Many-Core Scheduling of Data Parallel Applications using SMT Solvers

Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, Oded Maler
Verimag, Université Grenoble Alpes,
Centre Équation - 2, avenue de Vignate, 38610 Gières, France
Email: {pranav.tendulkar, petro.poplavko, Ioannis.Galanommatis, Oded.Maler}@imag.fr

***Abstract*—To program recently developed many-core systems-on-chip two traditionally separate performance optimization problems have to be solved together. Firstly, it is the parallel scheduling on a shared-memory multi-core system. Secondly, it is the co-scheduling of network communication and processor computation. This is because many-core systems are networks of multi-core clusters. In this paper, we demonstrate the applicability of modern constraint solvers to efficiently schedule parallel applications on many-cores and validate the results by running benchmarks on a real many-core platform.**

***Index Terms*—task graph, scheduling, multiprocessor, DMA**

I. Introduction

Optimal scheduling of application task graphs on multiprocessor platforms is a challenging task. Inherent complexity of optimal parallel scheduling problem is well-known [1]. However, *many-core platforms* are not only parallel, but also networked systems, where computation and communication workload should be properly distributed, balanced, and scheduled jointly. In this work we aim to address efficient deployment of parallel applications in this context. We focus on the signal processing applications like (I)DCT transformations, image and video decoding *etc.*, many of which can be represented as split-join graphs [2]. Split-join graphs are compact representation of task graphs with data parallelism.

In many-core platforms, a considerable number of cores is grouped together as clusters with local shared memory, plugged into an on-chip network with DMA (Direct Memory Access) transfers for networked communication between the clusters. The application deployment should satisfy different cost constraints like latency, maximum memory usage, maximum processor usage *etc.* It can be therefore considered a multi-criteria optimization problem. We search the solutions to such problems in the form of a Pareto set, representing the trade-offs between the cost constraints. To evaluate different cost trade-offs, we encode the cost feasibility problem as a set of combined logical/algebraic constraints that can be presented to SMT (SAT modulo theory) solvers.

Our approach to many-core scheduling is as follows. First, we use techniques similar to [3] to distribute the application tasks between the many-core clusters with balanced workload. Then, we perform optimal parallel scheduling on all clusters together, where the parallel intra-cluster computation tasks are scheduled jointly with inter-cluster communication tasks – the DMA transfers. The main topic of the paper is a detailed model of bounded-resource DMA communication and the corresponding method to let the SMT solver to jointly orchestrate parallel computations and multi-channel DMA transfers. The computed schedules for tasks and DMA transfers, are deployed using a run-time to execute real applications on a many-core platform with over 250 cores and 16 clusters.

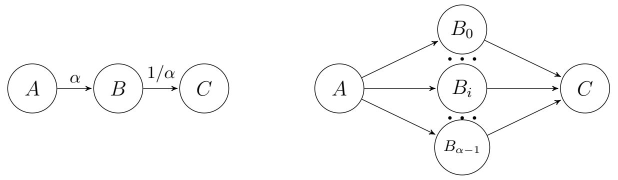


Fig. 1: A simple split-join graph and a task graph derived from it. Actor B has α instances.

The paper is organized as follows. In Sec. II we introduce the basic notions of the split-join and task graph. In Sec. III we introduce the many-core platform and the proposed inter-task communication scheme for split-join-graphs. Next, in Sec. IV we describe the design flow. In Sec. V we present the methodology to model the communication with bounded communication memory and network flow control. In the following section we present a model that represents joint scheduling of computation and communication (DMA) tasks. Then we show how that model is presented to an SMT solver in the form of constraints. Sec. VII presents the experiments with a set of application benchmarks. Sec. VIII concludes the paper and discusses the related and future work.

II. Split-Join Graphs

Definition 1 (Split-Join and Task Graphs): A split-join graph S is a tuple $S = (V, E, d, \alpha, \omega)$ 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 with no cyclic paths. The function $d : V \to \mathbb{R}_+$ defines the node delay, $\alpha : E \to \mathbb{Q}_+$ assigns a parallelization factor to every edge. An edge e can be either a *split*, *join* or *neutral*, depending on whether $\alpha(e) = a$, $1/a$ or $= 1$ for some integer $a > 1$, which counts the tasks at data split and join. $\omega(e)$ 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 if the edge is neutral. A split-join graph with $\alpha(e) = 1$ for every e is called a 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 task graph by data parallelism, explicitly represented by parallelization factors α. This is illustrated by the example in Fig. 1. 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 Fig. 1). A 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 task graph *tasks*. The edges of a split-join graph $e \in E$ are called *channels*, and those in a task graph $\epsilon \in \mathcal{E}$ are called *dependency arcs* or just dependencies.

We use the graph-theoretic models not only to model the tasks and dependencies of the application, but also those of the implementation. For this purpose, it is convenient to generalize the split-join graph model into a semantically richer model.

Definition 2 (Marked graph): The marked (split-join) graph S can be defined as a split-join graph extended by allowing cyclic paths and introducing an extra edge parameter – the marking: $m : E \rightarrow \mathbb{N}_{\geq 0}$. We assume that any split-join graph is a marked graph with zero marking.

