A Runtime Environment for Real-time Streaming Applications on Clustered Multi-cores

Pranav Tendulkar, Peter Poplavko, Jules Maselbas, Ioannis Galanommatis, Oded Maler

Verimag Research Report n° TR-2015-6

June 2015

A Runtime Environment for Real-time Streaming Applications on Clustered Multi-cores

Pranav Tendulkar, Peter Poplavko, Jules Maselbas, Ioannis Galanommatis, Oded Maler

June 2015

Abstract

Dataflow graphs with static rates, such as synchronous and cyclo-static dataflow models, can be used for multi-core programming of so-called streaming applications, which include a wide range of signal processing and coding. Given soft and firm real-time constraints on throughput, buffer size, and latency, many scheduling optimizers have been proposed for mapping dataflow models on multi-cores. However, not many of them are equipped with runtime environments that can deploy and execute the optimized schedule on the target platform. Among the publicly available runtimes, one can hardly find any that inherently support clustered multi-cores. Those are on-chip multiprocessors that consist of 'clusters', i.e. groups of multiple processors that share a local memory with each other and communicate via a global network-on-chip. This is an important possibility to ensure scalability in the number of cores, making it possible to provide a large (more than 100) number of processors on a single chip. Along with our 'StreamExplorer' open-source scheduling optimizer for static dataflow models we propose a runtime environment that deploys optimized scheduling solutions on a clustered multi-core platform.

Keywords: multi-rate dataflow graph, middleware, DMA, scheduling, schedulability, real-time systems, data parallelism, multi-core compiler, multi-core programming model, multiprocessor network-on-chip, many-core platforms, multiprocessor application benchmarks, multimedia, video and image processing

Reviewers:

How to cite this report:

@techreport {TR-2015-6,
title = {A Runtime Environment for Real-time Streaming Applications on Clustered Multi-cores},
author = {Pranav Tendulkar, Peter Poplavko, Jules Maselbas, Ioannis Galanommatis, Oded Maler},
institution = {{Verimag} Research Report},
number = {TR-2015-6},
year = {2015}
}

1 Introduction

There is abundant research literature on multi-core scheduling (and mapping) of so-called 'streaming' applications, i.e. those that compute output data streams as time-independent[1] functions of input data streams, generally represented by Kahn Process Networks [1]. Such applications include a wide range of audio, image, and video signal processing and coding applications [2]. These applications can be represented by dataflow graphs with static data communication rates, such as *synchronous and cyclo-static dataflow* [3, 4] (SDF, CSDF). When annotated with task execution times, period, throughput, and deadlines, i.e. maximum latency, constraints, these models represent real-time workload model. Based on different variants of this model various schedule optimization methods have been developed,e.g. [5, 6, 7, 8, 9, 10], to name a few.

Every such method is accompanied by an 'optimizer', i.e. an offline tool responsible to analyze the schedulability and to calculate the compile-time parameters of the scheduling problem solutions, according to different costs and platform constraints that are specified in the optimization problem. In order to produce the compile-time solutions the optimizer needs inputs in the form of constraints which represent the precedences between tasks, task execution times, the byte size of the data items communicated between tasks, the communication buffer memory constraints etc. In some frameworks, *cost-space* exploration is done, i.e. instead of solving the problem only for a particular set of processor, buffer memory, period and latency cost constraints, one explores different trade-off (Pareto) points in a multi-dimensional cost space. One of the optimizer frameworks that support cost-space exploration is our open-source software tool *StreamExplorer*. It incorporates the scheduling algorithms presented in [11, 12, 13, 14, 15].

Unfortunately, in the current research tools it is very often the case that the software infrastructure is incomplete, in the sense that though an optimizer is provided it is not backed by real deployment of applications on a target platform. Even when deployment is done, it often turns out to be restricted in many different ways: (1) by letting the user fill the task execution times for the optimizer by hand, (2) by assuming either purely shared- or purely distributed-memory platform, and (3) by assuming FIFO (first-in first-out) channels with strictly sequential reads and writes, which complicates efficient implementation of data parallel applications. Also, it is not always clearly shown in which terms the given runtime can be claimed to be free from timing anomalies, to warrant its use at least in soft and firm real-time systems, imposing period (or throughput) and deadline (or maximal latency) constraints.

In our infrastructure, we implemented a *runtime environment* which deploys the application and the corresponding compile-time scheduling solution calculated by StreamExplorer on a target platform. The structure of the application and scheduling solution is represented in a readable (XML) format, partially compatible with the SDF3 tool format [5]. The application task code should be written in C/C++ using to a custom synchronous-dataflow API. We believe that it is not difficult to customize our environment to different optimizers and target platforms. The runtime is equipped by the source code of application examples, mainly based on a port from StreamIT compiler benchmarks [2].

Our runtime environment is the main subject of this report. It addresses the restrictions mentioned above. Firstly, it is equipped with profiler functionality which can automatically measure and provide the task execution times to the optimizer tool. Secondly, it supports not only purely shared or purely distributed memory, but also *their combination*, in particular, the so-called *clustered* multi-cores, where the processor cores are grouped together into clusters that share local memory, whereas different clusters can communicate via message-passing, based on so-called DMA (direct memory access) transfers. Inside the clusters we support a generalization of FIFO buffer that provides simultaneous access to multiple processors to produce and consume data in parallel. Finally, we can argue that our runtime employs a scheduling policy which is (practically) free from timing anomalies.

Note that whereas the source code of StreamExplorer is available online, the runtime is available only under special request, because currently we have not yet made a clean separation

[1] subject to deadlines

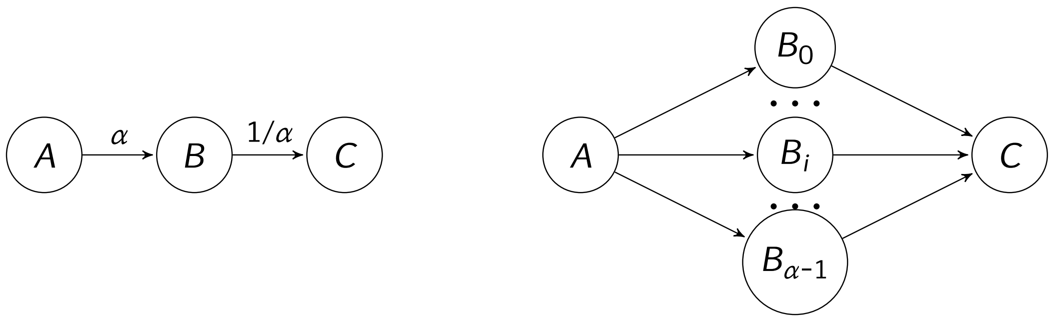


Figure 1: A simple split-join graph and its derived task graph. Actor B has α instances.

between platform independent (Posix-based) and platform-dependent code, where we assume the Kalray MPPA256 [16] platform.

This report is organized as follows. In the first part of the report, we give an overview of (1) the application programming model, (2) the target platform, and, finally (3) the design flow. Next to the programming model description, we list the available application benchmarks.

The second part of the report is dedicated to description of the runtime environment. We first give an overview of the environment, then we describe its real-time scheduling policy, and finally we focus on the generalized FIFO buffer, which is a special feature of our runtime environment for an advanced support of data parallelism. The final section summarizes this report and provides conclusions.

2 Application Programming Model

Our runtime and StreamExplorer have an almost complete support for the SDF model and allow straightforward extension to CSDF. However in this report, we emphasize data-parallel applications and therefore we would like to consider simpler models, specialized on data parallelism, while still covers most of the SDF applications. Thus, we define the *split-join graphs*, which can be thought as a sub-class of SDF graphs.

Split-join graph model is similar to the well-known in real time systems fork-join model, see e.g. [17]. Both models represent a well-structured nested loop computation which is typically observed in certain class of applications. In fork-join graphs, the tasks are divided in stages and segments. This model has a stricter requirement that tasks in the same stage can execute concurrently, while tasks in preceding stages must complete before. In split-join graph, we have a general expression of parallelism in which precedence constraints are expressed only via connected actors. Further we annotate the edges with the amount of data transfer to model the communication.

2.1 Split-join Graphs

Definition 1 (Split-Join and Task Graphs) – *A split-join graph S is defined by $S = (V, E, d, \alpha, \omega)$ where (V, E) is a directed acyclic graph (DAG), that is, a set V of actors, a set $E \subseteq V \times V$ of edges. The function $d : V \to \mathbb{R}_+$ defines the node execution time, $\alpha : E \to \{a, 1/a \mid a \in \mathbb{N}_+\}$ assigns a parallelization factor to every edge. An edge e is a* split, join *or* neutral *edge depending on whether $\alpha(e) > 1$, < 1 or $= 1$. $\omega(e)$ denotes the size of data tokens sent to each spawned task in the case of split, or received from each joined task in the case of join, or just sent and received once per execution if the edge is neutral. A split-join graph with $\alpha(e) = 1$ for every e is called an acyclic task-graph and is denoted by $T = (U, \mathcal{E}, \delta, \omega)$, where the four elements in the tuple correspond to V, E, d, and ω.*

The split-join graph is a generalization of the acyclic task graph by data parallelism, explicitly represented by parallelization factors α. This is illustrated by the example in Figure 1. The decomposability of a task into parallelizable sub-tasks is expressed as a numerical label (parallelization factor) on a precedence edge leading to it. A label α on the edge from A to B means that every executed instance of task A spawns α instances of task B. Likewise, a $1/\alpha$ label on the edge from B to C means that all those instances of B should terminate and their outputs be joined before executing C (see Figure 1). An acyclic task graph can thus be viewed as derived from the split-join graph by making data parallelism *explicit* .

