Odyn: Deadlock Prevention and Hybrid Scheduling Algorithm for Real-Time Dataflow Applications

Benjamin Dauphin, Andrea Enrici, Ludovic Apvrille, Renaud Pacalet

Odyn: Deadlock Prevention and Hybrid Scheduling Algorithm for Real-Time Dataflow Applications

Benjamin Dauphin*, Renaud Pacalet*, Andrea Enrici†, Ludovic Apvrille*
*LTCI, Télécom Paris, Institut polytechnique de Paris, France
firstname.lastname@telecom-paris.fr
†Nokia Bell Labs France, Centre de Villarceaux, 91620 Nozay, France
firstname.lastname@nokia-bell-labs.com

***Abstract*—In recent wireless communication standards (4G, 5G), the growing need for dynamic adjustments of transmission parameters (e.g., modulation, bandwidth, channel coding rate) makes traditional static scheduling approaches less and less efficient. The reason being that precomputed fixed mapping and scheduling prevent the system from dynamically adapting to changes of the operating conditions (e.g. wireless channel quality, available bandwidth).**
In this paper, we present Odyn, a hybrid approach for the scheduling and memory management of periodic dataflow applications on parallel, heterogeneous, Non-Uniform Memory Architecture (NUMA) platforms. In Odyn, the ordering of tasks and memory allocation are distributed and computed simultaneously at run-time for each Processing Element. Odyn also proposes a mechanism to prevent deadlocks caused by attempts to allocate buffers in size-limited memories. This technique, based on the static computation of exclusion relations among buffers in a target application, removes the need for backtracking that is typical of dynamic scheduling algorithms.
We demonstrate the effectiveness of Odyn on a testbench that simulates the interactions of randomly generated concurrent applications. We also demonstrate its deadlock prevention technique on a selection of use cases.

***Index Terms*—Dynamic scheduling, Embbeded systems, Deadlock prevention**

I. Introduction

Real-time dataflow applications with high processing power requirements are at the heart of many fields, from the Digital Signal Processing (DSP) in wireless communications to images or video analysis in autonomous driving. For performance reasons these applications are frequently deployed on heterogeneous platforms embedding General Purpose Processors (GPP), DSP processors, Graphics Processing Units (GPU), dedicated hardware accelerators, etc. To accommodate the very large memory bandwidth and short memory latency requirements of these applications, these platforms frequently adopt a Non-Uniform Memory Architecture (NUMA) where the memory resource is distributed over multiple, size-limited, physical memories. The drawback of NUMA is usually that the computing nodes can access only a limited sub-set of physical memories. Here, the input (output) data of *tasks*[1] must be stored before (after) tasks start executing.

In this paper the platforms that we target are typical of the ones used for wireless communications. They are heterogeneous and NUMA. GPPs are used for the control tasks while dedicated execution units (e.g., DSP, GPU, FPGA, hardware accelerators) are responsible for the DSP tasks. All these computing nodes are interconnected by communication resources (e.g., hierarchical buses, crossbars). Each execution unit is composed of a Processing Element (PE) and a local (scratchpad) memory. A PE can be general purpose (DSP) or very specialized (hardware accelerator). All data processed by the execution node are stored in the local memory. These local memories are not shared among PEs to prevent contention and guarantee deterministic access times during processing. Large shared memories, such as the ones used by the control GPPs, are not directly accessed by PEs. They are used as general purpose storage facilities and data blocks are transferred between shared memories and local private memories when needed.

In this context the search of the *(near) optimal mapping and scheduling*[2] is a well-known challenging problem. The mapping consists in allocating tasks to computation nodes, data buffers to physical memories and data transfers to physical links. It must fulfill all requirements imposed by the applications and the platform (e.g., compatibility between the nature of tasks and computing nodes, between computing nodes and physical memories). The scheduling consists in ordering the execution of all tasks and data transfers such that the data dependencies of the applications are satisfied, the real-time constraints are met and deadlocks are avoided.