A. The Semantics of Split-join and Marked Graphs

In split-join graphs, the tasks (*i.e.,* instances of an actor) can execute in parallel unless there are dependencies between them. In Fig. 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, a channel $e = (v, v')$ can be seen as a FIFO (first-in-first-out) buffer. The task instances of each actor v execute in a fixed order, which determines their index: v_0, v_1, v_2 *etc.*. 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 and the number of incoming arcs of the channel reader actor instances v'_r. We have:

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

The tokens have size ω bytes, and the amount of data produced by an instance of v and consumed by an instance of v' is:

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

Thus defined, the split-join graph can be viewed as a special case of SDF (synchronous dataflow) graphs [4], which consist of actors that write and read a fixed amount of data to FIFO channels. An important property of an SDF graph is its consistency [4], which requires that there must exist a positive integer function $c : V \rightarrow \mathbb{N}_{+}$ such that after executing $c(v)$ instances of all actors we have that at each channel the number of tokens produced is equal to the number of tokens consumed. The equations that express the consistency requirement are known as balance equations:

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

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

B. Derived Task Graph

The task graph derived from a split-join graph models one graph iteration.[1] For each actor v the task graph contains $c(v)$ tasks $\{v_0, v_1, \ldots v_{c(v)-1}\}$, *i.e.,* the *instances* of actor v.

Let us define the edges of the derived task graph. For a split-join channel (v, v') with $\alpha(v, v') = a/b$ let us consider the sequence of $a \cdot c(v)$ tokens produced in the FIFO buffer in one iteration. Let us number these tokens by index i in the order they are produced by the instances of actor $v : v_0, v_1, \ldots$. Obviously, token i is produced by task v_q where $q = \lfloor i/a \rfloor$. In sequential execution the actor instances consume tokens in the same order as they are produced (the FIFO order). Therefore, the first b tokens will be consumed by task v'_0, then the next b tokens by v'_1, etc. In general, token i is consumed by task v'_r where $r = \lfloor i/b \rfloor$. To model this *producer-consumer* dependency, the task graph should contain edge (v_q, v'_r).

A non-zero marking $m' = m(v, v')$ has the semantics of initial availability of m' tokens in the channel. Therefore, for the case of a marked graph in the above example we have to calculate r as $r = \lfloor (i + m')/b \rfloor$.

Definition 3 (Derived Task Graph): From a consistent marked split-join graph $S = (V, E, d, \alpha, \omega, m)$ we derive the task graph $T = (U, \mathcal{E}, \delta, \omega)$ as follows:

$$U = \{v_h \mid v \in V,\ 0 \leq h < c(v)\}$$

$$\mathcal{E} = \{(v_h, v'_{h'}) \mid (v, v') \in E \wedge \epsilon(v, v', h, h')\}$$

where ϵ is a predicate defined by:

$$\epsilon(v, v', h, h') \ : \ \exists\, i \in \mathbb{N} \ : \ h = \lfloor i/\alpha^{\uparrow}(v, v') \rfloor,$$
$$h' = \lfloor (i + m(v, v'))/\alpha^{\downarrow}(v, v') \rfloor,\ v_h, v'_h \in U$$

We also define that:

$$\forall (v_h, v'_{h'}) \in \mathcal{E} \quad \omega(v_h, v'_{h'}) = \omega(v, v'),$$
$$\forall v_h \in U \quad \delta(v_h) = d(v)$$

We use derived task graphs to define the scheduling constraints in Sec. VI.

III. The Many-core Platform

A. Many-core Hardware Architecture

Many-core processors can be seen as a hybrid of two simpler on-chip multiprocessor types. Those are multi-cores (shared-memory on-chip multiprocessors) and multiprocessor networks-on-chip. In many-cores, multiple multi-core clusters are networked on a chip. One of important usage scenarios is

[1]In SDF terminology, deriving a task graph is equivalent to deriving a homogeneous SDF graph.

where a host CPU runs the usual software stack and general-purpose tasks, whereas computationally expensive highly parallel kernels are forwarded to a many-core chip. MPPA [5] of Kalray, P2012 [6] of ST Micro, and SCC of Intel [7] are a few examples of many-core platforms.

In our experiments, we use Kalray MPPA. This platform contains 256 symmetrical cores. The cores are grouped in 16 compute clusters of 16 cores each. The clusters are connected by a network-on-chip (NoC) with a toroidal 2D topology.

Each cluster has 2MB of shared scratchpad memory. The cores have caches to access this memory, however, in our current work we disable them. A cluster also has a DMA control unit with 8 DMA channels. They are used for explicit data transfers to other clusters through the NoC. For efficiency, we use *asynchronous* transfers, *i.e.,* those that return the control back to the compute core before the transfer is completed. We characterize the delay $T(s)$ of a data transfer as per [8] as follows:

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

where I is the fixed initialization delay, s is the size of transfer in bytes, and g is the delay per byte, which depends on the throughput of the network communication links. It is important to note that during the time I both the compute core and the DMA channel are busy for setting up the transfer, whereas during the remaining time $G = g \cdot s$ only the DMA channel is still busy, pushing the data into the NoC, while the core can proceed to other tasks in parallel. DMA transfer completion status can be polled from any core inside the cluster. This operation is blocking until time $T(s)$ has elapsed plus some additional timing delay χ to return the control back to the thread waiting for transfer completion.

Definition 4 (Architecture Model): An architecture model A is a tuple (X, H, M, D, I, g, χ), where X is a set of clusters, $H \subseteq X \times X$ is a set of communication paths between pairs of clusters, whereas $|(x, x')|_H$ gives the topological distance between any two clusters. M and D are the number of cores and DMA channels per cluster respectively. $I, g, \chi \in \mathbb{R}_+$ are communication delay components introduced earlier.

B. Support of Communication Buffers

In the split-join graph application model, the actors communicate via channels. Given the limits of memory size of the platform, the channels should use a bounded memory. Moreover, to exploit the data parallelism of the split-join channels, it should be possible to execute different concurrent instances of channel writer and reader on different cores.

We created a software 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 data token place in the buffer is equipped by an individual *status record*, which can be updated independently and which indicates whether the given token is ready to be written or read. For a buffer of limited size we use standard circular-buffer wrap-around mechanism, with a wrap-around counter in each status record.