We call the nodes of the split-join graphs *actors* and those of the acyclic task graph *tasks*. The edges of a split-join graph $e \in E$ are called *channels*, and those in an acyclic task graph $\epsilon \in \mathcal{E}$ are called *dependency arcs* or just dependencies.

2.2 Split-join Graph Semantics

In split-join graphs, the tasks (i.e. instances of an actor) can execute in parallel unless there are dependencies between them. In Figure 1 actor A spawns α instances of actor B, which can execute in parallel. Still, for convenience, we explain the functional behavior from sequential-execution point of view.

In sequential execution, channel $e = (v, v')$ can be seen as a FIFO (first-in-first-out) buffer. Let the task instances of each actor v execute in a fixed order, which determines their index: v_0, v_1, v_2 etc. Let $\alpha^{\uparrow}(e)$ be the amount of tokens (also called *production rate*) that are produced by actor v on channel e. The instances v_q of the writer actor of channel (v, v') produce $\alpha^{\uparrow}(v, v')$ *tokens* each in the FIFO channel; in the derived task graph $\alpha^{\uparrow}$ also corresponds to the number of outgoing dependency arcs of v_q. Similarly, $\alpha^{\downarrow}(v, v')$ denotes the number of tokens consumed (called *consumption rate*) and the number of incoming dependencies of instances v'_r of the channel reader actor. Mathematically these rates can be described as:

$$\alpha^{\uparrow}(e) = \begin{cases} \alpha(e) & \alpha(e) \geq 1 \\ 1 & \alpha(e) < 1 \end{cases}; \qquad \alpha^{\downarrow}(e) = \begin{cases} 1 & \alpha(e) \geq 1 \\ \alpha(e)^{-1} & \alpha(e) < 1 \end{cases}$$

Practical split-join graphs should satisfy the consistency property, which is analogous to the one known in SDF graphs. Let $c(v)$ denote the *repetition count* of an actor v. The equations that express the consistency requirement are known as balance equations:

$$\bigwedge_{(v,v') \in E} \alpha^{\uparrow}(v, v') \cdot c(v) = \alpha^{\downarrow}(v, v') \cdot c(v') \tag{1}$$

Note that if Equations 1 have integer solution $c(v), v \in V$ then $k \cdot c(v), \forall k \in \mathbb{N}_+$ is a solution as well. However, we assume the *minimal* positive integer solution and use the notation $c(v)$ for it. Executing each actor $c(v)$ number of times is defined graph *iteration*.

The amount of data that is communicated in an iteration, on an edge can be then easily quantified. The tokens have size ω bytes, and the amount of data produced by an instance of v and consumed by an instance of v', for an edge e is:

$$w^{\uparrow}(e) = \alpha^{\uparrow}(e) \cdot \omega(e); \qquad w^{\downarrow}(e) = \alpha^{\downarrow}(e) \cdot \omega(e)$$

where $w^{\uparrow}(e)$ and $w^{\downarrow}(e)$ is total amount of data produced and consumed on edge e respectively.

Given that the consistency property of the split-join graph is satisfied, one can derive a task graph from it [11], see e.g. in Figure 1. In SDF graph literature this procedure is known as derivation of a homogeneous SDF graph (HSDF) from the original SDF graph.

2.3 Example: JPEG Decoder

Figure 2 shows a JPEG decoder expressed as a split-join graph. It has three main actors: variable length decoding (*VLD*), inverse quantization combined with inverse discrete cosine transform (*IQ/IDCT*) combined and color conversion (*CC*). The VLD actor is responsible for decoding the JPEG parameters which are added to the image as header. After the header is decoded, it performs

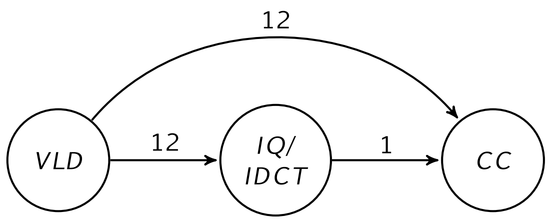


Figure 2: JPEG decoder

variable length decoding on the image data which is then converted into blocks. These image blocks are then passed to the IQ/IDCT actor which performs inverse quantization and inverse discrete cosine transform on these blocks. Finally the color actor performs color conversion which then finishes the decoding from JPEG image to Bitmap image format.

The feed-through channel from VLD to CC actor is used to communicate JPEG image parameters which are defined in the header and the image data. The parallelization factor is proportional to the size of image that is being decoded, in this case factor 12 corresponds to an image that consists of 12 MCU's (minimal coding units) of size 8×8 pixels (which would correspond for example to an image of size 32×24 pixels).

Note that, to be flexible in the input image size, in our programming model we could have interpreted the specified parallelization factors as maximal ones and allow input data for smaller factors, simply by not spawning the superfluous actor instances and not using the resources allocated to them. For example, setting 4800 in the JPEG example would allow us to accept images of certain maximal size, e.g. 640×480, and also smaller images. Changing the values of parallelization factors is possible in certain extensions of the SDF graph model and can be done online in between the graph iterations, however currently we did not implement this feature in our runtime environment.

2.4 StreamIT Benchmarks

Our runtime environment allows programming and executing split-join graph applications, the actor code being written in C/C++ and the structure being defined in XML format compatible to that of SDF3 [5]. The environment is equipped with a set of benchmarks, which consists of the previously described JPEG Image Decoder, and a subset of StreamIt benchmarks [2]. The characteristics of the benchmarks are summarized in Table 1, where the execution cycles are measured on Kalray MPPA256 architecture. Note that Column 4 gives the number of tasks in the derived task graph.

3 Clustered Multi-core Platform

A clustered multi-core platform is usually augmented by a general-purpose processor which can be located either on-chip or off-chip. This processor has full capabilities such as cache and cache coherency, coprocessor support, floating point engine etc., and is called *host CPU*. The '*accelerator fabric*', the multi-core system itself, contains an array of simplified processor cores which can accelerate computation by executing application code in parallel. The processor cores in the accelerator fabric are grouped in *clusters*. The processors (in the cluster) share a finite amount of resources like local memory and can function independently of each other. Figure 3 shows a functional diagram of such a platform.

Such multi-core architectures, although complex, can be efficiently utilized for various applications. One important usage scenario is when a host CPU runs all the usual software stack and general-purpose tasks, whereas computationally expensive highly parallel kernels are forwarded to the multi-core fabric. We discuss the components and terminology of these platforms below.

Table 1: Application benchmark characteristics

BenchMark	#Actors	#Channels	#Tasks	Total Exec. Time (cycles)	Total Comm. Data (bytes)
JPEG Decoder	3	2	25	934288	12384
Beam Former	8	7	53	342816	944
Insertion Sort	6	5	6	40033	320
Merge Sort	12	11	31	102347	704
Radix Sort	13	12	13	85464	768
Dct1	4	3	4	127496	768
Dct2	7	6	21	215525	1536
Dct3	5	4	12	129105	1024
Dct4	7	6	21	183890	1536
Dct5	7	6	21	216079	1536
Dct6	8	7	36	258304	1792
Dct7	8	7	29	218577	1792
Dct8	10	9	38	272514	2304
Dct Coarse	3	2	3	74401	512
Dct Fine	6	5	20	163708	1280
Comparison Count	5	5	20	141397	1280
Matrix multiplication	11	11	79	1087840	10656
Fft	13	12	96	640109	6144

3.1 Clusters

In order to facilitate the development of both software and hardware, multiple cores are grouped together as clusters which share some local resources like DMA engines, intra-cluster shared memory etc. The processors inside the cluster typically are light-weight processors (with limited capabilities in terms of pipeline stages, cache mechanism, or virtual addressing). Power consumption and design challenges are the fundamental reason behind such architecture. These processors are designed to perform computations independent of each other and communicate efficiently. Within a cluster, the processors can communicate using various mechanisms, primarily via shared intra-cluster memory. The communication outside the cluster is managed with help of asynchronous mechanisms such as DMA (discussed further). Typically, communication and synchronization inside a cluster is significantly faster than between clusters.