Traditionally this mapping and scheduling challenge has been tackled by static (offline) analyzers [1]. The main advantages of this approach are the high processing power and time that can be spent offline to explore solutions, the possibility to prove the functional and real-time correctness of the selected solution, and the low computation load at run-time for the control tasks: as the mapping and the scheduling are computed offline, they do not have to be recomputed online but "simply" to be applied. The main drawback is the lack of flexibility at run-time: with a precomputed fixed mapping and scheduling it is difficult to adapt the running system to changes of

[1]In this paper we use task as a synonym of data processing operation, not in the sense of operating system task.

[2]Depending on the specific case, optimality can be defined in terms of power consumption, resources usage, probability of missing real-time deadlines, etc.

the operating conditions (e.g. wireless channel quality, scene illumination).

However, the need for dynamic adjustments of dataflow applications recently became more and more important. Indeed, recent wireless communication standard such as 4G/5G constantly adjust the main transmission parameters (e.g., modulation, bandwidth, channel coding rate, allocation of radio resources to users) according to the environment state. As a consequence the processing power of tasks, their throughput, latency, and sometimes even the algorithm they implement, change during operation, potentially at a rate of the order of tens or hundreds of times per second. Moreover, the advent of new paradigms, like Software Defined Radio [2] (SDR), advocate for the sharing of the same execution platform between several more or less independent applications (e.g., 4G, 5G, WiFi, spectrum sensing). The same trends can be observed in other fields. Situations where a varying set of varying applications concurrently run on the same platform are becoming more and more frequent.

This paper presents Odyn[3], a new approach to the mapping and scheduling problem of time-varying real-time dataflow applications on parallel, heterogeneous, NUMA platforms. Part of the scheduling and mapping decisions are taken online, allowing the set of running tasks and their characteristics to change over time. It includes a deadlock avoidance strategy to guarantee the functional correctness. It mixes static and dynamic analysis in order to reduce the online computing power dedicated to the control tasks. Section II is an overview of previous work with similar goals, Section III presents the proposed approach, Section IV gives some experimental results, Section V concludes the paper and discusses several directions for improvement.

II. Related work

Scheduling problems have been widely studied over the years. A scheduling problem is usually composed of two parts: the mapping (or assignment) of tasks to PEs, and the execution ordering of tasks. These two aspects can be solved together or separately, and offline or online. Three categories of approaches exist: static, dynamic, and hybrid.

In *static* scheduling techniques such as [1], [3]–[5], both the execution order and the assignment of tasks to PEs are computed offline. These methods offer less flexibility and are often used for critical applications, e.g., avionics, where the scheduling correctness must be formally guaranteed.

Dynamic approaches such as those in Tomahawk [6], XKaapi [7], Spider [8] and StarPU [9], compute the mapping of tasks and their execution order online. These strategies are very flexible and reactive, particularly for homogeneous platforms where all units are capable of executing the same set of general-purpose operations. Yet, in the platforms we target, most PEs are highly specialized and only few PEs can execute a given class of operations (e.g., FFT, channel decoding or demodulation). In this case, it is our belief that the complexity overhead caused by the dynamic allocation of tasks to execution units (transfer of information among units, introduction of additional communication delays, decision-making process to select a mapping), is not justified.

Hybrid approaches offer a good balance in terms of flexibility and performance as they statically assign tasks to PEs but take scheduling and memory management decisions dynamically. Using a static mapping removes PEs from the burden of navigating the system's design space when ordering computations at run-time. Moreover, effective scheduling and memory management decisions can be distributed among PEs and taken at run-time.

PRUNE [10] is an example of hybrid approach that can be used to design dataflow applications that are analyzable and flexible. It uses an extension of the Synchronous Data-Flow (SDF) formalism where actors are allowed to dynamically vary their production/consumption rates. This MoC is used for compile-time analyses of deadlock freedom and memory usage. PRUNE offers the possibility to use a static or a dynamic assignment. PRUNE leaves the scheduling to the underlying OS (e.g GNU/Linux) since tasks are instantiated as threads using the pthreads library. PRUNE targets general purpose computer systems like personal computers while Odyn targets heterogeneous systems with hardware accelerators.