The communication protocol is more complex when the writer and reader actors are located in different clusters. In this case we split the FIFO buffer into *writer* and *reader sub-buffers*, the latter being located in the remote cluster. The data is copied from the writer sub-buffer to the reader by a DMA transfer. Along with this data transfer, also the *'writing finished'* status record update is sent for the reader sub-buffer. For its part, the reader sends the *'reading finished'* status update for the same sub-buffer back to the writer. This allows the writer, prior to transferring another data token through DMA, to safely estimate whether a remote buffer place is ready for this token. This backward signalling of buffer space availability is usually referred to as *flow control*. More details about DMA communication become apparent in later sections.

IV. Design Flow

Given a many-core architecture, the problem is to map and schedule an application model, represented by a split-join graph called *application graph* and denoted S^M=(V^M,E^M,d,α,ω).

The design flow is based on multi-criteria optimization, whereby we repeatedly query the SMT solver for solving the problem at different points in the multi-dimensional cost space, to approximate the Pareto front in a manner of a 'multi-dimensional binary search', as proposed in [3].

In general, combined scheduling, mapping and buffer allocation inside a shared-memory cluster is hard if one tries to solve it exactly using the SMT solvers, but one can still approximate the optimal solutions and bound the approximation distance, see *e.g.,* [2], [3]. To tackle the problem complexity, we solve it in three steps: *(i) partitioning, (ii) placement*, and *(iii) scheduling*. Thus the solutions found are optimal only locally, in the context of each step. Nevertheless, in the third step we combine several communication and computation decisions together to have a jointly optimal result.

In this section we briefly sketch the design flow as a whole, whereas in the rest of the paper we focus on the methodology of the scheduling step, the core contribution of this paper.

The *partitioning* step defines the groups of actors to be assigned to the same many-core cluster. We adapt the partitioning scheme of [3] to the split-join graph model.
Partitioning Problem Instance:
Application graph S^M=(V^M,E^M,d,α,ω) and the costs:

- C_τ : Maximum computation workload per partition
- C_η : Estimated total communication cost
- C_z : Number of partitions

The computation and communication workload are calculated as the sum of task execution delays or data token sizes in the derived task graph respectively. A channel contributes to the communication cost only if the channel writer and reader are assigned to different partitions.

Partitioning Problem Solution $z : V^M \to \mathbb{N}_{\geq 0}$
where $z(v)$ identifies the partition of actor v.

The approximate Pareto points in three-dimensional cost space obtained at the this step are propagated to placement and

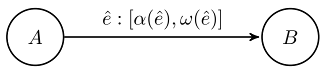


Fig. 2: Communicating Tasks

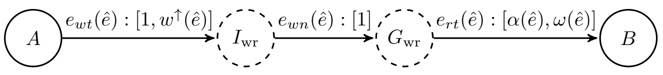

A
$e_{wt}(\hat{e}):[1,w^{\uparrow}(\hat{e})]$
I_{wr}
$e_{wn}(\hat{e}):[1]$
G_{wr}
$e_{rt}(\hat{e}):[\alpha(\hat{e}),\omega(\hat{e})]$
B

Fig. 3: Partition Aware Communicating Tasks

scheduling steps. The latter two are considered independent, as we currently ignore the network routing and contentions.

The *placement step* allocates a unique physical cluster to every partition. The purpose is to map the pairs of communicating partitions to clusters x, x' located as close as possible, to minimize the *total communication cost* according to metric $s \cdot |x, x'|_H$, where s is the communication data size. The SMT solver constraints at this step are again similar to those in [3].

In the *scheduling step* we perform a joint scheduling of the computation tasks (*i.e.,* the tasks of the application graph) and the communication tasks (*i.e.,* the DMA transfers). In this paper we first describe a *model* that captures the interactions between computation and communication tasks. Then we explain how this model is *encoded* in terms of SMT solver constraints. The model is derived from a series of *graph transformations*, which gradually change the application graph into a final *schedule graph*.

V. Modeling Communication

A. Partition Aware Graph

For defining the graph transformations, in particular for adding new actors, it is convenient to introduce notation $v : [d(v), z(v)]$, which represents an actor v with delay $d(v)$ and partition number $z(v)$. Similarly, $e : [\alpha(e), \omega(e), m(e)]$ represents an edge e with parallelization factor $\alpha(e)$, token size $\omega(e)$ and marking $m(e)$. The latter two parameters are sometimes omitted, the default values being zero. Note that if ω is zero, the given edge models an *execution order constraint* and does not involve a memory buffer for communication.

We assume to have a ready *partitioning solution* $z(v)$, calculated earlier in the design flow. In order to model the communication delay, we need to introduce additional actors in the split-join graph, representing the DMA transfers. Recall from Equality (2) that a DMA transfer consists of two phases: the initialization, modeled here by actor I_{wr}, and the network communication, modeled by actor G_{wr}. Fig. 2 shows an application graph channel $\hat{e} = (A, B)$ and Fig. 3 shows its partition aware graph for the case where actors A and B are assigned to different partitions. Recall that in this case the channel (A, B) is split into writer and reader sub-buffers. Edge $e_{wt}(\hat{e})$ models the writer sub-buffer; $\alpha = 1$ indicating that one writer instance is followed by exactly one data transfer, whereas $\omega(e_{wt}) = w^{\uparrow}$ indicates that all $\alpha^{\uparrow}$ tokens produced by an instance of the writer actor are encapsulated into a compound token. Edge $e_{wn}(\hat{e})$ reflects the sequential order between the two DMA phases. Edge $e_{rt}(\hat{e})$ corresponds to the reader sub-buffer. It has the same parameters as the original edge $\hat{e}$.