3.2 Shared Memory

A limited amount of local shared memory is generally made accessible to all the processors inside the cluster. This 'intra-cluster' memory is usually multi-bank in order to provide good performance scalability by preventing memory conflicts due to usage of different banks. This local memory ranges in size typically from some kilobytes to a few megabytes and has access latency usually less than 10 clock cycles. The main memory has access latency of around hundreds to a few thousand clock cycles and its size is of several hundreds of megabytes or a few gigabytes. Typically the program is first loaded in the main memory and the program execution is started. The clusters must fetch the program data into intra-cluster memory, and subsequently perform computations on it. The processors don't have direct access to the main memory, but can fetch data from main memory to local memory. The data movement between them is facilitated by asynchronous mechanisms namely DMA, explained further. The DMA and the hierarchical organization of memory provide the programmer possibility to overlap computation and communication by performing asynchronous data transfers between local and main memory or different local memories in parallel to the main computations.

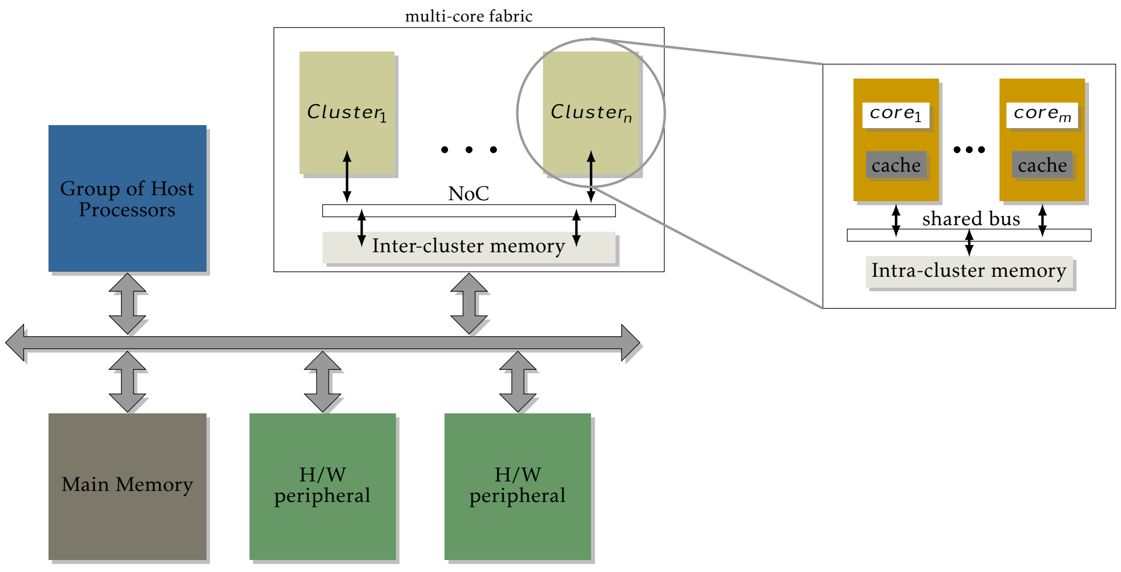


Figure 3: Clustered multi-core architecture

3.3 Network-On-Chip

The clustered multi-core platforms contain network on chip (NoC) in order to facilitate data movement between clusters or between cluster and other hardware peripherals like I/O devices. A processor which is connected to the NoC via a network interface initiates a data transfer, using DMA mechanisms. Data is pushed on the network in form of packets. The size of these packets depends on the NoC architecture. It may also contain other information like the route, the destination address etc. The role of the NoC is also to provide arbitration between packets with conflicting routes and to ensure correct delivery of data from the source to destination.

3.4 DMA

A DMA controller is a piece of hardware used to move data between isolated memories, possibly asynchronously with respect to the core program execution. The main memory access latency can reach few thousands of clock cycles. If the processor had access only to that memory, it will spend most of its time in blocking for memory access rather than performing useful computations. This scenario is improved by having a local memory in the cluster, such that data is fetched from the main memory to the local memory and then the processors perform the computation in the local memory. When the computation is finished, the results are written back to the main memory. This data movement is performed by the DMA controller. In order to efficiently overlap computation and communication, the DMA controller, after being initialized, can function without any intervention from the processor to move the data. Different parameters such as amount of data to be transferred, the source and destination address are specified by the programmer in the initialization of a DMA controller.

The DMA transfer time for a contiguous block of size s, denoted by $T(s)$, consists of two parts, a fixed initialization cost I and data transfer delay proportional to the data to be transferred. This delay is given by $g \cdot s$, where g refers to transfer delay in time units per byte, s refers amount of data transferred in bytes. The *minimal transfer initiation interval* TII, i.e. the time interval between two subsequent transfers to a remote destination originating from the same local source buffer is given in Equation 2.

$$TII(s) = I + g \cdot s + F \tag{2}$$

It is important to note that during the time I both the processor and the DMA channel are busy for initializing the transfer, whereas during time $G = g \cdot s$ only the DMA channel is still busy, pushing the data into the NoC, while the processor can proceed to other tasks in parallel. Before re-using the same local memory buffer from where the data is sent to the network, the processor will have to wait for the completion of the transfer. This operation is blocking until data has departed into the network plus some additional timing delay F to return the control back to the thread waiting for the transfer completion.

3.5 Kalray MPPA-256

Kalray MPPA-256 [16] platform consists of 256 symmetrical on-chip processors which are grouped together, 16 processors per group, into groups called *compute clusters*. These compute clusters are connected by a network-on-chip of 2D toroidal topology. A cycle accurate timer is also integrated inside a cluster, accessible by any processor in the cluster and used for time measurement. The timers of different clusters are not perfectly synchronized. The Kalray platform has a host processor (on a separate chip), an Intel CPU, which is connected to the multi-core chip via PCI bus.

Apart from compute clusters, the platform also has four on-chip I/O subsystems, which are groups of four processors each, called *I/O clusters*. They control the peripherals and act as mediators between the compute clusters on one hand and the host processor with external main memory on the other hand. Their cache memory is the cache for the main off-chip memory. It can be seen as the first level of memory hierarchy between the main memory and the compute clusters, therefore it corresponds to 'inter-cluster memory' in Figure 3.

Each compute cluster contains the processors and a shared intra-cluster memory of capacity 2 MB. Every processor also has independent 8KB instruction cache and 8KB data cache to hide the access latency to the 2MB shared memory. Note that to improve the predictability of our runtime environment we disable the data cache.

Kalray MPPA software library provides an abstraction layer which provides efficient communication and synchronization primitives by hiding low-level hardware details. It provides the API (Application Programmer Interface) for setting up the DMA, for communication between compute clusters, IO clusters and host. The host is connected to the IO cluster subsystem via PCI/e connection and communicates directly with the IO clusters, whereas the latter communicates with the compute clusters.

3.6 Platform Model

Though Kalray MPPA256 is so far the only clustered multi-core platform currently supported by our runtime, we can generalize to a certain class of multi-core architectures, not necessarily clustered, and identify the most important hardware-dependent parameters which would impact the performance of the software based on our runtime on such architectures. We collect these parameters into an abstract 'platform model'.

We consider the hardware platform as NoC-based interconnection of clusters. Each cluster consists of a fixed number of processors. The processors within the cluster are connected to a shared local memory and communicate with each other without any extra overhead or cost. The communication between the clusters is facilitated by DMA. Further, we assume NoC communication with a practically constant throughput g provided to each DMA channel at a negligible (compared to data copy time) network latency, thus we ignore the network congestion.

Given the notions above, we can formally define the platform model as :

Definition 2 (Platform Model) – *An architecture model $A = (X, M, D, I, g, F)$ is a tuple, X is a set of clusters, M and D are the number of processors and DMA data-send channels per cluster respectively. $I \in \mathbb{R}_{\geq 0}$ is the DMA initialization time and $g \in \mathbb{R}_0^+$ is the cost per byte for a transfer. $F \in \mathbb{R}_{\geq 0}$ refers to the constant time required to check if the DMA transfer is complete.*

Table 2 provides us the summary of parameters for Kalray MPPA256 as well as for some other multi-core platforms: a purely distributed-memory one (IBM CELL) and a purely shared memory one (Tilera TILE64).

Parameter	Symbol	Unit	IBM Cell	Tilera TILE64	Kalray MPPA
Number Of Clusters	$\lvert X \rvert$	Number	8	1	16
Proc. Per cluster	M	Number	1	64	16
On-chip memory per cluster		Bytes	256 kB	5.5 MB	2 MB
DMA Per Cluster	D	Number	16	n/a	8
DMA Init. Time	I	Cycles	400	n/a	6000
DMA Cost per byte	g	Cycles per byte	0.22	n/a	0.2818
DMA Completion time	F	Cycles per transfer	0	n/a	100

Table 2: Summary of hardware platform parameters

4 Optimized Deployment

4.1 Design Flow

Given a multi-core platform model, the problem is to map and schedule an application model represented by a split-join graph called *application graph*. The fact that we target *clustered* multi-cores makes the problem extra complex when compared to mapping and scheduling on either purely distributed-memory or purely shared-memory multiprocessors. The optimization problem in this case has more decision variables which affect optimal resource utilisation, such as load-balancing, communication, the number of used clusters, the number of cores and DMA channels used in each cluster, etc. If all these decision variables are presented to the optimizer in a monolithic optimization problem it would result in a very complex set of constraints, practically solvable only for a small number of tasks. Thus in order to be able to schedule larger applications the problem must be split into a few sub-problems. After splitting, the top-level decisions about cluster allocation and their load distribution are made first and later the scheduling inside the clusters is done, combined with mapping to processors and buffer size allocation. This makes the problem solving better tractable.

