Performance monitoring of throughput constrained dataflow programs executed on shared-memory multi-core architectures

Manuel Selva

Thèse

Pour obtenir le grade de
Docteur

Présentée devant
L'institut national des sciences appliquées de Lyon

Par
Manuel Selva

Performance Monitoring of Throughput Constrained Dataflow Programs Executed On Shared-Memory Multi-core Architectures

Encadré par Stéphane FRÉNOT, Lionel MOREL et Kevin MARQUET
Laboratoire CITI, INSA Lyon
École Doctorale Informatique et Mathématiques
Spécialité Informatique

En partenariat avec
Frédéric SOINNE, Bull
Stéphane ZENG, Bull

Soutenance prévue le 02 juillet 2015

Jury

Jean-François NEZAN	Professeur des Universités, *INSA Rennes*	Rapporteur
Eduard AYGUADÉ	Full professor, *University of Catalunya*	Rapporteur
Albert COHEN	Directeur de recherche, *Inria*	Président
Marco MATTAVELLI	Maître d'enseignement et de recherche, *EPFL*	Examinateur
Stéphane FRÉNOT	Professeur des Universités, *INSA Lyon*	Directeur
Lionel MOREL	Maître de conférences, *INSA Lyon*	Co-directeur
Kevin MARQUET	Maître de conférences, *INSA Lyon*	Encadrant

Contents

List of Figures

List of Tables

Glossary

numap Non Uniform Memory Access Profiling (`numap`) is a thin Linux library built on top of the `perf_event_open` system call dedicated to memory profinling on NUMA architectures. 3, 62, 70, 80, 91, 95, 98

AVC Advanced Video Coding (AVC) is a widspread video coding standard also known as H.264. 16, 17

CSDF Cyclo Static Dataflow (CSDF) is a dataflow computation model where actors rates are statically known and can vary periodically in a cyclic fashion. 21, 43, 96

DDF Dynamic DataFlow (DDF) is the most general dataflow computation model. 20, 21, 28, 32, 40–43, 46, 49, 50, 54, 55, 58, 59, 64, 65, 68, 80, 93, 95, 96

DPN Dataflow Process Networks (DPN) is a synonym of DDF. 20

FIFO a First In First Out (FIFO) data structure is a list of elements where elements are removed in arrival order (hence the name). 2, 14, 16, 19, 21, 24, 29–33, 35, 36, 41, 46, 48, 50, 51, 55, 56, 60, 63, 64, 68, 69, 79–82, 88, 89, 91, 95, 97

HEVC High Efficiency Video Coding (HEVC) is the new video coding standard following H.264. 81, 84–86, 88, 89, 91

KPN Kahn Process Networks (KPN) is is a simple model originally developed for modeling distributed systems. 19, 20

MoC a Model of Computation (MoC) defines the set of allowable operations and constraints used by programs conforming to the model. 16, 19, 21–24, 32, 43, 46, 48, 49, 62, 65, 96

MPI Standard defining the syntax and the semantic of a set of library routines for writing portable message-passing programs. 16

MPPA Many-core processor (256 or 512 cores) commercialized by Kalray. 9, 13, 93

NUMA Non Uniform Memory Access (NUMA) is a memory organization where memory access latency depends on the location of the requesting core and the location of the physical memory requested. 3, 10–12, 62, 64, 65, 80, 86, 88, 89, 91, 95–98

ORCC Open RVC-CAL Compiler (ORCC) is an open-source Integrated Development Environment based on Eclipse dedicated to dataflow programming. 28, 32, 40, 42, 50, 51, 54, 65, 68, 71, 78–82, 88, 91, 95, 96, 99

PAPI Performance API (PAPI) is a library specifying a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. 62, 98

PMU Performance Monitoring Unit (PMU) is a piece of hardware built into almost all modern processors providing hardware profiling capabilities. 60–65, 70, 86, 88, 91, 95

QPI Quick Path Interconnect (QPI) is the name of Intel's point to point processor interconnect. 11, 91

RVC Reconfigurable Video Coding (RVC) is an initiative from the MPEG group to provide an innovative framework of video coding developmen. 22

RVC-CAL Reconfigurable Video Coding Caltrop Actor Language (RVC-CAL) is a high level programming language for writing dataflow actors. 22, 42, 46, 47, 79–81, 86, 89, 91, 95

SADF Scenario Aware DataFlow (SADF) is an extension to SDF allowing an application to be described as several SDF graphs referred to as scenario. Runtime mechanisms allows to switch between scenario. 21, 43, 96

SAS Single Appearance Schedule (SAS) is an SDF schedule where each actor appears only once. 31, 36, 40

SDF Synchronous DataFlow (SDF) is a dataflow computation model where actors rates are statically known. 21, 28–34, 36, 37, 39–43, 46, 49, 50, 53–55, 57–59, 62, 64, 65, 68, 78, 80, 94, 96, 97

SPDF Schedulable Parametric Dataflow (SPDF) is a dataflow computation model where actors rates may vary at runtime according to parameters changes. 21, 23, 43, 96, 97

StreamIt Dataflow framework developped byt the MIT including a language rooted in the Synchronous Dataflow Model and a compiler performing agressive transformations with several backends. 28, 32–35, 42, 46, 65, 68–71, 73, 74, 77, 78, 93–95, 98

1 Introduction

Estimer correctement son degré d'ignorance est une étape saine et nécessaire

in "Patience dans l'azur" by Hubert Reeves

I started my PhD with hundreds of questions. I answered almost all of them, and now have thousands of new ones. This thesis summarizes my PhD work on *performance monitoring of throughput constrained dataflow programs executed on shared-memory multi-core architectures*. It presents the concepts, technologies and tools I learned about and the contributions I made during the last three years. Because I strongly agree with the epigraph above, this document also discusses about the new interrogations I now have regarding computer science.

This work has been done in the context of a collaboration between Bull and the Inria Socrate team from the CITI (Centre d'Innovation en Télécommunications et Intégration de services) at INSA de Lyon. From Bull's side, this work took place in the context of a project focusing on improving programmer efficiency when targeting parallel hardware with a focus on performances guarantees. The purpose of my work was to investigate and extend the dataflow programming model. I had to identify how the concepts from the dataflow programming model could be integrated in the programming model in use in the project. From the Socrate team's side, dataflow programming is a promising model to express the telecommunication applications being studied in the team.

1.1 The Multi-core Jungle

Because of physical limits, hardware designers have switched to parallel systems to exploit the still growing number of transistors per square millimeter of silicon. Indeed, since the beginning of the 21st century designers were not able to reduce the operating voltage as they did since the creation of the first computer while transistors were still smaller and faster. It resulted in an increase of the amount of waste heat to be dissipated by the processor from each square millimeter of silicon. Around 2004, the so-called *Power Wall* was reached. This wall corresponds to the point where systems can't reasonably dissipate the heat due to the impedance of the electronic circuits. In parallel designs, the heat to be dissipated is kept under acceptable limits by using maximum operating frequencies in the order of 3 GHz for each core. Additional performance comes from the ability to execute simultaneously several instruction flows on several cores. Today this

hardware parallelism is not only used to build super computers. It has reached other domains of computer science:

- All processors designed for the desktop market are multi-core processors;
- Smart-phones are almost all made of several cores;

This parallelism can be either homogeneous: all the cores of a system are the same, or heterogeneous: different kinds of cores are used in a single system. Moreover, the communication between cores can be handled in different ways depending on the memory architecture of the system. The contribution of this thesis that depends on the memory architecture focuses on homogeneous shared memory systems.

From the software perspective, this parallelism has a deep impact on programming and on performance analyses. How should we program these parallel systems? Most of the software in use today has been written using imperative sequential programming. In other words, programs are written has a single flow of dependent instructions. One solution to exploit parallelism is to perform automatic parallelization of these existing sequential programs. A large amount of work has been done in this area, nevertheless this is not a universal solution. There are some contexts where automatic parallelization can't find enough parallelism to fully exploit the available hardware resources. As a consequence, new programming models exposing parallelism have to be used if we want to efficiently exploit the capabilities of the forthcoming processors made of hundreds of cores. These models are called concurrent programming models. Moreover, software performance analysis is also impacted by hardware parallelism. Profiling mechanisms have to be developed to correlate low level hardware events with the concepts exposed by the concurrent programming model used to write applications. This work focuses on performance analyses for one of these concurrent programming model, namely dataflow programming.

1.2 Dataflow Programming

Dataflow programming consists in describing an application as a graph of actors communicating **only** through the use of explicit data dependencies with the usage of channels with a First In First Out (FIFO) semantic. Compared to sequential imperative programming where the programmer specifies the application's control flow, in dataflow as the name suggests, only the flow of data is specified.

I don't believe that a single ideal concurrent programming model can be used to program any application to be executed on any architecture. Nevertheless, I strongly believe that there are many places where dataflow programming has a strong role to play for the following reasons:

- The model matches killer applications: multimedia, big data processing, software defined radio, cryptography are all domains where algorithms are naturally described as dataflow graphs;
- The model provides different kinds of parallelism allowing compilers and runtimes to efficiently exploit hardware resources;

- The actors communication primitives provided by dataflow alleviates the burden of developers and allows to automatically compile programs for parallel systems with different communication architectures;

- Some restricted dataflow models provide interesting properties such as determinism, memory bounding, the possibility to identify deadlocks or minimum runtime overhead.

1.3 Problematics and Proposals

In the context of dataflow programming, many applications have intrinsic throughput constraints. It's the case for example for video decoders/transcoders, for software implementation of telecommunication protocols or for big data processing algorithm that must process inputs in real-time. The main questions I address in this work are **how to profile dataflow applications** and **how can we satisfy there throughput requirements** when they are executed on shared-memory multi-core architectures along with non dataflow applications ? Even if dataflow programming cannot be asserted as the only concurrent programming model, I strongly believe it can help in many contexts. I attach importance to scenarios with the presence of other applications because I believe that for dataflow to be widely accepted it needs to be integrated in existing general purpose systems.

To give a first answer to these questions, I make the following contributions in this thesis:

- I introduce mechanisms to express throughput requirements on dataflow programs [1],

- I show how to take the throughput requirements into account in compilation tool chains and exploit it along with static information when available [2, 3];

- I show how to profile dataflow applications to check whether or not throughput constraints are satisfied [3];

- I present profiling mechanism for system resources usage and correlate profiling information with the dataflow graph of the application to identify bottlenecks [3];

- I propose a library called Non Uniform Memory Access Profiling (`numap`) abstracting away the differences between processors micro-architectures about hardware performance counters for Non Uniform Memory Access (NUMA) architectures profiling[Section 4.5.2].

1.4 Thesis Organization

This thesis is organized in four main chapters. Chapters 2 introduces the context for this work. It focuses on the description of general purpose parallel hardware and gives an overview of some widely used concurrent programming models including dataflow programming. Finally, this chapter presents our motivation for focusing on dataflow and details the problematic I address in this work.

Chapter 3 introduces the notion of throughput constraints for dataflow programs. It shows how I extend dataflow languages to express this constraint and how this additional information can be propagated all along the dataflow graph in the case of static dataflow models.

Chapter 4 then presents how dataflow programs can be profiled. I first present in this chapter application-level profiling mechanisms dedicated to identify throughput constraints violation and their bottlenecks. Then, I introduce system-level profiling used to identify how dataflow programs exploit hardware resources when executed on top of homogeneous shared-memory multi-core architectures.

Chapter 5 shows how I applied and use together the concepts and mechanisms introduced in Chapters 3 and 4 in two different scenarios. The first one consists in building a throughput aware dataflow runtime system for a static dataflow language while the second one consists in a dataflow profiler for a dynamic dataflow language.

Finally, Chapter 6 concludes this thesis. It summarizes the contributions I made during the PhD before introducing the numerous perspectives for this work.

2 Context and Objectives

One early vision was that the right computer language would make parallel programming straightforward. There have been hundreds—if not thousands—of attempts at developing such languages ... Some made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming. Nor has any become as popular as the languages invented primarily for sequential programming

David Patterson

Chapter's outline

2.1 Introduction

This chapter introduces the context and the objectives for this work. Section 2.2 first places this work in the global context of computer science. Section 2.3 highlights the main motivations leading the hardware industry to the development of more and more parallel systems and then reviews the current status of general purpose parallel hardware. Section 2.4 gives a broad overview of some widely used programming models for parallel hardware before introducing dataflow programming which is the model we consider in this work. Finally, Section 2.5 presents the objectives of our work in the context of dataflow programs executed on modern parallel hardware.

2.2 General Context

Computers are today omnipresent in our lives. 39% of the world population is using Internet according to the International Telecommunication Union [4]. There are more than 1 billion smart-phones in use and current middle-of-the-range cars have at least 30 embedded processors [5]. Depending on the targeted usage, the size of computers can vary greatly. It ranges from more than 5 billions of computing units for supercomputers used in domains such as molecular dynamics simulations or weather forecasting to few processors in desktop computers, mobile phones or modern bike computers. Except for very small embedded systems, all the systems at both extremes of this size's range are parallel. The actual trend of the computer industry leads us to think that the number of computing units in all these systems will increase in the coming years. Indeed, to keep up increasing performances using the still growing number of available transistors on a single chip, hardware designers have shifted from sequential to parallel architectures.

This trend towards more and more parallel systems has a deep impact on the software community. How should programmers write code to be executed efficiently on parallel hardware? It's widely accepted [6, 7, 8, 9] that simple extensions to languages primarily designed for sequential systems is not a practical solution: the human brain can't apprehend the complexity of a huge number of possible interleaving resulting from the parallel execution of several tasks. As stated by the epigraph opening this chapter, numerous programming languages specially designed for parallel systems have been proposed in the past decades. Unfortunately, none of these languages has been widely accepted and "none has made it as fast, efficient, and flexible as traditional sequential programming". Nevertheless, some of these languages have made parallel programming easier, particularly when they were dedicated to a particular class of applications and to specific hardware. Among these languages, we can mention for example Erlang [10] that was invented for programming distributed fault-tolerant non-stop applications and OpenCL [11] that was designed to program heterogeneous systems made of central processing units and graphics processing units.

This work focuses on dataflow programming, another approach for programming parallel architectures, first introduced in 1974 both in Europe by Kahn [12] and by Dennis [13] in North America. Kahn's work was dedicated to the creation of a simple programming language targeting distributed systems allowing to write only deterministic programs whereas the goal of Dennis was to alleviate bottlenecks of Von Neumann hardware by creating a dataflow computer and a programming language for it.

In the dataflow approach, an application is described as a graph of computing entities operating over data exchanged only through explicit dependencies. Many of the killer

applications in use today are intrinsically dataflow applications and there popularity is increasing. Among numerous applications that exhibit dataflow processing we can mention:

- Video encoding/decoding;
- Image processing such as face recognition or analyses in medical images;
- Cryptographic algorithms;
- Software defined radio: software implementation of telecommunication protocols to reduce building costs and time to market and to increase flexibility;
- Big data processing commonly found in data centers used by Internet companies.

In addition to programming facility, the dataflow approach allows to efficiently execute applications on top of modern parallel architectures. The next section draws a global taxonomy for these parallel architectures. Section 2.4 then presents the dataflow approach in details and shows how it can answer to the programming challenges introduced by these modern architectures.

2.3 Parallel Architectures

For decades, computer scientists relied on the Moore's law, stating that the number of transistors in a dense integrated circuit doubles approximately every two years. Performance improvements came from increased processor frequencies and new logic enabling processors to carry out several operations at once, such as pipelining. Unfortunately, designers were not able to reduce the operating voltage as they did since the creation of the first computer while transistors were still smaller and faster. It resulted in an increase of the amount of waste heat to be dissipated by the processor from each square millimeter of silicon. Around 2004, the so-called Power Wall was reached. This wall corresponds to the point where systems can't reasonably dissipate the heat due to the impedance of the electronic circuits.

At this time, hardware vendors started to shift to multi-core processors designs. In these designs, the heat to be dissipated is kept under acceptable limits by using maximum operating frequencies in the order of 3 GHz for each core. Additional performance comes from the ability to execute simultaneously several instruction flows on several cores. In the rest of this dissertation we'll use the term *core* to refer to the smallest hardware unit capable of executing a single instruction flow. We'll use the term *processor* to refer to a single chip made of one or more cores with additional components such as caches and memory controllers.

Executing several instruction flows in parallel is not a new idea. Parallel computers with two or more processors have been built since the earliest days of computer science. Nevertheless, these parallel systems were expensive and dedicated to very specific markets. The shift operated in 2004, brought parallel computers into the desktop market. Multi processors systems are still built today, but they are made of several multi-core processors. The predicted follow-up of Moore's law in the next decade leads to think that processors are going to include more and more cores to improve performances. Hardware used today in markets ranging from big servers to embedded systems are almost all made with multi-core processors.

To get a global picture of the different kinds of parallel systems in use today, we propose to look at two criteria:

- The homogeneity criterion distinguishes parallel systems made only of identical cores from systems containing cores with different properties such as the instruction set or the operating frequency;
- The memory organization criterion distinguishes systems with a single central physical memory from systems with several distributed memories.

2.3.1 Computing Homogeneity

Parallel systems where historically homogeneous for practical reasons. It's easier to build a computer made of identical cores than one made of different cores. Nevertheless, with the advances of hardware technologies, heterogeneous systems are more and more used to handle very efficiently specific algorithms.

2.3.1.1 Homogeneous Computing

Homogeneous systems are systems where all computing units are identical. They are widely used in desktop computers, servers and supercomputers. All desktop computers are today made of a single processor with at least two identical cores. For example, an Apple MacBook Pro is today available with either an Intel Core i5 processor with 2 cores or an Intel core i7 processor with 4 cores.

In the server market, Intel's largest multi-core processors [14] include 14 cores. A single multi-processor-multi-core computer system, i.e. packaged in a single physical machine and managed by a single operating system, can be made by 4 of such processors resulting in a total of 56 physical cores.

In the supercomputers domain, many servers are used together to build very large machines. Centralized massively parallel supercomputers are clusters of homogeneous multi-processor-multi-core servers connected through an high speed network. The TERA-100 [15] jointly developed by Bull and CEA is such an example. It's one of the fastest computer in Europe, and it's made only of Intel Xeon multi-core processors. These Intel Xeon processors are the same than the one used in the desktop market. Homogeneous systems are easier and cheaper to built than the heterogeneous ones described below.

2.3.1.2 Heterogeneous Computing

Heterogeneous systems are systems made of different cores. This heterogeneity can be either at processor level, i.e. different cores are collocated into the same processor chip or at a higher level, i.e. different processors are collocated in the same machine. A famous example of an heterogeneous processor is the Cell [16] used in the PlayStation 3. This processor is made of one host core and eight co-processing cores dedicated to multimedia and vector processing. The widespread Exynos 5 Octa processor developed by Samsung for high-end smart-phones is an example of heterogeneous system made of different processors based on the ARM big.LITTLE architecture. In this architecture, a quad core processor cadenced at 1.6GHz is used in conjunction with another quad core processor cadenced at 1.2GHz. Depending on computing needs, either the big only, the little only, or both processors are activated. Recent supercomputers also started to use heterogeneous designs. The actual world's fastest supercomputer, Tianhe-2 [17], is made

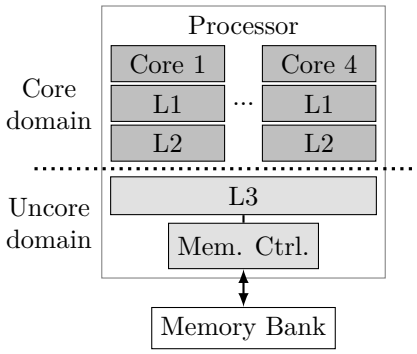


Figure 2.1: Centralized shared-memory architecture overview. In such architectures, all the cores share the same central physical memory and access it with the same latency.

of 16.000 computer nodes, each comprising two Intel Ivy Bridge Xeon processors and three Xeon Phi co-processor [18] chips. These heterogeneous systems are usually harder to program than homogeneous ones because software developers needs to care about the differences between the cores.

2.3.2 Memory Organization

The second criterion we propose to look at for the classification of parallel architectures concerns the memory organization and depends on the number of cores. This criterion is orthogonal to the homogeneous vs heterogeneous one. When the number of cores reaches a given limit, the centralized memory organization becomes a bottleneck and needs to be replaced with a distributed organization. We now describe in more details the differences between these memory organizations and their impact on software to understand the main challenges that applications and operating systems programmers face when targeting such architectures.

2.3.2.1 Centralized Shared-Memory Architectures

Centralized shared-memory architecture is a design where all cores access the same unique physical memory with the same latency. Historically, systems with centralized shared-memory organization were systems with several single core processors. Today, almost all systems made of a single multi-core processor have a centralized shared-memory organization as depicted on Figure 2.1. The current version of Intel processors, micro-architecture code name Haswell [14], has a maximum of 14 cores. AMD's biggest processors, micro-architecture code name Bulldozer, contain a maximum of 6 cores. Cores have small private caches, two in Figure 2.1 example and a last level cache is shared by all the cores. Inter-core communication is handled through the centralized shared memory. Moreover, most of centralized shared-memory architectures are cache coherent. In this case, the cache coherency system is responsible to ensure that cores see a coherent view of the memory. If a core requests for data being modified in another core's private cache, the cache coherency mechanisms must bring the data in the requesting core's private cache.

We also find a centralized shared-memory organization in upcoming many-core architectures. The Intel Xeon Phi co-processor [18] has a centralized shared-memory organization for all its cores and the Kalray (Multi-Purpose Processor Array (MPPA)

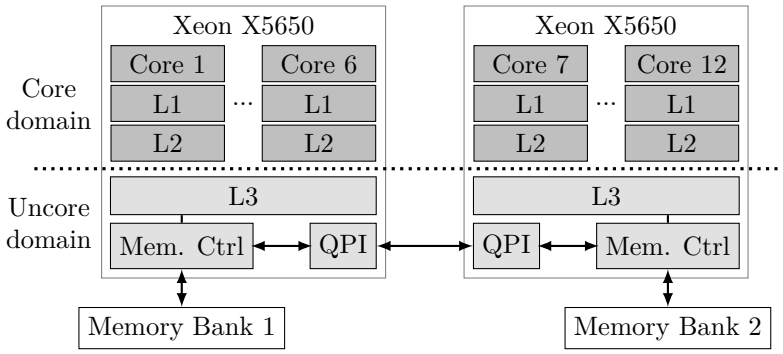


Figure 2.2: Intel's example of a dual processors NUMA architecture. The two processors can access the two memory banks through a unique physical address space. Memory accesses going outside the requesting processor are achieved by the inter processor network.

256 many-core [19] processor is made of several 16 cores clusters having a centralized shared-memory organization.

Compared to single core systems, programming a multi-core system having a centralized shared-memory organization requires a deep understanding of the underlying memory consistency model [20]. Fortunately, system libraries and high level languages have been developed to alleviate application's programmers dealing with low level concerns. Languages providing a lock based shared-state concurrent programming model and dataflow programming described in Section 2.4 abstract away these low level concerns.

2.3.2.2 Distributed Shared-Memory Architectures

The Memory Wall With the increase of the number of cores in multi-core processors, the so called memory wall becomes more problematic. This wall represents the disparity between the frequency of cores and the time required to access main memory located outside the processor. The increase in core numbers accessing the same centralized memory puts more pressure on the memory subsystem. One of the proposed solutions by hardware designers is the use of distributed shared-memory organization. This memory organization is also referred to as NUMA because memory access time depends on the location of the requesting core. Instead of having all cores accessing the same memory through the same controller, the global memory is split in several physical parts. Any core of the system can address all the memory through a single physical address space with memory access time depending on the physical memory location. Because the whole memory is directly addressable by all cores of the system, programs written for centralized shared-memory architectures can be executed without any modification on distributed shared-memory architectures.

Two kinds of NUMA architectures are used. In the first one depicted on Figure 2.2, several processors are used together in the same machine. Machines hosting web servers, or workstation used to perform mechanical simulations typically use a NUMA architecture because they require high computing power and execute applications that are easily expressed in a parallel way. In these machines, each processor has an attached memory controller/memory bank pair. On these architectures there are as many memory banks available as there are processors. Processors are connected through a point to point net-

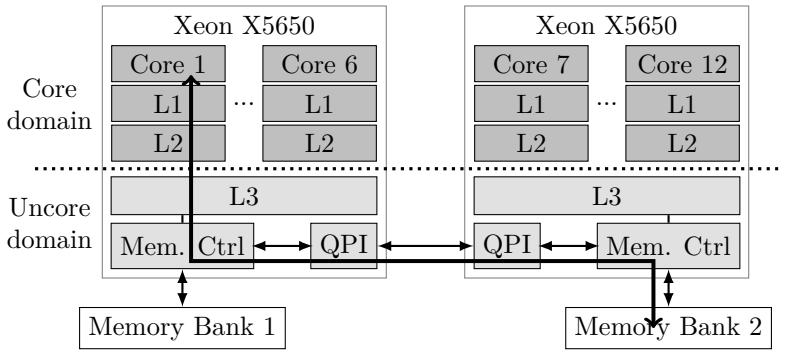


Figure 2.3: NUMA remote memory accesses. A request from core 1 to memory physically located on memory bank 2 must go through the processors interconnect network.

work. This network is used to communicate memory accesses going to memory outside the requesting processor.

Figure 2.2 is an example of Intel NUMA architecture where the processor interconnect technology is called Quick Path Interconnect (QPI). AMD proposes comparable NUMA architectures with an interconnect technology called Hypertransport.

The second kind of NUMA architecture is referred to as *on-chip NUMA*. In such architectures, several memory controllers are collocated on the same processor chip. It is used in processors such as the IBM power 7 [8] at the heart of a supercomputer made by the Defense Advanced Research Projects Agency (DARPA) but is today less frequent than multi processors NUMA.

NUMA Concerns The NUMA memory organization results in memory bandwidth improvements. On our example with two processors, the memory bandwidth is doubled without the need of complex modifications in the memory controller. Nevertheless, even if parallel programs written for centralized shared-memory can be executed without modification on NUMA architectures, special care should be taken into account to fully exploit the memory subsystem [21]. We now explain the three main concerns about the usage of the memory subsystem.

The first concern depicted in figC 2.3 is to avoid remote memory accesses as much as possible. Indeed, because these accesses require sending information over the interprocessor network, they have a greater latency. This overhead is about 30% on modern two-processors systems built today by Intel and AMD NUMA architectures [22, 23, 24].

The second consideration concerns shared data and is depicted on figC 2.4. On both centralized and distributed shared-memory architectures, the cache coherency protocol is responsible of keeping data synchronized between caches. In the case of distributed shared-memory, special care must be taken by the hardware if the producer and the consumer of shared data are located on different processors. In this case, as for remote memory accesses, the interconnect network has to be crossed to bring the correct version of the data in the consumer's local cache. Crossing the sockets has been identified as a "killer" feature [24]. The overhead of a remote cache access compared to a local one depends on the memory block cache coherency state. For Intel processors worst case, loading from a cache on a remote processor can be 5 times more expensive than loading from local cache [22, 24].

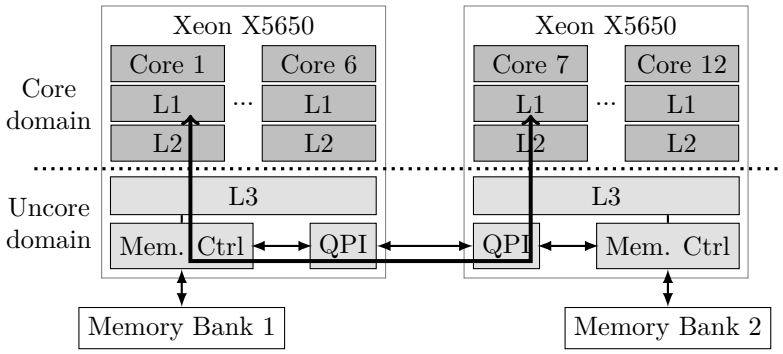


Figure 2.4: NUMA shared data synchronization. The cache coherency protocol ensuring that cores see a coherent view of the data has to cross the inter process network when a modified copy of the requested data is located in memory attached to another processor.

Finally, for memory intensive scenarios, memory controllers and processor interconnects may be saturated [25, 26, 27, 28]. This situation may arise when many cores are simultaneously accessing the memory and these accesses are not correctly balanced between the different memory banks. With the advances of hardware technologies, this last problem is becoming the more and more important compared to remote memory latency. Indeed, remote accesses were in the past 4 to 10 times slower than local ones [29] whereas they are in the order of 30% today as already stated.

The Impact of NUMA on Software The concerns introduced above have to be addressed by the software. This can be done at different levels. Applications can explicitly be written with the memory architectures in mind. Said differently, programmers may explicitly specify where data and code should be located. The memory architecture can also be taken into account by compilers to generate more efficient code or at runtime by operating systems to take better scheduling and memory allocations decisions. From these concerns, we can classify parallel applications running on top of NUMA hardware as either latency-or bandwidth-sensitive. Latency sensitive applications are applications that don't cause many simultaneous memory accesses. These applications don't saturate memory controllers but their performances may depend on the memory access time and thus on the location of accessed data relatively to the requesting core. On the other hand, bandwidth sensitive applications are applications that reach the memory bandwidth limit of the underlying hardware. To speed up latency sensitive applications on NUMA hardware, memory latency must be reduced, and this is usually done by collocating threads and memory on the same processor. To speed up bandwidth sensitive applications, all the available memory banks must be equally used to exploit all the available bandwidth provided by the hardware.