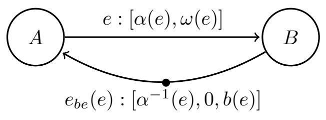

A
$e:[\alpha(e),\omega(e)]$
B
$e_{be}(e):[\alpha^{-1}(e),0,b(e)]$

Fig. 4: Buffer aware graph model for a channel without DMA

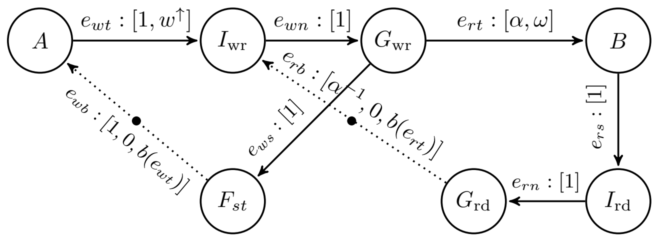


Fig. 5: Buffer aware graph model for a channel with DMA

Let $E^M_{\Delta Z}$ denote the channels crossing the partition boundaries and $E^M_{\overline{\Delta Z}}$ its complement. Formally,

$$\begin{aligned} E^M_{\Delta Z} &= \{(v, v') \in E^M \mid z(v) \neq z(v')\} \\ E^M_{\overline{\Delta Z}} &= E^M \setminus E^M_{\Delta Z} \end{aligned}$$

Definition 5 (Partition Aware Graph): A partition aware graph $S^P = (V^P, E^P, d, \alpha, \omega)$ is a split-join graph obtained by replacement of application graph edges $E^M_{\Delta Z}$ by a subgraph with new actors and edges as defined below.

The delay of the new actors is determined by the DMA initialization time I and the network sending time $g \cdot w^{\uparrow}(\hat{e})$.

$$V^P = V^M \cup \{I_{\text{wr}}(\hat{e}) : [I, z(v)],$$
$$G_{\text{wr}}(\hat{e}) : [g \cdot w^{\uparrow}(\hat{e}), z(v)] \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

The edges of partition aware graph are given by:

$$E^P = E^M_{\overline{\Delta Z}} \cup \{e_{wt}(\hat{e}), e_{wn}(\hat{e}), e_{rt}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Edge & Parameters	Connections
$e_{wt}(\hat{e}) : [1, w^{\uparrow}(\hat{e})]$	$e_{wt}(\hat{e}) = (v, I_{\text{wr}}(\hat{e}))$
$e_{wn}(\hat{e}) : [1]$	$e_{wn}(\hat{e}) = (I_{\text{wr}}(\hat{e}), G_{\text{wr}}(\hat{e}))$
$e_{rt}(\hat{e}) : [\alpha(\hat{e}), \omega(\hat{e})]$	$e_{rt}(\hat{e}) = (G_{\text{wr}}(\hat{e}), v')$

B. Buffer Aware Graph

After we obtain a partition aware graph, the next graph transformation is performed in order to model the bounded buffer memory allocated to the channels.

Definition 6 (Buffer Allocation): $b : E^P \to \mathbb{N}_+$ defines the bounded capacity assigned the channels of the partition-aware graph. It is part of the solution of the combined multiprocessor scheduling and buffer allocation problem.

Definition 7 (Buffer Aware Graph): is a cyclic marked graph $S^B = (V^B, E^B, d, \alpha, \omega, m)$ obtained from the partition-aware graph by adding marked channels that model the allocated buffer capacity and new actors that model the DMA polling and flow control.

In this graph, for each intra-cluster channel $\hat{e}$ we add a new *backward channel* – in the opposite direction, marked with the buffer allocation $b(\hat{e})$, modeling the 'communication' of the available space in the channel. The backward channel is the inversion of $\hat{e}$, with inversely proportional parallelization factor, see Fig. 4.

Recall that the marking models the number of tokens present in the channel at the start of the execution. In the backward channel the marking $b(\hat{e})$ indicates the free space that is available initially. At every execution, the writer takes $\alpha^{\uparrow}$ tokens of 'space' and produces $\alpha^{\uparrow}$ tokens of data, and the reader takes $\alpha^{\downarrow}$ tokens of data and produces $\alpha^{\downarrow}$ tokens of 'space'.[2]

If the channel is an inter-cluster channel, involving DMA, then we model the bounded space of the writer sub-buffer, e_{wt}, and the reader sub-buffer, e_{rt}, by separate backward channels. We also add additional actors, see Fig. 5.

Consider the writer sub-buffer, e_{wt}, serving as an input to the DMA transfer. The space in this sub-buffer becomes available when the transfer is completed. Recall that detecting the transfer completion costs processor time, this is modelled by new actor F_{st}. This actor produces the 'space' tokens on the backward channel of e_{wt}. Recall that for flow control the free space availability status of the reader sub-buffer, e_{rt}, has to be communicated back to the writer. The respective DMA transfer is modeled by I_{rd} and G_{rd}, and the latter feeds the 'space' tokens to the backward channel of e_{rt}.

Consider an example. Suppose that we use a double-buffering approach, such that $b(e_{wt}(A,B)) = b(e_{rt}(A,B)) = 2$ tokens. Let us simulate the execution of the model in Fig. 5 unfolded for three actor instances: A_0, A_1, A_2 – see Fig. 6.