In StreamExplorer we split (as suggested e.g. in [18]) the design flow into stages described below:

- **Software partitioning**: We partition the actors into groups (software clusters) with the intention that each software cluster is executed on one (hardware) cluster of the platform. Solutions are evaluated according to three criteria: the number of soft clusters (which determines the number of hardware clusters that will be used for the application), the maximal computational workload over the soft clusters, and the amount of communication between tasks belonging to different clusters. These are conflicting criteria and we provide a set consisting of the best trade-offs provided by our cost exploration procedure.
- **Placement of software onto hardware clusters**: (Since the results of this optimization stage are not yet accurately realized by our runtime environment, we skip the details here.)
- **Mapping and scheduling**: In this stage we map the tasks to the processors of their respective clusters and perform scheduling (i.e. static ordering) of tasks on processors and DMA transfers on DMA channels. To this end the original split-join graph is modified to include new nodes associated with DMA transfers as well as new edges that model the static ordering, communication buffer sizes, and DMA synchronization.

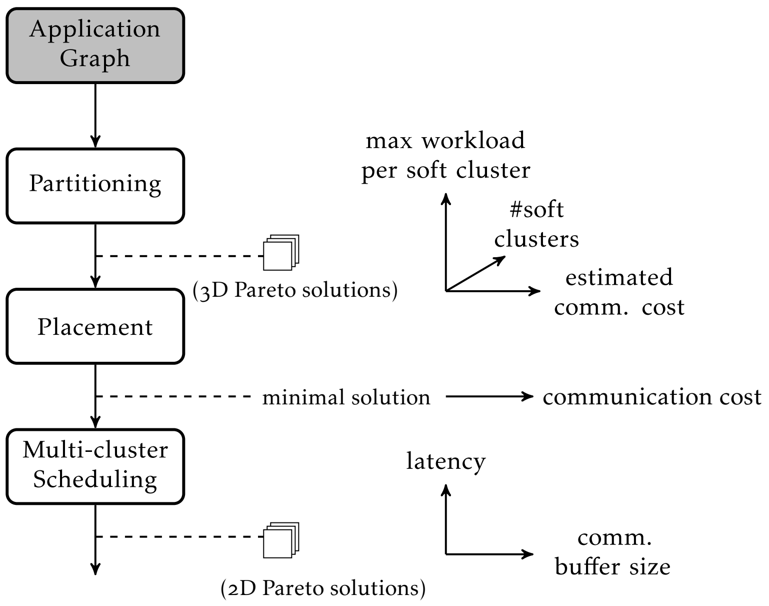


Figure 4: Multi-stage design flow in StreamExplorer

The outcome of this process is a set of solutions in terms of schedules, representing different trade-offs between latency or throughput, the amount of memory used in a cluster and number of clusters. Our runtime environment executes these multi-cluster schedules produced by the StreamExplorer optimizer tool, where we measure the latency or throughput and compare with the value predicted by the optimizer. The main methodology of StreamExplorer for solving the combinatorial optimization problems is to express the problem constraints in terms of so-called satisfiability modulo theory (SMT) solver, such as Z3 [19] and then query the solver for a solution. A detailed description of the design flow stages is beyond the scope of the report, but can be found in [15] for latency and [13] for throughput.

4.2 Tools

StreamExplorer is based on a collection of different algorithms which operate on split-join graphs. The brief structure of the tool is shown in Figure 5. It consists of following parts:

- **Split-Join Graph Core**: This part forms the core of the StreamExplorer, which contains the code to represent different components of the split-join graph model, like actors, channels, ports etc. It defines a graph object which is a collection of all these elements.
- **Property Analyzer**: This part performs different analyses on the split-join graph. For example it performs deadlock analysis and checks for the consistency of the graph. In addition it can provide different properties like total execution time in the graph, longest path etc. It also provides support functions to determine the connected actors / tasks, get split join factors etc.
- **Input Parsers**: The parsers form a vital part of the StreamExplorer which parses the input XML file, containing the information about the split-join graph model (rates, number of actors, interconnections etc.). It also contains a parser to parse the specification of hardware platform defined in subsection 3.6.
- **Solver Interface**: This part of the StreamExplorer performs the conversion of the split-join graph and platform model to constraints which are presented to the SMT solver. We have integrated Z3 SMT solver tightly with the StreamExplorer. The StreamExplorer creates variables and constraints for non-preemptive scheduling, from the split-join graph model

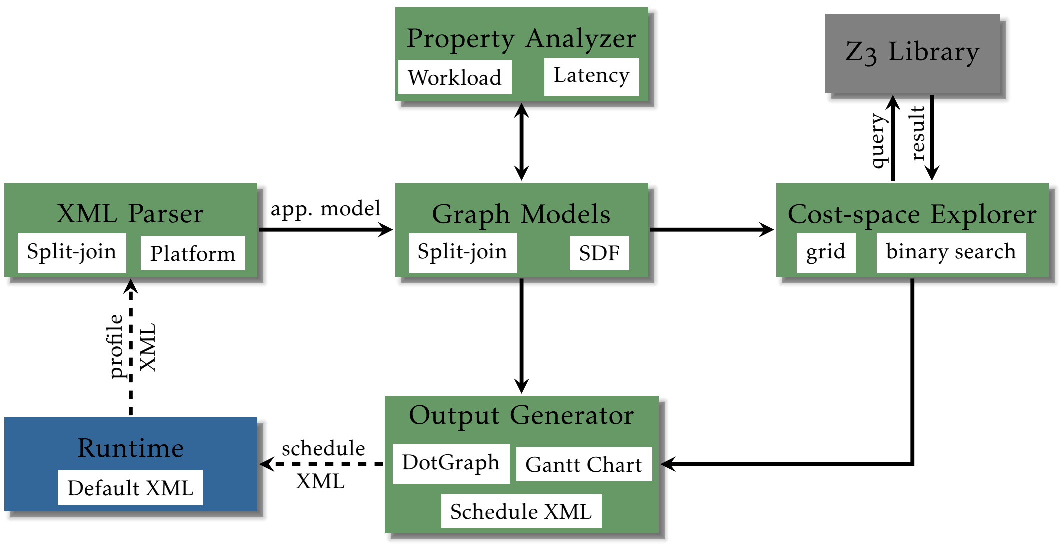


Figure 5: Structure of StreamExplorer

and platform parameters.

- **Cost-space Explorer**: This part of the StreamExplorer consists of the cost-space exploration algorithms (grid-based, binary search etc.). The algorithms in this part are generic and can be easily configured for different exploration like latency vs buffer size, latency vs processors etc.
- **Output Generators**: The output generators produce the results of different analyses and optimizations performed. The most commonly used for analysis of inputs and results, are the dot graph generation for graphical representation of split-join graphs, Gantt chart generation for a schedule, schedule XML generation for runtime environment.

StreamExplorer consists of total 25K+ lines of code written in Java.

The runtime environment consists of 15K+ lines of code written in C++. It is interfaced with the SMT solver using Z3 library. Z3 solver in the tool, can be easily replaced by another solver by performing minimal changes. The link between the StreamExplorer and the runtime environment is provided with bash scripts which automate the process.

5 Runtime Environment

As explained in the previous section, the runtime environment is a piece of software written in C++ that executes on the target multi-core platform. It has two different purposes in the design flow: (1) it ensures the execution time profiling; (2) it realizes the deployment and actual execution of applications. Thus, the former feeds an input to the design flow, whereas the latter actualizes its outputs.

5.1 Using the Runtime for Profiling

The profiling of the application is done mainly in order to measure the execution time of actors. The execution times are used for scheduling in the optimizer. We use the XML format similar to that in SDF3 tool [5] to present the split-join application graphs in the form of an equivalent SDF graph. The profiling of application is done on the platform, which uses the runtime environment for measurement and produces a XML file which contains the timing information of the application. In order to run the profiled application, the runtime environment must be initialized (for data-structure, FIFOs etc.) and hence must be provided with a schedule XML. Thus it is a cyclic requirement, where not only for profiling we need a schedule XML, but also for scheduling we need the execution times from profiling.