Singh et al. [11] propose a hybrid approach suitable for platforms with many tiles (sets of processor and memory resource). A static scheduling is computed for each tile based on Synchronous Data Flow graphs annotated with mapping information. The scheduling space of these graphs is explored to find the ordering with the highest throughput. At runtime, a mapping and its best-throughput scheduling are selected depending on the number of tiles available. In [11] the homogeneity of a target platform allows for dynamically adjusting mappings onto tiles that offer equal processing capabilities while we target heterogeneous architectures where PEs offer different processing capabilities.

Yang et al. [12] proposed a hybrid scheduling approach to minimize energy consumption while respecting time constraints. In their approach, a design time scheduler statically computes a set of Pareto-optimal schedulings for a set of applications. It then passes the characteristics of these Pareto curves to a runtime scheduler which uses them to select the best ordering so that the combined energy consumption of the entire system is optimal.

Similarly to Yang et al.'s approach, HYSTERY [13] statically computes a set of Pareto-optimal mappings and schedulings for a set of applications. Each scheduling is associated to a threshold temperature and is selected at runtime for each running application. During execution, schedulings are adapted as a consequence of frequency and voltage scaling if a component's temperature increases above its scheduling's threshold.

In contrast to Yang et al. work [12] and to HYSTERY [13], our scheduling decisions are entirely taken at runtime. Only the mapping of tasks to PEs and the deadlock prevention mechanism are computed statically.

[3]The name is a reference to the Odin god, and comes from French ***O****r****d****onanceur* ***dyn****amique* (dynamic scheduler)

To the best of our knowledge, the work of Calandrino et al. [14] is the closest to our contributions: it also proposes a hybrid real-time scheduling approach based on Earliest Deadline First (EDF), without precomputing a set of static schedulings that can be selected at runtime. However, tasks are assigned statically to clusters of general-purpose CPUs and GPUs whereas we assign and order tasks to more heterogeneous cores that can execute a specialized set of operations. Furthermore, their scheduling is preemptive and allows task migrations under certain conditions. To cover a larger number of platforms, Odyn does not currently support tasks preemption. This choice is justified by the fact that not all hardware accelerators efficiently support preemption.
Finally, Table I summarizes how the related work discussed in this section supports the characteristics that are relevant for the dynamic scheduling of data-flow applications. Entries marked with `undef.` (undefined) refer to characteristics for which support by a specific approach/tool could not be assessed because of lack of information in documents that are publicly available.

Last but not least, the risk of deadlock is a very common issue with scheduling algorithms. A deadlock is a situation where each task is waiting for a resource that another task possesses such that no task is able to continue its execution. *Deadlock prevention* techniques guarantee that deadlocks cannot occur under any circumstances. These methods can be overly pessimistic but remove the need to have deadlock monitoring at runtime. While common approaches to deadlock prevention are based on the use of Petri Nets [15], we present a novel approach based on the static computation of exclusion relations among memory buffers of an SDF graph. This approch guarantees the absence of deadlock without requiring data eviction at run-time.

III. Scheduling algorithm

In Odyn, tasks mapping to PE and tasks scheduling are separated. Indeed, Odyn assumes that tasks have already been annotated with mapping information e.g. at Design Space Exploration. Odyn also assumes that tasks have a Worst Case Execution Time (WCET).

A. Input

Different types of information serve as input to Odyn: (i) one or more application graphs annotated with real-time constraints (tasks' deadlines) and (ii) mapping information that assigns tasks to PEs, buffers to PEs' memories, specifies the size of each PE's memory, associates estimates of the tasks' Worst-Case Execution Time (WCET) and estimates for the throughput of communication links in the platform (to compute the duration of data transfers).