- A_0 executes and consumes one space token.
- Upon completion, A_0 triggers a DMA transfer initialization, modeled by I_{wr0}
- Data is sent to the network, which is represented by G_{wr0}. In parallel, A_1 picks the second space token and triggers another DMA transfer: I_{wr1} and G_{wr1}.
- When G_{wr0} finishes, B_0 receives the input data tokens. At the end it releases the space occupied by these tokens and triggers a DMA transfer to update the writer accordingly. This transfer is modeled by I_{rd0} and G_{rd0}.
- A_2 waits execution for task F_{st0} to finish, which releases the necessary space token in the backward channel.
- After A_2, we can start transferring its output by I_{wr2} when we receive the (flow-control) space token via G_{rd0}

Below we formalize the new actors and channels:

$$V^B = V^P \cup \{F_{st}(\hat{e}) : [\chi, z(v)], \quad I_{\mathrm{rd}}(\hat{e}) : [I, z(v')],$$
$$G_{\mathrm{rd}}(\hat{e}) : [\alpha^{\downarrow}(\hat{e}) \cdot \omega_0 \cdot g, z(v')] \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

The delay of network sending node $\alpha^{\downarrow} \cdot \omega_0 \cdot g$ corresponds to the delay of sending ω_0 bytes of status record for all $\alpha^{\downarrow}$ tokens read in the channel, where ω_0 depends on the implementation.

$$E^B = E^P \cup \{e_{ws}(\hat{e}), e_{wb}(\hat{e}), e_{rs}(\hat{e}),$$
$$e_{rn}(\hat{e}), e_{rb}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\} \cup$$
$$\{e_{be}(\hat{e}) : [\alpha^{-1}(\hat{e}), 0, b(\hat{e})] \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Edge & Parameters	Connections
$e_{ws}(\hat{e}) : [1]$	$e_{ws}(\hat{e}) = (I_{\mathrm{wr}}(\hat{e}), F_{st}(\hat{e}))$
$e_{wb}(\hat{e}) : [1, 0, b(e_{wt}(\hat{e}))]$	$e_{wb}(\hat{e}) = (F_{st}(\hat{e}), v)$
$e_{rs}(\hat{e}) : [1]$	$e_{rs}(\hat{e}) = (v', I_{\mathrm{rd}}(\hat{e}))$
$e_{rn}(\hat{e}) : [1]$	$e_{rn}(\hat{e}) = (I_{\mathrm{rd}}(\hat{e}), G_{\mathrm{rd}}(\hat{e}))$
$e_{rb}(\hat{e}) : [\alpha^{-1}(\hat{e}), 0, b(e_{rt}(\hat{e}))]$	$e_{rb}(\hat{e}) = (G_{\mathrm{rd}}(\hat{e}), I_{\mathrm{wr}}(\hat{e}))$

[2]Note that if we derive a task graph from a buffer aware graph then its structure will depend on the buffer sizes such that a smaller buffer size can only increase the critical path delay (a buffer-latency trade-off).

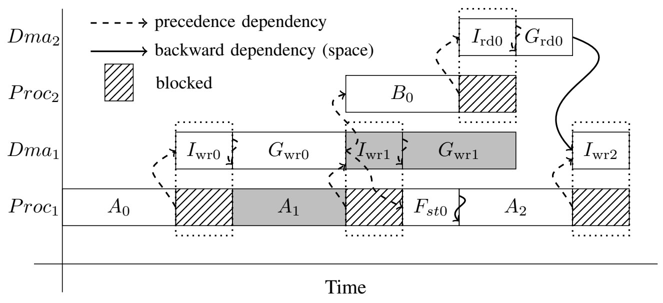


Fig. 6: Double buffering schedule example

C. Communication Aware Graph

In our implementation the compute core remains busy until the completion of the DMA initialization tasks (I_{wr} or I_{rd}) that follow after the corresponding computation actor (writer or reader, resp.). Moreover, if the computation actor instance starts multiple DMA transfers (as a writer/reader of multiple channels), then the next DMA transfer cannot start until the initialization of the previous transfer has finished. This restriction can be modeled by adding extra edges between the actors modeling the DMA initialization phase, to enforce a sequential execution order in the schedule.

Definition 8 (Communication Aware Graph): Communication Aware Graph $S^K = (V^K, E^K, d, \alpha, \omega, m)$ is obtained from the buffer aware graph S^B by adding the edges to model the ordering of the DMA transfers with regard of their transfer initialization phase.

The set of actors of the communication aware graph is the same as in the buffer aware graph $V^K = V^B$. To define the new edges, we first introduce function $I(v)$, representing an ordered set of the transfer initialization actors connected to the output of given actor v, *i.e.,*

$$I(v) = \{v' \mid (v, v') \in E^B \wedge$$
$$\exists\ \hat{e} \in E^M : v' = I_{\mathrm{wr}}(\hat{e}) \vee v' = I_{\mathrm{rd}}(\hat{e})\}$$

We select the order $(I_1, I_2, \ldots \mid I_k \in I(v))$ arbitrarily, ensuring that the implementation and the model follow the same order. We have:

$$E^K = E^B \cup \{(I_i, I_{i+1}) : [1] \mid I_i, I_{i+1} \in I(v),\ v \in V^B\}$$

The final transformation to model the communication is the derivation of the task graph $T^K = (U^K, \mathcal{E}^K, \delta, \omega)$ from the split-join graph S^K. This is done according to Def. 3, but in addition we require that the tasks inherit the partitioning from their actors: $\forall v_h \in U^K\ z(v_h) = z(v)$.

Having shown our model for buffered DMA communication, we proceed to the scheduling model in the next section.

VI. Scheduling

A. Schedule Graph

The schedule graph $T^S = (U^S, \mathcal{E}^S, \delta, \omega)$ is obtained from computation aware task graph T^K by adding the *mutual exclusion edges*, according to the given *schedule s* and *intra-cluster mapping μ*, where:

- $\mu : U^S \rightarrow \mathbb{N}_{\geq 0}$ maps every task to a core (for computation tasks) or a DMA channel (for transfer tasks);

$s : U^S \to \mathbb{R}_{\geq 0}$ associates each task with a start time.

The buffer allocation, b, the schedule and the mapping should be calculated by the solver in the scheduling step of the design flow, as explained later.