In order to break this cyclic dependency, StreamExplorer generates a default schedule XML file, which configures the split-join application graph to execute an arbitrary valid sequential schedule on a single processor in a single cluster. The knowledge of execution times is not required to generate such a schedule, as in this case we do not seek to optimize the timing costs and satisfy thee timing constraints. The sequential schedule is used to perform the execution time measurements for all actors when they are executed in the runtime environment. For profiling purposes the application is executed for 100 graph iterations and a statistical data is produced on the execution times. It contains minimum, maximum, average values for every actor present in the application. The profiling reports worst case and average case value of the actors and either of them can be chosen for the optimization purpose. The chosen characteristic execution times are automatically annotated in the application graph provided to the optimizer.

Note that during the execution of the sequential schedule employed for profiling by construction there is no shared memory bus interference between processors, as only one processor is running. Thus we measure the execution times under the assumption of no interference and hence the conclusions of the optimizer based on these execution time measurements may potentially be too optimistic from the timing point of view. However, in our experiments [12, 13] with the application benchmarks on Kalray MPPA256 the impact of bus interference turned out to be insignificant, which is presumably due to various hardware mechanisms employed in the cluster architecture to reduce and hide the memory access latency (even with disabled data caches).

5.2 Using the Runtime for Deployment and Execution

For the given solution produced by the optimizer, the runtime environment takes the corresponding input XML file that describes the split-join application graph and gives the links to the functional C++ code of the actors. The XML file also gives the binding of actors and DMA operations to the hardware resources and their scheduling ordering calculated by the optimizer. The runtime deploys the scheduling solutions on the platform in three phases:

- Initialization : allocate data structures for execution.
- Execution : execute different schedules on processors.
- Restitution : collect results and deallocate data structures.

5.2.1 Initialization

The first phase performs the initialization of the application graph on the platform. It first reads the communication channel buffer sizes allocated in the schedule, and it allocates the memory and initializes the data structures for them. Further, it initializes every processor that is supposed to execute tasks and provides it with the static order of tasks according to the schedule. The information of the static schedule described in the schedule XML file is discarded, however the ordering between the actors is retained. The names of actors in the schedule XML file are resolved to the call addresses of the corresponding software subroutines which constitute the functional code for the actors. When the name resolution is finished, the runtime environment calls the application-specific initialization subroutines which should perform setup operations, such as memory allocations specific to the application.

5.2.2 Execution

After the initialization is finished on the hardware platform, the runtime enters *Execution* phase. In this phase, the runtime spawns a thread on every processor to which a schedule is allocated. The remaining processors remain idle. Every processor executes its respective static-order schedule in a self-timed way for a fixed number of iterations. In a later subsection we explain our data-parallel communication buffer design, which ensures at run time the correct synchronization between actors for their proper communication. We enforce a barrier synchronization between iterations

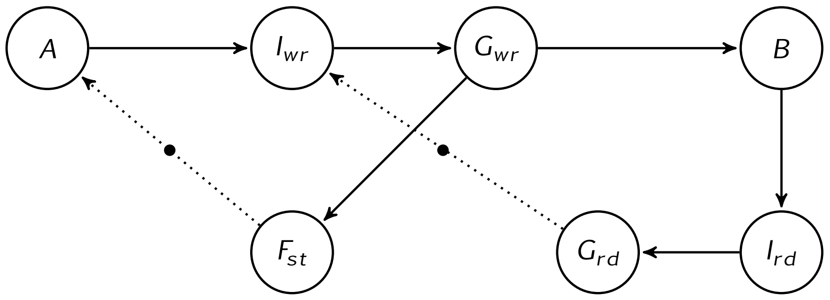


Figure 6: A model for an inter-cluster buffer with DMA

when we execute non-pipelined schedules. The runtime measures the time required for execution of the actor instance which is also used for debugging purpose. The latency is measured for every graph iteration on every processor. The maximum of the values measured on all the processors, gives the latency of the graph iteration.

5.2.3 Restitution

After the execution has finished, the runtime calls the application specific exit calls for application to release any allocated resources and transfer of the application output data (e.g. decoded JPEG image). Then it deallocates all the data structures and gathers the measurements that were done on the platform. Finally it performs statistical analysis on the measurements and generates the output XML. The output XML generated by the runtime environment contains statistical information about actor execution as well as latency. The latter is compared with the model predicted latency.

5.3 Inter-cluster Communication using DMA

Figure 6 shows a simplified HSDF model of inter-cluster communication buffers implemented in our runtime environment for the communication channels between actor pairs mapped to different clusters. The communicating actors are denoted A and B. An HSDF graph model, employed in the figure, is similar to task graphs, with the difference that, similar to Petri net places, the edges between actors may contain initial tokens. Similar to Petri net transitions, an HSDF actor can execute only when there is one token available at each input edge. After every execution the actor produces one token at each output edge. For example, in Figure 6, actor A is the first actor that can execute and after executing it will enable actor I_{wr}.

The communication buffer uses DMA transfers to copy the buffered data from the cluster memory of A to the cluster memory of B. We see actors named 'I' and 'G' on the path from A to B, with 'wr' subscript denoting that they are used to write data. The cyclic path composed of A, I_{wr}, G_{wr} and F_{st} models the transfer initiation time TII given in Equation 2. Actor F_{st} performs a check of the availability status (hence the subscript 'st') of the buffer memory, which only gets 'ready' after the completion of the previous transfer. The loop contains initial token that circulates in the loop at runtime, going between four different phases, calculating new data (A), initiating the data transfer (I_{wr}), pushing the data into the network (G_{wr}), and checking the memory availability status (F_{st}).

We see that I_{wr} takes another input except for the data from A. This input indicates availability of the memory space in the destination cluster, where actor B is located. Before the transfer starts the remote buffer should be ready to receive the data. The initial token present at this input indicates that the space is initially available. When the data transfer starts this space gets reserved, but it is released again when actor B finishes processing the received data and signals back to the cluster of A to notify that the buffer space has been freed up. For this, actor B employs another DMA data transfer, going in the backward direction. The backward transfer is modeled by I_{rd} and

G_{rd}. The signaling about the availability of space at the remote cluster via a backward network channel is commonly known as 'flow control'.

Note that DMA initialization, modeled by actors I_{wr} and I_{rd}, is always immediately followed by the corresponding action of pushing data into the network, modeled by actors G_{wr} and G_{rd}. Initialization and network copying can be seen as one atomic operation. However, we still model these two operations by two distinct actors because they occupy two different sets of resources. The DMA initialization keeps both the processor and the DMA channel busy, whereas the network copying occupies only the DMA channel, while the processor is available for executing other operations in parallel.

Note also that the communication buffer model shown in Figure 6 assumes that only a single-token buffer space is allocated in the clusters at both sides of the buffer. In general, a larger buffer capacity can be allocated, which can be modeled by a larger number of initial tokens in the edges. Also, the model assumes that the parallelization factor of channel (A, B) is equal to one. A larger parallelization factor α is implemented in our framework at the edge (G_{wr}, B). This means that the data producer, actor A, spawns only one DMA transfer, which takes all the tokens produced by A to be consumed by α instances of B. Upon arrival at the destination cluster, the tokens enable α instances of actor B, which may start processing them in parallel on different processors. The data-parallelism aspects of inter-cluster buffers are further explained in Section 6.3.

5.4 Online Scheduling Policy

To ensure the satisfaction of *soft real-time constraints*, i.e. the maximal *latency* (deadline) or maximal graph *iteration interval* (the period), in our runtime we use a variant of (non-preemptive) static-order scheduling policy extended by barrier synchronization and DMA support. The main concept that defines our scheduling policy is the *schedule graph*, which is the task graph obtained by extending the original application graph with extra nodes and edges to represent the DMA communication operations and the schedule ordering calculated by the optimizer. The optimizer verifies the timing constraints of the application by *timing analysis* of the schedule graph. The online scheduling policy should be consistent with the schedule graph model and robust against variations of execution times to ensure that the analysis results are valid and the timing constraints are met. The schedule graph details are given in [12, 15]. Technically, this graph is represented by the schedule XML file loaded into the runtime environment at the initialization phase.

5.4.1 Barrier and Environment Synchronization

The *barrier synchronization* is optional, it was introduced to ensure a synchronized start of split-join graph iterations on different processors in different clusters. It splits the system execution into *frames*, and inside each frame a single split-join graph iteration is executed. Ideally, the frame synchronization would be intended to synchronize the execution of graph iterations with the system IO peripherals for processing periodically incoming and outgoing data. However, our runtime currently does not support such real-time communication with the environment. Instead, in each frame the system gets the same input data, pre-loaded during the initialization phase. Similarly, the last calculated frame output is gathered up from the system at the restitution phase, whereas currently the sole goal of frames is to collect system statistics – the graph latency – based on repeating the same experiment multiple times. Of course, the runtime can be adapted in future work to realize frames that send and receive data to/from the environment on-the-fly.

