>
                <executionTime time='13'/>
                <memory>
                    <stateSize max="10848"/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='mbd'>
            <processor type='RH' default='true'>
                <executionTime time='6'/>
                <memory>
                    <stateSize max="6912"/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='mc'>
            <processor type='RH' default='true'>
                <executionTime time='5'/>
                <memory>
                    <stateSize max="22368"/>
                </memory>
            </processor>
        </actorProperties>
        <channelProperties channel="mc2me">
            <tokenSize sz="304128"/>
        </channelProperties>
        <channelProperties channel="me2mbc">
            <tokenSize sz="3072"/>
        </channelProperties>
        <channelProperties channel="mbc2vlc">
```

```
            <tokenSize sz="3072"/>
          </channelProperties>
          <channelProperties channel="mbc2mbd">
            <tokenSize sz="3072"/>
          </channelProperties>
          <channelProperties channel="mbd2mc">
            <tokenSize sz="3072"/>
          </channelProperties>
          <channelProperties channel="vlc2vlc">
            <tokenSize sz="8192"/>
          </channelProperties>
          <channelProperties channel="mc2mc">
            <tokenSize sz="304128"/>
          </channelProperties>
        <graphProperties>
          <timeConstraints>
            <throughput>0.00000003</throughput> <!-- 15fps (iterations) with
                500MHz clock -->
          </timeConstraints>
        </graphProperties>
      </sdfProperties>
  </applicationGraph>
  </sdf3>
```

- H.264 decoder

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='h264decoder'>
    <sdf name="h264decoder" type="H264decoder">
      <actor name="ed" type="A0">
                <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="256"/>
      </actor>
      <actor name="idctRecon" type="A1">
        <port name="p0" type="in" rate="16"/>
        <port name="p1" type="out" rate="16"/>
      </actor>
      <actor name="mc" type="A2">
        <port name="p0" type="in" rate="256"/>
        <port name="p1" type="out" rate="1"/>
      </actor>
      <channel name="ed2idct" srcActor="ed" srcPort="p1" dstActor="idctRecon"
          dstPort="p0"/>
      <channel name="idct2mc" srcActor="idctRecon" srcPort="p1" dstActor="mc"
          dstPort="p0"/>
      <channel name="mc2ed" srcActor="mc" srcPort="p1" dstActor="ed" dstPort=
          "p0" initialTokens='1'/>
    </sdf>
    <sdfProperties>
      <actorProperties actor="ed">
        <processor type="proc_0" default="true">
          <executionTime time="120"/>
          <memory>
            <stateSize max="14304"/>
          </memory>
        </processor>
      </actorProperties>
      <actorProperties actor="idctRecon">
        <processor type="proc_0" default="true">
          <executionTime time="2"/>
          <memory>
            <stateSize max="32768"/>
```

```
            </memory>
          </processor>
        </actorProperties>
        <actorProperties actor="mc">
          <processor type="proc_0" default="true">
            <executionTime time="9"/>
            <memory>
              <stateSize max="2048"/>
            </memory>
          </processor>
        </actorProperties>
        <channelProperties channel="ed2idct">
          <tokenSize sz="128"/>
        </channelProperties>
        <channelProperties channel="idct2mc">
          <tokenSize sz="128"/>
        </channelProperties>
        <channelProperties channel="mc2ed">
          <tokenSize sz="32768"/>
        </channelProperties>
        <graphProperties>
        <timeConstraints>
            <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
                MHz clock -->
        </timeConstraints>
        </graphProperties>
      </sdfProperties>
    </applicationGraph>
  </sdf3>
```

- H.264 encoder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='h264encoder'>
    <sdf name="h264encoder" type="H264encoder">
      <actor name="md" type="A0">
                <port name="p0" type="in" rate="1"/>
                <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="256"/>
      </actor>
      <actor name="dct" type="A1">
        <port name="p0" type="in" rate="16"/>