2.3.2.3 Distributed Private-Memory Architectures

To further increase cores number and alleviate the bottlenecks of shared memory organizations described in the previous sections, computer systems designers rely on a third memory organization called distributed private-memory. In such organization, cores can only access a subset of the memory that is distributed other several physical parts. Communication between cores that don't share any memory is handled through

message passing. This new communication style has a strong impact on programming. Indeed, the concurrent tasks with shared state model described in Section 2.4.1 need to be replaced with message passing models as described in Section 2.4.2.

The spectrum of distributed shared-memory architectures ranges from very large scale parallel systems to on-chip parallel architectures. Clusters are a famous example of large scale parallel systems with such memory organization. The communication between the different machines in the cluster are handled through message exchanges over an Ethernet network. The Kalray MPPA 256 many-core architecture [19] uses both centralized shared-memory and distributed private-memory organizations. On the MPPA chip, cores are grouped in clusters of 16 and must explicitly use hardware primitives to send messages across the cluster boundaries. Inside each cluster, cores communicate through a centralized shared-memory. Between large scale and on-chip systems we find on-machine systems also using a distributed private-memory organization. The Intel Xeon Phi co-processor [18] is dedicated to be used along with a general purpose processor in an on-machine distributed private-memory system. Communication between cores in the Xeon Phi is handled through shared memory while communication between the general purpose processor and the co-processor must be handled explicitly through message passing.

2.4 Concurrent Programming Models

From the application's programmer's point of view, programming parallel systems is a complex problem facing two major hurdles. The first challenge comes from Amdahl's law shown on Equation 2.1. This law gives a prediction of the theoretical maximum global speedup that can be achieved when parallelizing a sequential application. PF (Parallel Fraction) represents the fraction of the initial sequential application that can be parallelized. PS (Parallel Speedup) quantifies the speedup of the parallelized regions. A perfect parallel speedup corresponds to the situation where splitting a piece code in n parallel tasks leads to code n times faster and where all the code can be parallelized. In this case $PS = n$ and $PF = 1$ leading to a global speedup equal to n.

$$GlobalSpeedup = \frac{1}{\frac{PF}{PS} + (1 - PF)} \tag{2.1}$$

To understand how a small amount of code that can't be parallelized impacts the global speedup we consider an example. To achieve a speedup of 80 with 100 cores while considering that the parallel version scales ideally (i.e. is 100 times faster than the sequential one, $PS = 100$), Equation 2.1 shows that PF must not exceed 0.25%. In other words, only 0.25% of the original program must be sequential.

The second hurdle concerns communication's overhead. In the Amdahl's law given above, this communication overhead is hidden in the PS term. The time used to exchange information between the parallel parts of an application must not shadow the time gained from parallel execution. In other words, if this overhead is too large PS will approach 1 as will the global speedup.

From these two hurdles, compilers and runtimes must identify as much parallelism as possible while taking care of communication overhead to exploit hardware parallelism inside applications. This can be done either automatically by extracting instruction level parallelism from sequential programming models or by exploiting parallelism provided

by concurrent models. The main challenge for these concurrent models is thus to provide parallelism for compilers and runtimes while helping programmers and tools to reason on programs [6, 30]. This section reviews the following concurrent programming models:

- *Concurrent tasks communicating with shared state*: applications are defined as a set of tasks explicitly created by the programmer and inter tasks communication is done through shared memory;
- *Concurrent tasks communicating with message passing*: applications are also defined as a set of tasks explicitly created by the programmer but inter tasks communication is done with explicit send and receive requests;
- *Dataflow*: applications are defined as a graph of computing entities called actors and communicating only through explicit FIFO channels.

Each of these models has its own origins and motivations. The concurrent tasks communicating with shared state model is the simplest extension that can be done to the traditional sequential programming model with a program counter and a global memory. Explicit message passing requests was proposed as a replacement of the shared memory to let tasks communicate in parallel architectures with distributed private memory organization. Finally, dataflow programming is rooted both in distributed programming and in the signal processing domain. To understand the differences between these programming models and to highlight how it can be difficult to reason on programs, we propose the following criteria to classify them:

- *Determinism*: a concurrent programming model is said to be deterministic if programs written in the model are always deterministic. A deterministic program is a program where the outputs depend only on the input. For example, programs which outputs depend on the interleaving of concurrent tasks that may vary upon executions are not deterministic;
- *Targeted architectures*: programming models may be dedicated to a given type of architecture or may be used to write applications at a high level of abstraction allowing compilers to target different architectures;
- *Targeted applications class*: in addition to be dedicated to some architectures, programming models may be dedicated to a class of applications. Dataflow programming, for example, is dedicated to program applications focusing on data processing.

According to these criteria, we now review the three programming models mentioned above.

2.4.1 Concurrent Tasks with Shared State

The thread abstraction is probably one of the most used concurrent programming model when targeting both centralized and distributed shared-memory architectures. The model behind threads belongs to the concurrent tasks with shared state class of concurrent programming models. C/C++ with the Pthread library, Java and Python are

famous examples of programming languages providing the thread abstraction. General purpose processors and operating systems have been designed to support this model efficiently and sequential imperative languages require little syntactic changes to support it. This model directly reflects systems with shared memory where threads are concurrent execution flows communicating through this shared memory.

Unfortunately, programming with threads discards the determinism of sequential programs if synchronization is not correctly handled. The result of a threaded application's execution may depend on the execution order of threads usually decided by the operating system upon varying factors. Such factors are for example the interaction with other concurrent applications or the complex optimization mechanisms built into modern processors such as memory accesses reordering impacting the timing of instructions execution. The main goal of multi-thread programming is thus to remove this observable non determinism usually using synchronization primitive. When the number of threads needs to be increased (to keep improving performances with increasing hardware parallelism) the growing complexity of removing non determinism becomes a strong argument against the usage of threads [6, 30]. The human brain can't apprehend the growing complexity of a very large number of possible tasks interleaving. Programming a multi-thread application to be executed on a quad-cores processor is already a difficult task requiring a lot of care to ensure correct synchronization between tasks. With the upcoming general purpose processors made of more than 12 cores, programming will be even more difficult.

As a consequence, a large part of the parallel programming community argues for the use of deterministic models everywhere non determinism is not required [6, 30]. More recently [9], proposals investigate a solution limiting the non determinism to few different possible execution schedules. The idea behind this solution is that the problem with non determinism is not non determinism itself but the huge amount of different execution schedules allowed by non determinism. Reducing non determinism to few special cases allows the application developers to focus only on these cases.

To avoid rewriting legacy sequential applications using new deterministic models and to avoid the complexity of manually synchronizing threads, tools have been proposed to automatically create threads from sequential programs annotations. OpenMP is a widespread example of such a tool. It's an application programming interface consisting of a set of compiler directives, library routines and environment variables used to annotate and modify existing sequential applications in order to indicate to the compiler and to the runtime where and how the application should be parallelized. OpenMP mainly targets shared memory multiprocessors and implementations exist for the C, C++ and Fortran languages. In other words, programmers specify compiler directives and then the compiler along with runtime mechanisms are responsible to handle concurrency properly. With OpenMP, data parallelism (that execute the same code with different data) iterations is easily expressed with only one compiler directive. For example, heavy computing loops without dependencies between iterations can be data parallelized. Expressing task parallelism (different execution flows executing different pieces of code) among independent pieces of code is also done with simple compiler directive. Unfortunately, when parallelizing existing sequential code without clearly independent tasks, programmers must refactor existing code. Despite its widespread usage for scientific computing, it has been shown [7] that some common applications can't scale without large refactoring

using OpenMP because they don't contain independent loops or tasks.

2.4.2 Concurrent Tasks with Message Passing

Message passing programming models are used to program systems with distributed private-memory organization. Message Passing Interface (MPI) is probably the most widely used standard conforming to this model. It provides both synchronous and asynchronous message passing and targets a wide variety of parallel computers. The standard defines the syntax and the semantic of a set of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. Implementations of the MPI standard exists for different kind of distributed private-memory architectures. The most used MPI implementation targets clusters of machines communicating through an Ethernet network. Other implementations target on-machine distributed private-memory architectures such as a computer made of an Intel Xeon general purpose processor and an Intel Xeon Phi co-processor and on-chip distributed private-memory architectures such as Kalray MPPA 256. As with the concurrent tasks with shared state model it is possible to write programs with observable non determinism with this model because it doesn't specify when messages between concurrent activities are exactly sent and received. In other words, reasoning on MPI programs is not as easy as reasoning on programs based on a deterministic concurrent programming model.

2.4.3 The Dataflow Approach

In this work we focus on dataflow programming. A dataflow program is defined by a directed graph where vertices represent functional computation to be applied on data tokens feed by edges. In the following we use the word actor when referring to a node of a dataflow graph. The edge of the dataflow graph used to connect actors are FIFO channels and theoretically infinite. Actors *pop* tokens from input channels and *push* tokens to output ones. Actors without input channels are called *source* actors, and actors without output channels are called *sink* actors. Actors execution, also called *firing*, is an atomic step consisting in consuming an arbitrary number of tokens on input edges to produce another arbitrary number of data on output edges. The number of tokens consumed and produced by actors firings is either static or dynamic depending on the dataflow Model of Computation (MoC) as described in Section 2.4.3.1. Theoretically, to fire an actor, input tokens availability is the unique constraints to be satisfied. In practice, because FIFO channels are not infinite, enough room must be present on the output channels to fire the actor. Moreover, a computing resource must also be available to effectively fire the actor. Actors without any self dependency between consecutive firings are called *stateless* actors. These actors play an important role in parallelism extraction. Executing a dataflow graph consists in an infinite loop, or a loop running while there are more inputs to be proceed, executing actors when they can be fired.

Figure 2.5 shows a high level dataflow description of an MPEG-4 Advanced Video Coding (AVC) decoder also known as H.264. The first actor, called parser is a source actor reading the input stream (e.g. from a file or the network) and splitting the data to be decoded into the three Y, U and V image components. Once the parser has started to produce tokens, the texture decoding actors can be executed and then motion decoding actors. Finally, a merger actor construct back the final picture to be displayed from the

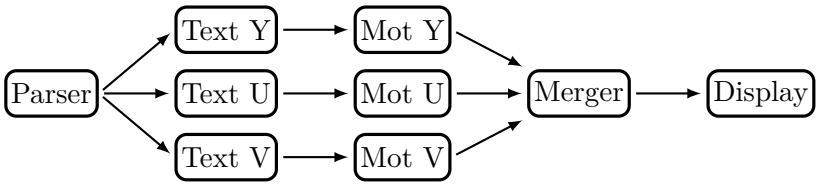


Figure 2.5: MPEG-4 AVC decoder expressed as a dataflow graph.

three Y, U and V decoded components.

The explicit and unique dependencies between dataflow actors allow the dataflow compiler and the dataflow runtime to exploit different kinds of parallelism among:

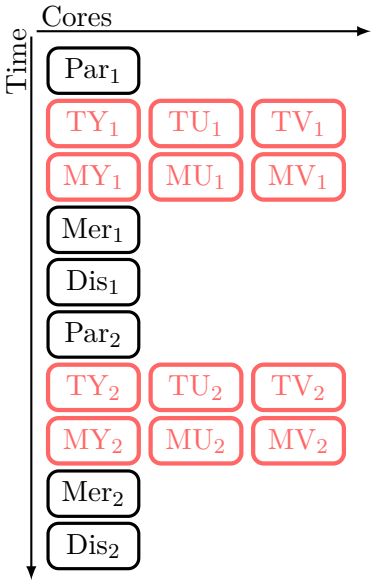
- *Task parallelism*: actors operate independently, each one encapsulating its own state, and thus can be executed concurrently;
- *Data parallelism*: stateless actors can be split into separate identical instances that operate over subsets of data;
- *Pipeline parallelism*: producer/consumer relationship are explicit allowing to let producer go ahead from consumer.

Figure 2.6 illustrates these three kinds of parallelism on the AVC decoder dataflow graph example of Figure 2.5. Figure 2.6a shows task parallelism where the texture and the motion decoding can be done in parallel on the three image components as expressed by the parallel branches in the dataflow graph. Figure 2.6b shows the addition of data parallelism. For the sake of clarity we consider that the parser actor is stateless and can thus be executed concurrently on different data. Finally, Figure 2.6c shows the addition of pipeline parallelism. A first initialization stage fills the pipeline and from there the merger and the display actors can be executed in parallel. As depicted by this example, dataflow compilers and runtimes can mix the different forms of parallelism to achieve the best performance possible.

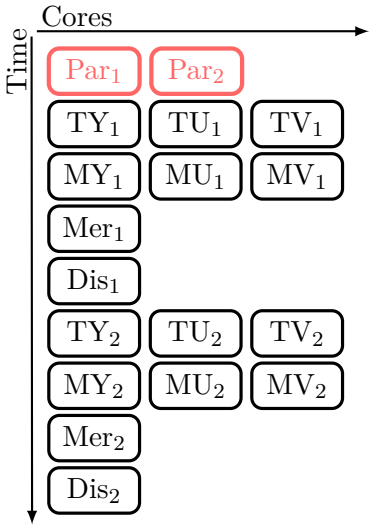
Because many killer applications in the near future will be intrinsically focused on data processing, we believe that this model has a strong role to play in the coming years. Such applications include video encoding and decoding, big data processing, face recognition algorithms and software defined radio protocols. Moreover, the communication abstraction provided by dataflow models allows dataflow compilers and runtimes to compile and execute applications on any one of the architectures introduced in Section 2.3. This is a strong advantage over many other concurrent programming models. To sum-up, we clearly identify three advantages to this model:

- Applications focused on data processing are naturally described as dataflow graphs;
- By decoupling computation from communication, this paradigm naturally exposes parallelism in several ways;
- Dataflow applications can be compiled and executed on many different architectures.

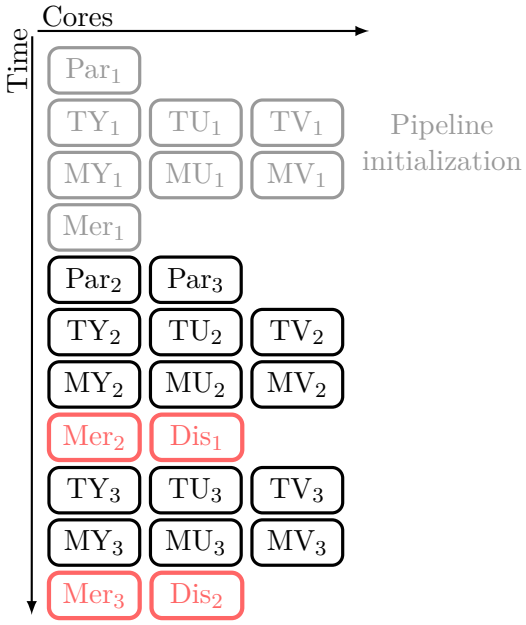
We now review the most influential dataflow models conform to the general model described in this section.



(a) Task parallelism. The parallel branches of the dataflow graph are executed concurrently.



(b) Task and data parallelism. Considering that the parser actor is stateless we can execute it in parallel on different data.



(c) Task, data and pipeline parallelism. Once pipeline has been filled, merger and the display can be executed in parallel.

Figure 2.6: The three kinds of parallelism available in an MPEG-4 AVC decoder expressed as a dataflow graph.

MoC	Determinism	Scheduling	Deadlock Det.	Bounded FIFOs	Year
KPN	Yes	Dynamic	No	No	1974
DDF	No	Dynamic	No	No	1993
SADF	Yes	Quasi-static	Yes	Yes	2006
SPDF	Yes	Quasi-static	Yes	Yes	2012
CSDF	Yes	Static	Yes	Yes	1995
SDF	Yes	Static	Yes	Yes	1987

Table 2.1: Dataflow MoCs properties. Recently, new MoCs have been proposed to extend expressiveness of static models while keeping as many properties as possible.

2.4.3.1 Dataflow Models of Computation

Several specific dataflow MoCs conform to the general model described above have been proposed in the last decades. These models define the set of operations used inside dataflow actors to describe the firing behavior and the properties required for actors to be fired. The main constraint specified by a dataflow MoCs is the number of tokens to be present on input ports for an actor to be fired. The execution order of actors, also called scheduling, the mapping of actors to hardware computing units and communications implementation are not specified by the model. All these concerns are under the responsibility of the execution model as described in Section 2.4.3.2. Even if the model of computation doesn't specify the scheduling, different models allow two main kinds of scheduling. The first one, called static scheduling is possible only for models where enough information is available to let the compiler decides actors execution order. The second one, for other MoCs is called dynamic scheduling and consist in checking and choosing fireable actors at runtime.

We now review the properties summed-up in Table 2.1 of the most influential dataflow models.

Kahn Process Networks Kahn Process Networks (KPN) [12, 31] is a simple model originally developed for modeling distributed systems. Since the first paper in 1974, this model has also proven its convenience for modeling signal processing systems. KPN is not a dataflow model of computation in the sense described above, because the firing notion doesn't exist in the model. In KPN, nodes represent sequential processes and edges are unbounded FIFO channels. Writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process. Reading from a channel is blocking, i.e. a process that reads from an empty channel stalls until the channel contains sufficient tokens. Processes are not allowed to test an input channel for existence of tokens without consuming them. Given a specific input tokens history for a process, the process must be deterministic so that it always produces the same output tokens. Timing or execution order of processes must not affect the result and therefore testing input channels for tokens is forbidden. Kahn processes have been shown to be monotonic; receiving more input at a process can only provoke it to send more output tokens and not change the one already sent. Also, a process does not need to have all of its input to start computing. Assuming

that each node performs a deterministic computation as required above, and using the monotonic property, it has been proven that the entire network is deterministic; the sequence of data items observed on the output channels is a deterministic function of those submitted to the input channels. However, it is not possible in the general case to statically:

- Schedule processes;
- Determine the amount of memory needed on the channels;
- Check whether the computation might deadlock or not.

Dynamic Dataflow Models Dynamic DataFlow (DDF), also called Dataflow Process Networks (DPN) formally described by Lee [32] is the most general dataflow model. This model allows to express any application including data processing applications requiring frequent reconfigurations leading to dynamic consumption and production rates. Cutting edge video codecs and upcoming telecommunication protocols such as the next generation of mobile protocol (5G) are typical examples of such applications.

In this model, each actor may have one or more input channels and one or more output channels and a set of firing rules. A firing rule specifies a number of tokens for each input channel required for one actor's execution. This number also specifies the number of tokens that will be consumed by the actor's execution. In DDF, the number of tokens consumed may be specified only at runtime. The firing of an actor removes tokens on its input edges and produces tokens on its output edges as specified by the firing rule.

Compared to KPN, DDF have the notion of actors firings conforming to the dataflow model. In other words, actors invocations are atomic meaning that once an actor firing is started the firing rules guaranty that enough tokens are present on input channels to complete the firing. DDF implementations then needs to save only actors state when switching between actors. In KPN, this firing notion doesn't exist. As a consequence, when an actor is blocked on a pop operation the current execution state of the actor must be saved before switching. DDF can thus be seen as an extension of KPN with this notion of atomic firings for processes. Another fundamental difference with KPN is that DDF allows to write non deterministic programs by the use of non deterministic actors. Intuitively, non deterministic actors are actors with output depending on the time at which inputs are produced. See Section 3.6.1 for details.

The DDF model is the most expressive of the dataflow computation models. The model doesn't enforce any restriction on the number of tokens consumed and produced by each actor firing. As a consequence this model is also the less analyzable, as for KPNs it is not possible to statically:

- Schedule actors
- Determine the amount of memory needed on the channels;
- Check whether the computation might deadlock or not.

Static Dataflow Models Synchronous DataFlow (SDF) first introduced by Lee [33, 34], rooted in the signal processing domain, can be seen as a specialization of DDF even if it has been formally described before it. It is typically used to model static kernel functions such as fast Fourier transforms, finite impulse response filters, Cholesky decomposition, CRC encoders. It can also be used to described complete applications that don't require production and consumption rates reconfiguration at all such as a DES encryption or a 802.11a transmitter [35].

In SDF graphs, actors always consume (respectively produce) the same number of tokens on each input (respectively output) FIFO channel. These numbers are called the *popRate* and the *pushRate* of the SDF actors.

SDF is the most restricted dataflow model, allowing several static analyses. First, tools can statically identify SDF graphs that can't be executed with finite memory. These graphs are called *inconsistent* graphs. Tools can also statically detect graphs that will deadlock because of insufficient initial tokens in FIFO channels along cycles. Finally, SDF graphs can also be scheduled statically avoiding the overhead of runtime scheduling.

Cyclo Static Data Flow (CSDF) [36] is a generalization of SDF where the number of tokens produced and consumed by a single node is also known at compile time, but can change periodically. The work by Bilsen, Engels, Lauwereins & al. [36] extends the consistency notion of SDF graphs to CSDF ones and proposes a way to compute a parallel schedule. Using this model, some applications difficult to be expressed in SDF can be expressed more easily while keeping the static properties of SDF.

Extended Static Dataflow Models In recent years, several new dataflow MoCs have been introduced to extend expressiveness of static models while trying to keep the static properties of SDF.

Scenario Aware DataFlow (SADF) [37] is an extension to SDF with the notion of scenario. A scenario represents an execution mode for the application and is itself described by an SDF graph. At runtime, application can switch between scenarios. The static knowledge for all the possible execution modes of an application allows to keep static properties of SDF.

Schedulable Parametric Dataflow (SPDF) [38] is another extension to SDF. As for SADF, the main goal is to increase SDF modeling capabilities while keeping as many static analyses as possible. In SPDF, actors production rates can be defined as relational equations of parameters. These parameters can be changed by one single actor. Because schedules for SPDF graphs depend on these parameters, runtime mechanisms are required to adapt the schedule when the parameters change. Such a schedule is called a quasi-static schedule. Static analyses to check boundedness from rate consistency and parameters change periods and to ensure liveness of an SPDF graph have been proposed. When SPDF graphs are consistent, a quasi-static schedule can be computed statically. This model of computation allows for example to express variable-length encoder/decoders present in almost all video standards or next generation telecommunication protocols such as the Long Term Evolution Advanced standard [39].

2.4.3.2 Dataflow Execution Model

The dataflow execution model is responsible to execute dataflow applications expressed in a dataflow language itself based on a given MoC described in the previous section. The execution model is responsible to preserve the semantic of the MoC in order to ensure applications correctness. As stated in Section 2.4.1, we believe that the concurrent tasks with shared state model is not a good choice to write applications. Nevertheless, this model is widespread and hardware and operating system have evolved toward mechanisms allowing to create an efficient execution model for it. As a consequence, dataflow execution models are often implemented on top of the concurrent tasks with shared state execution model, in other words on top of threads.

Dataflow execution models are implemented by the dataflow compiler and the dataflow runtime. Many dataflow compilers first convert dataflow programs to multi-thread programs. Theses multi-thread programs are then linked against a dataflow runtime. In the case of dataflow computation models where scheduling can be and is done statically, the runtime role is only to ensure data synchronization. In this case, the schedule is defined inside the multi-thread program generated by the compiler. It corresponds to the order of actors execution in the program. For computation models requiring runtime scheduling, the runtime also has the responsibility to choose when to fire actors. Dataflow execution models will be described in detail in Section 4.3.

2.5 Objectives

Dataflow programming is one of the concurrent programming model that can help programming modern parallel architectures. This work focuses on this model because we believe that it has a strong play in the near future. We aim at providing mechanisms to ensure that throughput constraints expressed on dataflow programs are respected at runtime. Before introducing our proposal, we first recall the applicative domains where dataflow programming makes sense and present why our approach differ from existing work.

2.5.1 Dataflow Applicative Domains

Even if the dataflow model is forty years old, there is an increasing interest in dataflow with the recent advent of parallel architectures. Dataflow has been and is successfully used today in different domains.

First, the video codec community heavily uses the dataflow model. The Reconfigurable Video Coding (RVC) [40] initiative from the MPEG group provides an innovative framework for video coding development. Based on the dataflow programming model, this framework offers a way to overcome the lack of interoperability between the different video codecs deployed in the market. Because all the codec standards rely on a common set of basic computing elements the RVC framework provides a standardized version for them using the Reconfigurable Video Coding Caltrop Actor Language (RVC-CAL) dataflow language [41]. The framework also provides a standardized way to connect dataflow actors together to define a codec. A set of mature development tools including dataflow analyses and compilation for different targets are available for this framework [42]. Operational codec implementations targeting different architectures based on this framework are available [43, 44, 45, 46].

A second domain where dataflow is widely used is software defined radio. This

domain aims at providing software implementations of telecommunication protocols to reduce building costs and time to market and to increase flexibility. Many work based on different dataflow MoC depending on the protocol needs have been proposed recently. Dardaillon, Marquet, Risset & al. [39] propose a SPDF based implementation of the 3 GPP LTE-Advanced demodulation for heterogeneous architectures. Salunkhe, Moreira and Van Berkel [47] propose a dataflow based modeling of a 4G-LTE receiver and Pelcat, Aridhi, Piat & al. [48] propose a dataflow based approach for the LTE physical layer.

Dataflow is also used in many other domains. Among these domains we can mention cryptographic algorithms. Such algorithms naturally describe computation on input data and thus are easily expressed in a dataflow language [35, 49]. We can also mention distributed real-time processing frameworks [50] based on the dataflow model and used in companies such as Twitter and Spotify for big data processing.

2.5.2 Throughput Constraints in Dataflow Programs

Independently of the considered application domain, almost all dataflow applications have natural throughput requirements. In the video codec domain, decoders must produce a minimum number of frames per second. Regarding the next generation telecommunication protocols, receivers must process input frames at a given rate. The current frame must be fully processed before the next one comes in. Finally, in the big data processing domain, many applications must process dynamic events with real time constraints. Because Tweeter for example, generates hundreds millions of tweets each day, a tweet processing system used to forward messages according to hash tags must ensure a minimum throughput to be able to process all the new incoming messages.

In this context, we aim at ensuring, through runtime mechanisms, that dataflow applications respect a given throughput constraint which is specified by the application developer/administrator/deployer. Our work differ from existing work in the following ways:

- We don't rely on static estimations of actors execution time;
- We don't suppose that dataflow applications are executed alone;
- We propose application- **and** system-level profiling mechanisms to understand why the throughput constraint is not met and allowing to take decisions;
- We don't want to maximize the throughput but to ensure a minimum.

Existing works based on static estimations of actors execution time maximizing throughput of dataflow applications are numerous [51, 52, 53, 54, 55]. Static estimation of the time required to execute a piece of code is a complex domain research made even more difficult in the context of modern hardware used in open systems where applications may come in and out. Deep cache hierarchy, branch prediction mechanisms and unpredictable applications make these static analyses unusable in practice. For these reasons, in this thesis, we take a runtime approach relying on actors effective execution time measured by profiling mechanisms.

To alleviate the problem of static estimation of actors execution time and to take dynamic dataflow models into consideration, runtime mechanisms have been proposed recently [56, 57, 58, 59]. As the solutions proposed in this thesis, all these mechanisms

rely on runtime profiling to increase the throughput of dataflow applications. Compared to these works, we propose fine grain system-level mechanisms to understand how the dataflow applications interact with the underlying hardware. Moreover, instead of maximizing the throughput, we focus on ensuring a specified throughput constraints. In many contexts, going further a given throughput constraint may not be interesting. Indeed, when the applicative requirements only specify a minimal throughput we can save processing resources and energy by only ensuring this minimal constraint.

Finally, in the case of non respect of the constraints we propose mechanisms allowing to identify the sources of the bottlenecks. To the best of our knowledge, this problem is not yet addressed in the particular case of dataflow applications. These bottleneck identification mechanisms aim at providing valuable information for runtime adaptation heuristics.

2.5.3 Proposal and Hypotheses

With the throughput guaranteeing objective in mind, the first question we need to address is how to express throughput constraints on dataflow programs and how these constraints can be exploited depending on the underlying MoC. Expressing the throughput is a simple question of syntax extensions in the considered language but the exploitation of this constraint leads to interesting questions. As will be demonstrated in Chapter 3, static dataflow MoCs allow to take into account the throughput in the compilation tool chain. As a result, the compiler is able to provide useful information for the runtime to identify where bottlenecks are in the dataflow graph. In the case of dynamic MoCs, runtime mechanisms are required to detect bottlenecks. These runtime mechanisms have to inspect the time spent by actors waiting for input tokens or waiting for room in output FIFO channels.

Our second goal is to set-up profiling mechanisms for dataflow programs to avoid relying on static actors execution time analyzes and to identify where are the bottlenecks and what is there origin. These profiling mechanisms can serve two purposes. They can be used to provide feedback to either programmers or compilers in order to improve generated code efficiency. The profiling results can also be used by dataflow runtime adaptation mechanisms for throughput constraint achievement. In the context of dynamic dataflow models it is clear that profiling mechanisms may be required to identify how applications behave at runtime. Indeed, even in scenarios where the dataflow applications to be executed on a given platform are all statically known, the compiler can't anticipate the dynamism inherent to the model. In the case of static dataflow models, we argue that profiling mechanisms are also required for several reasons. First, even if actors consumption and production rates are statically known, the internal behavior of actors may still be data dependent. Secondly, as stated in the previous section, the complexity of modern hardware mechanisms such as multi level caches, branch prediction and deep instruction pipelines make static analyses of actors execution time very complex. Finally, our ultimate goal is to create a dataflow aware general purpose operating system where applications may come in and out at any time. In such context, profiling mechanisms are required to understand the impact on hardware of the interaction between dataflow and non dataflow applications.