5.4.2 Static-order Policy with DMA Support

Inside every frame, the same static order scheduling is repeated. The actions of this scheduling policy are determined by the following configuration data provided in schedule XML file:

1. per processor and per DMA channel, an *ordered list of tasks and communication operations* statically scheduled on the given processor or DMA channel and their relative static order;

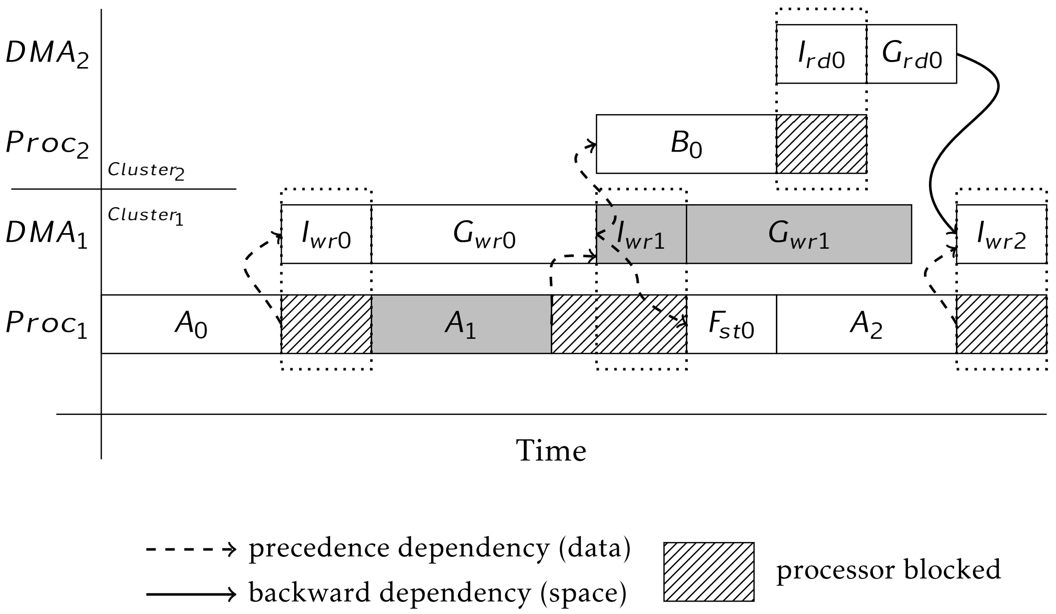


Figure 7: Sketch illustrating static-order policy with DMA support

2. the *schedule dependencies* between the tasks/operations placed in different lists; whereby the dependencies can be due to waiting for data, waiting for space in the buffer or waiting for the availability of the DMA channel.

On every processor and DMA channel, our runtime calls the tasks and communication operations specified in the static order specified in the lists. After being called, every task and communication operation waits, using inter-thread and DMA synchronization mechanisms, until all its predecessors according to the dependencies have finished. This is needed for synchronization between different processors and DMA channels.

The contribution of our variant of static-order scheduling policy is an advanced support of DMA communication and an implementation on a clustered multi-core system. The sketch in Figure 7 illustrates a Gantt chart of executing a static-order schedule that involves DMA transfers and two clusters, based on the model similar to the one that was shown in Figure 6. This example illustrates different features of our scheduling policy, whereas a more detailed description is given in [12, 15].

In the example, task A_0 is the first in the static-order list on processor $Proc_1$ and it spawns a DMA transfer (I_{wr0}, G_{wr0}) for writing data to the remote cluster. This pair of operations represent two phases of a single monolithic transfer: initialization and network copying. Being monolithic, this pair of operations correspond to a single entry in the static order list for DMA channel DMA_1, however in the schedule analysis we distinguish them by their different impact on the processor that spawns the transfer. To spawn the transfer the processor should first wait until the turn of the given DMA transfer comes in the list DMA_1 (in this case, it comes immediately) and then it performs a DMA asynchronous-write call. The call returns only after the completion of DMA initialization, that is why the processor remains blocked until its completion.

The transfer is spawned after the completion of A_0, therefore there is a dependency from A_0 to I_{wr0}. Transfer (I_{wr0}, G_{wr0}) is the first in the static-order list for channel DMA_1, therefore it starts immediately after its predecessor (A_0) finishes.

The second task in the list for $Proc_1$ is A_1, which also spawns a DMA transfer, (I_{wr1}, G_{wr1}). Although A_1 writes data to the same logical FIFO channel as A_0 (i.e. the application channel between actors A and B) the corresponding transfer could have been mapped to a different DMA channel in $Cluster_1$ (equally, A_1 could have been mapped to a different processor in the same cluster), however here it is not the case. Transfer (I_{wr1}, G_{wr1}) is the second transfer in the list for the same channel, DMA_1. Therefore, this time the processor has to wait until the previous transfer on DMA_1 completes, therefore it is blocked also during this waiting time.

In this example we assume a buffer size of two tokens, so after two writes to the logical channel (A, B) we should use F_{st0} to ensure that the first transfer has finished before we can reuse its local memory buffer space in task A_2. Needless to say that F_{st0} could have been mapped on any processor in $Cluster_1$. We assume the same buffer capacity (in this case, two tokens) in the local memory buffers of $Cluster_1$ and $Cluster_2$ for the logical channel (A, B). Therefore, when spawning the third transfer, (I_{wr2}, G_{wr2}), the processor should wait until the flow control, through the transfer (I_{rd2}, G_{rd2}), signals the availability of buffer space at the remote cluster. Therefore, there is a dependency from G_{rd0} to I_{wr2}, which is in fact managed by processor $Proc_1$ because the third transfer is spawned from this processor as well.

5.4.3 The Predictability of Static-order Policy

The static-order scheduling is common in embedded multiprocessor scheduling for signal applications [20]. The main motivations are:

1. static-order policies are *predictable*, i.e. no timing anomalies can occur (under the hypothesis of no bus, NoC, and cache interference); this property is crucial for the analysis for timing-critical applications;
2. *precedence dependencies* in applications and *lack of preemption* in many DSP and massively parallel multi-core platforms, which makes it problematic to apply many classical real-time scheduling approaches, such as those presented in multi-processor scheduling survey [21];
3. *regular execution/communication patterns* in signal processing algorithms.

The predictability property means that testing the timing constraints by a simulation with worst-case execution times (WCETs) is a correct way to prove the system schedulability. The opposite property is a possibility of timing anomalies, where reducing the execution time in some task may lead to delaying other tasks.

Our scheduling policy is predictable if the DMA channels and the processors are the only hardware resources shared between the tasks, because this policy puts the accesses to these resources in a static sequential order. Unfortunately, in reality the tasks and DMA transfers also implicitly use the shared memory buses inside the clusters and the network switches in the NoC. This can lead to bus and network bandwidth interference between the tasks and, in theory, to timing anomalies. However, as it was already mentioned before, from practice we did not notice any anomalies and the interference rarely had a noticeable impact on the performance. We believe that these observations support the hypothesis that our runtime is suitable at least for soft and firm real-time applications.

Note that due to no direct use of the system main memory cache for the computations and also due to disabling the local data caches of the processors there was no data-cache interference. Hypothetically, there was also no instruction-cache interference, except, possibly, in the starting phase of execution, as the actor code fitted into the instruction cache, whereas it had to be initially loaded there at start.

6 Data-parallel FIFO Buffers

In the split-join graph application model, the actors communicate via channels. The writer actor of the channel generates the tokens on the channel, while the reader actor consumes them. When we derive a task graph from a split-join graph each channel translates into a set of one-to-many or many-to-one task dependency arcs, which is essential for data parallelism. Usually, in dataflow runtimes these structures are implemented as fully sequential one-to-one FIFO buffers, requiring the user to manually insert channel 'splitters' if he wishes to exploit data parallelism. In this section we describe our alternative data-parallel implementation of split-join graph channels

6.1 FIFO Buffer Requirements

In any practical implementation the communication channels should use bounded memory, the buffer sizes being computed by the optimizer tool. To suit this requirement, the communication memory should be re-used efficiently. Moreover, to exploit the data parallelism of the split-join channels, it should be possible to execute different concurrent task instances of channel writer and reader on different processors. The requirements of the communication buffers can be listed as follows :

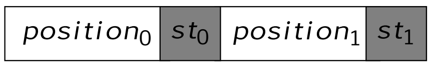
- Correct token delivery from the writer of the token to its reader.
- Reuse of the communication buffers (i.e. of the token space positions inside).
- Support for concurrent instances of writer and reader that access the same token position.
- In the case of inter-cluster channel, the DMA delivery of data tokens and their status information between writer and reader.

We created a software FIFO buffer implementation that satisfies these requirements. To support concurrent writers and readers we exploit the fact that in a split-join graph each writer/reader instance writes/reads statically known amounts of data at statically known offsets. Therefore, each token data item is extended with status record that can be updated independently, indicating whether the given token is ready to be written or read. For the reuse of bounded token memory, we use standard circular buffer with a wrap-around mechanism.