        <port name="p1" type="out" rate="16"/>
      </actor>
      <actor name="iq" type="A2">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
      </actor>
          <actor name="idct" type="A3">
        <port name="p0" type="in" rate="16"/>
        <port name="p1" type="out" rate="16"/>
      </actor>
          <actor name="rc" type="A4">
        <port name="p0" type="in" rate="256"/>
        <port name="p1" type="out" rate="1"/>
                <port name="p2" type="out" rate="1"/>
      </actor>
          <actor name="ip" type="A5">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
      </actor>
          <actor name="ime" type="A6">
```

```
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
  </actor>
      <actor name="intpol" type="A7">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
  </actor>
      <actor name="fme" type="A8">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
  </actor>
  <channel name="md2dct" srcActor="md" srcPort="p2" dstActor="dct"
       dstPort="p0"/>
  <channel name="dct2iq" srcActor="dct" srcPort="p1" dstActor="iq"
       dstPort="p0"/>
  <channel name="iq2idct" srcActor="iq" srcPort="p1" dstActor="idct"
       dstPort="p0"/>
      <channel name="idct2rc" srcActor="idct" srcPort="p1" dstActor="rc"
           dstPort="p0"/>
      <channel name="rc2ip" srcActor="rc" srcPort="p1" dstActor="ip"
           dstPort="p0"/>
      <channel name="ip2md" srcActor="ip" srcPort="p1" dstActor="md"
           dstPort="p1" initialTokens='1'/>
      <channel name="rc2ime" srcActor="rc" srcPort="p2" dstActor="ime"
           dstPort="p0"/>
      <channel name="ime2intpol" srcActor="ime" srcPort="p1" dstActor="
           intpol" dstPort="p0"/>
      <channel name="intpol2fme" srcActor="intpol" srcPort="p1" dstActor=
           "fme" dstPort="p0"/>
      <channel name="fme2md" srcActor="fme" srcPort="p1" dstActor="md"
           dstPort="p0" initialTokens='1'/>
</sdf>
<sdfProperties>
  <actorProperties actor="md">
    <processor type="proc_0" default="true">
      <executionTime time="27"/>
      <memory>
        <stateSize max="8000"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor="dct">
    <processor type="proc_0" default="true">
      <executionTime time="1"/>
      <memory>
        <stateSize max="32768"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor="iq">
    <processor type="proc_0" default="true">
      <executionTime time="3"/>
      <memory>
        <stateSize max="6688"/>
      </memory>
    </processor>
  </actorProperties>
      <actorProperties actor="idct">
    <processor type="proc_0" default="true">
      <executionTime time="3"/>
      <memory>
        <stateSize max="32768"/>
      </memory>
    </processor>
  </actorProperties>
      <actorProperties actor="rc">
```

```
    <processor type="proc_0" default="true">
      <executionTime time="21"/>
      <memory>
        <stateSize max="8000"/>
      </memory>
    </processor>
  </actorProperties>
       <actorProperties actor="ip">
    <processor type="proc_0" default="true">
      <executionTime time="5"/>
      <memory>
        <stateSize max="4096"/>
      </memory>
    </processor>
  </actorProperties>
       <actorProperties actor="ime">
    <processor type="proc_0" default="true">
      <executionTime time="4"/>
      <memory>
        <stateSize max="8000"/>
      </memory>
    </processor>
  </actorProperties>
       <actorProperties actor="intpol">
    <processor type="proc_0" default="true">
      <executionTime time="1"/>
      <memory>
        <stateSize max="8192"/>
      </memory>
    </processor>
  </actorProperties>
       <actorProperties actor="fme">
    <processor type="proc_0" default="true">
      <executionTime time="2"/>
      <memory>
        <stateSize max="4096"/>
      </memory>
    </processor>
  </actorProperties>
  <channelProperties channel="md2dct">
    <tokenSize sz="16"/>
  </channelProperties>
  <channelProperties channel="dct2iq">
    <tokenSize sz="16"/>
  </channelProperties>
  <channelProperties channel="iq2idct">
    <tokenSize sz="16"/>
  </channelProperties>
  <channelProperties channel="idct2rc">
    <tokenSize sz="16"/>
  </channelProperties>
  <channelProperties channel="rc2ip">
    <tokenSize sz="2048"/>
  </channelProperties>
  <channelProperties channel="ip2md">
    <tokenSize sz="2048"/>
  </channelProperties>