The profiling mechanisms we propose can be classified as either architecture dependent or architecture independent. As stated in Section 2.4.3, thanks to the communica-

tion abstraction provided by the model, dataflow programs can be compiled and executed on both centralized [40, 60, 61] and distributed memory architectures [39, 56, 62, 63]. In this work we focus on architectures with a centralized memory organization for architecture dependent profiling mechanisms. These architectures are the most widespread in the desktop and servers markets targeted by the industrial context of this work. We attach a particular importance to homogeneous architectures with a distributed shared-memory memory organization as these architecture are the de-facto standard in the server market. On such architectures the dataflow compiler and the dataflow runtime need to take the memory organization into consideration to fully exploit the memory subsystem. To that end, the coarse grained communication information provided by the dataflow programming model has to be exploited.

3 Throughput Constraints in Dataflow Programs

Chapter's outline

3.1 Introduction

The previous chapter introduced the motivations leading to new concurrent programming models and justified our choice to focus on the dataflow model. We also presented our motivations to add throughput constraints to dataflow programs. In this chapter we show how these constraint can be expressed and how they can be statically exploited in the case of SDF graphs. SDF is widely used in the signal processing community to write complete applications [35] but can also be used in larger contexts to write the static parts of dynamic applications [64, 65]. Our main goal is, assuming that application developers or integrators will specify throughput constraints at very high granularity level, to take benefits from SDF static information to produce valuable information for throughput aware dataflow runtime systems. Adding throughput constraints to an SDF graph leads to new interesting static analyses. In particular, we show how we are able to propagate the throughput information provided with the application along the edges of the SDF graph to compute timing requirements for all the actors of an SDF graph.

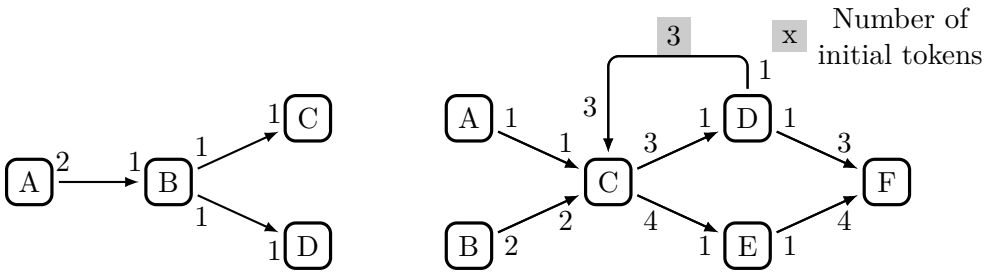
This chapter first presents existing work related to throughput constraints in dataflow graphs. It then introduces the SDF model in details and shows how SDF programs are scheduled on parallel architectures in Section 3.3. Section 3.4 then presents how throughput can be conceptually expressed on SDF programs and how we implemented these extension in the StreamIt framework. Section 3.5 then demonstrates how throughput constraints expressed on an SDF dataflow program can be propagated along the edges of the dataflow graph using the SDF schedule in order to compute local throughput values. Then Section 3.6 introduces the DDF model and shows how we extend the Open RVC-CAL Compiler (ORCC) framework to express throughput constraints. Finally, we conclude the chapter with several perspectives showing how our proposal could be extended to more expressive dataflow models.

We chose StreamIt to illustrate our SDF extensions because it includes both a SDF language with many applications available and a compiler infrastructure targeting different architectures and performing aggressive optimization transformations. Regarding the DDF extensions, we illustrate our proposal with the ORCC framework which is more and more used in the dynamic dataflow community.

3.2 Related Work

To the best of our knowledge, no existing work has proposed to express throughput constraints on SDF programs and use this information to provide timing requirements used by a throughput aware dataflow runtime system.

Nevertheless, statically computing the maximal theoretical throughput that can be achieved by an SDF program supposing an hardware platform with enough resources is an old research problem [66]. Indeed, a solution to compute the throughput consists in first converting the SDF graph to an Homogeneous SDF graph, a graph where all the consumption and production rates are equal to one, and then computing the maximum cycle mean of this transformed graph. The transformation from SDF to an homogeneous graph (a graph where all the consumption and production rates are equals) is always possible [33] and the throughput is the invert of the maximum cycle mean. More recent works [51, 53] propose a method to compute the theoretical maximal throughput directly on the SDF graph. Other works [52, 54, 55, 67] propose static solutions to find multicore SDF schedules trying to maximize the throughput. All these works rely on the execution



(a) Acyclic basic SDF graph **(b)** Cyclic SDF graph with initial tokens

Figure 3.1: SDF graphs examples

time of the actors composing the application. This execution time can either be obtained from profiling or code analysis.

In contrast to static works, Tan's proposal [68] defines runtime mechanisms to measure SDF actors effective execution times do perform online remapping. Compared to our work, Tan's main goal is to increase throughput where as we want to ensure a minimal throughput by using static information provided at applications level. Runtime identification of SDF bottleneck actors along with adaptations consisting in adding parallelism for these bottleneck actors have been proposed recently [56, 57]. In these works, bottleneck actors identification is based on FIFO channels filling levels and not on static information as in our proposal.

3.3 The SDF Model

SDF first introduced by Lee [33, 34] is a specialization of the general dataflow model rooted in the signal processing domain.

3.3.1 SDF MoC

In SDF graphs, nodes always consume (respectively produce) the same number of tokens on each input (respectively output) FIFO channel. These numbers are called respectively the popRate and the pushRate of the SDF actors. Figure 3.1 shows two examples of SDF graphs. SDF actors may have zero, one or more input and output ports. SDF graphs may also contain cycles. In the following we note:

- $popRate(act, in)$ the number of tokens consumed by the SDF actor act on the input FIFO channel in

- $pushRate(act, out)$ the number of tokens produced by the SDF actor on the output FIFO channel out.

Static rates allow different static analyses on SDF programs. The first one checks if an SDF graph is consistent or not. The consistency property means that the program can be executed infinitely without requiring infinite memory. The second analysis, for consistent graphs, identifies if the execution of an SDF graph can deadlock or not. Indeed, cyclic graphs may deadlock because of insufficient initial tokens on backward edges inside the graph such as the edge from actor D to actor C in Figure 3.1b.

3.3.2 SDF Scheduling

SDF graphs can be scheduled statically with bounded memory requirements. These schedules statically generated can be either sequential or parallel.

3.3.2.1 Sequential Schedules

A valid sequential schedule is an ordered list of actor firings guarantying finite memory usage and the absence of deadlocks. To compute a schedule, the dataflow compiler must first compute the *minimum repetition vector* noted p. This vector defines the minimum number of times each actor must be executed so that the number of tokens in all the FIFOs of the application falls back to its initial state. The repetition vector can be derived by finding the minimum positive integer solution to the *balance equations* for the SDF graph specifying that for every edge e in the graph connecting actors src and $dest$ p must satisfy:

$$p(src) * pushRate(src, e) = p(dest) * popRate(dest, e) \tag{3.1}$$

Executing a SDF graph consists in an infinite loop, or a loop running while there are more inputs to be proceed, executing each actor according to the repetition vector p. For any repetition vector q different from p we define the *blocking factor* noted J with Equation 3.2.

$$q = J * p \tag{3.2}$$

As described in the next paragraph, this blocking factor can help in the generation of more efficient parallel schedules by increasing the execution overlap between cores. $rep1$ (respectively $rep2$) shown below is the minimal repetition vectors for the example of Figure 3.1a (respectively of Figure 3.1b).

$$rep1 = (1, 2, 2, 2)\ for\ actors\ (A, B, C, D)$$
$$rep2 = (1, 1, 1, 1, 3, 4, 1)\ for\ actors\ (A, B, C, D, E, F)$$

From the repetition vector, the compiler computes a *steady-state schedule* or *iteration* that satisfies actors data dependencies. This schedule consists in a sequence of components. Each component act^x indicates to execute x times actor act. $sched1a$ and $sched1b$ shown below are two valid steady-state schedules for example of Figure 3.1a and $sched2a$ and $sched2b$ are two valid steady-states schedules for example of Figure 3.1b. Actors may appear only once in schedules such as in $sched1a$ and $sched2a$ or several times interleaved with other actors execution such as in $sched1b$ and $sched2b$.

$$sched1a = A^1B^2C^2D^2$$
$$sched1b = A^1B^1C^1D^1B^1C^1D^1$$

$$sched2a = A^1B^1C^1D^3E^4F^1$$
$$sched2b = A^1B^1C^1D^1E^2D^2E^2F^1$$

In the remainder of this thesis, because the compiler checks for graphs' consistency when computing the repetition vector, we consider only consistent graphs and we use the following notations:

- $q(act)$ denotes the entry corresponding to actor act in the repetition vector q; e.g. $q(B)$ in the vector $rep1$ above is 2 ;

- $activ(comp)$ denotes the number of activation of a component $comp$ in the steady-state schedule, i.e. $activ(act^x) = x$; e.g. in $sched2a$ above $activ(E^4) = 4$;

3.3.2.2 Schedules Properties

Two notable properties of SDF schedules are code memory requirements and FIFOs channels memory requirements. To minimize code memory requirement, the compiler must compute a Single Appearance Schedule (SAS) schedule. SAS are schedules where each actor only appears once. They implement the full repetition inherent in an SDF graph without requiring subroutines or code duplication. For acyclic graphs, a topological sort of the graph along with the repetition vector gives the SAS. For cyclic graph, it has been shown that an SAS exists only if the graph is weakly connected and algorithms to compute these schedules have been proposed [69].

FIFO channels memory requirements is the second notable property for an SDF schedule. It is informally defined as the sum of the maximum amount memory required for each channel. In our example, $sched1a$ has channels memory requirements of 6. This schedule has a maximum of 2 tokens in channel connecting A and B, 2 tokens in channel connecting B and C and 2 tokens in channel connecting B and D. $sched1b$ has channels memory requirements of 4. It has a maximum of 2 tokens in channel connecting A and B, 1 token in channel connecting B and C and 1 token in channel connecting B and D. Finding the minimal memory usage schedule has been showed to be an NP-Complete problem and heuristic must be used to compute efficient memory schedules [69].

3.3.2.3 Multi-core Schedules

When targeting multi-core systems, the sequential schedule introduced above has to be split between the available cores. A parallel schedule thus consists in a set of sub-schedules: one for each core of the target system. In sequential schedules, data precedence is enforced by the schedule. In the case of parallel execution of an SDF graph, actors must be synchronized in order to enforce the integrity of the schedule. This is done by synchronization mechanisms at runtime as described in Section2.4.3.2.

In parallel schedules, the load balancing objective is usually also considered in addition to code and buffers memory requirements. Intuitively, load balancing aims at minimizing the longest schedule among all the cores. For this goal to be taken into account statically, we need to know the execution time of each actor firing. This can be obtained by static code analyzes or by code profiling. The construction of a parallel schedule is divided in two stages [34]. First, an acyclic precedence graph must be constructed. Then, this graph can be used to map actors to cores so that precedence relations are satisfied. This need to be done while minimizing the longest schedule among all the cores. This problem is identical to assembly line problems in operations research. It can be solved for the optimal schedule, but the problem is NP-complete. This is not problematic for small SDF graphs. For large ones heuristics have been proposed that work well in practice. Note that increasing the blocking factor can help in minimizing the iteration period. We now describe our throughput constraints proposal that is valid for both sequential and parallel SDF schedules.

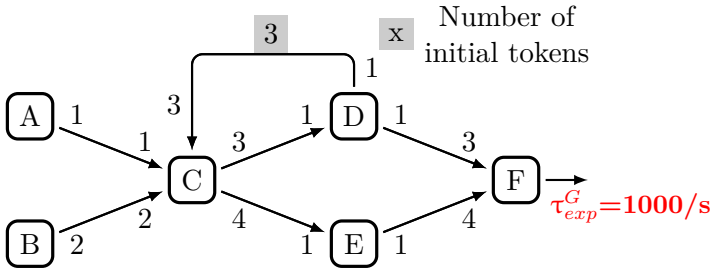


Figure 3.2: SDF graph example with τ^G_{exp} constraint

3.4 Extending SDF With Throughput

Throughput is defined as the rate at which tokens can be processed or the rate at which something can be produced. In this work we rely on throughput constraints expressed at dataflow source programs level. These constraints should be specified either by the programmer or a system integrator with the knowledge of the expected throughput for the dataflow applications considered. For a dataflow graph $G = (V, E)$ we define the global expected throughput for a FIFO channel $c \in E$, noted $\tau^G_{exp}(c)$, as the number of tokens produced per unit of time in the FIFO channel. We intentionally don't specify the time unit here, because some applications may have requirements at microsecond granularity whereas others will use seconds. This section first introduces the notion of global throughput for dataflow programs independently of any specific MoC or any specific language. Then we show how we have implemented throughput expression in StreamIt with simple syntactic extensions and in the DDF language supported by the ORCC framework.

3.4.1 Extensions at MoC Level

The global expected throughput value for an SDF or a DDF graph, noted $\tau^G_{exp}(c)$, is expressed on one channel of the dataflow graph. Figure 3.2 illustrates the notion of global expected throughput on an SDF example. The program is annotated with information specifying that the final actor must produce 1000 tokens per second. In the case of SDF graphs, throughput can be specified on any of the channel of the SDF graph because of the consistency property described in the previous section. As will be described in the next section, if the graph is consistent the throughput expressed on any channel of the graph will define all the other channels throughput as described in the Section 3.5. If several throughput are defined, we check for their consistency. In the case of DDF graphs, several throughput constraints can be expressed on the graph, and the runtime mechanisms described in Chapter 4 will have to check all of them.

Depending on the dataflow language implementation, sink actors may or may not be have explicit output FIFO channels. On the example depicted on Figure 3.2 we suppose that sink actors have these output FIFO channels. If these channels are not explicitly defined and the throughput constraint is only known for sink actors, extensions to the language are required to allow the last actor in the graph to express the expected throughput value or a new identity actor (don't doing anything) may be added to create an output FIFO channel. This is an implementation concern and we now describe how we solve it for the StreamIt language.

3.4.2 StreamIt Extensions

StreamIt [60, 70] is a dataflow programming language rooted into SDF bringing some extensions to the initial model. It allows to write in the same language both the dataflow network and the actor's internals. This section briefly introduces the language and then shows how we extend it to let applications developers specify a throughput constraint.

3.4.2.1 The StreamIt Language

Applications written in StreamIt are made of actors which are called filters in the StreamIt terminology. StreamIt provides one unique language to write the internal body of actors and to create the network of filters. The first particularity of StreamIt concerns the dataflow graph topology. Filters can't be connected arbitrarily. They have one unique input stream and one unique output stream and must be connected through the use of connectors provided by the language. Figure 3.3 shows that:

- `Pipeline` connectors allow to connect actors in sequence.
- `Split` and `join` connectors are used to create parallel branches in the dataflow graph. Split can be either duplicating split or round robin split. A duplicate `split` just forwards each one of its input tokens to every filter connected to the split. A round robin `split` distributes its inputs to its successors according to a pattern specified statically. On the example of Figure 3.3, the second `split` is a round robin one with a pattern defined by $(1, 1, 1)$ specifying that the split equally distributes its 3 input tokens to the three output channels. `Join` connectors are round robin connectors.
- `Feedbackloop` connectors allow to create cycles in the graph. A loop is constructed using a split and join pair splitting and joining on exactly two streams as shown on Figure 3.3. In this example, the filter B represents the loop's body and the filter C is a filter that may affect or not the token fed back to the loop's joiner. C can be an identity filter - a filter just forwarding its inputs - in the case where the loop output must be provided back without any modification. Loops also have initial tokens. These tokens are required to initially start the application.

The second extension brought by StreamIt to the SDF model is the notion of peeking filter. A StreamIt filter can read more tokens than the number of consumed tokens to produce its output. This extension allows to easily write sliding windows actors such as a moving average. Thus, in addition to their *pushRate* and *popRate* information, filters have a *peekRate* information. Compared to the standard SDF model, this extension only requires an initialization stage at the beginning of execution to ensure enough tokens are present on FIFOs to execute the graph.

3.4.2.2 Language Extensions

To specify the expected throughput of StreamIt applications we extend the part of the language used to create the network of actors. The language provides a special `add` keyword used to add actors to the graph. Figure 3.4a shows StreamIt source code used to create the graph of the example of Figure 3.3. Figure 3.4b shows the source code for the creation of the same graph but with a throughput constraint expressed on the last channel of the graph. The throughput constraint is expressed using the additional `mon`

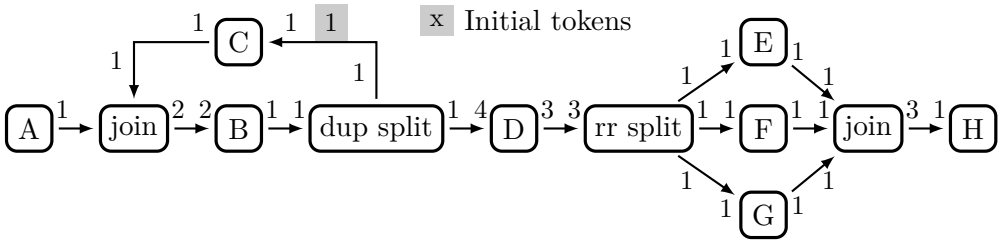


Figure 3.3: StreamIt graph example illustrating split/join, pipeline and feedback loop constructors

```
add(A());
add(Feedbackloop(){
  add(Join(RR, (1,1)));
  add body(B());
  add feedback(C());
  add(Split(RR, (1,1)));
});
add(D());
add(SplitJoin(){
  add(Split(RR, (1,1,1)));
  add(E());
  add(F());
  add(G());
  add(Join(RR, (1,1,1)));
});
add(H());
```

(a) StreamIt original program

```
add(A());
add(Feedbackloop(){
  add(Join(RR, (1,1)));
  add body(B());
  add feedback(C());
  add(Split(RR, (1,1)));
});
add(D());
add mon 1000 s(SplitJoin(){
  add(Split(RR, (1,1,1)));
  add(E());
  add(F());
  add(G());
  add(Join(RR, (1,1,1)));
});
add(H());
```

(b) StreamIt extended program

Figure 3.4: StreamIt language extensions

(for monitored) keyword with the expected throughput in tokens per time unit when adding the filter to be monitored. The time unit is chosen among s ms µ and ns. In this example, we specify $\tau^G_{exp} = 1000$ tokens per second on the channel connecting the second joiner and H.

3.4.2.3 StreamIt Graph's Transformations Follow-up

The StreamIt compilers performs aggressive graph's transformations to produce implementations improving throughput or memory usage according to the targeted platform [63, 71]. Indeed, the static rates along with the structured graph allow to fuse and split filters. In such cases, the compiler needs to correlate the initial global throughput information expressed on the application with the final graph produced by the compiler [2].

Two kinds of graph transformation are performed by dataflow compilers for SDF languages. The first one consists in fusing actors. Fusion is used for example when the number of actors is larger than the number of execution units dedicated to the application. Fusion can merge consumer/producer pairs of actors into a single actor as shown on Figure 3.5. In this case, the compiler puts in sequence in a new actor the code of the fused actors.

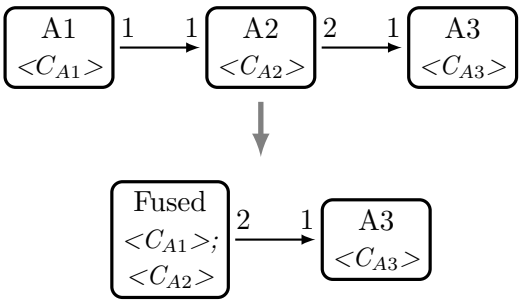


Figure 3.5: SDF fusion, consumer and producer filters directly connected can be fused in one bigger filer.

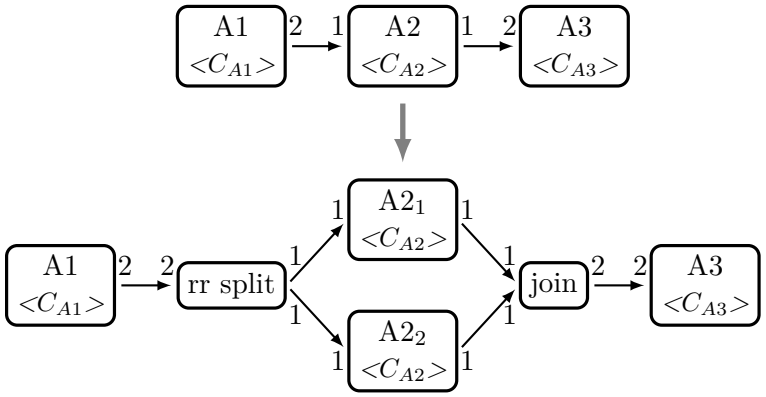


Figure 3.6: Data parallelism introduction. Stateless filters can be duplicated to work on several data tokens in parallel. The compiler ensure that the outputs of the parallel version of the filter are provided in the same order as the sequential one using a join to downstream filters.

The second class of transformation adds parallelism to the dataflow graph. Figure 3.6 shows how data parallelism is added to a graph. Stateless actors are duplicated to allow them to work on several data sets in parallel. The compiler is also able to introduce pipeline parallelism by splitting one actor into a pipeline of two smaller actors.

When the compiler applies such transformations, the information of throughput on FIFO channels must be correctly preserved.

To handle the fusion case, we simply forbid the compiler to fuse two actors if they are connected by a FIFO channel where a throughput constraint has been expressed. In practice, as will be described in Chapter 5 this constraint has no impact because we add an identity actor at the end of the StreamIt applications to express the throughput. In other words, this constraint only prevent the fusion of this identity actor with its predecessor.

Figure 3.7 shows how the compiler preserves the throughput information when a stateless actor is parallelized. When the initial actor's output channel is tagged with throughput information, we add a throughput information on each created duplicate. In the case of pipeline parallelism introduction, the expected throughput information of the split actor is reported on the last actor of the created pipeline.

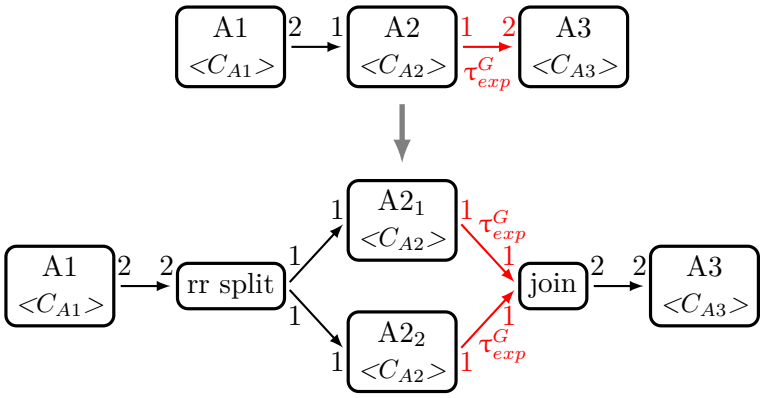


Figure 3.7: Data parallelism introduction follow-up to propagate expected throughput. Each duplicate has its output channel tagged with expected throughput information.

3.5 SDF Throughput Propagation

In the case of SDF programs, we can use the static consumption and production property of the actors to statically compute a local expected throughput, noted $\tau^L_{exp}(c)$, for all FIFO channels c of the transformed graph from τ^G_{exp}. $\tau^L_{exp}(c)$ is the number of tokens that must be produced on the FIFO channel c per unit of time to satisfy τ^G_{exp}. Because actors production rates are static in SDF, computing $\tau^L_{exp}(c)$ values translates to computing the number of time each actor must be fired per unit of time in order to satisfy τ^G_{exp}. We define this required frequency noted $f_r(act)$ for an actor act as the required number of actor's firings per unit of time. The link between $\tau^L_{exp}(c)$ and $f_r(act)$ is defined by Equation 3.3 where act is the actor feeding c.

$$\tau^L_{exp}(c) = f_r(act) * pushRate(act, c) \tag{3.3}$$

From this we can compute an average expected execution time for the actors noted $\kappa_{exp}(act)$ and defined by Equation 3.4.

$$\kappa_{exp}(act) = \frac{1}{f_r(act)} = \frac{pushRate(act, c)}{\tau^L_{exp}(c)} \tag{3.4}$$

Depending on how the SDF program is executed, this expected execution time maybe required to take benefit of the propagation. Consider for example a sequential execution using the SAS schedule of an SDF application. If an actor in the middle of the graph is too slow to produce it's outputs, it will slow down all the subsequent actors. The too low throughput on the slow actor's output FIFO channels will automatically lead subsequent actors to wait for data tokens. We can't identify which actor among the guilty actor and its successors is the bottleneck. In this case, measuring the execution time of one actor firing and comparing this execution time to its κ_{exp} values allows us to identify it as a bottleneck.

Table 3.1 sums up all the notations introduced so far and used in the next two sections by two different algorithm for throughput propagation.

Notation	Definition
$\tau^G_{exp}(c)$	global expected throughput on channel c
$\tau^L_{exp}(c)$	local expected throughput on channel c
$\kappa_{exp}(act)$	expected execution time of actor act
$f_r(act)$	required activation frequency of actor act

Table 3.1: Notations about throughput used in this thesis.

3.5.1 Propagation By Graph Traversal

The first algorithm we present to propagate the τ^G_{exp} is a naive graph traversal. A depth first version is presented in Figure 3.8. The `propagate` function is first called on the channel where the τ^G_{exp} is expressed with the actor feeding this channel. This function first computes (line 2 to 24) the τ^L_{exp} for all the input and output channels of actor act where it is not already computed using $\tau^L_{exp}(c_{ref})$. The τ^L_{exp} value for a channel c depends on the type of c, i.e. if it is an output or an input channel, and on the type of c_{ref}. The function then marks the actor act as visited (line 25). Finally the function recursively calls itself (line 26 to 35) on all the neighbors of act not already visited.

Considering weakly connected dataflow graphs, the channel where τ^G_{exp} is expressed can be any channel of the graph. In any case, this algorithm traverses the graph until all nodes have been visited. In the particular case where more than one τ^G_{exp} values have been specified, lines 4, 9, 15 and 20 must check that the throughput expressed are consistent with the computed ones instead of only checking if the channel is already tagged or not.

Applying the `propagate` function on the example of Figure 3.2 leads to a graph traversal in the order F, D, C, A, B, E. The result is depicted on Figure 3.9.

We now present a simpler solution to propagate τ^G_{exp} consisting in using the repetition vector introduced in Section 3.3.2.

3.5.2 Propagation Using SDF Repetition Vector

τ^L_{exp} values can also be computed from the repetition vector. Because all the dataflow compilers for SDF languages must compile this repetition vector to compute the schedule as described in Section 3.3.2, we favor this solution. This avoids navigating the graph as described in the previous section.