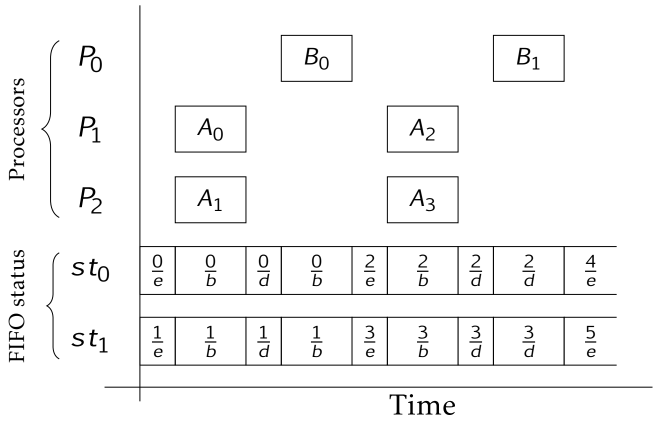
It is very important to note the following. The term FIFO here indicates a certain ordering restriction between the concurrent instance of writer and reader that write and read to the same reused token position in the buffer. The actor instances with a smaller index should *start* no later than those with a larger index. This restriction on the schedule coincides with the notion of task symmetry introduced in [11], where we prove that restricting the schedule like this does not lead to sub-optimality. Note that here we stressed the word 'start', as many instances of the same actor[2] may start at the same time or concurrently on different processors.

Our *FIFO* buffer is an array of a data structure that consists of token data and token status. The status record for every token consists of two entries. The first entry is the *index* of the token, while the second one gives the state of the token which can be either *{'e'-empty, 'b'-busy,* or *'d'-contains data}*. Every writer task of the FIFO buffer can acquire a token only when it is empty, similarly, a reader task of the FIFO buffer can acquire a token only when it contains data. Both the reader and writer change token's state to *busy* after acquiring it. When a writer task finishes, it updates the state entry from '*busy*' to '*contains data*'. Similarly when a reader task finishes, it updates the state entry from '*busy*' to '*empty*'. Complementary to the state entry, the index entry identifies the ordering in which the token would have been handled in a sequential FIFO buffer. In a channel with rates $\alpha^{\uparrow}$ and $\alpha^{\downarrow}$ the instance '0' of the writer writes the tokens in the index range $[0, \alpha^{\uparrow})$, instance '1' writes the tokens $[\alpha^{\uparrow}, 2 \cdot \alpha^{\uparrow})$ and so on. In general, any instance i of a writer writes the tokens in range $[i \cdot \alpha^{\uparrow}, (i+1) \cdot \alpha^{\uparrow})$. Similarly, any instance j of the reader reads the tokens having index $[j \cdot \alpha^{\downarrow}, (j+1) \cdot \alpha^{\downarrow})$. In a bounded buffer of size b tokens, a token space at position k is first used for token k and then reused for tokens $k+b$, $k+2b$ etc. Therefore, the value of index is monotonically increasing in time. Every instance of the reader that reads from the given token position increments its index by b upon its completion. Both the writer and the reader instances access statically known positions in the FIFO buffer which can be easily calculated from their instance index. Thus, a combination of index and state gives an accurate information about which actor instance is the next eligible task to access the given token position in the FIFO buffer.

An important point to note is that the presented FIFO design does not make any assumption about the execution time of its producers and consumers, thus being robust against variation of task execution times.



a FIFO structure in memory, st_i is status


Processors
P_0
P_1
P_2
B_0
B_1
A_0
A_2
A_1
A_3
FIFO status
st_0
st_1
$\frac{0}{e}$ $\frac{0}{b}$ $\frac{0}{d}$ $\frac{0}{b}$ $\frac{2}{e}$ $\frac{2}{b}$ $\frac{2}{d}$ $\frac{2}{d}$ $\frac{4}{e}$
$\frac{1}{e}$ $\frac{1}{b}$ $\frac{1}{d}$ $\frac{1}{b}$ $\frac{3}{e}$ $\frac{3}{b}$ $\frac{3}{d}$ $\frac{3}{d}$ $\frac{5}{e}$
Time

b FIFO token and status on executions

Figure 8: FIFO token example for actors A and B with buffer size = 2

6.2 Data-parallel Buffer Inside a Cluster

Suppose (A, B) is a split-join graph channel with parameters $\alpha = 1/2$, $\alpha^{\uparrow}_{AB} = 1$, $\alpha^{\downarrow}_{AB} = 2$, and buffer size $b_{AB} = 2$. Figure 8a shows how the FIFO buffer is allocated in the local memory of the processor and Figure 8b shows an execution of this example. We use the notation $\frac{index}{state}$ for the status record. The execution works as follows:

- Initially $position_0$ and $position_1$ are marked with status $\frac{0}{e}$ and $\frac{1}{e}$, indicating two empty tokens with index 0 and 1 respectively.
- writer instance A_0 acquires $token_0$ and marks it as busy. Similarly A_1 marks $token_1$ as busy.
- After A_0 finishes execution, $token_0$ contains data which changes its status to $\frac{0}{d}$ indicating it contains data. Similarly A_1 changes the status of $token_1$.
- reader instance B_0, who should read from both $token_0$ and $token_1$, waits till both of them contain data. At the beginning of its execution it marks both them as busy.
- After B_0 finishes execution and increments the index of $token_0$ and $token_1$ to 2 and 3 respectively. Also it marks them as empty.
- A_2 and A_3 waited for token status $\frac{2}{e}$ and $\frac{3}{e}$ respectively. Now they can continue execution and the rest follows similarly.

Table 3 shows the token status before and after execution of each instance. The 'before' status is the status that the given actor instance awaits before it can start. The 'after' status is the token status written by the given task at its completion. Thus we can conclude that irrespective of the execution time and mapping of the reader and writer instances in a sequential or parallel schedule, they will wait for each other to maintain the correct order of production and consumption of tokens. This logic guarantees that the tokens are delivered correctly.

FIFO buffer used by Erbium compiler [22] to map streaming applications on shared memory platforms use similar notions.

6.3 Data-parallel Buffer for Communication between Clusters

In a clustered multi-core platform we have a possibility of allocating communicating actors on different clusters if memory or processor resources of a single cluster are insufficient. In such a case the FIFO buffer must be adapted to transport the tokens from one cluster to another. The runtime checks the cluster assignment of writer and reader to detect inter-cluster FIFO. Such a

[2]unless it is a 'stateful' actor [2]

Table 3: FIFO buffer token status before and after execution of tasks

instance	$token_0$ before	$token_0$ after	$token_1$ before	$token_1$ after
A_0	$\frac{0}{e}$	$\frac{0}{d}$	-	-
A_1	-	-	$\frac{1}{e}$	$\frac{1}{d}$
A_2	$\frac{2}{e}$	$\frac{2}{d}$	-	-
A_3	-	-	$\frac{3}{e}$	$\frac{3}{d}$
B_0	$\frac{0}{d}$	$\frac{2}{e}$	$\frac{1}{d}$	$\frac{3}{e}$
B_1	$\frac{2}{d}$	$\frac{4}{e}$	$\frac{3}{d}$	$\frac{5}{e}$

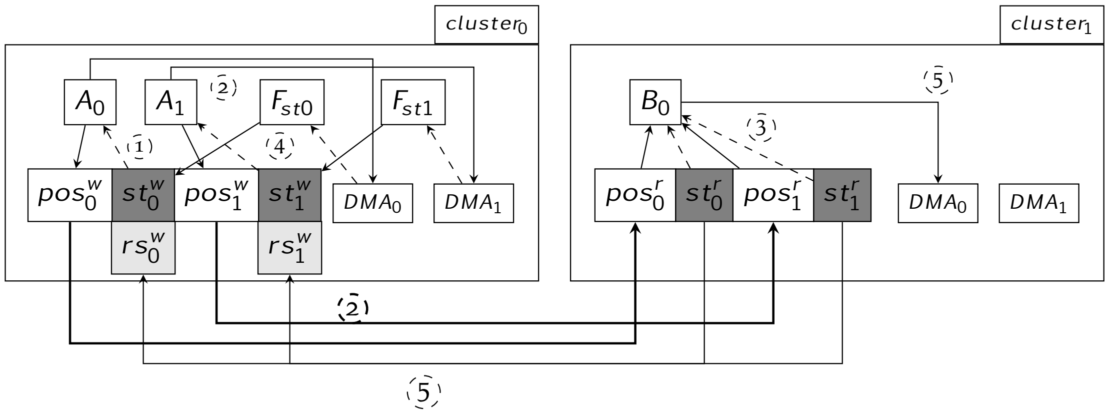


Figure 9: Implementation of an inter-cluster FIFO buffer

FIFO buffer has the writer actor and reader actor allocated on two different clusters. Currently we assume that the instances of the same actor cannot be spread between different clusters. In our FIFO buffer implementation, the same amount of buffer memory is allocated at the writer and the reader cluster. Also, the writer side of the buffer maintains, in addition to its own token status, also the estimated 'remote' status of the reader side. The remote status is updated by the flow control communication from the reader side backward to the reader side, in line with the model shown in Figure 6.