  <channelProperties channel="rc2ime">
    <tokenSize sz="2048"/>
  </channelProperties>
  <channelProperties channel="ime2intpol">
    <tokenSize sz="2048"/>
  </channelProperties>
  <channelProperties channel="intpol2fme">
    <tokenSize sz="2048"/>
  </channelProperties>
```

```
        <channelProperties channel="fme2md">
          <tokenSize sz="2048"/>
        </channelProperties>
        <graphProperties>
        <timeConstraints>
            <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
                MHz clock -->
        </timeConstraints>
        </graphProperties>
      </sdfProperties>
    </applicationGraph>
  </sdf3>
```

- JPEG decoder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='jpegdecoder'>
    <sdf name="JPEGdecoder" type="G">
        <actor name="a" type="a">
            <port name="IN" type="in" rate="1"/>
            <port name="OUT" type="out" rate="6"/>
          </actor>
          <actor name="b" type="b">
            <port name="IN" type="in" rate="1"/>
            <port name="OUT" type="out" rate="1"/>
          </actor>
          <actor name="c" type="c">
            <port name="IN" type="in" rate="1"/>
            <port name="OUT" type="out" rate="1"/>
          </actor>
          <actor name="d" type="d">
            <port name="IN" type="in" rate="1"/>
            <port name="OUT" type="out" rate="1"/>
          </actor>
          <actor name="e" type="e">
            <port name="IN" type="in" rate="1"/>
            <port name="OUT" type="out" rate="1"/>
          </actor>
          <actor name="f" type="f">
            <port name="IN" type="in" rate="6"/>
            <port name="OUT" type="out" rate="1"/>
          </actor>
          <channel name="f2a" srcActor="f" srcPort="OUT" dstActor="a" dstPort
              ="IN" initialTokens="1"/>
          <channel name="a2b" srcActor="a" srcPort="OUT" dstActor="b" dstPort
              ="IN" />
          <channel name="b2c" srcActor="b" srcPort="OUT" dstActor="c" dstPort
              ="IN" />
          <channel name="c2d" srcActor="c" srcPort="OUT" dstActor="d" dstPort
              ="IN" />
          <channel name="d2e" srcActor="d" srcPort="OUT" dstActor="e" dstPort
              ="IN" />
          <channel name="e2f" srcActor="e" srcPort="OUT" dstActor="f" dstPort
              ="IN" />
        </sdf>
        <sdfProperties>
        <actorProperties actor="a">
          <processor type="proc_0" default="true">
          <executionTime time="28"/>
          <memory>
            <stateSize max="9"/>
          </memory>
```

```
  </processor>
    </actorProperties>
    <actorProperties actor="b">
  <processor type="proc_0" default="true">
    <executionTime time="2"/>
    <memory>
      <stateSize max="9"/>
    </memory>
  </processor>
    </actorProperties>
    <actorProperties actor="c">
      <processor type="proc_0" default="true">
                <executionTime time="2"/>
                <memory>
                  <stateSize max="9"/>
                </memory>
  </processor>
    </actorProperties>
    <actorProperties actor="d" >
          <processor type="proc_0" default="true">
                <executionTime time="7"/>
                <memory>
                  <stateSize max="9"/>
                </memory>
  </processor>
    </actorProperties>
    <actorProperties actor="e" >
          <processor type="proc_0" default="true">
                <executionTime time="2"/>
                <memory>
                  <stateSize max="9"/>
                </memory>
  </processor>
    </actorProperties>
    <actorProperties actor="f">
                  <processor type="proc_0" default="true">
                <executionTime time="36"/>
                <memory>
                  <stateSize max="9"/>
                </memory>
   </processor>
 </actorProperties>
  <channelProperties channel="a2b">
  <tokenSize sz="512"/>
</channelProperties>
<channelProperties channel="b2c">
  <tokenSize sz="512"/>
</channelProperties>
<channelProperties channel="c2d">
  <tokenSize sz="512"/>
</channelProperties>
<channelProperties channel="d2e">
  <tokenSize sz="512"/>
</channelProperties>
<channelProperties channel="e2f">
  <tokenSize sz="512"/>
</channelProperties>
<channelProperties channel="f2a">
  <tokenSize sz="3072"/>
</channelProperties>
<graphProperties>
<timeConstraints>
    <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