Let us recall and introduce some notations.

- $U^S = U^S_T \cup U^S_C$ is the task partitioning into:
  - *transfer tasks* U^S_T, derived from DMA transfer actors: I_{wr}, G_{wr} I_{rd}, and G_{rd}.
  - *computation tasks* U^S_C
- $\mathcal{I}(u)$ are the transfer tasks connected to the output of task u, by analogy to $I(v)$ of the computation-aware graph.
- U^S_{C+} is the set of computation tasks connected to DMA transfers: $U^S_{C+} = \{u \in U^S_C \mid \mathcal{I}(u) \neq \emptyset\}$
- $U^S_{C\emptyset}$ is the set of the remaining computation tasks, *i.e.,* those with no DMA transfer tasks at the output.

Note that not all tasks introduced for communication are transfer tasks. In fact, F_{st} (see Fig. 5) are computation tasks, as they are executed on the compute cores.

Due to a limited number of the compute cores and DMA channels multiple tasks are mapped to the same core or channel, and should therefore be executed sequentially. This requirement is modeled by adding mutual exclusion edges. The edges of the schedule graph are therefore defined by:

$$\mathcal{E}^S = \mathcal{E}^K \cup \mathcal{E}^\mu_T \cup \mathcal{E}^\mu_{C\emptyset} \cup \mathcal{E}^\mu_{C+}$$

$\mathcal{E}^\mu_T$ are the mutual exclusion edges for the transfer tasks mapped at the same DMA channel.

$$\mathcal{E}^\mu_T = \{(u,u') : [1] \mid u, u' \in U^S_T,\\ z(u) = z(u'), \mu(u) = \mu(u'),\ s(u') \geq s(u)\}$$

Similarly, we insert an edge for the computation tasks mapped on the same core, first for the case of no DMA transfers:

$$\mathcal{E}^\mu_{C\emptyset} = \{(u,u') : [1] \mid u \in U^S_{C\emptyset}, u' \in U^S_C\\ z(u) = z(u'), \mu(u) = \mu(u'),\ s(u') \geq s(u)\}$$

Finally, given task u that starts some DMA transfers, let $\mathcal{I}_{\max}(u)$ represent the last transfer in the ordered set ($\mathcal{I}_1(u)$, $\mathcal{I}_2(u)$, $\ldots$). The compute core becomes available to a later task u' only when the last transfer initialization is completed:

$$\mathcal{E}^\mu_{C+} = \{(\mathcal{I}_{\max}(u), u') : [1] \mid u \in U^S_{C+}, u' \in U^S_C,\\ z(u) = z(u'), \mu(u) = \mu(u'),\ s(u') \geq s(u)\}$$

B. Scheduling Problem on SMT solver

In the previous sections we described a sequence of rules to derive a schedule graph, which models the timing effect of a scheduling/buffering solution calculated by the SMT solver. The solver constraints for this problem are therefore obtained directly from the schedule graph. Below we present the corresponding definition of the optimization problem dealt with at the scheduling step of the design flow.

Problem Instance:

- partition-aware graph S^P
- hardware architecture model A,
- costs for multi-criteria optimization:
  - C_B : maximal buffer memory size per cluster
  - C_L : schedule latency constraint

Solution: (T^S, b, μ, s), where $T^S = (U^S, \mathcal{E}^S, \delta, \omega)$ is the schedule graph obtained from S^P by adding extra edges based on buffering b, mapping μ and scheduling s.

Schedule constraints

The schedule should respect all the dependencies, including the application dependencies, bounded buffer space, DMA transfer ordering, and mutual exclusion, all represented by dependency arcs in the schedule graph:

$$\bigwedge_{(u,u') \in\ \mathcal{E}^S(b,s,\mu)} s(u') \geq s(u) + \delta(u)$$

As we explicitly indicate here, the set of schedule dependencies is function of the problem solution. However, the SMT solvers require a static set of constraints. Therefore, observing that the set U^S is static we rewrite the constraints as:

$$\bigwedge_{u,u' \in U^S} \epsilon^S(u,u',\mu,s,b) \implies s(u') \geq s(u) + \delta(u)$$

where ϵ^S is a predicate asserting the presence of dependency arc (u,u'). For mutual exclusion arcs this predicate can be trivially obtained from the definition of sets $\mathcal{E}^\mu$. For backward edges it can be obtained from $\epsilon(v,v',h,h')$ in Def. 3, by substituting the buffer allocation to marking m. However, we did not yet implement in the SMT constraints for the backward edges, therefore we replaced them by equivalent constraints proposed in [2], slightly adapted to the channels with DMA.

Resource constraints

Bounded number of DMA channels and cores per cluster.

$$\bigwedge_{u \in U^S_T} \mu(u) < |D| \quad \wedge \quad \bigwedge_{u \in U^S_C} \mu(u) < |M|$$

Bounded buffer memory per cluster:

$$\bigwedge_{z' \in Z} \quad \sum_{e=(v,v') \in E^P : z(v')=z'} b(e) \cdot \omega(e) \quad \leq \ C_B$$

Latency constraint

$$\bigwedge_{u \in U^S} s(u) + \delta(u) \leq C_L$$

Extra constraints

In our implementation we require the writer and reader sub-buffers to have equal buffer memory:

$$\bigwedge_{\hat{e} \in E^M_{\Delta Z}} b(e_{wt}(\hat{e})) \cdot \omega(e_{wt}(\hat{e})) = b(e_{rt}(\hat{e})) \cdot \omega(e_{rt}(\hat{e}))$$