After each execution the writer starts a DMA transfer from its own cluster to the reader cluster to deliver the tokens that it has produced. However, it can start the transfer only when the remote token status indicates empty tokens at the required positions. Similarly, each time when the reader finishes an execution it starts a DMA transfer to communicate the token status at the positions that it has read back to the writer side, for flow control purposes. This classical mechanism allows the reader and writer sides of the FIFO buffer to be in sync with each other.

We implemented an inter-cluster FIFO which transparently handles the data transfer and synchronization between the readers and writers of the FIFO. To explain how it works we use the same example as in Section 6.2, where we assumed a simpler intra-cluster case.

Suppose we have a split-join graph where (A, B) is a channel with parameters $\alpha = 1/2$, $\alpha^{\uparrow}_{AB} = 1$, $\alpha^{\downarrow}_{AB} = 2$. Recall that an inter-cluster buffer is split into writer and reader sub-buffers located at the reader and the writer cluster memory. Suppose that the buffer size allocated to both sub-buffers is two tokens. Figure 9 illustrates the implementation of such an inter-cluster buffer for logical channel (A, B). Tasks A_0, A_1, F_{st0} etc. are mapped to some processors of the writer cluster

($cluster_0$). Recall that F_{st} is a special actor which is needed to detect the completion of a DMA transfer in the writer cluster. The token data and token status entries in the FIFO buffer located in the writer sub-buffer are denoted as pos^w_k, st^w_k, $k = 0, 1$. The corresponding data structures at the reader cluster ($cluster_1$) are denoted as pos^r_k, st^r_k. The writer cluster has an additional status denoted by rs^w_k which indicates FIFO status at reader cluster for this position. The following steps illustrate how the inter-cluster communication takes place.

- **Step** 1: Task A_0 polls the status record st^w_0 in the FIFO until the token at position zero (pos^w_0) contains an empty token. As soon as the token state is 'e'-empty, A_0 marks the token as 'b'-busy and starts to produce data in this position of the FIFO. Similarly task A_1 checks for status st^w_1 and starts to produce data at position 1 of the FIFO. A point to note here is that A_0 and A_1 are not necessarily synchronized. They can execute on the same or different processors. Flags rs^w_0 and rs^w_1 are initialized empty before the execution of the schedules indicating free space for tokens on $cluster_1$.
- **Step** 2: After task A_0 finishes, the position 0 of the FIFO contains data, which is notified by setting its status record to 'd'. Then the task A_0 starts a DMA transfer of token data and status together, but not before the remote status rs^w_0 indicates an empty token space at remote cluster. Similarly, upon its completion task A_1 also starts another DMA to copy the data and status at position 1 to the remote cluster, but not before rs^w_1 indicates availability of an empty space at the destination.
- **Step** 3: The DMA transfers copy the data from pos^w_k and st^w_k to the reader cluster pos^r_k and st^r_k respectively. Until the DMA transfers initiated by tasks A_0 and A_1 are finished, task B_0 is continuously polling the status records at position 0 (st^r_0) and position 1 (st^r_1). It awaits them to indicate the availability of data so that task B_0 can start executing.
- **Step** 4: Task F_{st0}, executing in $cluster_0$ is polling the status of DMA transfer started by task A_0. Once it detects completion of this transfer it marks the status record (st^w_0) as empty and available for reuse. Similarly, task F_{st1} marks st^w_1 as empty when the transfer from task A_1 completes. Note that Steps 3 and 4 execute concurrently.
- **Step** 5: After task B_0 finishes execution, its marks the local status st^r_0 and st^r_1 of token as free. Then it initiates a DMA transfer that copies st^r_0 to rs^w_0 and st^r_1 to rs^w_1.

After these steps, the next instances of actors A and B, namely the tasks A_2, A_3 and B_1, execute the same set of communication and synchronization operations as A_0, A_1 and B_0 respectively, possibly running on different processors and using different DMA channels. Note that the writer cluster can immediately reuse the FIFO buffer positions after the DMA transfer has finished transferring their data into the network, so the writer actor can start again without waiting for the reader to finish.

7 Conclusions

In this report we described the runtime environment of our 'optimizer' tool StreamExplorer that supports Kalray MPPA256 clustered multi-core platform. We believe our runtime to be customizable to other optimizers and multi-core platforms. The optimizer calculates for a given split-join application graph an optimized schedule, whereby if C/C++ the source code of the application tasks is provided then the runtime automatically annotates the task parameters required to perform optimization, namely the measured estimations of the worst-case execution times. At the same time, the main role of our runtime is to deploy and execute the application under the schedule provided by the optimizer.

Our runtime is based on a (practically) predictable scheduling policy, which should warrant its use at least for soft and firm real-time applications. The advantages of our runtime is support of clustered multi-cores and data parallelism present in the classical task fork/join operations, while preserving all the flexibility of mapping and scheduling the parallel tasks to an optimal set of processors an optimal order (which should be calculated by the optimizer).

In future work we consider to make the platform-dependent parts in the runtime decoupled from platform independent ones and then to add the runtime to the online distribution of

StreamExplorer. We also plan to improve the support of pipelined schedules for throughput optimization (which is currently restricted to single-cluster schedules) and to ensure online I/O data communication with the system environment.

References

[1] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing '74: Proceedings of the IFIP Congress* (J. L. Rosenfeld, ed.), pp. 471–475, New York, NY: North-Holland, 1974. 1

[2] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *international conference on Parallel architectures and compilation techniques*, PACT '10, (NY, USA), pp. 365–376, ACM, 2010. 1, 2.4, 2

[3] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, sept. 1987. 1

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *1995 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '95, Detroit, Michigan, USA, May 08-12, 1995*, pp. 3255–3258, 1995. 1

[5] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design 6th International Conference ACSD 2006 Proceedings*, pp. 276–278, IEEE Computer Society Press Los Alamitos CA USA, June 2006. 1, 2.4, 5.1

[6] A. Franceschelli, P. Burgio, G. Tagliavini, A. Marongiu, M. Ruggiero, M. Lombardi, A. Bonfietti, M. Milano, and L. Benini, "MPOpt-Cell: A High-performance Data-flow Programming Environment for the CELL BE Processor," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, (New York, NY, USA), pp. 11:1–11:2, ACM, 2011. 1

[7] W. Thies, *Language and compiler support for stream programs*. PhD thesis, Massachusetts Institute of Technology, 2009. 1

[8] J. Zhu, I. Sander, and A. Jantsch, "Constrained global scheduling of streaming applications on mpsocs," in *ASPDAC*, 2010. 1

[9] D. Liu, J. Spasic, J. T. Zhai, T. Stefanov, and G. Chen, "Resource optimization for csdf-modeled streaming applications with latency constraints," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, (3001 Leuven, Belgium, Belgium), pp. 188:1–188:6, European Design and Automation Association, 2014. 1

[10] B. Bodin, A. M. Kordon, and B. D. de Dinechin, "Periodic schedules for cyclo-static dataflow," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia, Montreal, QC, Canada, October 3-4, 2013*, pp. 105–114, 2013. 1

[11] P. Tendulkar, P. Poplavko, and O. Maler, "Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores," in *Formal Modeling and Analysis of Timed Systems* (V. Braberman and L. Fribourg, eds.), vol. 8053 of *Lecture Notes in Computer Science*, pp. 228–242, Springer Berlin Heidelberg, 2013. 1, 2.2, 6.1

[12] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-Core Scheduling of Data Parallel Applications using SMT Solvers," in *Proceedings of the 2014 17th Euromicro Conference on Digital System Design*, DSD '14, Aug 2014. 1, 5.1, 5.4, 5.4.2

[13] P. Tendulkar, P. Poplavko, J. Maselbas, and O. Maler, "Pipelined Scheduling of Acyclic SDF Graphs using SMT Solvers," in *Proceedings of the 1st International Workshop on Investigating Dataflow in Embedded computing Architecture*, IDEA '15, Jan 2015. 1, 4.1, 5.1

[14] P. Tendulkar, P. Poplavko, and O. Maler, "Strictly periodic scheduling of acyclic synchronous dataflow graphs using smt solvers," Tech. Rep. TR-2014-5, Verimag Research Report, 2014. 1

[15] P. Tendulkar, *Mapping and Scheduling on Multi-core Processors using SMT Solvers*. Ph.D. thesis, Universite de Grenoble, Grenoble, France, Oct 2014. 1, 4.1, 5.4, 5.4.2

[16] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pp. 1–6, 2014. 1, 3.5

[17] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pp. 215–224, July 2013. 2

[18] S.-H. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele, "Multi-objective mapping optimization via problem decomposition for many-core systems," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, pp. 28–37, IEEE, 2012. 4.1

[19] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver.," in *TACAS* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008. 4.1

[20] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition ed., 2012. 5.4.3

[21] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, Oct. 2011. 2

[22] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton, "Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes," in *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, (New York, NY, USA), pp. 11–20, ACM, 2010. 6.2