        MHz clock -->
</timeConstraints>
</graphProperties>
```

```
        </sdfProperties>
    </applicationGraph>
</sdf3>
```

- MPEG4 decoder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name='mpegdecoder'>
    <sdf name="MPEGdecoder" type="G">
        <actor name="fd" type="A0">
                <port name="p1" type="out" rate="99"/>
                <port name="p2" type="out" rate="1"/>
                <port name="p3" type="in" rate="1"/>
            </actor>
            <actor name="vld" type="A1">
                <port name="p0" type="out" rate="1"/>
                <port name="p1" type="in" rate="1"/>
                <port name="p2" type="out" rate="1"/>
                <port name="p3" type="out" rate="1"/>
                <port name="p4" type="in" rate="1"/>
            </actor>
            <actor name="idct" type="A2">
                <port name="p0" type="out" rate="1"/>
                <port name="p1" type="in" rate="1"/>
                <port name="p2" type="out" rate="1"/>
                <port name="p3" type="in" rate="1"/>
            </actor>
            <actor name="rc" type="A3">
                <port name="p0" type="out" rate="1"/>
                <port name="p1" type="in" rate="1"/>
                <port name="p3" type="out" rate="1"/>
                <port name="p4" type="in" rate="1"/>
                <port name="p5" type="out" rate="1"/>
                <port name="p6" type="in" rate="99"/>
            </actor>
            <actor name="mc" type="A4">
                <port name="p0" type="in" rate="99"/>
                <port name="p1" type="in" rate="1"/>
                <port name="p2" type="out" rate="1"/>
                <port name="p3" type="in" rate="1"/>
                <port name="p4" type="out" rate="1"/>
                <port name="p5" type="in" rate="1"/>
            </actor>
          <channel name="fd2vld" srcActor="fd" srcPort="p1" dstActor="vld"
              dstPort="p4"/>
            <channel name="fd2mc" srcActor="fd" srcPort="p2" dstActor="mc"
                dstPort="p3"/>
            <channel name="vld2idct" srcActor="vld" srcPort="p2" dstActor="
                idct" dstPort="p3"/>
            <channel name="vld2mc" srcActor="vld" srcPort="p3" dstActor="mc
                " dstPort="p0"/>
            <channel name="idct2rc" srcActor="idct" srcPort="p2" dstActor="
                rc" dstPort="p6"/>
            <channel name="rc2mc" srcActor="rc" srcPort="p5" dstActor="mc"
                dstPort="p1" initialTokens='1'/>
            <channel name="rc2fd" srcActor="rc" srcPort="p3" dstActor="fd"
                dstPort="p3" initialTokens='3'/>
            <channel name="mc2rc" srcActor="mc" srcPort="p2" dstActor="rc"
                dstPort="p4"/>
            <channel name="vld2vld" srcActor="vld" srcPort="p0" dstActor="
                vld" dstPort="p1" initialTokens='1'/>
```

```
        <channel name="idct2idct" srcActor="idct" srcPort="p0" dstActor
            ="idct" dstPort="p1" initialTokens='1'/>
        <channel name="rc2rc" srcActor="rc" srcPort="p0" dstActor="rc"
            dstPort="p1" initialTokens='1'/>
        <channel name="mc2mc" srcActor="mc" srcPort="p4" dstActor="mc"
            dstPort="p5" initialTokens='1'/>
</sdf>
<sdfProperties>
<actorProperties actor='fd'>
            <processor type='proc_0' default='true'>
                <executionTime time='25'/>
                <!-- worst-case execution time in time units -->
                <memory>
                    <stateSize max='10848'/>
                    <!-- worst-case state size in bytes -->
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='vld'>
            <processor type='proc_0' default='true'>
                <executionTime time='16'/>
                <memory>
                    <stateSize max='400'/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='idct'>
            <processor type='proc_0' default='true'>
                <executionTime time='11'/>
                <memory>
                    <stateSize max='400'/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='rc'>
            <processor type='proc_0' default='true'>
                <executionTime time='35'/>
                <memory>
                    <stateSize max='400'/>
                </memory>
            </processor>
        </actorProperties>