1) Application model: In Odyn, a workload is defined as a set S of independent periodic applications. Each application $s \in S$ is represented as a Synchronous Data-Flow (SDF) graph. In an SDF graph $G = < A, E >$, the set of nodes A (called actors) represents tasks interconnected by a set of

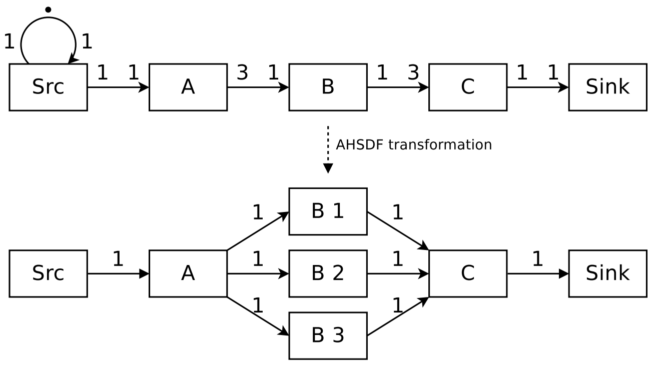


Fig. 1. Example of an application SDF graph and its corresponding AHSDF graph.

edges E that are First-In First-Out (FIFO) buffers. In the SDF MoC, an actor starts execution (firing) when its incoming FIFO(s) contains enough tokens, it cannot be preempted and produces tokens onto its outgoing FIFO(s). The number of tokens consumed/produced by each firing is a fixed scalar that is annotated to the graph's edges. As actors have no state in SDF graphs, if enough tokens are available, an actor can start several executions in parallel. For this reason, SDF graphs naturally express the parallelism of signal-processing applications and can be statically analyzed for several types of optimizations (e.g., memory allocation, scheduling). An example SDF graph can be seen at the top of Fig. III-A1.
In Odyn, each application SDF graph $s \in S$ is first transformed, as described in [16], into a directed Acyclic Homogeneous SDF graph (AHSDF graph). In a AHSDF graph $H = < T, D >$, tasks in T are associated to identical (homogeneous) production and consumption rates on FIFOs in D. The result of this transformation on the example SDF graph can be seen at bottom of Fig. III-A1. This transformation is necessary to expose data parallelism and memory allocation options (Homogeneous SDF) as well as to isolate one iteration of the algorithm captured by the original SDF graph (Acyclic Homogeneous SDF).
In Odyn, dynamic scheduling and memory management decisions are taken independently for each Processing Element. For each PE Odyn considers a cluster of partitioned AHSDF graphs. Given the set P of Processing Elements in the target platform, for each $p \in P$, the partitioned AHSDF graph of an application s, is the graph $H_p = < T_p, D_p >$ where the nodes $T_p \subseteq T$ are the tasks $t \in T$ mapped to p. "Artificial" source and sink tasks are added to guarantee the semantical correctness of the partitioned graph H_p with respect to the semantics of the SDF MoC. The set of edges $D_p \subseteq D$ is composed of the edges $d \in D$ for which both the producer task $t_{cons} \in T$ and consumer task $t_{prod} \in T$ are mapped to p as well as the edges that connect tasks in T_p with the "artificial" source and sink tasks. The cluster of partitioned AHSDF graph is computed as the disjoint union of the partitioned AHSDF graphs for all applications $s \in S$.

	Odyn^{H}	StarPU^{D}	Tomahawk^{D}	XKaapi^{D}	SynDEx^{S}	PRUNE^{H}	Calandrino et al.^{H}	Singh et al.^{H}	Yang et al.^{H}	HYSTERY^{H}
Deadlock analysis	yes	yes	no[1]	no	yes	yes	no	no	no	no
Deadlock prevention	yes	yes	no[1]	no	yes	no	no	no	no	no
NUMA support	yes	yes	yes	yes	yes	undef.	no	yes	yes	yes
Heterogeneity	CPU/DSP/ FPGA	CPU/GPU	CPU/DSP/ FPGA/GPU	no	CPU/DSP/ FPGA/GPU	CPU/GPU	no	no	no	CPU/GPU
Dynamic ordering	yes	yes	yes	yes	no	yes	yes	no	no	no
Dynamic assignment	no	yes	yes	yes	no	yes	yes	no	no	no
Targeted system	real-time	HPC	real-time	HPC	real-time	general purpose	real-time	real-time	low-power, real-time	low-power

[1] To ensure the absence of deadlocks a set of constraints on timing properties and data consumption of tasks must be respected.