In this solution, we first compute the required frequency of all the actors. To compute these required frequency we only need the repetition vector of the schedule and don't care about actors firings order. Equation 3.5 defines this required frequency for the actor where τ^G_{exp} is defined.

$$f_r(act_{\tau^G_{exp}}) = \frac{\tau^G_{exp}(c)}{pushRate(act_{\tau^G_{exp}}, f)} \tag{3.5}$$

We can then compute the required frequency for all the actors with Equation 3.6.

$$f_r(act) = f_r(act_{\tau^G_{exp}}) * activRatio(act) \tag{3.6}$$

1: propagate(c_{ref}, act):
2: **if** c_{ref} is an output channel **then**
3: **for** c_o in act.outputChannels **do**
4: **if** c_o is not tagged **then**
5: $\tau^L_{exp}(c_o) = \frac{\tau^L_{exp}(c_{ref}) * pushRate(act, c_o)}{pushRate(act, c_{ref})}$
6: **end if**
7: **end for**
8: **for** c_i in $act.inputChannels$ **do**
9: **if** c_i is not tagged **then**
10: $\tau^L_{exp}(c_i) = \frac{\tau^L_{exp}(c_{ref}) * popRate(act, c_i)}{pushRate(act, c_{ref})}$
11: **end if**
12: **end for**
13: **else**
14: **for** c_i in $act.inputChannels$ **do**
15: **if** c_i is not tagged **then**
16: $\tau^L_{exp}(c_i) = \frac{\tau^L_{exp}(c_{ref}) * popRate(act, c_i)}{popRate(act, c_{ref})}$
17: **end if**
18: **end for**
19: **for** c_o in $act.outputChannels$ **do**
20: **if** c_o is not tagged **then**
21: $\tau^L_{exp}(c_o) = \frac{\tau^L_{exp}(c_{ref}) * pushRate(act, c_o)}{popRate(act, c_{ref})}$
22: **end if**
23: **end for**
24: **end if**
25: mark actor visited
26: **for** $inputAct$ in $act.inputFifos.sources$ **do**
27: **if** $inputAct$ is not visited **then**
28: propagate($inputAct$)
29: **end if**
30: **end for**
31: **for** $outputAct$ in $act.outputFifos.dests$ **do**
32: **if** $outputAct$ is not visited **then**
33: propagate($outputAct$)
34: **end if**
35: **end for**

Figure 3.8: τ^G_{exp} propagation in SDF

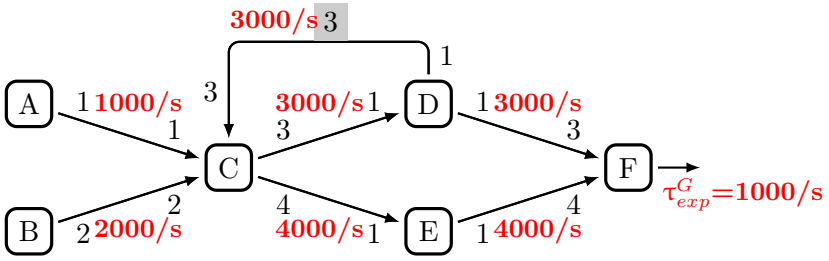


Figure 3.9: Throughput propagation result

where $activRatio(act)$ denotes the ratio of the number of activation of the actor act to the number of activation of the actor $act_{\tau^G_{exp}}$ (the actor where τ^G_{exp} is expressed) as shown on Equation 3.7.

$$activRatio(act) = \frac{q(act)}{q(act_{\tau^G_{exp}})} \tag{3.7}$$

The required frequency only gives for an SDF schedule the required frequency at actor granularity. We now easily compute the final schedule required frequencies at component level, noted *comp*, in an SDF schedule using Equation 3.8.

$$f_r(comp) = f_r(act) * compRatio(act) \tag{3.8}$$

where $compRatio(act)$ denotes the ratio of the number of activation of the actor act in the component $comp$ to the total number of activation of the actor act as defined by equation 3.9.

$$compRatio(act) = \frac{activ(comp)}{q(act)} \tag{3.9}$$

We now illustrate the propagation on the example of Figure 3.2. We first compute the required frequency for the actor F where τ^G_{exp} is defined using Equation 3.5.

$$f_r(F) = \frac{\tau^G_{exp}(c)}{pushRate(F, f)} = 1000$$

Then using Equations 3.6 and 3.7 we can compute $f_r(A)$, $f_r(B)$, $f_r(C)$, $rf_r(D)$, $rf_r(E)$. For example for actor C we get:

$$f_r(C) = f_r(F) * activRatio(C) = 1000$$

Then we can compute the τ^L_{exp} values for the two output channels of actor C noted c_{CD} and c_{CD} using Equation 3.3:

$$\tau^L_{exp}(c_{CD}) = f_r(C) * pushRate(C, c_{CD}) = 3000\, tokens/s$$

$$\tau^L_{exp}(c_{CD}) = f_r(C) * pushRate(C, c_{CE}) = 4000\, tokens/s$$

We can also compute the required frequency for each component in the two valid schedules introduced in Section 3.3.2.1 and recalled here:

$$sched2a = A^1B^1C^1D^3E^4F^1$$

$$sched2b = A^1B^1C^1D^1E^2D^2E^2F^1$$

Using Equations 3.8 we get the following results for the first schedule $sched2a$:

$$f_r(A^1) = f_r(A) * compRatio(A^1) = 1000$$

$$f_r(B^1) = f_r(B) * compRatio(B^1) = 1000$$

$$f_r(C^1) = f_r(C) * compRatio(C^1) = 1000$$

$$f_r(D^3) = f_r(D) * compRatio(D^3) = 1000$$

$$f_r(E^4) = f_r(E) * compRatio(E^4) = 4000$$

$$f_r(F^1) = f_r(F) * compRatio(F^1) = 1000$$

This results in the same required frequency as the required frequency computed globally at actors level because $sched2a$ is a SAS. For $sched2b$ which is not a SAS we get the following results:

$$f_r(A^1) = f_r(A) * compRatio(A^1) = 1000$$

$$f_r(B^1) = f_r(B) * compRatio(B^1) = 1000$$

$$f_r(C^1) = f_r(C) * compRatio(C^1) = 1000$$

$$f_r(D^1) = f_r(D) * compRatio(D^1) = 1000$$

$$f_r(E^2) = f_r(E) * compRatio(E^2) = 2000$$

$$f_r(D^2) = f_r(D) * compRatio(D^2) = 1500$$

$$f_r(E^2) = f_r(E) * compRatio(E^2) = 2000$$

$$f_r(F^1) = f_r(F) * compRatio(F^1) = 1000$$

The propagation presented in this section is usable when targeting multi-core architectures because it is done at actors components level. Indeed, a multi-core SDF schedule is made of several single core schedules. Each one of this single core schedule is itself made of actors components.

3.6 Extending DDF With Throughput

The extensions described in Section 3.4.1 are valid for all also valid for DDF graphs. Nevertheless, the throughput propagation introduced in Section 3.5 can't be applied to DDF graph because of their dynamic nature. This section describes the DDF model in details and show how we extended the ORCC framework with throughput constraints.

3.6.1 The DDF Model

Compared to SDF, actors in a DDF graph don't have to consume and produce the same number of tokens each time they are fired. This dynamic behavior is expressed through the use of multiple firing rules. Such a rule specifies a condition on the number of tokens that must be present on the actors input FIFO channels to be fired. Each actor with s input edges with $s > 0$ can have N firing rules, where N is unbounded. The set of firing rules for an actor is noted:

$$\Upsilon = \{R_1, R_2, ..., R_N\}$$

A firing rule is a set of patterns, one for each input edge and is noted:

$$R_i = \{P_1, P_2, ..., P_s\}$$

We don't give here the formal definition for a pattern P_j, but intuitively it defines an acceptable sequence of tokens on the j^{th} input edge. The special wildcard pattern noted $*$ is used to indicate that at least one token is required on the associated input edge. The other special symbol $\perp$ denotes any sequence of tokens, including the empty one. An actor can fire if and only if one or more of its firing rule is satisfied. When several rules are available at the same time, the choice of which rule to execute is left to the runtime implementation.

In practice, in addition to specify input patterns, firing rules specify also output patterns. Indeed, the theoretical model rely on infinite FIFO channels where as implementation must work with finite memory. Output patterns allow to indicate whether or not there is enough room in output channels for an actor firing.

A fundamental difference between SDF and DDF is that DDF allows observable non determinism by the use of non deterministic actors. Non deterministic actors are actors with output depending on the time at which inputs are produced. Non determinism may be desirable to construct dataflow programs interacting with multiple external events. This non determinism is expressed through the use of non sequential firing rules as described in [32]. Informally, rules are sequential if they can be tested in a pre-defined order using only blocking reads. Figure 3.10 shows an example of a non deterministic merge actor having two inputs and one output. This actor has two non sequential firing rules, one for each input requiring at least one token on this input:

$$\Upsilon = \{R_1, R_2\}$$

$$R_1 = \{[*], \perp\}, R_2 = \{\perp, [*]\}$$

For this actor, the sequence of output depends on the time of inputs arrival because this actor just forwards its inputs in arrival order if it has the time to be fired between two inputs arrival. Otherwise in the case where the two rules are satisfied at the same time because there is at least one input on both inputs, the behavior of the actor is unspecified.

The DDF model is the most expressive of the dataflow computation models. Compared to SDF it doesn't enforce any restriction at all on the number of tokens consumed and produced by each actor firing (through the use of several firing rules for an actor). As a consequence, this model is also the least analyzable. It is neither possible to statically schedule a DDF graph nor statically bound FIFO sizes nor to statically identify

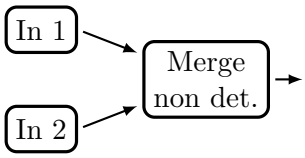


Figure 3.10: DDF non deterministic merge actor. This actor forwards its inputs as soon as available on its unique output channel. The output flow depends on the time at which inputs are available.

```
<XDF name="Top_mpeg4_part2_SP_decoder">
    <Instance id="decoder">
       <Class name="org.sc29.wg11.mpeg4.part2.sp.RVC_decoder"/>
    </Instance>
    <Instance id="source">
       <Class name="org.sc29.wg11.common.Source"/>
    </Instance>
    <Instance id="display">
       <Class name="org.sc29.wg11.common.DisplayYUV"/>
    </Instance>
    <Conn dst="decoder" dst-port="bits" src="source" src-port="O"/>
    <Conn dst="display" dst-port="B" src="decoder" src-port="VID">
       <ExpectedThroughput value="25" unit="second">
    </Connection>
    <Conn dst="display" dst-port="W" src="decoder" src-port="W"/>
    <Conn dst="display" dst-port="H" src="decoder" src-port="H"/>
</XDF>
```

Figure 3.11: ORCC program annotated with throughput constraint

deadlocks on feedback loops. Because of its dynamic nature, runtime mechanisms are required to execute correctly a DDF application. The actors scheduler is responsible at runtime to check which actors can be fired (i.e. having at least one firing rule enabled) and to fire them. Chapter 4 presents how this scheduling can be implemented.

3.6.2 ORCC extensions

ORCC [42] is a framework dedicated to write and execute DDF applications. We will not dig into the details of ORCC in this chapter but we just give an example of an annotated graph. Chapters 4 and 5 will deeply introduce this framework.

In ORCC, actors internal is written using the RVC-CAL programming language and actors graphs are described using an xml description called XDF. To express the throughput constraints on ORCC applications we simply extend this xml description with a new attribute for connection nodes. Figure 3.11 shows an annotated version of the top level dataflow graph for an MPEG4 part 2 video decoder. The channel connecting the decoder and the display actors is annotated with a throughput constraint of 25 frames per second.

This section as shown how we implement throughput expression in the StreamIt language and in the ORCC framework. We now switch back to the general SDF model to show how the throughput information can be propagated all along the dataflow graph using the static information provided by the model.

3.7 Discussion

This chapter has first introduced how a throughput constraint can be expressed on a dataflow program. Then we showed how this information is exploited at compile time for SDF programs to propagate a throughput constraint all along the dataflow graph. These propagated constraints can then be used to compute an average required execution time for all the dataflow actors of the program. This information provides a simple mean to let runtime mechanisms identify bottleneck actors. Indeed, comparing the statically computed average execution time to satisfy the throughput with the measured execution time indicates if an actor is a bottleneck or not. Chapter 4 describes in details how we use this information.

The throughput expression introduced in this chapter is valid for any dataflow MoC. Indeed, this throughput constraint is just a value on a specific edge of the dataflow graph, and the concept of edge is present in all the dataflow MoCs. Nevertheless, the propagation algorithm is valid only for SDF graphs where consumption and production rates are statically known. Even if many widespread computations such as a fast Fourier transform, a finite impulse response filter, a Cholesky decomposition, or CRC encoder are naturally expressed using SDF real life applications are rarely fully expressible using this model. As a consequence, extending the proposal introduced in this chapter to models allowing to express more application and preserving as much static analyzes as possible seems an attractive perspective. In particular, it would be interesting to identify how this throughput propagation algorithm allowing to compute statically an average execution time for all the actors could be extended to CSDF [36] SADF [37] and SPDF [38].

Regarding DDF models it's clear that the expected throughput value can't be propagated to compute local information about the actors. A first interesting perspective to extend our proposal to dynamic models consist in studying the impact of including our proposal in the static sub graphs of DDF graphs. Another solution to address dynamic models consists in setting up runtime mechanisms to identify where are the bottlenecks in the dataflow graph. The next chapter presents these runtime mechanisms.

4 Dataflow Programs Profiling

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories instead of theories to suit facts

Sherlock Holmes in "A Study in Scarlet" by Arthur Conan Doyle

Chapter's outline

4.1 Introduction

Profiling is a crucial activity to get insight of how a program behaves when executed on the final target. It consists in runtime mechanisms measuring and gathering different metrics related to the execution of programs. These metrics may then be used either to provide feedback to programmers or compilers to improve the result of compilation or by runtime mechanisms to perform adaptations. In the first case, profiling is required because static tools are not able to model the complete interaction that the program will have at runtime with the hardware. This is mainly due to several complex mechanisms built into modern processors such as caches, long instruction pipelines, out of order execution or branches prediction. In the second case, profiling allows runtime mechanisms to handle the dynamism inherent to general purpose systems where applications may come in and out at of the system any time. Other sources of dynamism such as changes of computing requirements inside applications may also lead to profiling needs.

The previous chapters introduced the notion of throughput constraints on dataflow programs and how this information initially expressed on one FIFO channel of the program could be statically propagated to the whole graph in the case of SDF applications. This chapter focuses on runtime profiling and assumes that for both dynamic and static dataflow models the dataflow runtime knows at least the expected throughput on one of the FIFO channels of the graph. In this chapter, we also assume that a general purpose best effort operating system is running on top of a homogeneous shared-memory multi-core hardware.

Using the throughput constraint information alongside the dataflow graph structure, we introduce runtime mechanisms required to monitor the effective throughput of dataflow programs during execution and mechanisms required to build meaningful profiling results. Most of these mechanisms can be used both for SDF and DDF applications but with methodologies that vary depending on the MoC. As a consequence, we intentionally don't specify in details in this chapter how these mechanisms are used together. Chapter 5 will give details on how we used it to built a StreamIt throughput aware runtime and to profile RVC-CAL DDF applications.

Section 4.2 first presents work related to the profiling of dataflow application. Section 4.3 introduces the execution model we consider to run dataflow applications on multi-core hardware. Section 4.4 then shows how the runtime can measure the effective throughput for comparison with the expected one in order to check whether the application's requirements are satisfied or not. In case of non respect of the expected throughput we show how to identify bottleneck actors for SDF graphs. Section 4.5 finally introduces profiling mechanisms at system level regarding processor and memory usage to identify the source of bottlenecks.

4.2 Related Work

Application profiling is a common practice because it's the only way to get insight about how an application behaves when executed on complex general purpose multi-core processors or alongside other applications. Indeed, processors are now so complex that statically modeling the exact runtime behavior of an application is a very hard task. Moreover, static analyses can't predict the interaction with other applications that may come in and go out at any time in an open system. In this chapter, we focus on the profiling of dataflow applications executed on top of general purpose multi-core

processors with a shared-memory organization. The profiling mechanisms we present serve two main goals. We want to identify how the computing resources are distributed among the dataflow actors and we want to identify communication bottlenecks and their origins. This section reviews existing work related to these goals starting with the profiling of dataflow actors computing resources usage.

Turnus [72] is a profiler specifically designed for dynamic dataflow programs written in the RVC-CAL language. This is a high level profiler mainly targeting to identify which actors of the source program need to be optimized and which mapping of the application may improve performance. This profiler proposes different analyses all based on an execution trace, called the causation trace. This causation trace is an acyclic oriented graph where nodes represent actors firings and edges represent firing dependencies. This trace is obtained by simulation and concerns only a given run of the application on a given input stimulus. The dependencies between actors may be either tokens dependencies between two actors, or actors state dependencies between two firing of the same actor. In addition to firings dependencies, the causation trace also includes actors firings execution times. This execution time, represents the number of RVC-CAL instructions executed by the actor. The execution trace is thus a weighted graph. The execution trace contains only the dependencies required to ensure the correctness of the program. Incidental dependencies such as the limited number of physical resources able to run the program are not captured by the causation trace.

From this trace, the compiler computes the critical path. It represents the heaviest path on the graph between all the combination of source and sink actors. The profiler also proposes methods to estimate how much the critical path length could be improved by each actor in the critical path to help the programmer focusing on the actors needing refactoring. Turnus also proposes heuristics targeting different goals to map the actors onto computation units of a target platform using the execution trace. Compared to Turnus, the mechanisms we propose add profiling of application throughput and profiling of memory usage. Like Turnus, the profiling mechanisms presented in Section 4.5.1 aims at identifying potential imbalance between the computing resources provided by the hardware.

The TAU (Tuning and Analysis Utilities) Parallel Performance System [73] is a large framework dedicated to the performance analyses of parallel systems. Compared to the proposition we make in this chapter focusing on dataflow programs executed on shared-memory multi-core architectures, the main objective of TAU is to provide abstraction about performance analyzes in the context of different parallel architectures and different concurrent programming models.

Regarding memory profiling, many solutions [23, 74, 75] have been proposed in the context of distributed shared memory. All these tools are dedicated to the profiling of applications written in C. They mainly aim at identifying remote memory accesses and pinpoint where in the application source code these accesses come from. They rely on hardware profiling counters to identify these accesses. The runtime mechanisms proposed by Carrefour [25] also rely on hardware profiling mechanisms to identify these accesses and memory controllers overload. From these memory accesses, Carrefour proposes thread migrations, page migrations or page replications to alleviate the bottlenecks. In this chapter, we also propose memory profiling based on hardware mechanisms but in the context of dataflow programs. We aim at identifying memory bottlenecks and at

building memory profiles for each actor of a dataflow application.

Recently, the Aftermath tool [76] has been proposed to identify bottlenecks in task-parallel programs. This tool focuses on applications written using OpenStream [77], an OpenMP extension to support streaming. Aftermath proposes both application- and system-level profiling mechanisms based on hardware performance counters. The work we present in this chapter has the same objectives than Aftermath in the context of dataflow application without the epxlicit notion of tasks but the notion of actor. Moreover, we propose to use memory sampling mechanisms to evaluate the cost of memory accesses.

Farhad et al. also proposed [78] a way to measure communication costs of dataflow programs executed on top of shared-memory multi-core architectures. Compared to the memory sampling mechanism that we present in this chapter giving measured communication overhead, they approximate this communication overhead by comparing the execution time of an actor when it is executed on the same core than the actors feeding its inputs channels with the execution time when the actor and its producers are located on different cores.

4.3 Dataflow Execution Model

The role of the execution model is to describe how to execute dataflow programs while respecting the semantics of the dataflow model of computation and to fix execution concerns not specified by the MoC. In other words, the execution model describes

- How to ensure that actors are fired only when enough input tokens and enough room in output FIFO channels are available
- Ensure actors are executed atomically regarding each others
- Which actor to fire when several actors can be fired at the same time

In addition to the respect of the semantics of the programs, dataflow execution models often target goals such as optimizing throughput, minimizing latency or minimizing memory usage. Because profiling aims at understanding how applications behave at runtime, the execution model can have an impact on profiling.

This chapter introduces both application level profiling regarding application's throughput and system level profiling concerning allocation and management of resources. On the one hand, application level profiling described in Section 4.4 is not impacted by the way dataflow applications are executed. As a consequence, it could be implemented on top of any execution model.

On the other hand, system level profiling described in Section 4.5 clearly depends on the execution model. Indeed, the main purpose of this profiling is to make a direct link between dataflow programs execution and the underlying resources. In Chapter 2 we argued that the concurrent tasks with shared state programming model, i.e. the threading model, is not a good candidate to program multi-core architectures. Nevertheless, as already stated in section 2.4.1, hardware and operating systems have evolved toward mechanisms allowing to create an efficient execution model for the thread programming model. As a consequence, we rely on this execution model made of tasks running concurrently and sharing memory to execute dataflow applications. We now review how

dataflow programs are compiled into multi-threaded programs to be executed according to this execution model.

4.3.1 Dataflow Compilation Overview

Many dataflow programming languages are either new languages [40, 60] relying on a source-to-source compiler or C/C++ extensions through libraries [39, 79]. In the former case as depicted on Figure 4.1a, dataflow programs are compiled to multi-threaded programs in a language supporting the thread programming model. In the latter case shown on Figure 4.1b, people often extend existing compilers. These extensions mainly consist in extracting the parallelism exposed by the dataflow graph.

In both cases, the code produced by the dataflow compiler is linked to a dataflow runtime in charge of initializing the graph's execution and in charge of scheduling actors in the case of dynamic dataflow MoCs. In the following we illustrate our proposals with examples supposing that the dataflow compilation tool chain uses an intermediate language before generating binary code. We use C as the intermediate language. We take this hypothesis because the two dataflow tool chains used in our experiments described in Chapter 5 work this way. Nevertheless, our profiling contributions could be integrated into dataflow compilers generating binary code directly. The dataflow tool chains we use also both convert the internal function executed each time an actor is fired into a C function that we'll call the actor's *step()* function in the following.

On shared-memory architectures, the communication between actors is implemented through the shared-memory. Actors produce and consumer tokens by writing and reading into the shared-memory. As a consequence, the communication time is hidden behind memory write and read instructions. Moreover, actors are also using the memory system for their internal computations, and this usage is not clearly exposed by the dataflow MoC. Actors internal behavior is often described in an imperative sequential language for which it's very difficult to statically compute memory usage. The memory usage profiling mechanisms proposed in Section 4.5.2.3 aim at identifying how the memory system is used by actors.

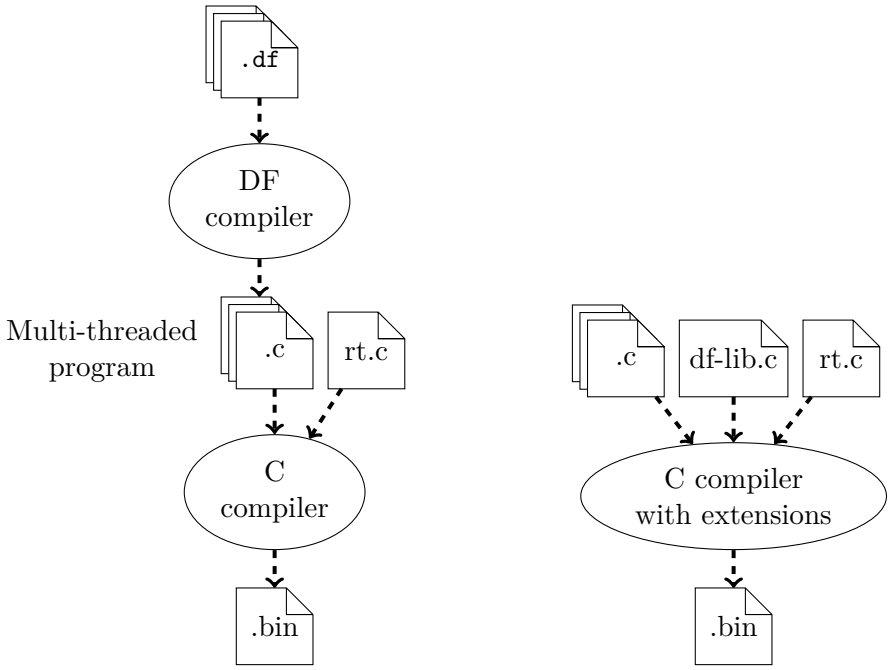
4.3.2 Sequential Execution Model

Before introducing how dataflow programs can be executed in parallel, we introduce how the dataflow compiler along with the dataflow runtime ensure a correct sequential execution of the application for both SDF and DDF programs.

4.3.2.1 SDF

A sequential execution model for SDF programs consists in having a single task responsible to execute all the actors of the dataflow graph. The schedule of the actor is statically decided by the compiler as explained in Section 3.3.2.1 and encoded into the sequential generated code. It corresponds to the execution order of actors. No synchronization at all is required in this case because all the data exchanges and their order are known statically. As a consequence, communication between actors is implemented by static arrays which size is computed from the schedule.

To illustrate this execution model, we consider the example depicted on Figure 4.2. As described in Section 3.3.2.1 $(1, 1, 1, 1, 3, 4, 1)$ is a repetition vector for the actors (A, B, C, D, E, F) of this graph and thus $A^1B^1C^1D^3E^4F^1$ is a valid sequential schedule.



(a) Dataflow programming with new languages

(b) Dataflow programming in existing languages

Figure 4.1: Dataflow programming languages are either new languages relying on a source-to-source compiler or C/C++ extensions through libraries. In both cases, one of the main objective for the compiler and the runtime system is to have a binary program able to exploit the parallelism provided by the underlying hardware.

Choosing this schedule, the dataflow compiler generates the code on Figure 4.3 for sequential execution. This example supposes that the compiler has generated a separate function for the internal work of each actor as shown on Figure 4.4 for actor C. We suppose in this example that the FIFO channels between the actors carry `float` values. These actors functions `pop` and `push` tokens into the FIFO channels. The `pop` and `push` functions are implemented by reading and writing to the global communication arrays. To allow the actor to *pop* and *push* tokens from anywhere in the *step* function, most of the SDF languages implementation rely on an index updated each time a token is read or written as depicted on this example.

4.3.2.2 DDF

In the case of DDF programs, the static schedule computed by the compiler and encoded into the generated code presented in the previous section for SDF programs doesn't exist. A runtime scheduler is thus required to fire actors only when enough input tokens are available. Moreover, we can't compute an upper bound for the size of the static arrays used for communication as in the SDF case.

One solution, available in the ORCC framework that we use in the experiments Chapter 5, is to use a round robin sequential scheduler along with fixed size FIFO implementations [80] as shown on Figure 4.5. FIFO channels are implemented with statically allocated memory, which size is a compiler option. The round robin scheduler

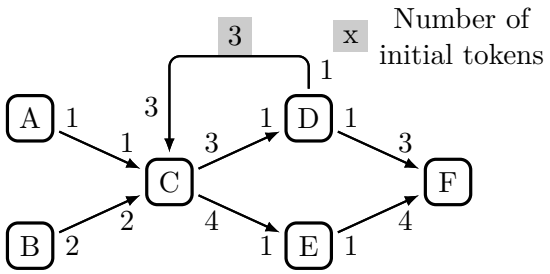


Figure 4.2: Cyclic SDF graph with initial tokens used to illustrate the notion of execution model. Initial tokens must be present on feedback edges to start the execution.

```
int[1] AC;
int[2] BC;
int[3] CD;
int[4] CE;
int[3] DC;
int[3] DF;
int[4] EF;
while(true) {
  step_A();
  step_B();
  step_C();
  for (int i = 0; i < 3; i++) {
    step_D();
  }
  for (int i = 0; i < 4; i++) {
    step_E();
  }
  step_F();
}
```

Figure 4.3: SDF sequential execution main loop. The loop infinitely executes the chosen static schedule $(1, 1, 1, 1, 3, 4, 1)$ by calling each actor's `step` function the number of times specified in the schedule.

starts by checking and executing the first actor in topological order if it's fireable and then moves to the next actor. When the last actor is reached, the scheduler loops back to the first one.

Checking if an actor is fireable consists in searching for one firing rule where constraints on inputs are satisfied. Moreover, because FIFO channels are not infinite in practice, each firing rule also specifies the number of tokens produced on each output FIFO channel. The scheduler thus also checks that there is enough room on the output channels before firing the actor. In the ORCC framework, the scheduler keeps executing the same actor while possible. In other words, once chosen by the round robin scheduler, an actor is executed as long as it has input tokens to be processed and there is space in output FIFO channels.

Another scheduling strategy, also available in the ORCC framework, consists in using a data driven sequential scheduler [80]. In this case, instead of switching to the next

```
int AC_idx, BC_idx, CD_idx, CE_idx;
float pop_AC() {
  return AC[AC_idx++];
}
float pop_BC() {
  return BC[BC_idx++];
}
void push_CD(float value) {
  CD[CD_idx++] = value;
}
void push_CE(float value) {
  CE[CE_idx++] = value;
}

step_C() {
  AC_idx, BC_idx, CD_idx, CE_idx = 0;

  // Pop inputs
  int AC1 = pop_AC();
  int BC1 = pop_BC();
  int BC2 = pop_BC();

  ... //Body computing outputs

  // Push results
  push_CD(...);
  push_CD(...);
  push_CD(...);
  push_CE(...);
  push_CE(...);
  push_CE(...);
  push_CE(...);
}
```

Figure 4.4: SDF actors `step` function. `pop` and `push` functions are implemented using statically allocated arrays. For clarity, the initialization of these arrays (AC, BC, CD, CE) is not shown here.

```
act = round_robin_next(null);
while(true) {
  while(is_firable(act) && enough_output_room(act)) {
    step(act);
  }
  act = round_robin_next(act);
}
```

Figure 4.5: DDF sequential execution main loop. Before executing an actor, we must check through the `is_firable` function that enough input tokens are available to fire an actor.

actor in a round robin fashion, the scheduler switch to one of the actor that will allow the current actor to be scheduled again. If the execution of the current actor is stopped because insufficient data on an input channel, the actor producing tokens on this channel is chosen. If the execution of the current actor is stopped because of insufficient room in one output channel, the actor consuming these tokens is chosen.

4.3.3 Parallel Execution Model

One of the main motivations for dataflow programming is to efficiently exploit hardware parallelism using the parallelism exposed by the model. Exploiting this parallelism requires to map actors to the hardware execution units. Some dataflow frameworks [39, 40, 60, 79] fix at compilation time the mapping of actors. Others [56, 57, 58, 62, 68, 81, 82] modify the initial mapping at runtime using information collected during execution. The profiling mechanisms described in this chapter aim at providing detailed information to reconsider these mapping choices either at run- or compile-time as will be demonstrated in Chapter 5.

4.3.3.1 SDF

As stated in Section 3.3.2.3, a parallel schedule consists in one sub-schedule for each core of the target system. These schedules are computed by the dataflow compiler. A parallel execution model for SDF graphs thus consists in n tasks where n is the number of parallel computing units of the underlying hardware to use for the execution of the graph. In the following we refer to these tasks executing dataflow actors as *dataflow tasks*. When these tasks are executed on top of a general purpose operating system, system primitives are used to pin each task to a specific core of the system.

To execute the parallel schedule computed by the compiler while preserving data dependencies, the execution model must set-up synchronization mechanisms between the tasks. For this, the dataflow tasks progress in a synchronized way through the usage of barrier at the end of the steady state [63]. We call this synchronization mechanism *steady state synchronization*. Figure 4.6 shows how the graph of Figure 4.2 is executed by two tasks and steady state synchronization. On this example, the chosen multicore schedule is $A^1D^3E^4$ for the first task, and $B^1C^1F^1$ for the second.