        <actorProperties actor='mc'>
            <processor type='proc_0' default='true'>
                <executionTime time='34'/>
                <memory>
                    <stateSize max='8000'/>
                </memory>
            </processor>
        </actorProperties>
<channelProperties channel="fd2vld">
                <tokenSize sz="512"/>
            </channelProperties>
            <channelProperties channel="fd2mc">
                <tokenSize sz="512"/>
            </channelProperties>
            <channelProperties channel="vld2idct">
                <tokenSize sz="512"/>
            </channelProperties>
            <channelProperties channel="vld2mc">
                <tokenSize sz="512"/>
            </channelProperties>
            <channelProperties channel="idct2rc">
                <tokenSize sz="512"/>
            </channelProperties>
            <channelProperties channel="rc2mc">
```

```
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="rc2fd">
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="mc2rc">
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="vld2vld">
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="idct2idct">
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="rc2rc">
                        <tokenSize sz="50688"/>
                      </channelProperties>
                      <channelProperties channel="mc2mc">
                        <tokenSize sz="50688"/>
                      </channelProperties>
      <graphProperties>
      <timeConstraints>
          <throughput>0.00000003</throughput> <!--15fps (iterations) with 500
              MHz clock -->
      </timeConstraints>
      </graphProperties>

        </sdfProperties>
  </applicationGraph>
</sdf3>
```

- MP3 decoder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
         .xsd">
  <applicationGraph name='mp3decoder'>
    <sdf name='mp3decoder' type='MP3decoder'>
      <actor name='huffman' type='Huffman'>
          <port name='p0' type='out' rate='2'/>
          <port name='p1' type='out' rate='2'/>
          <port name='p2' type='in' rate='1'/>
          <port name='p3' type='out' rate='1'/>
          <port name="_p5" type="in" rate="2"/>
          <port name="_p6" type="in" rate="2"/>
      </actor>
      <actor name='req0' type='Req'>
          <port name='p0' type='in' rate='1'/>
          <port name='p1' type='out' rate='1'/>
          <port name='p2' type='in' rate='1'/>
          <port name='p3' type='out' rate='1'/>
          <port name="_p5" type="out" rate="1"/>
      </actor>
      <actor name='reorder0' type='Reorder'>
          <port name='p0' type='in' rate='1'/>
          <port name='p1' type='out' rate='1'/>
      </actor>
      <actor name='req1' type='Req'>
          <port name='p0' type='in' rate='1'/>
          <port name='p1' type='out' rate='1'/>
          <port name='p2' type='in' rate='1'/>
          <port name='p3' type='out' rate='1'/>
          <port name="_p5" type="out" rate="1"/>
      </actor>
```

```
<actor name='reorder1' type='Reorder'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='stereo' type='Stereo'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='in' rate='1'/>
    <port name='p2' type='out' rate='1'/>
    <port name='p3' type='out' rate='1'/>
</actor>
<actor name='antialias0' type='Antialias'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='hybridsynth0' type='HybridSynth'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='freqinv0' type='FreqInv'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='subbinv0' type='SubbInv'>
    <port name='p0' type='in' rate='1'/>
</actor>
<actor name='antialias1' type='Antialias'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='hybridsynth1' type='HybridSynth'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='freqinv1' type='FreqInv'>
    <port name='p0' type='in' rate='1'/>
    <port name='p1' type='out' rate='1'/>
</actor>
<actor name='subbinv1' type='SubbInv'>
    <port name='p0' type='in' rate='1'/>
</actor>
<channel name='huffman2req0' srcActor='huffman' srcPort='p0' dstActor='
    req0' dstPort='p0'/>
<channel name='huffman2req1' srcActor='huffman' srcPort='p1' dstActor='
    req1' dstPort='p0'/>
<channel name='req02reorder0' srcActor='req0' srcPort='p1' dstActor='
    reorder0' dstPort='p0'/>
<channel name='req12reorder1' srcActor='req1' srcPort='p1' dstActor='