TABLE I
A comparison between related approaches (s: static, h: hybrid, d: dynamic)

2) Platform model and mapping information: A generic logical architecture of the platforms that Odyn targets can be seen in Fig. III-A3. These platforms are composed of a set of n Processing Elements (PEs) and their local memory units connected together by a set of buses and interconnects. Direct Memory Access (DMA) engines can be used to transfert data among PEs' local memories.

3) Hypothesis for the scheduling analysis: The scheduling analysis that is performed by Odyn is based on the following list of hypothesis.

- Application:
  - H_1: All applications have the same period
- Architecture:
  - H_2: There is exactly one memory node per PE
- Mapping:
  - H_3: Buffers are dynamically allocated in the memory assigned to their consumer or producer PE and are not allowed to migrate at run-time
  - H_4: Task migration across PEs is not allowed
  - H_5: Estimates for the duration of data transfers can be computed only based on the throughput of the communication links in the target platform and the transfers' sizes
- Scheduling:
  - H_6: Each PE can execute only one task at a time, without preemption, until the task's completion
  - H_7: For each application, the overlapping of schedulings for different periods is not allowed (i.e., all tasks of period n are executed before any task of period $n+1$ can be executed)
  - H_8: Data transfers start as soon as they are requested by a consumer task
  - H_9: There is no risk of deadlock or livelock on the interconnect
  - H_{10}: There is no data loss during transfer
  - H_{11}: Buffers are entirely transferred from a producer PE to a consumer PE: there is no partial transfer of buffers. In other words, data-transfers cannot be interrupted and resumed at a later time instant.

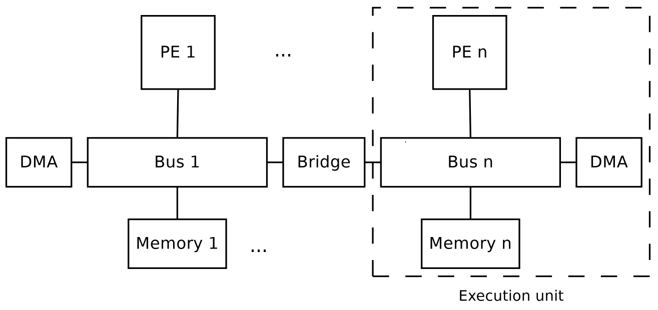


Fig. 2. Example logical architecture of a target platform.

B. Local dynamic scheduling algorithm

The task-ordering decisions are distributed. Each PE is managed by a separate dedicated scheduler that takes two inputs: (i) a flow of control events generated by the schedulers of all PEs in the platform, (ii) the cluster of partitioned AHSDF graphs. The algorithm of a scheduler is given in Algorithm 1. An event signals the timestamps of allocation and data availability of a specific buffer. Since each Processing Element can execute only one task at a time, without preemption (H_6), the well-known Earliest Deadline First (EDF) algorithm cannot be used. Instead, we use the Earliest Due-Date first (EDD) algorithm that is similar in principle to EDF, but is compatible with our hypotheses on targeted systems. If multiple tasks share the smallest due-date, one of them is selected randomly. The due-date metric is an upper-bound on the completion time of a task, which if missed will cause a deadline miss for said task or at least one of its descendants. We use the method introduced by Adyanthaya et al. in [17] to compute due-dates. This method takes into account that multiple successors of a task mapped to the same PE will be executed sequentially, providing a tighter bound than the usual as-late-as-possible start time. If no tasks are ready, a virtual idle task is inserted to force the PE to wait for the next event to occur. Once a task is selected, its completion time is estimated using the task's WCET, and the lifetimes of the task's buffers are updated accordingly. This generates a new event for each task output buffer. These events are sent to the scheduler of the PE executing the buffer's consumer task. For a given buffer, if its producer and consumer tasks are executed by two different PEs, a communication occurs to transfer the buffer's data. The communication time is estimated using the provided model of communications.