The network phase of a DMA transfer should run immediately after the initialization phase on the same channel:

$$\bigwedge_{\hat{e}=(v,v') \in E^M_{\Delta Z}} \ \bigwedge_{0 \leq h < c(v)} \begin{array}{c} s(I_{\text{wr}\,h}(\hat{e})) + I = s(G_{\text{wr}\,h}(\hat{e})) \\ \wedge \\ \mu(I_{\text{wr}\,h}(\hat{e})) + I = \mu(G_{\text{wr}\,h}(\hat{e})) \end{array}$$

and:

$$\bigwedge_{\hat{e}=(v,v') \in E^M_{\Delta Z}} \ \bigwedge_{0 \leq h' < c(v')} \begin{array}{c} s(I_{\text{rd}\,h'}(\hat{e})) + I = s(G_{\text{rd}\,h'}(\hat{e})) \\ \wedge \\ \mu(I_{\text{rd}\,h'}(\hat{e})) + I = \mu(G_{\text{rd}\,h'}(\hat{e})) \end{array}$$

where h and h' are indices of the task instances of channel writer and reader.

Last but not the least, we add *task and processor symmetry breaking* constraints, which improve the performance of the constraint solvers [2].

TABLE I: Application Benchmarks

Benchmark	#Actors	#Channels	#Tasks	Total Exec. Time (cycles)	Total Comm. Data (bytes)
JpegDecoder	3	2	25	934288	12384
BeamFormer	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
DctCoarse	3	2	3	74401	512
DctFine	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

$$\begin{tikzcd} vld \arrow[r, "{e:[12,268]}"] & iq/idct \arrow[r, "{e:[1,76]}"] & color \end{tikzcd}$$

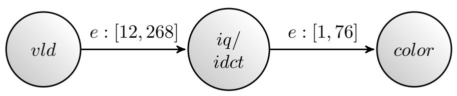


Fig. 7: JPEG Decoder Application Graph

We also found that the solutions produced by the SMT solver can profit from a post-processing. According to our constraints, the solver is allowed to schedule the tasks later than the earliest time when they are enabled, often resulting in 'lazy' schedules. Attempts to add extra constraints to ensure non-lazy schedules have resulted in unacceptable increase in the solver calculation times. Therefore, instead, after each successful round of solving we modify the latency cost C_L by setting it to the critical path delay in the schedule graph. In the context of our cost space exploration method this sometimes yields Pareto points with improved C_L.

VII. Experiments

In this section we give an empiric evaluation of the validity and limitations of our many-core scheduling approach using a set of application benchmarks.

Our application benchmarks consist of the JPEG Image Decoder [2] and a subset of StreamIt benchmarks [9]. The characteristics of the benchmarks are summarized in Tab. I.

The exploration experiments use version 4.1 of the Z3 Solver [10] running on a Linux machine with Intel Core i7 processor at 1.73 GHz with 4 GB of memory. We use a global time-out which stops a cost space exploration in 20 minutes, while every SMT query is given a local timeout of 3 minutes, after which the exploration proceeds to the next query [2].

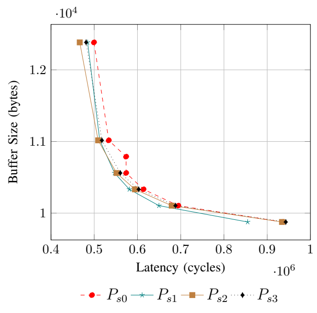
We use the JPEG decoder as an example to illustrate the experiment done for each benchmark. The application graph is shown in Fig. 7, it has three actors: VLD (variable length decoding), IQ-IDCT (inverse quantization and inverse discrete cosine transform) and color conversion. The parallelization factor 12 is set for a particular image size.

For partitioning step, the exploration is done in a 3-dimensional cost space: (C_τ, C_η, C_z) *i.e.,* the workload per partition, communication cost and the partition count. There is a trade-off between these costs, and our grid-based exploration strategy [2] finds four Pareto points, see Table II.

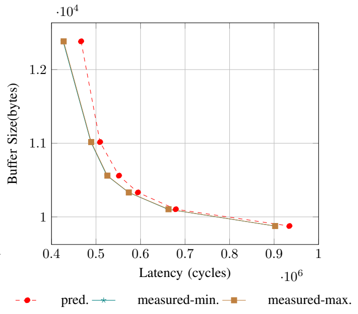
At the placement step, the partitions are mapped to neighbor clusters. In the scheduling step, the exploration is done for

TABLE II: Partitioning Pareto Points of JPEG decoder

Solution	Allocated group			Exploration Cost		
	vld	iq	color	C_τ	C_η	C_z
P_{s0}	0	1	2	424012	12384	3
P_{s1}	0	0	1	758116	2736	2
P_{s2}	0	0	0	934288	0	1
P_{s3}	0	1	1	510276	9648	2



(a) 25 scheduling solutions for 4 partitioning solutions



(b) sched. solutions for P_{s2} measured on Kalray platform

Fig. 8: JPEG Decoder : solutions and their measurements

each partitioning solution in a 2-dimensional space (C_L, C_B), *i.e.,* latency and maximal buffer size per partition. We plot all four Pareto fronts in Fig. 8a. We observe that some Pareto fronts cross. This is because they differ in the number of available partitions, which leads to the following effect. On the top-left side of the cost diagram a large buffer memory per partition is allowed. Therefore, even a single-partition solution has enough buffer memory to minimize the latency, and it dominates the solutions with more partitions because it does not use any DMA transfers. On the bottom-right side, the parallelism available to the solutions with less partition count is restricted by small amount of buffer memory, whereas adding more partitions results in a larger total buffer memory, allowing to run more tasks in parallel and get a better latency. The combination of these individual Pareto fronts can be pruned, retaining only the overall Pareto solutions. These solutions will be a mix of multi-cluster and single-cluster ones, showing interesting deployment trade-offs.