    reorder1' dstPort='p0'/>
<channel name='reorder02stereo' srcActor='reorder0' srcPort='p1'
    dstActor='stereo' dstPort='p0'/>
<channel name='reorder12stereo' srcActor='reorder1' srcPort='p1'
    dstActor='stereo' dstPort='p1'/>
<channel name='stereo2antialias0' srcActor='stereo' srcPort='p2'
    dstActor='antialias0' dstPort='p0'/>
<channel name='stereo2antialias1' srcActor='stereo' srcPort='p3'
    dstActor='antialias1' dstPort='p0'/>
<channel name='antialias02hybridsynth0' srcActor='antialias0' srcPort='
    p1' dstActor='hybridsynth0' dstPort='p0'/>
<channel name='antialias12hybridsynth1' srcActor='antialias1' srcPort='
    p1' dstActor='hybridsynth1' dstPort='p0'/>
<channel name='hybridsynth02freqinv0' srcActor='hybridsynth0' srcPort='
    p1' dstActor='freqinv0' dstPort='p0'/>
<channel name='hybridsynth12freqinv1' srcActor='hybridsynth1' srcPort='
    p1' dstActor='freqinv1' dstPort='p0'/>
<channel name='freqinv02subbinv0' srcActor='freqinv0' srcPort='p1'
    dstActor='subbinv0' dstPort='p0'/>
```

```
    <channel name='freqinv12subbinv1' srcActor='freqinv1' srcPort='p1'
        dstActor='subbinv1' dstPort='p0'/>
    <channel name='huffman2huffman' srcActor='huffman' srcPort='p3'
        dstActor='huffman' dstPort='p2' initialTokens='1'/>
    <channel name='req02req0' srcActor='req0' srcPort='p3' dstActor='req0'
        dstPort='p2' initialTokens='1'/>
    <channel name='req12req1' srcActor='req1' srcPort='p3' dstActor='req1'
        dstPort='p2' initialTokens='1'/>
    <channel name="_huffman2req0b" srcActor="req0" srcPort="_p5" dstActor="
        huffman" dstPort="_p5" initialTokens="2"/>
    <channel name="_huffman2req1b" srcActor="req1" srcPort="_p5" dstActor="
        huffman" dstPort="_p6" initialTokens="2"/>
  </sdf>

<sdfProperties>
    <actorProperties actor='huffman'>
      <processor type="encoder" default="true">
        <executionTime time="75"/>
        <memory>
          <stateSize max="48544"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='req0'>
      <processor type="proc_0" default="true">
        <executionTime time="72"/>
        <memory>
          <stateSize max="832"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='reorder0'>
      <processor type="proc_0" default="true">
        <executionTime time="34"/>
        <memory>
          <stateSize max="18816"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='req1'>
      <processor type="proc_0" default="true">
        <executionTime time="72"/>
        <memory>
          <stateSize max="832"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='reorder1'>
      <processor type="proc_0" default="true">
        <executionTime time="34"/>
        <memory>
          <stateSize max="18816"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='stereo'>
      <processor type="proc_0" default="true">
        <executionTime time="53"/>
        <memory>
          <stateSize max="544"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor='antialias0'>
      <processor type="proc_0" default="true">
        <executionTime time="409"/>
```

```
      <memory>
        <stateSize max="5088"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='hybridsynth0'>
    <processor type="proc_0" default="true">
      <executionTime time="74"/>
      <memory>
        <stateSize max="80"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='freqinv0'>
    <processor type="proc_0" default="true">
      <executionTime time="49"/>
      <memory>
        <stateSize max="128"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='subbinv0'>
    <processor type="subbinv" default="true">
      <executionTime time="93"/>
      <memory>
        <stateSize max="3736"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='antialias1'>
    <processor type="proc_0" default="true">
      <executionTime time="4"/>
      <memory>
        <stateSize max="5088"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='hybridsynth1'>
    <processor type="proc_0" default="true">
      <executionTime time="7"/>
      <memory>
        <stateSize max="80"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='freqinv1'>
    <processor type="proc_0" default="true">
      <executionTime time="4"/>
      <memory>
        <stateSize max="128"/>
      </memory>
    </processor>
  </actorProperties>
  <actorProperties actor='subbinv1'>