C. Deadlock conditions

Coffman et al. [18] determined four necessary and sufficient conditions for deadlocks to occur:

- Mutual exclusion: a resource must be non-shareable.
- Hold and wait: a process is holding at least one resource while requesting additional resources.
- No preemption: a resource ownership cannot be suspended or canceled until the end of the process holding the resource.
- Circular wait: there is a set of tasks $\{t_1, t_2, \ldots, t_n\}$ such that t_1 is waiting for a resource held by t_2, t_2 is waiting a resource held by t_3, and so on until t_n waiting for a resource held by t_1.

In our architecture, we have three kinds of resources: processing elements, communication units, and memory units. Ports used to access memory units are considered as communication resources, and memory resources only refer to the memory space. The state of the deadlock conditions for the different types of resource are presented in Table II, and explained below.

Mutual exclusion does not occur in the case of (purely theoretical) infinite memory because new buffers can always be allocated in new memory space.

The hold and wait situation can arise in memory units as some input buffers can be already allocated while others are yet to be. It cannot occur in processing and communication nodes as we enforced the constraint that all input and output buffers must be already allocated in order to launch the execution of a task or a communication.

There is no preemption in processing nodes in the targeted architecture. There is also no preemption on memory as we decided that allocated buffers can only be freed once their consuming tasks have been executed.

There cannot be circular waits for processing nodes as each task is executed on one and only one node, and a task requests its processing node only once all other needed resources (e.g., in/out buffers) have already been allocated.

This means that only memory units with limited capacity can produce deadlocks. In the next subsection, we propose a technique to prevent this type of deadlocks based on Memory Exclusion Graphs.

D. Deadlock prevention using a static analysis of Memory Exclusion Graphs

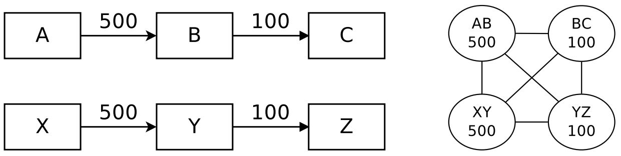
Desnos et al. introduced Memory Exclusion Graphs (MEGs) in [19]: a MEG is an undirected weighted graph where vertices represent indivisible memory objects that correspond to communication buffers in an SDF graph, the working memory of SDF actors and feedback FIFOs that store initial tokens in an SDF graph. Edges in a MEG represent exclusion relations, i.e., the impossibility to share physical memory. In Odyn, a MEG differs from the definition in [19] as it is derived from

```
input
    $I$: set of inbound events;
    $H = (T, D)$: AHSDF dependency graph;
        $T$: set of tasks;
        $D$: set of dependencies;
define
    $R$: tasks ready to be fired;
    $W$: tasks waiting for input buffers;
    $P_t$: tasks/events producing input buffers of task $t$;
    $Start_t$: start time of task $t$;
    $End_t$: end time of task $t$;
    $Alloc_{a,b}$: allocation time of task-$a$-to-$b$ buffers;
    $Free_{a,b}$: deallocation time of task-$a$-to-$b$ buffers;
    $send\_event(node, event, time)$: inform scheduler of
      $node$ that $event$ will occur at $time$;
    $receive\_events()$: get new event notifications from
      other schedulers;
$W \leftarrow T$; $current\_time \leftarrow now$;            // init
while $W \neq \emptyset$ do            // while tasks waiting
    for $event \in I$ do
        if $time(event) \leq current\_time$ then
            $I \leftarrow I \setminus event$;
            for $succ \in successors(event)$ do
                $P_{succ} \leftarrow P_{succ} \setminus event$;
            end
        end
    end
    for $t \in W$ do
        if $P_t = \emptyset$ then  $R \leftarrow R \cup t$;
        ;
    end
    $W \leftarrow W \setminus R$;
    if $R \neq \emptyset$ then
        // Select task to schedule
        $t \leftarrow$ task with smallest due date in $R$;
        $Start_t \leftarrow current\_time$;
        $End_t \leftarrow current\_time + wcet(t)$;
        // Update buffers lifetime
        for $succ \in successors(H, t)$ do
            $Alloc_{t,succ} \leftarrow current\_time$;
            $P_{succ} \leftarrow P_{succ} \setminus t$;
            if $succ$ is an outgoing communication then
                $End_{com} = End_t + wcet(succ)$;
                $send\_event(dest(succ), succ, End_{com})$;
            end
        end
        for $pred \in predecessors(H, t)$ do
            $Free_{pred,t} \leftarrow End_t$;
        end
        schedule $t$ at $Start_t$;
        $current\_time \leftarrow End_t$;
    else
        // Idle until next incoming event
        $current\_time \leftarrow \min(I)$;
    end
    // Get new incoming event
    $I \leftarrow I \cup receive\_events()$;
end
```