The horizontal lines represent barriers. We clearly see a first initialization stage required to fill up the pipeline allowing to execute the multicore schedule above (e.g. executing A and C in parallel requires pipelining because they are not on parallel branches in the application graph). Then we see two steady state executions synchronized by the

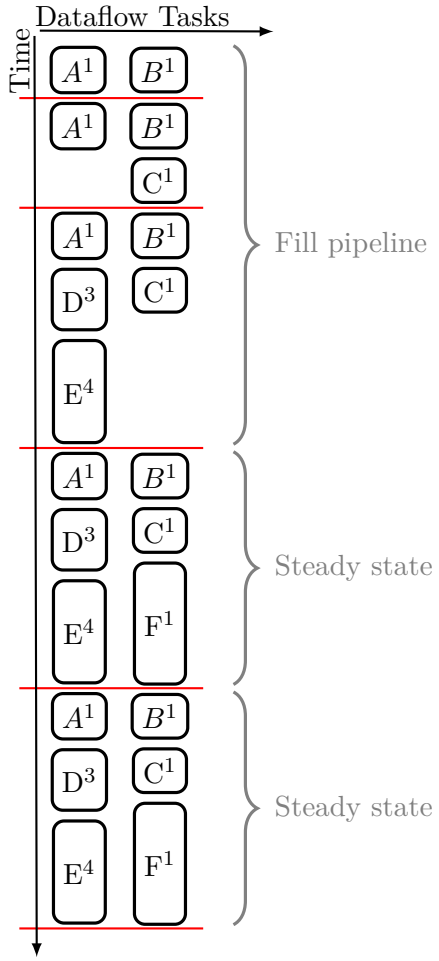


Figure 4.6: SDF parallel execution model. The steady state schedule is spitted into several dataflow tasks and the tasks synchronize through a barrier to ensure data dependencies are satisfied. The steady state can be executed infinitely in finit memory.

barrier. This steady state can then be executed infinitely without increasing the memory footprint of the program.

4.3.3.2 DDF

One solution to execute in parallel DDF graphs consists in extending the sequential round robin scheduling strategy to multiple cores. Again, we focus on the ORCC framework scheduling strategies. As for SDF, the parallel execution model is built using one dataflow task for each computing unit of the underlying hardware. Each task runs its own round robin scheduler on a subset of the graph's actors.

Using this solution, no synchronization other than the checks (i.e. the call to the `is_fireable(...)` function in Figure 4.5) performed by the scheduler before firing an actor are required. Indeed, an actor is executed only if it has enough inputs and if enough room is available in output channels.

Measurement	Expected value	Observed value
Global throughput	$\tau^G_{exp}(c)$	$\tau^G_{obs}(c)$
Local throughput	$\tau^L_{exp}(c)$	$\tau^L_{obs}(c)$
Execution time	$\kappa_{exp}(act)$	$\kappa_{obs}(act)$

Table 4.1: Notations about expected throughput used in this thesis. The first column shows expected statically available values while the second column shows their runtime equivalent called observed values.

4.4 Throughput Profiling

In this section we first show how the global expected throughput is used by the dataflow runtime to check whether or not the application's requirements are satisfied. Then we demonstrate in the case of SDF programs, how bottleneck actors can be identified using instrumentation of the actors code and the local expected execution time introduced in Chapter 3. The profiling mechanisms described in this section are independent of the chosen execution model. Before introducing global throughput monitoring, Table 4.1 recalls the notations about throughput introduced in Chapter 3. This table also shows the pending notations defined in this section used to refer to the values observed by the profiling mechanisms.

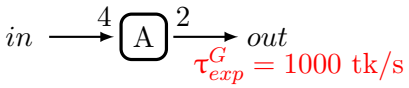
4.4.1 Global Throughput

At runtime, we need a way to observe the effective throughput. In the following we note τ^G_{obs} this observed throughput. Comparing this value with the information of expected throughput provided by the programmer described in Chapter 3 allows us to identify whether or not the throughput conforms to expectations.

A simple mechanism to compute at runtime the effective throughput consists in incrementing a counter each time a token is written into the FIFO of the actor where τ^G_{exp} has been expressed. Figure 4.7b shows this counter increment for the SDF actor depicted on Figure 4.7a. This actor has a single input channel *in* and a single output channel *out* where τ^G_{exp} has been expressed. Its *popRate* and *pushRate* values are respectively 4 and 2. Each time the output token is pushed, the τ^G_{obs} counter is incremented by one.

This global throughput monitoring solution is valid for both static and dynamic dataflow models. In the case of dynamic models, the only difference is the fact that the number of counter increments can vary between firings. Because actors push rates may vary, the number of counter increments would also vary. As a consequence, we need several increment instructions compared to the SDF case where one instruction is sufficient. Figure 4.8b shows an example of how τ^G_{obs} profiling work for the DDF actor A depicted on Figure 4.8a. This actor as two input channels *data* and $conf$ and a single output channel *out* where τ^G_{exp} has been expressed. Each time this actor is fired, its push and pop rates for the *data* and the *out* channels are updated with a value read on the $conf$ input channel. In this example, *loop* tokens are pushed each time the actor is fired, and as a consequence the τ^G_{obs} counter is incremented by *loop*.

For both SDF and DDF cases we extend the dataflow compiler in order to generate the additional lines in charge of creating and incrementing the global throughput counter

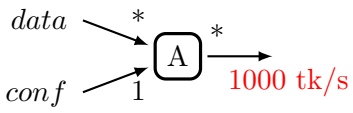


(a) SDF actor with τ^G_{exp} constraint

```
step_A() {
  int p1 = pop_in();
  int p2 = pop_in();
  int p3 = pop_in();
  int p4 = pop_in();
  int res1 = ...
  int res2 = ...
  push_out(res1);
  push_out(res2);
  τ^G_obs Count += 2;
}
```

(b) SDF τ^G_{obs} profiling

Figure 4.7: SDF global throughput profiling. We add a fixed counter increment in the code of the `step` function of the actor where the τ^G_{exp} is specified.



(a) DDF actor with τ^G_{exp} constraint

```
int loop = 2; //Init value
step_A() {
  for(int i=0;i<loop;i++){
    int p = pop_data();
    int res = ...
    push_out(res);
    τ^G_obs Count += 1;
  }
  loop = pop_conf();
}
```

(b) DDF τ^G_{obs} profiling

Figure 4.8: DDF global throughput profiling. We add a counter increment in the code of the `step` function of the actor where the τ^G_{exp} is specified.

as shown on Figures 4.7b and 4.8b.

Computing the global throughput value consists in reading this counter at different times. To get the throughput, we just divide the difference of two successive values of the τ^G_{obs} counter by the time elapsed between the two reads. Finally, comparing this observed τ^G_{obs} value with the expected τ^G_{exp} one provided by the programmer as described in Chapter 3 allows to identify potential violation of throughput requirements.

When dataflow program profiling is targeted at satisfying a throughput constraint, identifying such violation is of course the first step done by the dataflow runtime. In case of non-respect of the throughput constraint, or when the observed value is getting close to the expected one, the runtime needs to identify bottleneck actors. When to activate the bottleneck identification mechanism has an impact on the reaction time of the runtime has will be described in chapter5.

The overhead induced by τ^G_{exp} monitoring is negligible. Indeed, as shown on Figures 4.7b and 4.8b it only adds one increment instruction each time a token is produced on the FIFO channel where τ^G_{exp} is expressed.

```
int time = getTime();
for(int i = 0; i < 4; i++){
  step_E();
}
time = getTime() - time;
κobs = time / 4;
```

Figure 4.9: SDF τ^L_{obs} profiling. We measure time at component level to reduce the overhead introduced by the timing call.

4.4.2 Identify Bottleneck Actors In SDF Graphs

When the global throughput value is not conform to the requirements, we want to be able to identify where the problem comes from. For this, in the context of SDF languages we use the local expected execution times coming from the propagation introduced in Chapter 3. Comparing this information with effective actors execution times allows us to identify bottleneck actors.

Chapter 3 introduced the notion of an actor's local expected execution time noted κ_{exp}. We now introduce the runtime equivalent, called actors effective observed execution time noted κ_{obs}. When dataflow applications are executed alongside non dataflow applications not under our control, the dataflow tasks can be preempted. To be comparable with κ_{exp} computed statically, κ_{obs} must represents the execution time of the actor only. In other words, it must exclude preemption time. Excluding preemption time allows to distinguish cases where a dataflow actors is intrinsically too slow from cases where the actor has not been provided enough processor time to execute by the operating system scheduler. This is explained in Section 4.5.1.

To compute $\tau^L_{obs}(c)$, the compiler generates some additional code. Timing measurements are performed before and after each loop executing a component of the SDF schedule as shown on Figure 4.9. In this example, the SDF schedule leads to 4 execution's of actor E. To compute the average $\tau^L_{obs}(c)$, we divide the measured time by the number of times the actor *act* has been fired in the component. Measuring outside the loop allows to reduce the number of system calls and thus the overhead introduced by timing measurements. Chapter 5 evaluates the overhead of these system calls for different use cases.

To identify bottleneck actors in an SDF graph, we compare $\kappa_{exp}(act)$ and $\kappa_{obs}(act)$ values for all the actors. Such actors are intrinsically too slow. Their execution time, excluding preemption, is not conform to execution times required to satisfy the global throughput constraint.

4.5 System-Level Profiling

In the context of shared-memory multi-core architectures, the different computing units (i.e. the cores) are shared by all the running applications. The hardware required to access the memory is also shared by all the applications. The system-level profiling mechanisms we described in this section aim at identifying how these shared resources are used by all these applications. To exploit hardware parallelism on shared-memory multi-core architectures, the computing resources must be equally used and the memory subsystem must not be saturated. This section first describes how we identify imbalance

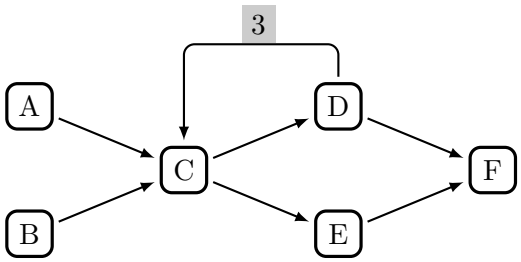


Figure 4.10: Dataflow graph example used to illustrate tasks imbalance. We use the same graph than the one in Figure 4.2 but without the static consumption and production rates because the mechanisms described in this section are valid for both SDF and DDF dataflow graphs.

problems between the cores. We then show how memory bottlenecks can be identified and related to dataflow actors.

4.5.1 Cores Load

The parallel execution model introduced in Section 4.3.3 needs a mapping between actors and cores. The initial static mapping decided by the compiler can be changed at runtime. In other words actors can be migrated between dataflow tasks to balance the load. Imbalance between dataflow tasks load can be caused by either imbalance between the dataflow actors executed by the tasks or because the dataflow tasks are preempted by other applications. To illustrate our claims, we use in this section again the same example depicted on Figure 4.10. Actors static consumption and production rates have been removed because the mechanism we propose in this section is valid for both SDF and DDF graphs.

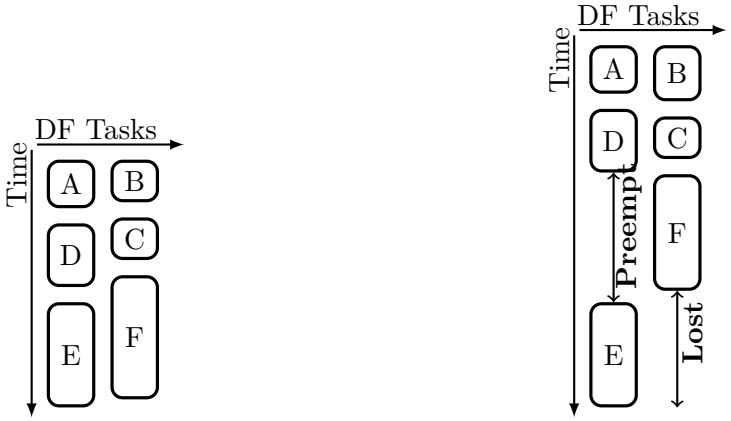
In a scenario where a dataflow application is executed alone on a multi-core platform using one task per core, imbalance between dataflow tasks leads to CPU waste. Independently of the underlying model (SDF or DDF), under-loaded cores will have to wait for the over-loaded ones. For static dataflow models, this imbalance is caused by the inaccuracy of the actors execution time used in the compiler's mapping algorithm. It can also be the result of actors with data dependent computing requirements. For example, the `step` function of an actor may contain a loop which upper bound is a value provided by one input channel. In this case, the execution time is dependent on this value that change at runtime.

In the case of dynamic dataflow models, the unknown number of firings for each actor is another source of imbalance. Figure 4.11 shows an example of poor static execution time estimation for the example of Figure 4.10. On this example, we consider that the dataflow graph is executed alone on dual core system with the one task per core and steady state synchronization execution model. The execution of actor F is shorter than estimated. In this case, migrating A from its initial dataflow task to the second one will lead to better performances by reducing the time lost by the second task.

The second source of imbalance we consider occurs when the system runs dataflow applications along with other dataflow and non dataflow applications. Figure 4.12 shows an example of such preemption for the example of Figure 4.10. On this example, we consider that the dataflow graph is executed on dual core system along with other

(a) Static estimation **(b)** F is shorter than estimated

Figure 4.11: Dataflow tasks imbalance caused by wrong static estimation of actors execution times. F is shorter than estimated, a better load balancing exists.



(a) Static estimation **(b)** F is shorter than estimated

Figure 4.12: Dataflow tasks imbalance caused by preemption by other applications.

applications and with the one task per core and steady state synchronization execution model. On the first core, the dataflow task is preempted by another application. This leads to CPU time lost on the second core.

In both cases of imbalance, the dataflow runtime needs a way to measure actors execution times and preemption time to compute the load for each dataflow task. For actors execution time profiling in the case of SDF programs, we use the same mechanisms than the one introduced in Section 4.4.2 for SDF actors bottleneck identification. Actors firing are surrounded with time measurement calls as shown on Figure 4.13a. For DDF programs, we also surround actors firing with timing calls as shown on Figure 4.13b.

For preemption time we also rely on operating system services. On Linux, we read the `/proc/stat` virtual file to get this information.

4.5.2 Memory Subsystem Load

When targeting multi-core systems with a distributed shared-memory architectures, the software has to take care of memory usage. In the dataflow programming context, it means that the dataflow compiler along with the dataflow runtime must take into account

```
int time = getTime();
for(int i = 0; i < 4; i++){
  step_E();
}
time = (getTime() - time) / 4;
```

(a) SDF actors

```
act = round_robin_next(null);
while(true) {
  while(is_firable(act) &&
        enough_out_room(act)) {
    int time = getTime();
    step(act);
    time = getTime() - time;
  }
  act = round_robin_next(act);
}
```

(b) DDF actors

Figure 4.13: Cores imbalance profiling using actors execution time. For SDF programs we surround components execution as we did to identify bottleneck actors. For DDF programs we surround actors firing with timing calls.

the underlying memory organization to efficiently use it. As shown by recent work [25], on modern distributed private-memory architectures, memory system overload can be an important source of performance degradation.

As a consequence, we need a way to identify whether dataflow programs are facing memory congestion or not. In the case of memory controllers overload, by correlating actors execution with memory usage we can exploit the dataflow programming to alleviate bottlenecks by changing actors mapping and/or FIFO channels mapping. To identify perform memory subsystem profiling we rely on the processor's Performance Monitoring Unit (PMU).

4.5.2.1 PMU

The PMU provides software means to characterize hardware usage through hardware performance counters. Intel, AMD and ARM all include a PMU in their processors.

The counters provided by the PMU can be configured by software to count some specific hardware events among a huge number of possibilities. As shown on Figure 4.14, these counters are either located at the level of cores or at the level of memory controllers. At core level, example of hardware events that can be profiled are instruction cycles, number of floating point instruction, number of level 1 cache misses or number of branch mispredictions. At memory level, counters can be configured to count for example the exact number of memory read or memory write requests.

In addition to the counting mode, almost all PMUs provide a sampling mode. In sampling mode, instead of counting the number of a specific event, the PMU is configured

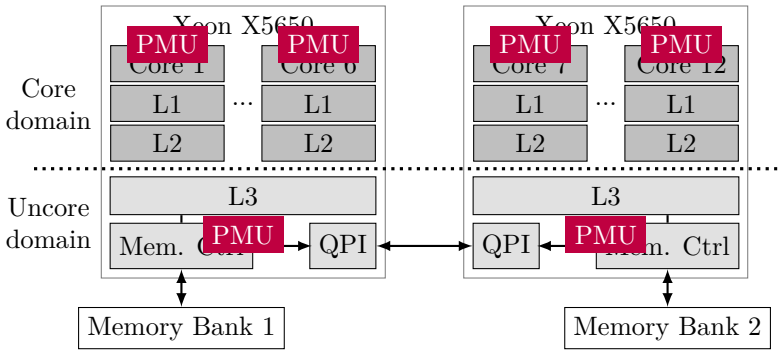


Figure 4.14: Performance monitoring counters on an Intel dual processors NUMA architecture. Performance counters are either located at cores level or at the uncore level. Uncore counters are not able to distinguish events depending on the originated core.

to generate a sample with detailed information every time the event occurred a specified number of times. The information provided by the samples allow to perform complex analyses compared to the ones enabled by counting only.

For example, when sampling mode is activated for a hardware event located at core level, the sample can contain detailed information about the processor's state at the time the event was generated. Among the available detailed information we can mention the value of the program counter and the core identifier. Moreover, when the sampled event is a memory event such as a memory read or memory write, the sample can also provide the memory address involved in the memory access, the memory level in memory hierarchy where the data was read or written and the latency (i.e. the time required to serve the memory request) of the memory request. We rely on this information to construct actors memory profile.

4.5.2.2 Memory Controllers Imbalance

We use the PMU to count the exact number of memory requests reaching each memory controller. From this we then compute the effective memory bandwidth for each memory controller. Comparing this effective bandwidth with the maximal bandwidth reachable allows us to identify overloaded memory controllers.

The usage of the performance monitoring counters requires a deep understanding of the processor's architecture and requires very low level code writing. It also requires to be in processor's supervisor mode.

In the case where dataflow applications are executed directly on top of the hardware without an operating system, performance counters would have to be manually programmed to identify memory controllers overload as shown on the left part of Figure 4.15.

In our experiments described in Chapter 5, we run dataflow applications on top of Linux. In this case, there is two ways to access the PMU as shown on Figure 4.15. The first one consists in using the `/dev/cpu/msr` kernel module as illustrated on the right part of Figure 4.15. This module only allows to access the PMU configuration registers in supervisor mode but doesn't provide any abstraction at all. The second solution is to use the `perf_event_open` system call. This system call provides a first level of abstraction to access to hardware performance counters. Nevertheless, to start

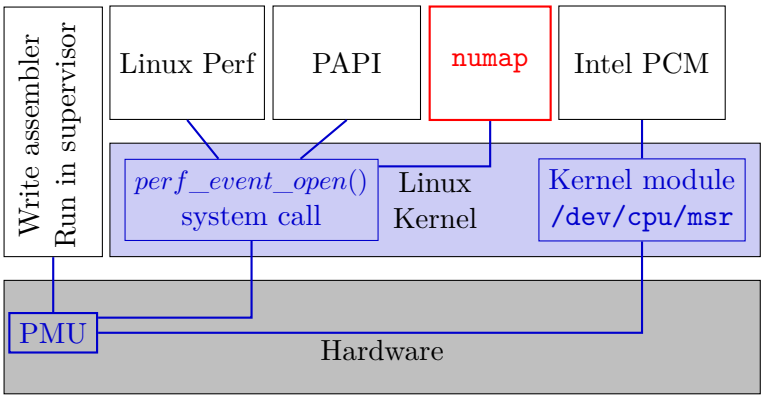


Figure 4.15: Different PMU usages on top of Linux. The PMU can be accessed either through the `perf_event_open()` system call or through a kernel module to access the Model Specific Registers (MSR) controlling it. For commodity reasons, the `numap` profiling library we propose is built on top of `perf_event_open()`.

counting an event using this system call, we still need to setup a lot of complex parameters depending on the underlying architecture. For example, to count memory read and write requests on the particular hardware we use in our experiments, we use the *UNC_QMC_NORMAL_READS.ANY* and *UNC_QMC_WRITES.FULL.ANY* events as described in Intel's Software Developer Manual chapter 18 [83]. This system call is used by the Linux Perf command line profiler [84].

To abstract the complexity of using `perf_event_open`, we implemented a thin library dedicated to memory profiling called `numap` as shown by Figure 4.15. Compared to Performance API (PAPI) [85], our library abstracts the PMU differences regarding memory profiling between NUMA architectures. Another added value provided by `numap` compared to PAPI is an abstraction of memory sampling using the PMU as described in the next section. Even if the implementation of `numap` for our experimental platform only contains few hundreds of line of code, it was quite complex to get it operational. Indeed, using the PMU requires a deep understanding of the targeted hardware and requires to extract the good information from the Intel Software Developer Manual. This complexity is the main motivation that lead us to write `numap`. At the time of writing this thesis, we are not aware of any open source library providing such an abstraction and we plan to see how `numap` could be released.

4.5.2.3 Sampling of Memory Accesses

The mechanism to detect the memory controller's overload only allows to identify memory bottlenecks. In particular, this mechanism doesn't say anything about the origin of the memory accesses. The dataflow model itself provides a first level of information about memory accesses. It indicates which actors communicate together, and in the case of SDF MoCs it also gives the exact amount of information exchanged by actors.

Nevertheless, how the data exchange information provided by the dataflow model relates to hardware memory access is not trivial. Indeed, the complex memory architecture, mainly the cache system, prevents dataflow compilers to statically know which actors pop and push operations will lead to memory accesses. Moreover, as already stated actors are also using the memory system for their internal computations. This

```
step_A() {
  int p1 = pop_in();
  int p2 = pop_in();
  int p3 = pop_in();
  int p4 = pop_in();
  int res1 = ...
  int res2 = ...
  push_out(res1);
  push_out(res2);
  A_end_label:
}
```

Figure 4.16: Additional end of function label on an SDF actor. We use this label to associate PMU samples to dataflow actors. Even if the example is given on and SDF actor, this mechanism is valid for both DDF SDF actors.

usage is not clearly exposed by the dataflow MoC. As a consequence, we propose to set up actors memory usage profiling mechanisms using the PMU memory sampling mechanisms introduced in Section 4.5.2.1.

To correlate memory accesses samples provided by the PMU to dataflow actors, we have to associate program counters values in the PMU samples to actors execution functions. We must also correlate the memory addresses in the samples to either dataflow FIFO channels accesses or actors stack used by the internal of actors execution functions. In the particular case where dataflow programs are first compiled to C multi-threaded programs we propose to modify the compiler to generate additional code allowing the runtime to do the correlation between samples generated by the PMU and the dataflow model.

For the correlation of programs counter values with dataflow actors we rely on a simple mechanism based on function pointers and an additional *end-of-function* label generated by the compiler at the end of each actor's function. Figure 4.16 shows this additional label on an example actor called A. The internal of the computing of outputs from input is not shown but only represented with dotted lines.

From this label and the function's pointer, we are able to know whether or not a sample belongs to the actor's A. At application launch time, we construct a list of actors functions ordered by addresses. A binary search in this list allows to know which actor the sample belongs to in $O(log(n))$ time with n representing the number of actors. In the case, where the sample doesn't belong to any actor, it means that it belongs to the dataflow runtime used to coordinate actors execution.

For the correlation of memory addresses to FIFO channels we use a similar mechanism in the case of execution models relying on C as intermediate language. The dataflow compiler generates either memory allocation calls for the FIFO channels in the initialization function of the C files or static arrays. In both cases, we extend the compiler to generate code that keeps track of the addresses of the memory used to implement FIFO channels. This way, we know exactly where each channel of the graph is allocated in memory. Based on this and knowing the size allocated in memory for each channel we can correlate memory samples to channels. Determining the memory size allocated for the implementation of a dataflow FIFO channel is under the responsibility of the

execution model, and thus the runtime has access to this information.

To correlate memory addresses with FIFO channel, we first search for the actor associated to the sample using the program counter value provided in the sample. As described above this is done in $O(log(n))$ time where n is the number of actors. If an actor is found, we search if the sample belongs to one of the channel of the actor also using a binary search in the actor's channels ordered list. If it's not the case, we know that the sample belongs to the actor's private stack. It concerns internal memory required by the actor to compute it's outputs.

Once the memory samples have been associated to actors and FIFO channels of the dataflow graph we propose to compute statistics to analyze these samples to get a better understanding of the application performances. Using the memory access latency information provided by the PMU as described in Section 4.5.2.1, we can evaluate for example, how much is the overhead induced by mapping to actors communicating a lot on cores from two different NUMA nodes.

The memory profiling mechanisms described in this section, allow us to build a memory profile for each actor and/or for each FIFO channel of the graph. This profiling can be activated on demand. For example, it can be activated from the start to the end of the execution of an application to construct memory profiles over all the execution of an application for offline analyses. It can also be activated on demand by runtime mechanisms to identify if a bottleneck actor identified by mechanisms described in Section 4.4.2 is facing long memory latency. Section 5.3 will present how we use memory sampling profiling results for offline analysis of DDF applications and what conclusion we can draw from these results.

4.6 Discussion

We have presented in this chapter dataflow profiling mechanisms. First, we showed how to create profiling mechanisms regarding the throughput constraint provided by the programmer at application level. We showed how to profile the effective throughput of a dataflow application to check whether or not it conforms to its throughput requirements. In the case of SDF graphs, we showed how to identify bottleneck actors using the expected execution time computed statically for each actor. We also presented system-level profiling mechanisms allowing to identify computing and memory resources contention. The mechanisms dedicated to profile computing resources usage are all based on timing information wheras the memory profiling mechanisms rely on hardware performance counters provided by modern processors.

All the mechanisms presented in this chapter are built in user space on top operating system primitives, we rely on:

- System timing calls to get actors execution time;
- System timing calls to get preemption time;
- System calls to access to the PMU.

All these system calls have a cost, and we believe that the profiling mechanisms introduced in this section would benefit from being implemented directly in the operating system kernel. This will first reduce the profiling overhead, and will allow a better integration between dataflow and non dataflow applications.

The NUMA memory profiling library we introduced in this chapter is dedicated to abstract PMU differences regarding NUMA concerns between different architectures. We have a functional implementation for a Xeon processor based on the Westmere-EP micro architecture. It should be interesting to add implementation for other Intel micro architectures and for other processors such as AMD ones. As already stated, such implementation work will require to deeply understand the PMU of the underlying architecture to use the right hardware counters to measure NUMA effects.

We intentionally didn't give details on the way these mechanisms can be used together in this chapter. We introduced these mechanims as general mechanisms that could be applied to different dataflow programming languages based on different dataflow MoCs. The next chapter presents how we exploit these mechanisms together along with the throughput propagation algorithm introduced in Chapter 3 in order to:

- Build a throughput aware SDF dataflow runtime based on the StreamIt framework;
- Construct profiling results for DDF applications written using the ORCC IDE.

5 Profiling Mechanisms Exploitation

If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice

Donald Knuth

Chapter's outline

5.1 Introduction

This last chapter is not only an experimental validation of the concepts introduced in the previous chapters. It also describes how we assemble and exploit the profiling mechanisms proposed in this thesis. The chapter is divided in two distinct sections:

- Section 5.2 introduces a SDF throughput aware runtime system built in the StreamIt framework. This runtime system is built on top of Linux and uses profiling results to perform runtime adaptations for throughput constraint satisfaction.
- Section 5.3 presents the implementation of our profiling mechanisms into the ORCC framework dedicated to DDF programming. The profiling results are used to provide feedback to the application developer.

We associate in this chapter the runtime adaptations to SDF applications and offline analyses to DDF ones. Nevertheless, this is a practical choice only and our profiling mechanisms can be used in both cases for offline profiling **and** for runtime adaptations. As will be described in Sections 5.2 and 5.3, we made this choice because the StreamIt runtime system was easier to adapt than the ORCC one to work in an open system where other applications can come in and out at any time.

Moreover, we illustrate the memory profiling only in the context of the ORCC framework. We decided to focus on the memory analysis of the realistic applications provided alongside ORCC compared to the StreamIt micro-benchmarks.

5.2 SDF Throughput-Aware Runtime

In this section we present how we use the concepts introduced in Chapters 3 and 4 to implement a throughput-aware runtime in the StreamIt framework. We show how the profiling mechanisms are used in conjunction with the information computed statically from the global throughput constraint to identify cases where some actors are bottlenecks. In cases where no bottleneck actors are found, we show how we use dataflow tasks profiling to identify imbalance between cores. To build a proof-of-concept validating our profiling mechanisms, we also present a simple adaptation heuristic.

The choice to prefer StreamIt over ORCC to illustrate how the profiling mechanisms can be used for runtime adaptations comes from the runtime system implementation. Both StreamIt and ORCC runtime systems are based on the concept of dataflow tasks as described in Chapter 4, and they both assume that dataflow applications are executed alone on the hardware. The two runtime systems use busy waiting to synchronize dataflow tasks and thus consume 100% of the CPU resources all the time. To use either StreamIt or ORCC in an open environment where applications may come in and out we have to modify their runtime systems in order to replace busy waiting with mechanisms allowing to put dataflow tasks in a sleep state and wake them up when they can again execute some actors. On the one hand, we extend the StreamIt runtime very easily to not use 100% of the computing resources. For this, we replace the original busy waiting barrier at the end of the steady state by a non-busy one using `pthread` barriers. On the other hand, because of the dynamic nature of DDF, the ORCC runtime system needs to be designed differently to not use all the computing resources all the time. Using one dataflow task each actor could be a solution. In this case, each dataflow task has to sleep when its actor is blocked waiting for input tokens or for room in output FIFO

channels and to wake-up when new input tokens have been produced or when room has been made in output channels.

5.2.1 Language Compiler And Runtime Support

We extend the StreamIt language and compiler as described in Chapter 3. The language is extended to let application developers specify the global expected throughput. Then, the compiler takes this information into account to compute the local expected execution time for each actor.

We also extend the runtime introduced in Section 4.3.3.1 so that it handles several StreamIt applications. The information obtained by the throughput propagation is attached to each application, and includes its global expected throughput and the expected execution time for each actor. This is done by having the compiler generating a global data structure including this information.