    <processor type="subbinv" default="true">
      <executionTime time="93"/>
      <memory>
        <stateSize max="29888"/>
      </memory>
    </processor>
  </actorProperties>
  <channelProperties channel="huffman2req0">
    <bufferSize sz="2"/>
    <tokenSize sz="4608"/>
  </channelProperties>
  <channelProperties channel="huffman2req1">
```

```
                <bufferSize sz="2"/>
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="req02reorder0">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="req12reorder1">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="reorder02stereo">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="reorder12stereo">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="stereo2antialias0">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="stereo2antialias1">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="antialias02hybridsynth0">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="antialias12hybridsynth1">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="hybridsynth02freqinv0">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="hybridsynth12freqinv1">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="freqinv02subbinv0">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="freqinv12subbinv1">
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="huffman2huffman">
            <tokenSize sz="8192"/>
          </channelProperties>
          <channelProperties channel="req02req0">
            <tokenSize sz="512"/>
          </channelProperties>
          <channelProperties channel="req12req1">
            <tokenSize sz="512"/>
          </channelProperties>
          <channelProperties channel="_huffman2req0b"/>
            <tokenSize sz="4608"/>
          </channelProperties>
          <channelProperties channel="_huffman2req1b"/>
            <tokenSize sz="4608"/>
          </channelProperties>
          <graphProperties>
            <timeConstraints>
                    <throughput>0.000000026</throughput> <!-- 26ms per frame (
                        iteration) with 500MHz clock -->
            </timeConstraints>
          </graphProperties>
        </sdfProperties>
      </applicationGraph>
    </sdf3>
```

- Samplerate:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf
        .xsd">
  <applicationGraph name="samplerate">
    <sdf name="samplerate" type="Samplerate">
      <actor name="a" type="A">
        <port name="p1" type="out" rate="1"/>
        <port name="_p2" type="out" rate="1"/>
        <port name="_p3" type="in" rate="1"/>
        <port name="_p4" type="in" rate="1"/>
      </actor>
      <actor name="b" type="B">
        <port name="p1" type="in" rate="1"/>
        <port name="p2" type="out" rate="2"/>
        <port name="_p3" type="out" rate="1"/>
        <port name="_p4" type="in" rate="1"/>
        <port name="_p5" type="out" rate="1"/>
        <port name="_p6" type="in" rate="2"/>
      </actor>
      <actor name="c" type="C">
        <port name="p1" type="in" rate="3"/>
        <port name="p2" type="out" rate="2"/>
        <port name="_p3" type="out" rate="1"/>
        <port name="_p4" type="in" rate="1"/>
        <port name="_p5" type="out" rate="3"/>
        <port name="_p6" type="in" rate="2"/>
      </actor>
      <actor name="d" type="D">
        <port name="p1" type="in" rate="7"/>
        <port name="p2" type="out" rate="8"/>
        <port name="_p3" type="out" rate="1"/>
        <port name="_p4" type="in" rate="1"/>
        <port name="_p5" type="out" rate="7"/>
        <port name="_p6" type="in" rate="8"/>
      </actor>
      <actor name="e" type="E">
        <port name="p1" type="in" rate="7"/>
        <port name="p2" type="out" rate="5"/>
        <port name="_p3" type="out" rate="1"/>
        <port name="_p4" type="in" rate="1"/>
        <port name="_p5" type="out" rate="7"/>
        <port name="_p6" type="in" rate="5"/>
      </actor>
      <actor name="f" type="F">
        <port name="p1" type="in" rate="1"/>
        <port name="_p2" type="out" rate="1"/>
        <port name="_p3" type="in" rate="1"/>
      </actor>
      <channel name="a2b" srcActor="a" srcPort="p1" dstActor="b" dstPort="p1"
          />
      <channel name="b2c" srcActor="b" srcPort="p2" dstActor="c" dstPort="p1"
          />
      <channel name="c2d" srcActor="c" srcPort="p2" dstActor="d" dstPort="p1"