A scheduling solution is represented by (b, μ, s), giving the buffer allocation, the core mapping and the scheduling. The solution is interpreted by our runtime environment, together with the cluster placement of actors. The environment allocates the buffer memory and orchestrates the clusters, cores, and DMA channels according to the given solution, executing the tasks and transfers in a self-timed manner, while preserving the same order per core and per channel as in the schedule s. This is in line with a common practice for DSP multi-cores. While executing the application, we measure its real latency on the platform, collecting statistics over 100 repeated iterations and compare it with the model-predicted latency.

Fig. 8b shows the results for one of the partitioning solutions. We plot the minimum and maximum observed latency on the platform as well as the predicted one. The maximum error that was observed in this configuration, as well as for the entire JPEG experiment, was 8%.

Fig. 9 shows a summary of results for all benchmarks.

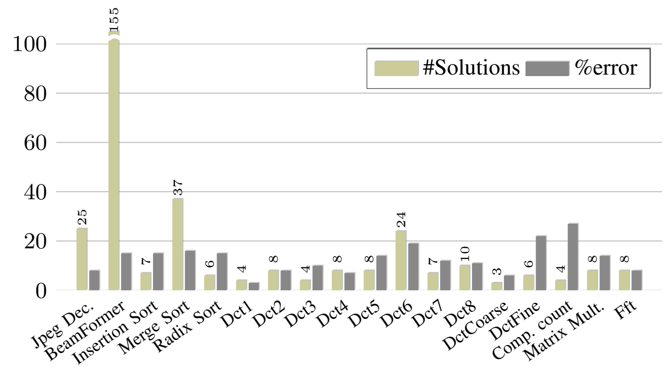


Fig. 9: Application benchmarks: summary of results

Firstly, we have plotted the total number of scheduling solutions obtained for given benchmark, adding up those obtained for different partitioning solutions. Note that the solution count depends on the amount of parallelism available in the application as well as on the structure of the application graph.

Secondly, we plot the maximum error of the predicted latency vs the one measured on the platform for 100 iterations. The overall maximum error (27%) was observed in the 'Comparison Count' benchmark. In the schedule that resulted in this error we observed that there were 8 simultaneous DMA transfers between a pair of clusters, which contributed to network contention and hence a less accurate latency prediction. We believe that another factor contributing to inaccuracy is the contention on the shared memory bus inside the clusters. We also believe that the error could be larger if we ran with enabled cache, a subject for future work.

VIII. Discussion and Related Work

The contribution of this paper can be summarized as follows. We demonstrate a compile-time scheduling framework for data-parallel applications on many-core platforms, *i.e.,* networks of multi-core systems. Our application model is a sub-class of popular synchronous dataflow (SDF) graphs. We present an accurate model of buffered communication channels that support multiple parallel writers and readers and involve network DMA transfers with flow control. The core of our contribution is a method to employ such models in constraint based solvers for combined scheduling of computation and communication. We validate the method using a dozen of real applications from a well-known StreamIt set of signal-processing benchmarks. For the optimal schedules generated for benchmarks, we performed latency measurements on real many-core hardware. Despite the fact that we ignored the network contention in scheduling, the maximal error of scheduler's timing estimation was only 27%, exploiting non-cached shared local memory system at the cluster level. We plan to support network routing and contention analysis, as well as caching, which would reduce the memory bus contention but possibly introduce unpredictable cache delays.

Our scheduling approach is based on approximation of Pareto points in buffer-latency cost space using an SMT solver. We reused the symmetry breaking proposed in [2] to ensure as high solver performance as we can, for a closer approximation of the exact solutions and for managing larger problem sizes. Still, the maximal benchmark size we could handle well had 96 tasks, where the solver's performance started to saturate. In future we intend to investigate methods to improve the scalability to larger problem sizes.

The work [11] proposes implementation-aware graphs similar to our schedule graphs. However, the processor blocking for initializing the DMA was not modeled and model validation on real hardware was not done. Moreover, this and some other related works employ detailed models of buffered communication only for validation or refinement of a pre-computed multiprocessor mapping solution. The work [12] demonstrates scheduling with memory constraints. Interestingly, their DMA model turns out to be coherent to ours. They also used an SMT solver for calculating the schedules. However, that work is restricted to a single-processor system.

Many previous works use constraint solvers to solve related problems. The work [13] presents a task graph scheduling methodology for SMT solvers. They propose an interesting procedure of providing a problem-specific feedback to the SAT solving engine for better efficiency. However, their work does not consider buffer allocation and DMA communication. Several other works apply constraint solving to schedule SDF graphs on multiprocessors. The advantage of [14] is that they consider the network routing and pipelined scheduling. However, they do not combine their the network communication model with parallel scheduling in shared-memory clusters. Pipelined scheduling and handling the network interference are future work subjects for us.

References

[1] Y.-K. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998*.

[2] P. Tendulkar, P. Poplavko, and O. Maler, "Symmetry breaking for multi-criteria mapping and scheduling on multicores," in *FORMATS*, 2013.

[3] S. Cotton *et al.*, "Multi-criteria optimization for mapping programs to multi-processors," in *SIES*, 2011.

[4] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75.

[5] B. de Dinechin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *HPEC, 2013*.

[6] D. Melpignano *et al.*, "Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications," in *DAC*, 2012.

[7] J. Howard *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *ISSCC, 2010*.

[8] S. Saidi *et al.*, "Optimizing explicit data transfers for data parallel applications on the cell architecture," *ACM TACO*, 2012.

[9] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[10] L. Moura and N. Bjorner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[11] P. Poplavko *et al.*, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *CASES*, 2003.

[12] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *ECRTS, 2013*.

[13] W. Liu *et al.*, "Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, 2011.

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