Figure 5.1 shows an overview of our runtime system. The streaming applications execute on top of a mainstream operating system (Linux in this instance). Because we are running on top of a general purpose operating system, some non-dataflow applications may cohabit in the system with the streaming ones. By default, the StreamIt compiler compiles application to n threads, where n is the number of cores of the hardware. We modified a little bit this behavior to compile applications to $n - 1$ threads. To avoid perturbations in our results, we keep the first core available to run all the operating system tasks that can be moved to a specific core.

We add two components, detailed in the next section, to the original StreamIt runtime system. The first one is a runtime `Monitor` running in a dedicated thread. The `Monitor` is responsible to check whether throughput constraint is satisfied or not. In the case of non respect of this constraint, the `Monitor` is responsible to identify bottleneck actors and eventual cores imbalance and could be extended to identify memory system overload. In the current implementation, the `Monitor` is pinned on the first core of the system reserved for the operating system. In the context of multi- and many-core we believe that it's affordable to dedicate a core for this kind of tasks. Executing the `Monitor` on an other core alongside the applications will lead to additional overhead depending on the monitoring frequency. Each time the monitor is waked up, the system will have to perform a context switch.

The second component we add to the original StreamIt runtime system is a `Adapter`. This component is activated on-demand from the `Monitor` thread and thus also executes on the dedicated core. It is responsible to perform adaptations to alleviate the bottlenecks identified by the `Monitor`. The `Adapter` also reports information about the bottlenecks to the developer.

To support the `Adapter`, we add mechanisms to to the StreamIt runtime allowing to suspend and to resume dataflow applications either individually or globally. This mechanism is required to safely perform adaptations. We can't move an actor from one core to another while it's being executed. For this, we rely on threading synchronization primitives provided by the operating system. We also extend the original StreamIt runtime system with migration mechanisms. To migrate actors between cores, function pointers are moved between threads. In the context of runtime system also performing memory monitoring, we could rely on the pages migration mechanism of the underlying operating system to migrate FIFO channels.

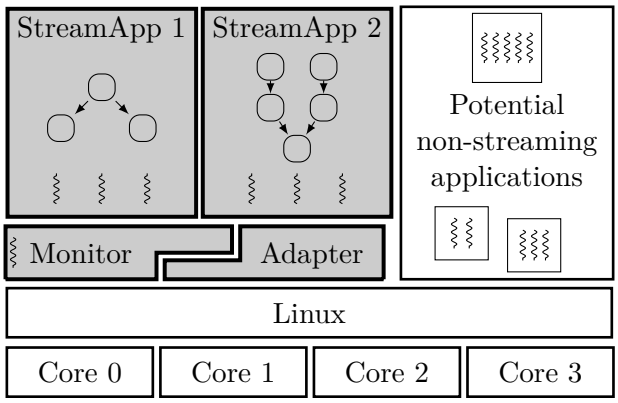


Figure 5.1: Throughput aware StreamIt runtime system overview. Compared to the original StreamIt runtime system, our runtime system can handle several streaming applications and includes additional components responsible to monitor and adapt streaming applications executed alongside non-streaming applications.

5.2.2 Runtime Monitoring

The `Monitor` is responsible for:

- determining if throughput constraints are satisfied or not (Section 5.2.2.1);
- identifying which actors in the graph are the bottlenecks (Section 5.2.2.2);
- identifying CPU imbalance (Section 5.2.2.3).

This section describes conditions under which the runtime layer starts monitoring actors' activity and how it identifies bottleneck actors and impacted resources. Figure 5.2 gives an overview of the job performed by the `Monitor` component for one dataflow application.

As already stated and shown on Figure 5.1, the `Monitor` runs in a specific thread pinned on a dedicated core. It is automatically activated at application start-up time where it enters stage 1. This stage consists of periodically checking the value of $\tau^G_{obs}(c)$ as described in Section 5.2.2.1. When the throughput reaches the local monitoring threshold $\tau^G_{exp}(c) + \delta$, the `Monitor` enters in stage 2 by activating actors execution time monitoring. Section 5.2.2.1 discusses about how to fix the value of δ. The `Monitor` stays in this stage for a given duration that we call the stage's period. During this period, each dataflow task (as defined in Section 4.3.3.1) records information about the execution time of its actors. Section 5.2.2.2 describes how we fix the value for the period of stage 2 and gives the details of this stage. At the end of the period of stage 2, depending on the existence of bottleneck actors or not, the `Monitor` either enters stage 3 where it performs memory monitoring as described in Section 4.5.2.2, or stage 4 where it performs CPU load monitoring as described in Section 5.2.2.3. For the memory monitoring, we use the Intel Westmere-Ep implementation of the `numap` memory profiling library introduced in Section 4.5.2.2. This implementation uses the PMU memory events available on our experimental platform as described in the Intel software development manual [83]. As for stage 2, the `Monitor` stays in stage 3 or 4 for a given duration, that we also call the stage's period. At the end of this period the `Monitor` either reports information to the application developer or invokes adaptation mechanisms described in Section 5.2.3.

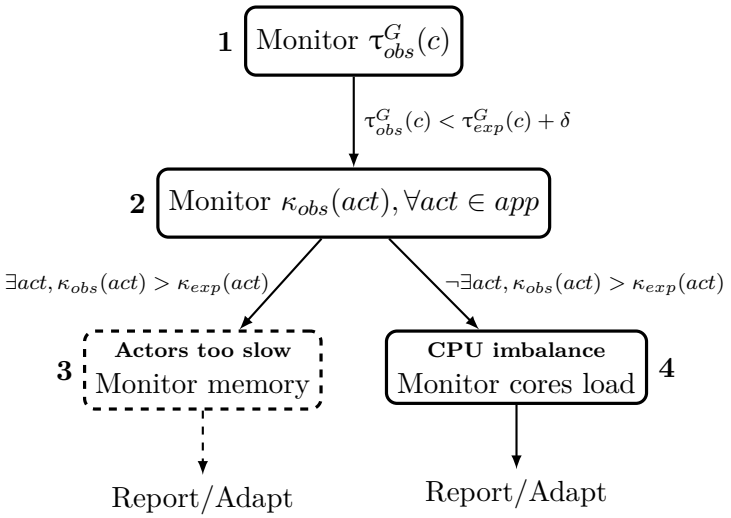


Figure 5.2: Monitoring stages overview. The `Monitor` permanently checks the global throughput (stage 1). When the global throughput is getting close to the throughput constraint, the `Monitor` starts actors execution time monitoring (stage 2) to identify potential bottlenecks. Finally, depending on the result of stage two, memory or CPU imbalance monitoring is started.

The memory monitoring stage has not been integrated into the StreamIt runtime system. Nevertheless, the memory profiling extensions to the ORCC runtime system presented in Section 5.3.1 can be applied to the StreamIt runtime system. We made this choice to focus on memory profiling of realistic complex applications such as the video decoders provided by the ORCC system.

5.2.2.1 Monitoring The Global Throughput

In order to check at runtime if the throughput expected for an application is obtained, we rely on the monitoring code generated for the actor where the global throughput constraint τ^G_{exp} has been expressed has explained in Section 4.4.1.

By checking periodically the value of this counter, the runtime layer computes the *global observed throughput* $\tau^G_{obs}(c)$ of the application. If it is too low, the *Monitor* enters stage 2. In our experiments, performed on benchmarks provided by the StreamIt framework, we set the value of the δ parameter (controlling when to enter stage 2) to 0 for simplicity. Our main goal is to demonstrates how our runtime system can identify throughput constraint violations and their origins. In a production system, this value should be set depending on expected reaction time for the system. In some cases, we may want to react before the throughput gets below the specified constraint.

Overhead from this monitoring system comes from two places. Incrementing a counter each time data is written on the channel where τ^G_{obs} is expressed is negligible. The overhead of the simple arithmetic used to compute τ^G_{obs} on regular intervals using the counter is also negligible since the monitoring thread is executing on a separate core.

5.2.2.2 Monitoring Actors Execution Times

When the *Monitor* enters stage 2, it activates for all the actors of the graph the monitoring code generated by the compiler introduced in Section 4.4.2. This is done through a boolean flag generated by the compiler that allows to switch on and off actors' execution time monitoring. Recalling that $\kappa_{obs}(act)$ excludes preemption time, bottleneck actors are the ones where κ_{obs} is greater than κ_{exp}.

At the end of the actor execution time monitoring period, the $\kappa_{obs}(act)$ for each actor is compared to its $\kappa_{exp}(act)$ to tag the bottleneck actors. When all actors respect their expected execution time, that means the dataflow tasks are badly balanced and the `Monitor` must identify why. This is discussed in section 5.2.2.3. Otherwise, i.e. there is at least one actor with $\kappa_{obs}(act)$ greater than $\kappa_{exp}(act)$, the actors being too slow are known, and the `Monitor` tries to identify why. To that end, the `Monitor` enters stage 3 where it activates the memory profiling mechanisms introduced in Section 4.5.2.

5.2.2.3 Monitoring Cores Imbalance

When the global throughput is too low and all actors respect their execution time (bottom right branch of Figure 5.2), we know that we are facing at best a core imbalance, at worst a globally overloaded system. The later corresponds to scenarios where all the cores are used 100% of the time. In these cases, there is no room for CPU balancing improvements and the best we can do is to report the overload information. To identify cores imbalance cases the `Monitor` activates the mechanisms introduced in Section 4.5.1. For the execution time of dataflow tasks, this is again done through the boolean flag generated by the compiler allowing to activate actors execution time monitoring.

For preemption time, on Linux, the *Monitor* reads the `/proc/stat` file at the start and at the end of the cores imbalance monitoring period. Because this file is a virtual file which holds an image of the kernel structures living in memory, it provides real time information about cores usage. In other words, each time you read this file, the kernel simply copies the data structures it holds internally into the provided userland buffer. For each core, this file gives the time spent in different states since boot time. Among others, this file include time spent executing:

- normal processes in user mode;
- niced processes in user mode;
- processes in kernel mode;
- nothing (idle time).

We use the idle times to know what are the over-and under-loaded cores. The over-loaded core is the one with the lowest idle time value where as the under-loaded core is the one with the highest idle time value.

Then, if we identify cores imbalance, we apply a simple adaptation heuristic consisting in migrating actors from over-loaded cores to under-loaded ones.

5.2.3 Reporting and Adaptations

When the `Monitor` has identified bottleneck actors, it always reports information to the application developer. This reporting is done into a file generated by the `Monitor`. This file contains for each bottleneck actor:

```
int ovld;
int ovldCore;
int unld;
int unldCore;
int toMove;
int moved;
actor_t act;
getLoad(&ovld, &ovldCore, &unld, &unldCore);
toMove = (ovld - unld) / 2;
moved = 0;
act = nextHeaviestActor(ovldCore);
while (moved < toMove and act != null) {
  if (moved + κ_obs(act) < toMove) {
    move(act, ovldCore, unldCore);
    move += κ_obs(act);
  }
  act = nextHeaviestActor(ovldCore);
}
```

Figure 5.3: CPU load balancing heuristic. The `Adapter` moves actors from the over-loaded to the under-loaded core to balance their load.

- the name of the actor;
- the expected execution time κ_{exp};
- the measured execution time κ_{obs}.

When the `Monitor` leaves stage 4, it invokes the `Adapter`. Using runtime mechanisms allowing to migrate actors between threads pinned to cores, the `Adapter` is able to implement adaptations heuristics.

For now, we only implemented a simple heuristic consisting of migrating actors from the most overloaded core to the one with the lowest load in the context of a single dataflow application running along with non-dataflow applications. As stated in the introduction for this section, the main goal for this heuristic is only to build a proof-of-concept validating our monitoring proposal. Including state of the art load balancing algorithms is kept for future work.

Figure 5.3 shows this heuristic. It moves actors between the core with the highest CPU load to the one with the lowest CPU load using actors measured execution times to minimize the difference of work amount between theses two cores. The `getLoad()` function provides the results of dataflow thread imbalance monitoring introduced in the previous section. The `nextHeaviestActor(int core)` function returns actors in decreasing execution time order (i.e. the actor with the largest κ_{obs} value is returned first).

5.2.4 Results

We now present the results of our runtime system under different scenarios illustrating the different kinds of bottlenecks that dataflow applications may encounter when executed alongside non dataflow applications. We use applications from the StreamIt

Processors	Westmere-EP: 2x Intel Xeon5650 (hexa core)
Core frequency	2.66 GHz
L1d cache size	32 KiB
L1i cache size	32 KiB
L2 cache size	256 KiB
Shared L3 cache size	12 MiB
Memory	6 x 8 GiB DDR3-1333
Operating system	Linux kernel 3.11

Table 5.1: System configuration

benchmarks suite [35]. The experiments were conducted on a workstation running Linux 3.11 on top of two hexa-core Intel Westmere-EP processors. Details are shown on Table 5.1.

The hardware platform we use has a total of twelve cores. As already mentioned, StreamIt applications used in the experiments are compiled to 11 threads. We keep core 0 available for running all the system's processes that can be migrated and the benchmark scripts in order to minimize the operating system impact on the experimental results. The monitoring thread is also running on the core 0.

5.2.4.1 Scenario 1: Reaction To Preemption By Other Applications

In this scenario, we run one StreamIt application with an attached throughput constraint along another non-dataflow application. This non dataflow-application is executed on one of the core dedicated to run the application (cores 1-11). For the applications used in this scenario, $\kappa_{obs}(act)$ values are smaller than $\kappa_{exp}(act)$ values for each actor, i.e. there are no bottleneck actors. In other words, in this scenario the `Monitor` threads always switch to stage 4 at the end of stage 2. We first run the application alone on the system and save the value of $\tau^G_{obs}(c)$ at an arbitrary frequency of 20 Hz. Because the monitoring thread is running on a dedicated core, this frequency does not impact streaming applications performances but only changes the time before the runtime may react. Then, we run a non-dataflow application, in this case a C micro-benchmark performing simple arithmetic in an infinite loop that yields the cpu approximately every microsecond, on core 5 for 10 seconds along with the dataflow application and see how our runtime reacts. We experimentally evaluated the value of the cores imbalance monitoring period described in Section 5.2.2.3 to 10 milliseconds. With this value we are able to correctly identify core 5 as the overloaded core.

Figures 5.5 and 5.4 show the evolution over time of $\tau^G_{obs}(c)$ in tokens per microsecond for two applications (fft and $filterbank$) along with the $\tau^G_{exp}(c)$ values and the perturbation introduction. Repeating the scenario several times leads to very small throughput variations but clearly shows the same three phases. First, the application is launched without any perturbation showing a stable state. During this period, from time 0 to 8, the `Monitor` is in stage 1. Then we clearly see a throughput drop when perturbation is introduced. The `Monitor` enters stage 2 and because it doesn't detect any bottleneck

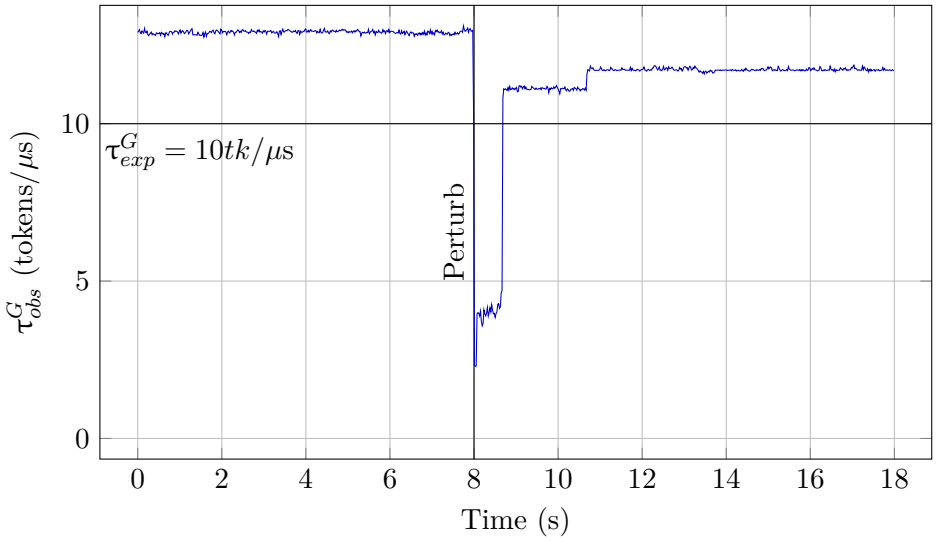


Figure 5.4: Scenario 1 - filterbank. Throughput evolution over time along with the expected throughput value (horizontal line) and the time when perturbation is introduced (vertical line). We see three phases: a stable state, a throughput drop down when perturbation is introduced, and a throughput rise up after rebalancing.

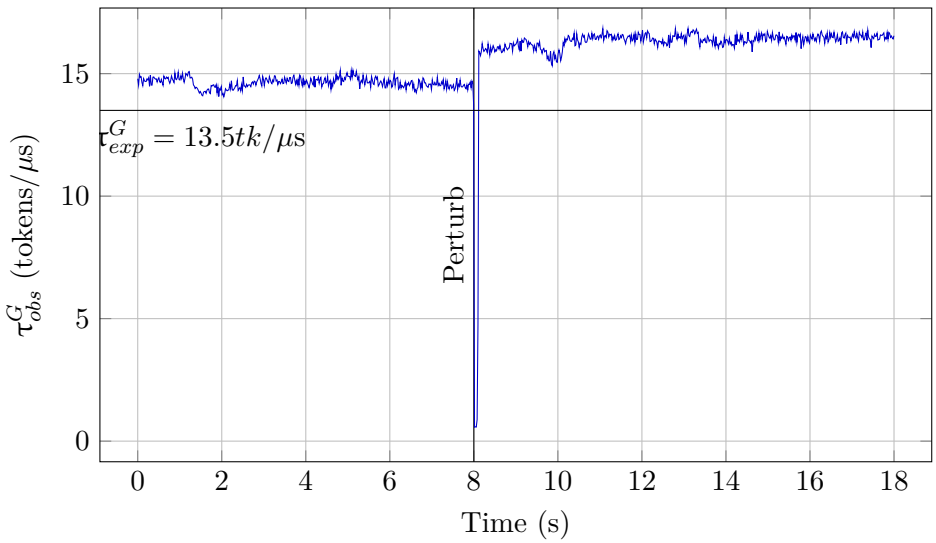


Figure 5.5: Scenario 2 - fft. Throughput evolution over time along with the expected throughput value (horizontal line) and the time when perturbation is introduced (vertical line). As for the filterbank application, we see three phases: a stable state, a throughput drop down when perturbation is introduced, and a throughput rise up after rebalancing.

Actor	$\kappa_{exp}(\mu s)$	$\kappa_{obs}(\mu s)$
fir_filter	780	12.231
combine	780	**3508.678**
down_sample	97.500	16.078

Table 5.2: Actors execution times. *combine* is identified as a bottleneck because its observed execution time is almost five time longer than its expected execution time.

actor, it moves to stage 4 and to the adaptation. Finally, after balancing we see that the throughput is reaching back an acceptable value.

For both applications, because the perturbation introduced consumes a lot of CPU time, all actors of the graph are moved away from the perturbed core. For $filterbank$, it results in a new throughput slightly lower than the initial throughput obtained without any perturbation. This is explained by the fact that this application is compute intensive with few communications between actors and scales well with the number of core. After rebalancing, it executes on 10 cores whereas it was previously executed on 11 cores, which makes a difference in terms of computations for this application. We were not able to explain the artifact leading to a period of half a second between the end of the rebalancing and the throughput rise up.

For fft, the new throughput is higher than the original one. This is also explained by the scaling property of the application. fft steady-state time is small and this application has a lot of communication. The application is thus more efficient when mapped to 10 cores than to 11 cores.

5.2.4.2 Scenario 2: Identification Of Bottleneck Actors

In this scenario, we highlight the effectiveness of bottleneck detection mechanisms. For this purpose we changed the $filterbank$ application used in scenario 1 in order to increase the execution time for the actor called *combine* with artificial computing code. In this case, as soon as the application is started, τ^G_{obs} is smaller than τ^G_{exp} and the `Monitor` enters stage 2. It then identify a bottleneck actors and because memory monitoring is not yet implemented, it reports the information. Table 5.2 shows the average execution times for 3 actors at the end of the local monitoring period. The local monitoring clearly identifies the modified *combine* actor as a bottleneck actor: κ_{obs} is almost five times longer than κ_{exp}.

Once *combine* is identified as a bottleneck actor we start memory monitoring as described in Section 5.2.2. In this fictive scenario, our memory monitoring subsystem does not detect any memory bottleneck. The monitored memory throughput on the two memory controllers is below 1 gigabyte per second whereas the theoretical maximum throughput is 32 gigabytes per second. The runtime then concludes that *combine* execution time can't be reduced using memory adaptations and is only able to report the bottleneck.

5.2.4.3 Runtime Monitoring And Adaptation Overhead

We now evaluate the overhead of the proposed mechanisms.

App.	τ^G_{obs}(tk/μs)	τ^G_{obs} with local mon(tk/μs)	Slowdown(%)
audiobeam	2.04	2.44	-19.80
fft	14.85	16.24	-9.36
filterbank	12.69	12.48	3.65
fmradio	3.82	3.88	-1.52

Table 5.3: Local monitoring overhead. The additional timing system call impacts the performance of applications.

Global throughput monitoring τ^G_{obs} As stated in Section 5.2.2.1 the overhead induced by the global monitoring system is negligible because we only add a counter increment to the original application and because the monitoring thread is executing on a separate core.

Actors execution time monitoring κ_{obs} To evaluate the overhead of local execution time monitoring we first evaluated the cost of the system call `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` that we use to monitor actors execution time excluding preemption. On our platform, the system calls execution time averages to 170 nanoseconds. The overhead on our application is thus in theory $2*170$ multiplied by the number of components in the steady state chosen by the compiler (the system call is made at the beginning and at the end of the component's execution). Table 5.3 shows the average experimental overhead over 20 runs computed by monitoring τ^G_{obs} with and without local monitoring activated.

For 3 of the 4 applications shown on this table, activating local monitoring increases throughput. This is explained by the very short length of the steady-state (the synchronization time is greater than the steady state time for $fmradio$ and $audiobeam$ and equal to the steady state time for fft). These applications spend a lot of time synchronizing on the steady-state barrier and adding time measurement affect the synchronization time positively. We suspect complex mechanisms inside the operating system scheduler to explain these numbers. For the $filterbank$ application, the steady state is longer and is about 10 times the synchronization time spent on the barrier. In this case, adding time measurement increases the steady state length and has a negative impact on the throughput reaching almost 4%.

Balancing To preserve the application semantic while performing the load balancing, we need to suspend applications. As a consequence, applications are not able to output any tokens during balancing. We thus want to minimize the time required to provide a new mapping of the application. Our simple first balancing algorithm has a complexity of $O(n)$ where n is the number of actors. In our two examples of scenario 1, the average balancing times over 20 runs are 489 and 474 microseconds for fft and $filterbank$ respectively.

5.2.5 Discussion

In this section we demonstrated how to exploit the proposals made in Chapters 3 and 4 to extend the StreamIt runtime system with throughput constraint awareness. Using

the micro-benchmarks provided by the StreamIt framework we clearly showed that our runtime system equipped with a `Monitor` is able to identify bottleneck actors and perturbation introduced by non-dataflow applications.

Several perspectives have been opened by this work. First, we implemented all our runtime extensions into user space. The operating system kernel has no knowledge about dataflow applications at all. There are two consequences to that. First, we rely on system calls to ask the kernel for help, and this has a cost. Second, because our runtime system has not the control over non-dataflow applications, its choices may interfere and not be coherent with the choices made by the operating system scheduler. To reduce this overhead, and more importantly to integrate smoothly with the operating system scheduler and legacy applications, we believe that the dataflow knowledge should be added to the operating system kernel.

The second perspective concerns adaptation heuristics. We have only implemented a basic heuristic as a proof-of-concept for our proposal. Static heuristics to efficiently map dataflow actors onto multi-core systems have been extensively studied [63, 86, 87]. It could be interesting to integrate existing work in this domain to our context of dataflow applications executed alongside legacy applications requiring runtime load balancing. The main challenge here consists in keeping the balancing time as low as possible in order to be able to do it at runtime.

A third perspective, linked with the elaboration of more complex adaptation heuristics is to exploit the capability of our runtime system to execute several dataflow applications at once. Indeed, when several dataflow applications are executed at the same time, the monitoring system results are available for all the running applications and global decision have to be taken.

Finally, we want to apply our profiling mechanism to more complex and more realistic applications. We used in this section the SDF model which is quite restrictive in terms of expressiveness. The benchmarks provided by StreamIt are only applications or parts of applications that fit this restrictive model. The next section discusses about the integration of our profiling mechanisms into the ORCC framework that comes with operational state of the art video decoders.

5.3 DDF Programs Profiling

This section describes how we use the mechanisms introduced in Chapters 3 and 4 to profile RVC-CAL applications written and compiled using the ORCC IDE [42]. The profiling results are dedicated to offline optimization. These results aim at identifying how the parallelism provided by the applications is exploited by the runtime system and to accurately understand how hardware resources are used. We validate our proposal using realistic applications provided alongside ORCC.

5.3.1 ORCC Extensions

The profiling mechanisms proposed in this section are integrated into the ORCC IDE.

5.3.1.1 Throughput Constraint Expression

We extend the ORCC framework to allow developers specify throughput constraints on RVC-CAL applications as described in Section 3.6.2. This extension is integrated in the graphical editor provided by ORCC to construct the application's graph from individual actors. Throughput constraints can be added on any FIFO channel of the dataflow graph.

We use the throughput constraints to indicate to the compiler which FIFO channels must be profiled. At profiling time, we use the constraints value to report information about the observed throughput values that we gather during execution and the expected value.

For all the examples in the following sections, we add this constraint on the last edge of graph where it is naturally expressed. For example, on image processing applications the constraint is a frame rate on the FIFO channel feeding the `display` actor responsible to display the decoded frames. Nevertheless, this profiling mechanism could be used to annotate more than one FIFO channel. It can be used for example to annotate all the channels for which the developer has an accurate knowledge of the minimal throughput required to satisfy the global application throughput requirements. The throughput profiling results for these channels allow to identify more accurately the bottlenecks of the application and to highlight some actors that can be bottlenecks.

5.3.1.2 Profiling

In addition to the throughput constraint extension, we extend ORCC to support the profiling mechanisms introduced in Chapter 4.

Application-level profiling Regarding application-level profiling, i.e. the profiling of actors execution time, we rely on a mechanism already proposed by ORCC very similar to the one introduced in Section 4.5.1. It consists in measuring time before and after firing an actor and saving the information in a per actor data structure of the runtime system. To understand the results presented in the next sections, it's worth mentioning that this execution time is incremented only when the actor is effectively fired. In the following we refer to this time as *work time*.

The time spent by the scheduler to see that actors can't be fired is also measured and stored in a data structure. In the following we refer to this time as *scheduling time*. This information is crucial to understand the scaling behavior of an RVC-CAL application.

System-level profiling Our ORCC extensions for system-level profiling consist mainly in memory profiling using the `numap` library introduced in Section 4.5.2. We don't profile here preemption time but only actors execution time because we focus on DDF applications being executed alone on the targeted system. The memory profiling is made of two distinct parts:

- Memory accesses sampling;
- Memory bandwidth profiling for each NUMA node of the underlying hardware.

Because sampling of memory accesses requires additional code generation to enable the correlation between samples and RVC-CAL actors, the compiler that transforms RVC-CAL code to C must be told that we want to do memory sampling. For this purpose, we extend the code generation dialog provided in the ORCC IDE. We also modified the part of ORCC runtime system responsible to start and stop the application. On application start-up, we instruct `numap` to start memory sampling for all the dataflow tasks of the application. When the application is stopped, we correlate the samples recorded by `numap` to RVC-CAL actors and dump the results in a file for offline analysis.

For memory bandwidth profiling we don't need modifications in the C generated code. All the modifications are located in the part of ORCC runtime system responsible to start and stop the application. When application starts, we launch a new thread whose work consists in periodically reading and storing the number of bytes read and written from and to each NUMA node. The period for this thread is chosen through a command line argument provided when the application is launched. When the application ends, the recorded memory read and write counts are also dumped to a file for offline analysis.

Our two memory profiling extensions are portable across hardware platforms thanks to the usage of `numap` provided that it has been ported.

5.3.2 Experimental Setup

The hardware platform used to validate our DDF profiling mechanisms is the one we already used for our SDF runtime system described in Section 5.2.4. It is an Intel NUMA architecture made of two Xeon hexa-core resulting in a total of twelve cores. See Table 5.1 for the details.

As described in Section 4.3.3.2, several important parameters have to be decided when executing DDF applications among the mapping of actors to dataflow tasks, the size of FIFO channels and the way to schedule actors on each dataflow task. We now review the different values we use in our experiments for these parameters.

Actors mapping For the mapping parameter we use two strategies provided by ORCC. The first one consists of a simple Round Robin (RR) mapping of actors to dataflow tasks. The second one, called Weighted Load Balancing (WLB), is based on actors execution time. To get this information, the application is first executed sequentially (using one dataflow task only) with actors execution recording switched on. Then from these execution time, the weighed load balancing algorithm sort actors by decreasing execution times and map them onto the dataflow task with the lowest total load. In both round robin and weighted load balancing cases, the generated mappings are saved into files provided to the runtime system at application start-up through a command line parameter.

Channels size, actors scheduling and inputs ORCC imposes that the size of FIFO channels is a power of two for performances reasons. This size allows to replace a modulo operation by a less costly right shift operation in the circular buffer implementation of FIFO used by ORCC. The size unit is a number of tokens (and not number of bytes).

Regarding the scheduling of actors, we report results for the default round robin scheduler only because the data driven scheduler provided by ORCC is still reported as being in a beta state. Nevertheless, we plan to use our profiling mechanisms on this scheduler in the short term to see how it impacts performances and resources usage.