          />
      <channel name="d2e" srcActor="d" srcPort="p2" dstActor="e" dstPort="p1"
          />
      <channel name="e2f" srcActor="e" srcPort="p2" dstActor="f" dstPort="p1"
          />
      <channel name="a2a" srcActor="a" srcPort="_p2" dstActor="a" dstPort="
          _p3" initialTokens="1"/>
      <channel name="b2b" srcActor="b" srcPort="_p3" dstActor="b" dstPort="
          _p4" initialTokens="1"/>
      <channel name="c2c" srcActor="c" srcPort="_p3" dstActor="c" dstPort="
          _p4" initialTokens="1"/>
```

```
    <channel name="d2d" srcActor="d" srcPort="_p3" dstActor="d" dstPort="
        _p4" initialTokens="1"/>
    <channel name="e2e" srcActor="e" srcPort="_p3" dstActor="e" dstPort="
        _p4" initialTokens="1"/>
    <channel name="f2f" srcActor="f" srcPort="_p2" dstActor="f" dstPort="
        _p3" initialTokens="1"/>
    <channel name="_a2bb" srcActor="b" srcPort="_p5" dstActor="a" dstPort="
        _p4" initialTokens="1"/>
    <channel name="_b2cb" srcActor="c" srcPort="_p5" dstActor="b" dstPort="
        _p6" initialTokens="4"/>
    <channel name="_c2db" srcActor="d" srcPort="_p5" dstActor="c" dstPort="
        _p6" initialTokens="8"/>
    <channel name="_d2eb" srcActor="e" srcPort="_p5" dstActor="d" dstPort="
        _p6" initialTokens="14"/>
    <channel name="_e2fb" srcActor="f" srcPort="_p4" dstActor="e" dstPort="
        _p6" initialTokens="5"/>
  </sdf>
  <sdfProperties>
    <actorProperties actor="a">
      <processor type="proc_0" default="true">
        <executionTime time="50"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="b">
      <processor type="proc_0" default="true">
        <executionTime time="20"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="c">
      <processor type="proc_0" default="true">
        <executionTime time="30"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="d">
      <processor type="proc_0" default="true">
        <executionTime time="10"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="e">
      <processor type="proc_0" default="true">
        <executionTime time="40"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
    <actorProperties actor="f">
      <processor type="proc_0" default="true">
        <executionTime time="60"/>
        <memory>
          <stateSize max="100"/>
        </memory>
      </processor>
    </actorProperties>
```

```
        <channelProperties channel="a2b">
                <bufferSize sz="1" />
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="b2c">
                <bufferSize sz="4" />
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="c2d">
                <bufferSize sz="8" />
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="d2e">
                <bufferSize sz="14" />
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="e2f">
                <bufferSize sz="5" />
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="a2a">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="b2b">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="c2c">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="d2d">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="e2e">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="f2f">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="_a2bb">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="_b2cb">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="_c2db">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="_d2eb">
                <tokenSize sz="512"/>
        </channelProperties>
        <channelProperties channel="_e2fb">
                <tokenSize sz="512"/>
        </channelProperties>
        <graphProperties/>
    </sdfProperties>
  </applicationGraph>
</sdf3>
```

Appendix B

Application Mapping Order

The order of mapping applications in the simulation in Section 4.3 was as listed below:

1. MPEG4 decoder
2. H.263 decoder
3. JPEG decoder
4. Samplerate
5. MPEG4 decoder
6. H.264 encoder
7. MP3 decoder
8. MP3 decoder
9. H.264 decoder
10. MP3 decoder
11. H.263 encoder
12. JPEG decoder
13. H.264 decoder

14. MPEG4 decoder
15. H.264 encoder
16. MPEG4 decoder
17. MPEG4 decoder
18. MP3 decoder
19. H.263 decoder
20. H.263 decoder
21. MP3 decoder
22. JPEG decoder
23. MPEG4 decoder
24. MP3 decoder
25. JPEG decoder
26. H.263 decoder
27. Samplerate
28. JPEG decoder
29. H.263 decoder
30. Samplerate