Because RVC-CAL is first dedicated to image decoding and image processing, the applications we used in this experiments are image processing applications. We thus profile these applications using different kinds of input video streams to see their impact on the performances.

Scripting To choose among all these different parameters and to make our results easily reproducible we developed a suite of python scripts allowing to:

- Compile RVC-CAL applications to C and build the C code;
- Launch the compiled applications;
- Parse the profiling results to generate graphs and table that will be presented in the next section.

The automatic compilation from RVC-CAL to C allows to easily change the compile time parameters that change the generated C code.

5.3.3 HEVC

Our first experiments are performed on an High Efficiency Video Coding (HEVC) video decoder [88]. HEVC, also known as H.265, is the coming standard for video encoding succeeding to H.264. It is jointly developed by the Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG). To handle the constraints imposed by high image resolutions such as 4k and 8K, HEVC improves data compression rate compared to H.264 while keeping the same video quality. ORCC comes with a standardized version of HEVC decoding. We use this decoder in this section [89].

5.3.3.1 Scaling

We first report on the scaling of the video decoder when it's executed with a number of dataflow tasks that vary between one and twelve (recall that each dataflow task is pinned to a specific core of the underlying hardware).

Global tendency Figure 5.6 shows the speedup compared to the sequential version with four different input video sequences. In addition to the input sequences parameter, we used two different FIFO channels size, 8192 and 65536 tokens, and the two different mappings introduced previously, round robin and weighted load balancing.

We use this plot to highlight tendencies before looking into the details. First, we notice that the decoder doesn't scale well above 5 dataflow tasks for any of the four input sequences that we used. Even if we are using up to twelve dataflow tasks, the highest

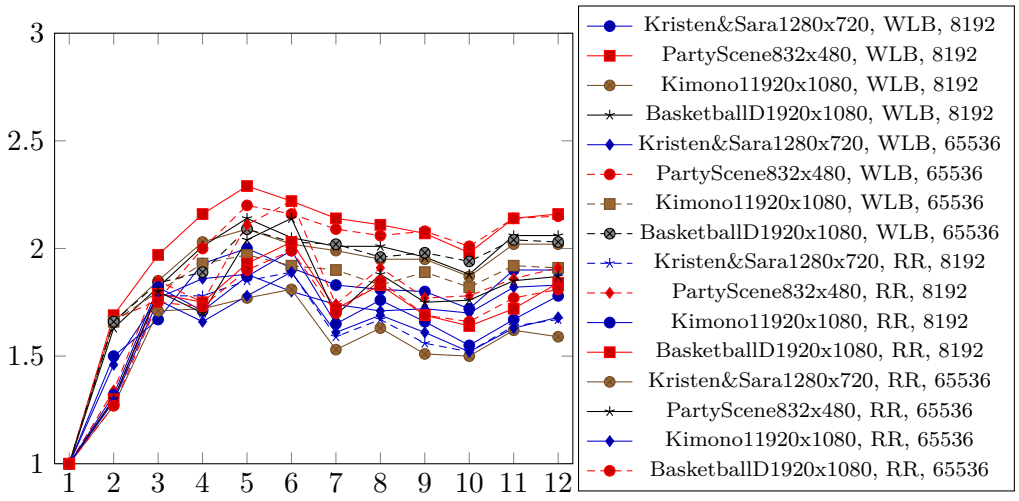


Figure 5.6: HEVC scaling. The X axis is the number of dataflow tasks used to execute the decoder and the Y axis is the speedup over the sequential execution. The legend specifies the video input stream with its resolution, the mapping strategy used (RR or WLB) and the FIFO channel size (8192 or 65536).

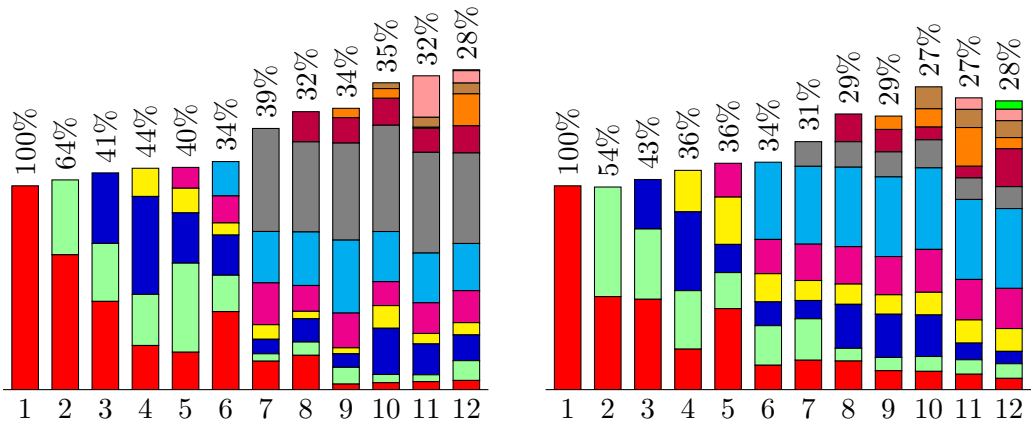
speedup reached is 2.29 for the input sequence *PartyScene*, with a FIFO channels size of 8192 tokens, weighted load balancing mapping and with five dataflow tasks.

The second conclusion we can draw from this plot is that the speedup tendency is globally the same for all the input video sequences and the different FIFO channels size. As a consequence, we reduce the number of values for the parameters in the following when trying to understand why the application doesn't scale. For the input video sequence we focus on the *Kimono* stream because its resolution of 1920x1280 corresponds to the so-called widespread full HD resolution. For FIFO channels size, we keep only the ORCC suggested size of 8192 tokens. For the mapping, we keep the two different strategies as the following will illustrate the obvious fact that the weighted load balancing strategy is more efficient than the round robin one.

To understand why the application doesn't scale above five/six dataflow tasks, we first propose to look at the work load balancing between the dataflow tasks as described in Section 4.5.1.

Dataflow tasks balancing To compute the work load for a dataflow task we sum the work load of all the actors mapped onto the task. Figures 5.7a and 5.7b show the dataflow tasks load balancing for both round robin and weighted load balancing strategies when the number of dataflow tasks vary between one and twelve. For each execution of the application (a bar in the plot), the Y axis shows how the work load is distributed among the dataflow tasks. In these two graphs, scheduling time is excluded. The value on top of the plot, shows how many percents of the total work load are executed by the most loaded core.

From these plots, we clearly see that for both mapping strategies, there are no more clear mapping improvements above five dataflow tasks. Said differently, the work done by the most loaded core doesn't decrease anymore, or only very slowly. This load ranges from 34 to 28 percents for the round robin mapping strategies and from 31 to 28 percents



(a) Round robin mapping **(b)** Weighted load balancing mapping

Figure 5.7: HEVC dataflow tasks load balancing using the *Kimono* input a fifo size of 8192 tokens for both mapping strategies. The X axis represents the number of dataflow tasks used to execute the application and the Y axis is the useful work executed by each dataflow task. The value on top of the plot, shows how many percents of the total work time are executed by the most loaded core.

for the weighted load balancing one.

Note that, as already mentioned, the weighted load balancing mapping strategy performs better. As shown by the results for six and seven dataflow tasks in Figures 5.7a, the round robin strategy can generate mapping to x+1 tasks that are worst than mapping to x tasks. In this example, the mapping to seven tasks leads to tasks with a maximal load of 38% whereas the mapping to six tasks leads to a maximal load of 34%. In the case of the weighted load balancing mapping strategy, a mapping to $x + 1$ tasks is in theory always better than a mapping to x tasks. It's not the case for eleven and twelve cores in this example. This is certainly due to the fact that some actors are slower in the twelve mapping scenario than in the eleven one. Indeed, as depicted by these two plots showing work time only (i.e. excluding scheduling time), the total work time increases with the number of core. This is in part due to memory effects as will be discussed in details in the next section.

Using the profiled information of actors execution times and knowing the mapping from actors to dataflow tasks we now report how the application load is spread among actors. This will demonstrate why the mapping algorithm is not able to reduce the load of the most loaded dataflow tasks above a given number of cores.

Table 5.4 shows how the load is balanced between actors for the execution of the decoder with six dataflow tasks and using the weighted load balancing mapping strategy. We arbitrarily present here the numbers for an execution with six dataflow tasks because the results don't vary significantly with the number of dataflow tasks. We clearly see here that the load of 34% for the most loaded dataflow task shown on Figure 5.7a is made of the execution of one single actor, namely the `HevcDecoder_InterPrediction` actor. After this actor come three other "big" actors limiting parallelism. Together, these four actors represent 67% of the application's load. In order to pretend to better speedups, these actors need to be refactored in order to expose more parallelism to the

Actor	Dataflow task	Load (%)
HevcDecoder_InterPrediction	5	34
HevcDecoder_DecodingPictureBuffer	4	15
HevcDecoder_SAO	0	9
HevcDecoder_DBFilter_DeblockFilt	1	9
HevcDecoder_SelectCU	3	6
HevcDecoder_Algo_Parser	1	4
HevcDecoder_xIT_Block_Merger	2	3
HevcDecoder_IntraPrediction	2	2
HevcDecoder_xIT_IT32x32_IT32x32_1d_0	3	2
HevcDecoder_xIT_IT32x32_IT32x32_1d_1	1	2
display	0	1
HevcDecoder_xIT_IT_Splitter	2	1
HevcDecoder_xIT_IT_Merger	2	1
HevcDecoder_generateInfo_MvComponentPred	3	1
HevcDecoder_DBFilter_GenerateBs	2	1

Table 5.4: HEVC work load by actors using the *Kimono* input, $FS = 8192$, $M = WLB$ and six dataflow tasks. The second column shows on which task actors are mapped. Only the "biggest" actors are shown.

runtime system. Nevertheless, even if not enough parallelism is exposed by the HEVC decoder as demonstrated in this paragraph, the decoder should in theory scale until it reaches the limit imposed by the available parallelism. We now review why this is not the case in practice.

Work time versus scheduling time Even in the case of an optimal mapping algorithm, i.e. a mapping able to split the work load into x equal partitions when using x dataflow tasks, the speedup may be smaller than x. Indeed, getting a speedup of x implies that all the work between the dataflow tasks can be fully executed in parallel. In our case of dataflow applications, even if actors are independent from each other, they require input tokens and room in output channels in order to be fired. As a consequence, the dataflow tasks will have sometime to wait. Moreover, even if a dataflow task can always fire an actor, it requires some work to identify which of the actors are fireable. This scheduling leads to useless time as previously mentioned.

Table 5.5 reports the measured speedups, the theoretical maximal speedup and the percentage of useful work among total work. Theses values have been measured on the *Kimono* input with weighted load balancing mapping strategy and a channel size of 8192 tokens. The optimum speedup column (Speedup Opt) gives a number on the scaling limitation resulting from the impossibility of the mapping strategy to parallelize the

Nb tasks	FPS	Speedup	Speedup Opt	Useful Time%
1	4.68	1.00 (100.00%)	1.00	98.39
2	7.76	1.66 (89.01%)	1.86	90.41
3	8.65	1.85 (81.85%)	2.26	84.65
4	9.53	2.03 (78.69%)	2.59	79.52
5	9.79	2.09 (83.06%)	2.52	85.28
6	9.48	2.02 (76.71%)	2.64	77.06
7	9.34	1.99 (76.17%)	2.62	76.31
8	9.13	1.95 (76.00%)	2.57	76.86
9	9.15	1.95 (76.38%)	2.56	76.59
10	8.74	1.87 (74.62%)	2.50	74.64
11	9.44	2.02 (79.08%)	2.55	78.46
12	9.44	2.02 (78.90%)	2.55	78.28

Table 5.5: HEVC measured speedup vs optimum speedup and percentage of useful work using the *Kimono* input, $FS = 8192$ and $M = WLB$. The optimum speedup is computed by dividing the work load of the sequential version by the work load of most loaded dataflow task.

execution of the big actors. This optimum speedup is computed as following:

$$\frac{Sequential\ work\ load}{Work\ load\ of\ the\ maximaly\ loaded\ core}$$

We see that the best speedup that can be achieved by the HEVC decoder is 2.64 because of the `HevcDecoder_InterPrediction` as already said. The useful work percentage column gives us additional information to understand why the optimal speedups are not reached. The runtime system is spending a part of its time doing useless work. This last column is in accordance with the measured speedup. Spending only XX percent of time doing useful work leads to a speedup that is XX percent of the optimal speedup.

The scaling results and the balance between actors we introduced above are in perfect accordance with results recently published by Jerbi et al [46]. Compared to this work, we presented in this section additional information allowing to better understand why the HEVC decoder doesn't scale to the theoretical limit imposed by the mapping of actors to dataflow tasks. Moreover, we highlighted the fact that the time required to do useful work increases with the number of dataflow tasks used. We now review the results for our memory profiling mechanisms to understand why the execution time increases with the number of dataflow tasks.

5.3.3.2 Memory Profiling

As described in Section 5.3.1.2 , we profile memory bandwidth usage and sample memory accesses to get a better understanding of the application behavior.

Memory Bandwidth The hardware platform we use is equipped with two NUMA nodes. Each of these nodes comes with 3 x 8 GiB of DDR3-1333 memory, leading to a theoretical memory bandwidth of $3 * 1033 * size\ of\ memory\ bus\ (8\ bytes) = 32\ GB/s$. Using a micro-benchmark coming from `minime` [90] and performing 64-bits memory load instructions, we evaluated in practice a maximal read bandwidth of 23 GB/s. To reach this value, the benchmark has to use 8 threads, each pinned to a dedicated core. These threads perform sequential 64 bits memory load instructions. When using more than 8 threads, the memory bandwidth doesn't increase anymore. For the write bandwidth, we evaluate, also using `Minime`, a maximal bandwidth of almost 20 GB/s. These results are conformed to results published for the same Intel micro architecture [22].

This upper bound on the memory read and write bandwidth allows us to get a rough idea on the fact that the HEVC video decoder could be limited by the memory bandwidth.

Table 5.6 reports the measured read and write bandwidth for the two NUMA nodes of the platform. Theses values have been measured on the *Kimono* input with weighted load balancing mapping strategy and a channel size of 8192 tokens. The bandwidth is computed by sampling the value of the memory controllers read and write PMU register 200 times per second. On our hardware platform, it corresponds to the *UNC_QMC-_NORMAL_READS.ANY* and *UNC_QMC_WRITES.FULL.ANY* events as described in Intel's Software Developer Manual chapter 18 [83].

From this table, it is clear that the RVC-CAL HEVC decoder is not limited by the memory bandwidth. Regarding the write bandwidth, the application maximal traffic is just below 1.8 GB/s for the first NUMA node when we use twelve dataflow tasks. For the read bandwidth, the maximal bandwidth for a single NUMA node is reached when using ten dataflow tasks. In this case, the average bandwidth on the first NUMA node is around 10 GB/s. Nevertheless, this value is only half of the maximal bandwidth we got with micro benchmarks leading to think that it's not a bottleneck for this application.

From this table, we can also note that the read memory bandwidth increases sharply as soon as the second NUMA node starts to be used. In this case, we suspect that the memory traffic is not only incurred directly by data but by the cache coherency protocol responsible to keep caches between the two NUMA nodes coherent.

Memory Load Sampling To get a more accurate understanding of the memory usage, we now report the results from the memory accesses sampling. On the Intel hardware platform we use in these experiments, the PMU only provides memory loads sampling. This sampling mechanisms is known as *Load Latency Performance Monitoring Facility* in the Software Development Manual chapter 18 [83]. As a consequence we report only memory load sampling. More recent Intel hardware, also provide memory write sampling capabilities known as *Precise Store Facility*.

Figure 5.8 shows how memory loads are distributed according to the level of the memory hierarchy that served it. Again, the results have been obtained on the *Kimono* input with weighted load balancing mapping strategy and a channel size of 8192 tokens. The X axis represents the number of dataflow tasks used to execute the application and the Y axis is the percentage of total memory access time for each level of the memory hierarchy. The value on top of the plot is the average memory load latency in cycles. This plot only includes memory samples associated to actors firing. In particular, it

Nb Dataflow Tasks	Numa Node	Read Bdw (avg/%RSD)	Write Bdw (avg/%RSD)
1	0	116.41/1.13	81.89/1.15
	1	1.11/0.88	0.53/3.05
2	0	190.80/0.81	135.16/0.81
	1	1.27/0.97	0.55/1.54
3	0	212.54/0.87	150.97/0.88
	1	1.46/0.90	0.63/1.37
4	0	235.12/0.86	166.17/0.87
	1	1.33/1.02	0.58/1.80
5	0	248.70/0.85	171.91/0.86
	1	0.52/1.14	0.54/19.58
6	0	238.80/0.76	166.09/0.77
	1	2.12/0.79	1.12/1.24
7	0	4246.18/0.34	957.00/0.24
	1	254.50/0.78	94.50/0.77
8	0	5454.60/0.30	1212.38/0.17
	1	3226.28/0.50	698.60/0.30
9	0	6810.68/0.09	1484.24/0.09
	1	3898.66/0.11	869.08/0.11
10	0	10513.84/0.06	2157.43/0.08
	1	3046.14/0.19	912.21/0.18
11	0	7264.78/0.07	1768.76/0.05
	1	6191.71/0.12	1379.21/0.05
12	0	8546.34/0.07	1757.84/0.04
	1	4185.00/0.10	1011.86/0.05

Table 5.6: HEVC read and write memory bandwidth in MiB/s using the *Kimono* input, $FS = 8192$ and $M = WLB$. The table shows the average bandwidth along with the relative standard deviation (%RSD) for the two NUMA nodes of the platform.

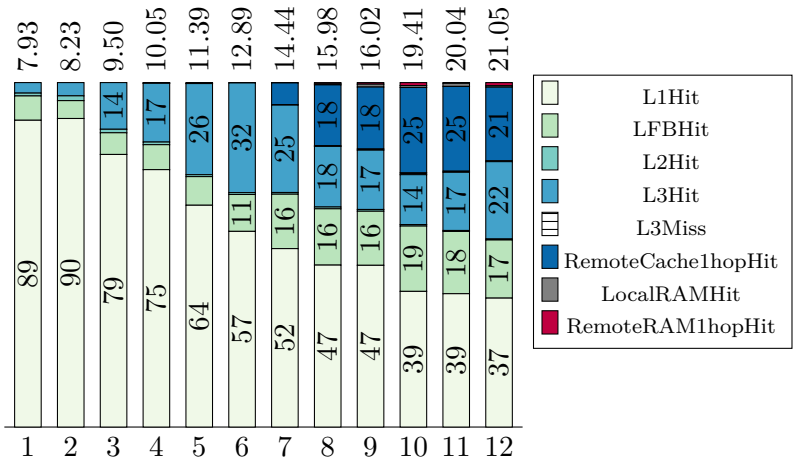


Figure 5.8: HEVC memory load distribution using the *Kimono* input, $M = WLB$ and $FS = 8192$. The X axis represents the number of dataflow tasks used to execute the application and the Y axis is the percentage of total memory access time for each level of the memory hierarchy. The value on top of the plot is the average memory load latency.

excludes the accesses generated by the runtime system scheduler. The sampling rate is 50000; i.e. a sample is recorder every 50000 memory loads. Moreover, the PMU sampling mechanism only allows to sample data memory accesses. As a consequence, this plot doesn't report memory accesses generated by the instructions fetcher.

From this table, it is clear that the memory sub-system impacts the execution time of the actors. In the sequential execution, 89 percents of the memory access time is spent by accesses served by the L1 cache. In the execution with twelve cores, this number falls to 37 percents and we see that 21 percents of the memory access time is spent by accesses served by the remote cache. These accesses correspond to reads into FIFO channels connecting two actors that are mapped on a different NUMA node. Indeed, the weighted load balancing algorithm that we used only takes into account actors execution time and not data exchanges.

The numbers on the top of each bar showing the average memory access time is the main explication of the increase of actors execution time that we noted from Figure 5.7. L1 and L2 data caches are private to each core on our platform. Because FIFO channels producer and consumer can be located on different cores, the actors that consume the data rely on the cache coherency protocol to bring the data in the shared L3 cache. The cache coherency overhead is the highest when the consumer and the producer are located on cores from different NUMA nodes.

5.3.4 MPEG4-part2

We now show how our profiling mechanisms explain the scaling performances on an other video decoder provided by the ORCC framework, namely the MPEG4-part2 decoder. Because, this section presents the same graphs has the one introduced for the HEVC decoder we do not detail the graphs here but only present the conclusion that we can draw from the results.

Figure 5.9a shows an the MPEG4 decoder scales using two different input streams,

two different FIFO channels size and weighted load balancing mapping strategy. The global shape of this scaling plot is the same than the scaling plot for the HEVC decoder. The four curves scale up to 6 dataflow tasks. Then, for three of them we clearly see a drop down when we start to use the second NUMA node (transition from 6 to seven dataflow tasks). After that the four curves have a different shape, that we can explain using dataflow tasks balancing profiling and memory accesses sampling. We now dig into these details for the curve representing the decoding of the *oldtowncross* input stream with FIFO channels having a capacity of 8192 tokens.

Figure 5.9b shows the dataflow tasks load balancing for the *oldtowncross* input, a fifo size of 8192 tokens and weighted load balancing strategy. Figure 5.9c shows the results of memory sampling for the same parameters. The three graphs are presented side by side to make easier the correlation between them.

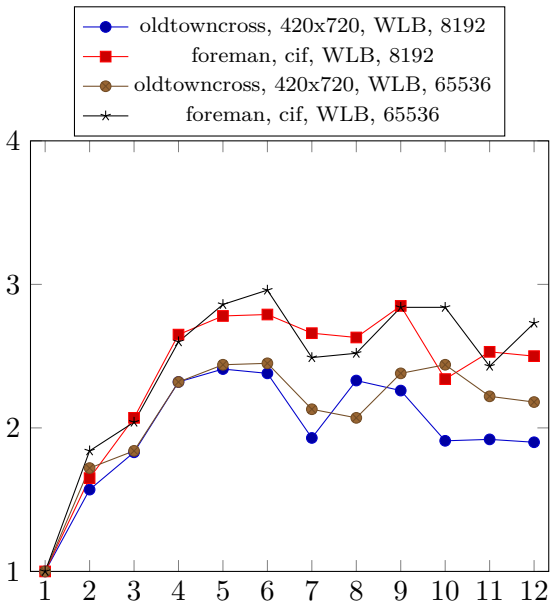
From Figure 5.9b, we clearly see that as for the HEVC decoder, the work time increases with the number of dataflow tasks. Again, this is partly explained by the memory accesses time that also increases with the number of dataflow tasks as depicted on Figure 5.9c. The twelve dataflow tasks bar on Figure 5.9b height is about twice the height of the bar for the sequential execution and the twelve dataflow tasks average memory sample latency is also twice the average memory sample latency for the sequential execution. Nevertheless, we can't explain the big jump in total work time from the six to the seven dataflow tasks using the memory samples latency information only. Indeed, from six to seven dataflow tasks the average memory latency only changes from 10.20 to 10.76. We explain this big jump in total work time by an increase of the execution time of the actors' action scheduler. Indeed, in RVC-CAL an actor can be made of several `step` functions. The choice of which `step` function to execute when an actor is given CPU time by the dataflow task round robin scheduler is left to the actor. It usually made on the current state of the actor. This hypothesis has to be confirmed by profiling at action levels instead of doing profiling at actor level.

These graphs also show an interesting point regarding the transition from eight to nine dataflow tasks. Regarding the speedup, Figure 5.9a tells us that the decoder performances are the almost the same for eight and nine dataflow tasks. Figure 5.9b shows that the mapping is clearly improved when using nine tasks compared to eight (26% to 18% for the most loaded task) but it also shows that the total work time has increased a lot. These two factors compensate each other and lead to almost the same speedup for eight and nine dataflow tasks. In this case, memory samples access time on Figure 5.9c explains the major part of the increase in total work time. The average memory latency jumps from 11.04 to 13.

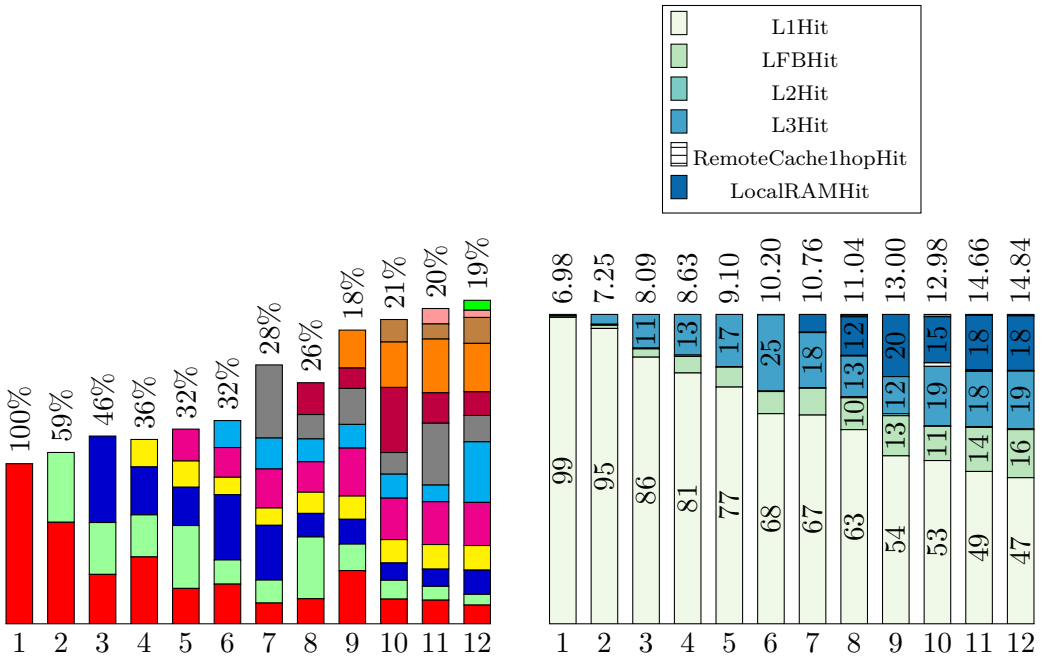
5.3.5 Perspectives

We presented in this section the results for our application- and system-levels profiling mechanisms on the standardized version of the RVC-CAL HEVC decoder. We clearly identified three important points that may lead to optimizations:

- The parallelism exposed by the application is not sufficient (Figure 5.7);
- The scheduling time spent by the runtime system is not negligible (Table 5.5);
- The NUMA architecture has to be taken into account to efficiently exploit the memory sub-system (Figure 5.8).



(a) Scaling. The Y axis is the speedup over the sequential execution. The legend specifies the video input stream with its resolution, the mapping strategy used is always WLB and the FIFO channel size (8192 or 65536).



(b) Dataflow tasks load balancing. The Y axis is the useful work executed by each dataflow task. The value on top of the plot, shows how many percents of the total work time are executed by the most loaded core.

(c) Memory accesses cost per memory level. The Y axis is the percentage of total memory access time for each level of the memory hierarchy. The value on top of the plot is the average memory load latency.

Figure 5.9: MPEG4 analysis. In all the three graphs the X axis represents the number of dataflow tasks used to execute the application. The two bottom graphs have been generated with the *oldtowncross* input, a fifo size of 8192 tokens and weighted load balancing strategy.

Regarding the first point, Jerbi et al [46] recently propose a new version of the HEVC decoder. This version has not yet been standardized but it should be soon. In this optimized design of the HEVC decoder, the big actors we identified using application-level profiling have been spitted. Currently these actors operate sequentially on the three image components (Y, U and V). Jerbi and al [46] propose to have one actor for each image component to expose more parallelism. Moreover, to speedup decoding time and to get closer to widespread decoders such as `ffmpeg`, they propose to use native code relying on x86 SSE instructions instead of using C code generated from RVC-CAL for the critical actors. We started to evaluate this optimized version with our profiling mechanisms. Application level profiling confirms that the application exposes more parallelism. For the memory profiling, we need to identify how our profiler could be extended to take into account the native code coming from a library dynamically linked to correlate memory samples with actors. Then it will be interesting to see how the usage of SSE instructions impacts the memory behavior.

For the second point, we plan to apply our profiling mechanisms using the data driven scheduler provided by the ORCC runtime system. The results should give us accurate information on the efficiency for this scheduling algorithm.

Regarding the third point about the NUMA architecture, we strongly believe that there is room for many performance improvements. First, we plan to apply our memory profiler on new mapping algorithms recently added to the ORCC framework taking into account the amount of tokens exchanged between actors [59]. The memory sampling mechanisms will allow us to confirm the efficiency of these algorithms by numbers. Second, from preliminary results not presented here, it seems that many memory accesses are not generated by FIFO channels accesses. We plan to extend our profiling mechanisms to associate memory samples with global data and stack data. If this hypothesis is confirmed, we may be able to optimize the runtime system with a clever allocation of global data structures to limit the NUMA effects. Then, because the PMU provides the required events to compute the QPI interconnect bandwidth, we plan to extend the `numap` library with support for monitoring this metric. This information will allow to know whether a given mapping saturates the processor interconnect network or not. Finally, we plan to extend our profiler to count the number of instructions to build roofline plots [91]. This performance model recently introduced, makes precise the notions of memory- and compute-boundness. It provides an insightful visualization of bottlenecks by plotting the operational intensity (number of instructions per memory access) against performances along with the memory and the CPU limits imposed by the hardware.

6 Conclusion and Perspectives

Tu deviens responsable pour toujours de ce que tu as apprivoisé.

in "Le Petit Prince" by Antoine de Saint-Exupéry

Multi-core processors are today used in almost all desktop computers, all servers, and in more and more smart-phones. Many-core processors such as Intel Xeon Phi [18], Kalray MPPA [19] or Tilera's processors starts to be used in specific contexts. The actual trend of the computer industry leads us to think that the number of cores in all these systems will increase in the coming years. From the software perspective, this trend has a big impact. To exploit the performances provided by these parallel platforms, the software has to be spitted in many independent activities. To do that, computer scientists either rely on automatic parallelization of sequential code, or on concurrent programming model. In the later case, applications are written by programmers as a set of concurrent activities. This work focused on the dataflow programming model being one of the many concurrent programming model in use today. We investigated how to profile dataflow applications executed on shared-memory multi-core architectures with the objective to ensure a given throughput requirement.

To conclude this thesis, I now summarize the work presented in the three previous chapters. Then I present the perspectives opened by this work. Finally, I give personal thoughts about the acceptance of dataflow programming.

6.1 Summary

This section summarizes the work I did during my thesis. It recalls the main contributions and emphasizes the main difficulties encountered for each of them. I start with the contribution introduced in Chapter 3 about the propagation of throughput constraints in SDF graphs. Then I summarize the work concerning the StreamIt throughput aware runtime system before discussing about the DDF profiling mechanisms while pinpointing the difficulties encountered to apply the profiling mechanisms introduced in Chapter4.

6.1.1 SDF Throughput Propagation

Expressing throughput constraints in dataflow graphs was a question of languages syntax. I proposed extensions for two dataflow languages in this work, but extending any other dataflow language could be easily done. The only question to answer here is, what is the most practical language extension from the programmers point of view ?

The main contribution of the Chapter 3 is the proposition to statically exploit information provided by the SDF model along with a throughput constraint to compute actors execution times that can be used at runtime to identify bottlenecks. It took time to mature this simple idea, mainly to identify how to exploit the throughput constraint propagated on all the edges of the dataflow graph. Indeed, even if this throughput constraint on all the edges of the graph is available, it can't be used directly. An actor that doesn't produce enough tokens in the dataflow graph is either to slow or not feed fast enough for input tokens. A walk-up in the dataflow to chain until finding an actor producing enough tokens allows to distinguish between these two cases. In the particular case of dataflow graphs without cycles. To easily use this information in any SDF graph I proposed to compute the so-called actors expected execution time. Because this information concerns only one firing of the actor, it can be use in any cases.

This contribution is valid for any language conformed to the SDF model. It is clearly independent from any target architecture and independent from the execution model. In particular, this proposal is independent from the level of parallelism used to execute the applications, i.e. it works for SDF executed sequentially and for SDF applications executed in parallel by any number of computing units. As described in the next section, the main challenge regarding the exploitation of this proposal is an implementation concern.

6.1.2 SDF Throughput Aware Runtime

Section 5.2 proposes a SDF throughput aware runtime system based on concepts introduced in Chapters 3 and 4. I demonstrated how a throughput constraint statically propagated on a StreamIt graph can be used at runtime to identify bottleneck actors. Even if the throughput propagation algorithms I introduced in Chapter 3 are straightforward, implementing them into the StreamIt compiler was challenging. Indeed, StreamIt has grown over the last ten years from a basic SDF framework, to a very big infrastructure targeting many hardware architectures and implementing a lot of different, often incompatible, SDF optimizations. Many people have contributed to the framework with different objectives in mind. As result, finding the right places into the compiler source code where to add the throughput constraint propagation code was quite difficult. Moreover, once I identified where to add this throughput propagation code, it was also very difficult to effectively do it without breaking the existing compiler's code.

The runtime system extension implementation was also challenging. The main difficulty I encountered was about the implementation of mechanism to start and stop applications required to safely perform runtime adaptations. This mechanism involves synchronization between dataflow application tasks and the runtime system monitoring tasks. Getting an operational deadlock free implementation for this mechanism was difficult. These synchronization difficulties I encountered reinforced my belief that multi-thread programming should be avoided as much as possible and is not a serious candidate for the incoming more and more parallel processors. The step from sequential programming to multi-thread programming is definitely not as simple as it seems to be [6].

Last but not least, because the SDF throughput aware runtime is running on top of Linux which is a very complex piece of software, it was very difficult to properly integrate with the operating system. As will be discussed in the next section, I strongly

believe that the dataflow knowledge should be added into the operating system as will be discussed in Section 6.2.2.

Despite these main difficulties, I was able to build a proof-of-concept prototype showing the interest the approach. With the dataflow structure knowledge, the runtime system is able to take decisions and to perform adaptation that wouldn't have been possible only with legacy threads. Dataflow actors can be migrated between dataflow tasks, and data placement can be made with the information about who is communicating with who.

6.1.3 DDF Profiling

In Section 5.3 I presented a profiler based on mechanisms introduced in Chapter 4 for DDF applications written and executed using the ORCC framework. I first introduced mechanisms allowing to profile actors execution time and the imbalance between dataflow tasks. Then, compared to the existing dataflow profilers, I introduced memory profiling mechanisms based on hardware performance monitoring capabilities. This profiler has been applied to two standardized video decoders provided alongside the ORCC framework. Using these profiling mechanisms I explained the scaling behavior of these two video decoders. The memory profiling mechanisms I propose in this thesis allow to explain with numbers a part of the non scaling behavior observed when using more than one NUMA node of the underlying hardware. Most of the additional time required to perform the same amount of work when using more cores is caused by memory overhead.

Thanks to the modularity of the ORCC implementation, extending the framework to express and take into account throughput constraints was far more easy than in the StreamIt framework. The main difficulty encountered to build this profiler was about the memory profiling. As stated in Chapter 4, it is required to get a deep knowledge of the underlying hardware and to dig into processors manuals to find the right hardware monitoring mechanisms to set up. Even between two Intel micro-architectures, the PMU can vary significantly and requires a completely different piece of software to analyze memory behavior. To alleviate the burden on programmer, I proposed the `numap` library providing hardware performance counters abstractions allowing to understand how the memory system is used by applications in NUMA architectures. This thin layer, built on top of `perf_event_open()` Linux system call abstracts away these differences behind a simple API. I implemented `numap` for the Intel Westmere-EP micro-architecture that we used in Chapter 4 to validate our DDF profiling mechanisms. This implementation required few decade lines of code, but getting these lines right was quite challenging and time consuming. I exchanged several mails on the Linux `perf_event_open()` mailing list and spent long hours reading the Intel software developer manual [83] before getting the final version of the `numap` implementation for my Westmere-EP experimental platform.

In addition to this difficulty caused by low-level concerns about memory profiling, I faced an issue related to the compile- and run-time parameters provided by the ORCC framework. To execute applications written with ORCC, one has to configure many parameters. These parameters include the size of FIFO channels, the mapping algorithm, the scheduling strategy and some optimization parameters in the generation of C code from RVC-CAL one (such as the use of actors fusion or inlining). Moreover, several optimized versions of the video decoders exist in addition to the one that have been standardized. Because of these parameters, a huge number of profiling results had to be

generated and analyzed. This huge number make difficult the analyze, and make difficult to find which parameter are the most important ones regarding performances. To face these difficulties, I developed Python scripts allowing to easily select and generate plots for any combination of the parameters. As will be discussed in the next section, I also plan to work in collaboration with the ORCC team to better identify the accurate cases where my memory dataflow profiler should be used.

6.2 Perspectives

In this section I describe the perspectives I see for this thesis work. The section is divided into three parts. The first one presents the perspective opened by the SDF throughput constraint propagation algorithm introduced in Chapter 3. It discusses how the proposal could be extended to quasi static dataflow models. I then discuss about the benefits that could result of the integration of dataflow knowledge inside the operating system kernel. Finally, I discuss about the need for an open library dedicated to NUMA profiling.

6.2.1 Throughput Constraints in Quasi Static Dataflow Models

The throughput constraint propagation algorithm introduced in Chapter 3 is valid for SDF graphs. The main perspective for this proposal consists in considering how it could be extended to more expressive dataflow models.

An extension to DDF is excluded, since this model doesn't provide any information about actors consumption and production rates. Nevertheless, this proposal can be integrated into DDF applications by considering static sub-graphs only. This idea of considering static sub-graphs of DDF applications has already been applied successfully to statically schedule part of a DDF graph for performances improvements [92]. The throughput constraint propagation proposal could be applied to each static sub-graph provided that a throughput constraint is provided for each one of them.

Extending the throughput constraint propagation proposal to the CSDF model should be straightforward. Indeed, this model has been proven equivalent to the SDF. Nevertheless, some applications are more easily written using a language based on CSDF than using a language based on SDF.

With the increase of parallelism and the fact that more and more killer applications are focused on data processing, new dataflow MoCs have been introduced in the last years. Compared to the old SDF programming model, these models allow to express more applications while trying to keep the static properties of SDF. Among these models, we can mention SADF [37]. In this model, an application is described as several scenarios. Each scenario is itself described by a SDF graph. As for the integration into DDF graphs, the throughput constraint propagation algorithm could be applied to each scenario provided at least a throughput constraint has been expressed on it.

The throughput constraint propagation algorithm could also be integrated into the recently introduced SPDF model [38]. This model, successfully used to model the next generation of telecommunication protocols [39], allows express graphs with consumption and production rates that can vary during the execution of the application. When these changes can happen depends depend on the particular specification of the model, but in all cases the changes are based on parameters. Indeed, to keep static properties about consistency and finite memory usage as in SDF the model can't allow arbitrary changes. In SPDF, a quasi-static schedule including parameterized consumption and production

rates can be computed statically. Extending the throughput propagation proposal to SPDF will require to take these parameters into account to have actors execution times that also depend on parameters.

6.2.2 Towards a Dataflow Aware Operating System

The second main perspective for this work concerns the integration of dataflow information into the operating system kernel itself. To my opinion, this integration is mandatory to have a proper integration of dataflow applications into legacy systems and not only in specific and dedicated contexts. Indeed, all the dataflow runtime systems [42, 58, 68, 77] I encountered during my thesis, including mine presented in Section 5.2, were implemented on top of an existing operating system. In my case, the motivation for this choice was purely practical. It's far more easy to build a runtime system for scheduling dataflow actors on top of Linux than directly integrated into the kernel scheduler and the kernel memory management layer. This choice leads to implement dataflow runtime systems that either make the assumption that the dataflow applications are the only one running on the system or relying on operating system primitives. The first case excludes the usage of dataflow application along side non dataflow applications because the dataflow runtime system often uses all the hardware resources. In the second case, as demonstrated in Section 5.2, relying on kernel primitives induces overhead and can lead to strange behaviors caused by the kernel scheduler heuristics.

As consequence, and he because operating system is the layer responsible to virtualize hardware resources for applications I believe that the efficient execution of dataflow programs must be under its responsibility. The kernel is the entity having the knowledge of all the applications currently running and how they use hardware resources. In addition to provide concurrent activities through the notion of actors in the same way legacy applications do with threads, the dataflow programming model provides a fundamental information about data exchanges. This information has to be taken into account by the kernel memory management layer for efficient memory usage. This is particularly true on NUMA architectures. Recent Linux kernels implement memory migration heuristics on NUMA architecture [93]. These migration mechanisms are based on a hack in the virtual memory manager to generate fictive access violation faults on memory pages just to compute statistics about pages accesses. These information could be provided for free by the applications, because they are explicit in the programming model, to the kernel memory management layer of a dataflow aware operating system.

Moreover, compared to thread scheduling which is based on complex heuristics inside the Linux kernel, the data dependencies exposed by the dataflow programming model can be exploited to efficiently schedule actors. When an actor is blocked because it doesn't have anymore input tokens to produce or because it doesn't have anymore room on output FIFO channels the actors to be executed are known. In the first case, the actors feeding the blocked actor must be executed in order to provide new input tokens. In the second case, the actors consuming the tokens produced by the blocked actor have to be executed in order to consume tokens in the FIFO channels that are full. Dataflow runtime systems exploiting this information are numerous [58, 59, 77], nevertheless they are all implemented on top of an existing operating system as already stated. All these works don't take into account the cases where dataflow application are executed alongside non dataflow applications.

6.2.3 An Open NUMA Profiling Library

The last main perspective for this work concerns the `numap` library. As stated in Chapter 4 I implemented it only for the intel Westemer-Ep micro-architecture that I used for my experiments because this platform was the one used in the industrial context for this thesis. To prove the benefits for such a library, it must be ported onto other NUMA architectures and used to build other profiling tools.

Few months ago, I started a discussion on the PAPI [85] mailing list to discuss about the abstraction provided by `numap`. Including the abstraction provided by `numap` into PAPI seems to require major changes to the PAPI's existing very basic sampling interface. These changes were on the todo list for the version 6 of PAPI. I don't know exactly when this next major version is planned and what is the status for the memory sampling feature in it. I plan to contact again the PAPI team for serious discussions on this subject.

The complexity of setting up memory sampling on a single Intel micro-architecture was real. Because NUMA architectures are more and more used, and because the memory sub-system is the bottleneck for many killer applications, I strongly believe that this complexity must be abstracted. Moreover, this thesis demonstrated the need for profiling tools that correlate low level profiling statistics with high level concurrent programming models. As a consequence, the purpose of the abstraction library I am speaking about, is not to build yet another NUMA profiler [23, 74, 75] but allowing to easily build such profilers in different concurrent programming model contexts.

6.3 Thoughts on the Dataflow Programming Paradigm

To close this thesis, I want to give few words on my personal belief about the (non) adoption (yet) of the dataflow programming model from the industry. Indeed, even if this programming model is theoretically attractive, for the reasons we already mentioned many times in this thesis, it is not yet widely used in the industry.

During my thesis I had the opportunity to take on some classes at INSA Lyon. From this experience, and my own one, I noticed that we, in the sense of software engineer, are not really prepared to get away from the multi-thread programming model. Indeed, we logically start to learn computer science with the sequential imperative programming model because it closely matches the Von Neumann architectural model basically made of global program counter and a global updatable memory. Unfortunately, for the same reasons that we invoked in the Chapter 2 to explain the wide adoption of multi-thread programming despite its strong drawback(s), this programming model is the one used to introduce concurrent programming to future engineers. Indeed, it's very easy to start programming a parallel computer just by tweaking a piece of simple sequential code to add several threads. Paradoxically, I personally put many efforts and a lot of motivation trying to explain students how to get threads synchronization right where as I was struggling in my own PhD work with deadlock in the StreamIt throughput aware runtime system. I don't know how a shift towards other **deterministic**, or at least less error prone, concurrent programming models could be operated in computer science teaching, but I think it is required. The choice of which concurrent to use is dependent on the teaching context and objectives. Hopefully, this shift already started in many places and I am just saying here that after this personal PhD experience on dataflow programming, I now totally agree with this shift.

Moreover, to be adopted, dataflow programming languages need to be supported with mature and efficient compilers, runtime systems, and profilers. All the software stack should have the data dependency knowledge when applications are written as dataflow graphs. I believe that the framework provided by the ORCC team is a first promising step in this direction. Indeed, the ORCC IDE provides programmers with fully operational tools.

My PhD was a great experience and as stated in the conclusion's epigraph, this work is now under my responsibility forever because "je l'ai apprivoisée". Nevertheless as stated in the introduction, even if I answered almost all the questions I had about computer science before starting this work, I now have many new ones that are more interesting.

References

[1] Manuel Selva, Lionel Morel, Kevin Marquet, and Stéphane Frénot. **QoS Monitoring System for Dataflow Programs**. In *Proceedings of the Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)*, CFSE track, 2013. (p. 3)

[2] Manuel Selva, Lionel Morel, Kevin Marquet, and Stéphane Frénot. **Extending Dataflow Programs with Throughput Properties**. In *Proceedings of the First International Workshop on Many-core Embedded Systems*, MES '13, pages 54–57, New York, NY, USA, 2013. ACM. (p. 3, 34)

[3] Manuel Selva, Lionel Morel, Kevin Marquet, and Stéphane Frénot. **A Monitoring System for Runtime Adaptations of Streaming Applications**. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23nd Euromicro International Conference on*, March 2015. (p. 3)

[4] `www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx`. (p. 6)

[5] `www.nytimes.com/2010/02/05/technology/05electronics.html`. (p. 6)

[6] Edward A. Lee. **The Problem with Threads**. *Computer*, **39**(5):33–42, May 2006. (p. 6, 14, 15, 94)

[7] Victor Pankratius, A. Jannesari, and W.F. Tichy. **Parallelizing Bzip2: A Case Study in Multicore Software Engineering**. *Software, IEEE*, **26**(6):70–77, 2009. (p. 6, 15)

[8] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. **IBM POWER7 multicore server processor**. *IBM Journal of Research and Development*, **55**(3):1:1–1:29, May 2011. (p. 6, 11)

[9] J Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. **Determinism is not enough: Making parallel programs reliable with stable multithreading**. *Communications of the ACM*, 2014. (p. 6, 15)

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. **Concurrent Programming in ERLANG**, 1993. (p. 6)

[11] John E. Stone, David Gohara, and Guochun Shi. **OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems**. *IEEE Des. Test*, **12**(3):66–73, May 2010. (p. 6)

[12] Gilles Kahn. **The Semantics of Simple Language for Parallel Programming**. In *IFIP Congress*, pages 471–475, 1974. (p. 6, 19)

[13] J.B. Dennis. **First version of a data flow procedure language**. In *Symposium on Programming*, pages 241–271, 1974. (p. 6)

[14] `http://ark.intel.com/products/codename/42174/Haswell`. (p. 8, 9)

[15] `www.cea.fr/defense/top500-tera-100-supercalculateur-le-plus-puiss-58066`. (p. 8)

[16] `www.research.ibm.com/cell/`. (p. 8)

[17] `www.top500.org/system/177999`. (p. 8)

[18] Intel. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, 2012. (p. 9, 13, 93)

[19] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. **Time-critical Computing on a Single-chip Massively Parallel Processor**. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 97:1–97:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. (p. 10, 13, 93)

[20] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011. (p. 10)

[21] Ulrich Drepper. **What Every Programmer Should Know About Memory**, 2007. (p. 11)

[22] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. **Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System**. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society. (p. 11, 86)

[23] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. **MemProf: A Memory Profiler for NUMA Multicore Systems**. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association. (p. 11, 47, 98)

[24] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. **Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask**. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM. (p. 11)

[25] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. **Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems**. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM. (p. 12, 47, 60)

[26] Zoltan Majo and Thomas R. Gross. **Memory System Performance in a NUMA Multicore Multiprocessor**. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM. (p. 12)

[27] Zoltan Majo and Thomas R. Gross. **Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead**. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM. (p. 12)

[28] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. **A Case for NUMA-aware Contention Management on Multicore Systems**. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association. (p. 12)

[29] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. **Operating System Support for Improving Data Locality on CC-NUMA Compute Servers**. *SIGOPS Oper. Syst. Rev.*, **30**(5):279–289, September 1996. (p. 12)

[30] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004. (p. 14, 15)

[31] Gilles Kahn and David Macqueen. **Coroutines and Networks of Parallel Processes**. Research report, 1976. (p. 19)

[32] Edward A. Lee and T.M. Parks. **Dataflow process networks**. *Proceedings of the IEEE*, **83**(5):773 –801, may 1995. (p. 20, 41)

[33] Edward A. Lee and D.G. Messerschmitt. **Synchronous data flow**. *Proceedings of the IEEE*, **75**(9):1235 – 1245, sept. 1987. (p. 21, 28, 29)

[34] Edward Ashford Lee and David G. Messerschmitt. **Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing**. *IEEE Trans. Comput.*, **36**(1):24–35, January 1987. (p. 21, 29, 31)

[35] William Thies and Saman Amarasinghe. **An empirical characterization of stream programs and its implications for language and compiler design**. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, page 365, 2010. (p. 21, 23, 28, 74)

[36] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. **Cyclo-static data flow**. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, **5**, pages 3255 –3258 vol.5, may 1995. (p. 21, 43)

[37] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. **A scenario-aware data flow model for combined long-run average and worst-case performance analysis**. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, July 2006. (p. 21, 43, 96)

[38] P. Fradet, A. Girault, and P. Poplavko. **SPDF: A schedulable parametric data-flow MoC**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 769–774, 2012. (p. 21, 43, 96)

[39] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, and Henri-Pierre Charles. **A Compilation Flow for Parametric Dataflow: Programming Model, Scheduling, and Application to Heterogeneous MPSoC**. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 8:1–8:10, New York, NY, USA, 2014. ACM. (p. 21, 23, 25, 49, 53, 96)

[40] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges. **Reconfigurable video coding on multicore**. *Signal Processing Magazine, IEEE*, **26**(6):113 –123, november 2009. (p. 22, 25, 49, 53)

[41] Johan Eker and Jörn Janneck. **CAL language report**. Technical report, EECS Department, University of California, Berkeley, 2002. (p. 22)

[42] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. **Orcc: Multimedia Development Made Easy**. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866. ACM, 2013. (p. 22, 42, 79, 97)

[43] J. Gorin, M. Raulet, Y.-L. Cheng, H.-Y. Lin, N. Siret, K. Sugimoto, and G.G. Lee. **An RVC dataflow description of the AVC Constrained Baseline Profile decoder**. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 753–756, Nov 2009. (p. 22)

[44] E. Bezati, M. Mattavelli, and M. Raulet. **RVC-CAL dataflow implementations of MPEG AVC/H.264 CABAC decoding**. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 207–213, Oct 2010. (p. 22)

[45] M. Chavarrias, F. Pescador, M. Garrido, E. Juárez, and M. Raulet. **A DSP-Based HEVC decoder implementation using an actor language dataflow model**. *Consumer Electronics, IEEE Transactions on*, **59**(4):839–847, November 2013. (p. 22)

[46] Khaled Jerbi, Daniele Renzi, Damien de Saint-Jorre, Hervé Yviquel, Mickaël Raulet, Claudio Alberti, and Marco Mattavelli. **Development and optimization of high level dataflow programs: the HEVC decoder design case**. In *48th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, United States, November 2014. (p. 22, 85, 91)

[47] Hrishikesh Salunkhe, Orlando Moreira, and Kees van Berkel. **Mode-controlled Dataflow Based Modeling & Analysis of a 4G-LTE Receiver**. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 212:1–212:4, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. (p. 23)

[48] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-Franois Nezan. *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer Publishing Company, Incorporated, 2012. (p. 23)

[49] Junaid Jameel Ahmad, Shujun Li, Ahmad reza Sadeghi, and Thomas Schneider. **CTL: A Platform-Independent Crypto Tools Library Based on Dataflow Programming Paradigm**. In *in Proceedings of 16th International Conference Financial Cryptography and Data Security (FC 2012). 2012*, 2012. (p. 23)

[50] `www.storm.apache.org/`. (p. 23)

[51] A.-H. Ghamarian, M. C W Geilen, S. Stuijk, T. Basten, A. J M Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. 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, 2006. (p. 23, 28)

[52] S. Stuijk, T. Basten, M. C W Geilen, and H. Corporaal. **Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs**. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 777–782, 2007. (p. 23, 28)

[53] A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. **Parametric Throughput Analysis of Synchronous Data Flow Graphs**. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 116–121, March 2008. (p. 23, 28)

[54] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. **An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 897–902, March 2010. (p. 23, 28)

[55] Sergiu Carpov, Loïc Cudennec, and Renaud Sirdey. **Throughput constrained parallelism reduction in cyclo-static dataflow applications**. *Procedia Computer Science*, **18**:30–39, 2013. (p. 23, 28)

[56] Rebecca L. Collins and Luca P. Carloni. **Flexible Filters: Load Balancing Through Backpressure for Stream Programs**. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 205–214, New York, NY, USA, 2009. ACM. (p. 23, 25, 29, 53)

[57] Yoonseo Choi, Cheng-Hong Li, Dilma Da Silva, Alan Bivens, and Eugen Schenfeld. **Adaptive Task Duplication Using On-line Bottleneck Detection for Streaming Applications**. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 163–172, New York, NY, USA, 2012. ACM. (p. 23, 29, 53)

[58] Changwoo Min and Young Ik Eom. **DANBI: Dynamic Scheduling of Irregular Stream Programs for Many-core Systems**. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 189–200, Piscataway, NJ, USA, 2013. IEEE Press. (p. 23, 53, 97)

[59] Hervé Yviquel, Emmanuel Casseau, Mickaël Raulet, Pekka Jääskeläinen, and Jarmo Takala. **Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms**. In *International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 725 – 730, Trieste, Italy, September 2013. (p. 23, 91, 97)

[60] William Thies, Michal Karczmarek, and Saman Amarasinghe. **StreamIt: A language for streaming applications**. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002. (p. 25, 33, 49, 53)

[61] Zheng Fang, C. Venkatramani, R. Wagle, and K. Schwan. **Cache Topology Aware Mapping of Stream Processing Applications onto CMPs**. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 52–61, July 2013. (p. 25)

[62] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. **Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures**. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009. (p. 25, 53)

[63] Michael I Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, MIT, 2010. (p. 25, 34, 53, 78)

[64] Matthieu Wipliez. *Compilation infrastructure for dataflow programs*. Theses, INSA de Rennes, December 2010. (p. 28)

[65] Matthieu Wipliez and M Raulet. **Classification and transformation of dynamic dataflow programs**. *Design and Architectures for Signal and . . .*, pages 303–310, 2010. (p. 28)

[66] Richard M. Karp. **A characterization of the minimum cycle mean in a digraph**. In *Discrete Mathematics*, 1978. (p. 28)

[67] Thomas W. Bartenstein and Yu David Liu. **Rate Types for Stream Programs**. *SIGPLAN Not.*, **49**(10):213–232, October 2014. (p. 28)

[68] Ceryen Tan. *A hybrid static/dynamic approach to scheduling stream programs*. Master's thesis, MIT, 2009. (p. 29, 53, 97)

[69] Shuvra S. Bhattacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. (p. 31)

[70] William Thies. *Language and compiler support for stream programs*. PhD thesis, MIT, 2009. (p. 33)

[71] Michael I. Gordon, William Thies, and Saman Amarasinghe. **Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs**. *SIGARCH Comput. Archit. News*, **34**(5):151–162, October 2006. (p. 34)

[72] S. Casale-Brunet, E. Bezati, C. Alberti, G. Roquier, M. Mattavelli, J.W. Janneck, and J. Boutellier. **Design space exploration and implementation of RVC-CAL applications using the TURNUS framework**. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 341–342, Oct 2013. (p. 47)

[73] Sameer S. Shende and Allen D. Malony. **The Tau Parallel Performance System**. *Int. J. High Perform. Comput. Appl.*, **20**(2):287–311, May 2006. (p. 47)

[74] Collin Mccurdy and Jeffrey Vetter. **Memphis: Finding and fixing numa-related performance problems on multi-core platforms**. In *In Proceedings of ISPASS*, 2010. (p. 47, 98)

[75] Xu Liu and John Mellor-Crummey. **A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures**. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM. (p. 47, 98)

[76] Andi Drebes, Pop Antoniu, Karine Heydemann, Albert Cohen, and Nathalie Drach. **Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems**. In *Seventh Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, Vienna, Austria, January 2014. (p. 48)

[77] Antoniu Pop and Albert Cohen. **OpenStream: Expressiveness and Dataflow Compilation of OpenMP Streaming Programs**. *ACM Trans. Archit. Code Optim.*, **9**(4):53:1–53:25, January 2013. (p. 48, 97)

[78] S. M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. **Profile-guided Deployment of Stream Programs on Multicores**. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 79–88, New York, NY, USA, 2012. ACM. (p. 48)

[79] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. **Sigma-C: A Programming Model and Language for Embedded Manycores**. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, **7016** of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2011. (p. 49, 53)

[80] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. **Efficient multicore scheduling of dataflow process networks**. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198 –203, oct. 2011. (p. 50, 51)

[81] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. **GRAMPS: A Programming Model for Graphics Pipelines**. *ACM Trans. Graph.*, **28**(1):4:1–4:11, February 2009. (p. 53)

[82] Sander Stuijk, Marc Geilen, and Twan Basten. **A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour**. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, 2010. (p. 53)

[83] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, January 2015. (p. 62, 70, 86, 95)

[84] `https://perf.wiki.kernel.org/index.php/`. (p. 62)

[85] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. **Using PAPI for Hardware Performance Monitoring on Linux Systems**. In *In Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001. (p. 62, 98)

[86] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. **Mapping Stream Programs Onto Heterogeneous Multiprocessor Systems**. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 57–66, New York, NY, USA, 2009. ACM. (p. 78)

[87] Sardar M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. **Orchestration by Approximation: Mapping Stream Programs Onto Multicore Architectures**. *SIGPLAN Not.*, **47**(4):357–368, March 2011. (p. 78)

[88] G.J. Sullivan, J. Ohm, Woo-Jin Han, and T. Wiegand. **Overview of the High Efficiency Video Coding (HEVC) Standard**. *Circuits and Systems for Video Technology, IEEE Transactions on*, **22**(12):1649–1668, Dec 2012. (p. 81)

[89] `git://github.com/orcc/orc-apps.git`. (p. 81)

[90] `https://github.com/fgaud/Minime`. (p. 86)

[91] G. Ofenbeck, R. Steinmann, V. Caparros, D.G. Spampinato, and M. Puschel. **Applying the roofline model**. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 76–85, March 2014. (p. 91)

[92] Ruirui Gu, JörnW. Janneck, Mickaël Raulet, and ShuvraS. Bhattacharyya. **Exploiting Statically Schedulable Regions in Dataflow Programs**. *Journal of Signal Processing Systems*, **63**(1):129–142, 2011. (p. 96)

[93] `http://kernelnewbies.org/Linux_3.8`. (p. 97)