Algorithm 1: Scheduler of an execution node

Condition	Processing	Commu-nication	Memory (infinite)	Memory (limited)
Mutual exclusion	true	true	false	true
Hold and wait	false	false	true	true
No preemption	true	true	true	true
Circular wait	false	true	false	true
Deadlock risk	no	no	no	yes

TABLE II
DEADLOCK RISK DEPENDING ON RESOURCE



(a) Example cluster of partitioned AHSDF graphs (b) Corresponding MEG

Fig. 3. Example applications, potentially subject to deadlock

a PE-specific cluster of partitioned AHSDF graphs. Formally, in Odyn, a MEG is defined as a weighted undirected graph $M = < B, R >$ where the vertices B are the buffers in D_p that capture data dependencies between tasks that run on a given PE, T_p. Similarly to [19], vertices in B are weighted with the sizes of buffers D_p and edges R represent memory exclusion relations. As in [19], in Odyn, we consider that the memory allocated to buffers D_p is reserved from the execution start of the producer actor until the completion of the consumer actor in T_p.

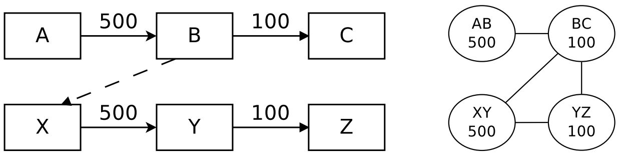
As described in [19], AHSDF graphs can be updated with scheduling constraints. These constraints have the effect of removing exclusion relations (edges) in the corresponding MEG.

As explained above, a deadlock may occur if a circular wait happens due to a memory being unable to host a buffer. This situation can be detected by inspecting cliques in a MEG M. A clique is defined as a subset of an undirected graph's vertices such that every two distinct vertices in the clique are adjacent. The weight of a MEG's clique defines the maximum amount of memory that must be allocated to store the buffers that compose the clique. This is regardless of the scheduling policy for the tasks that produce/consume the buffers represented by a MEG's nodes. Therefore, if the weight of a clique exceeds the PE's memory capacity, a deadlock may occur. We call these cliques *oversized cliques*. We define a minimal oversized clique as an oversized clique such that removing any buffer from it would lead to a clique that is not oversized.

In case of unscheduled MEGs, the presence of an oversized clique is a necessary (but not sufficient) condition for a deadlock. For scheduled MEGs, the presence of an oversized clique becomes a necessary and sufficient condition. For example, let's consider the scenario where two applications can run simultaneously, as shown in Fig. 3a that represents the corresponding cluster of partitioned AHSDF graphs. These applications run on a platform with one processing node and one memory unit of capacity 1000 bytes. Executing these applications will deadlock if producer tasks A and X are scheduled to run before both consumer tasks B and Y. This is because neither output buffer BC nor YZ can be allocated. This is visible in the corresponding MEG (Fig. 3b) as the clique $\{AB, BC, XY\}$ is of size 1100 bytes, thus larger than the available memory capacity.

```
input
    $M$: Memory Exclusion Graph to analyze;
    $mem\_size$: size of memory unit;
output
    $OC$ minimal oversized clique of $M$ (or empty set if
      none found);
$OC \leftarrow \emptyset$;                    // initialization
$max\_cliques \leftarrow find\_maximal\_cliques(M)$;
for $clique \in max\_cliques$ do
    if $size(clique) > mem\_size$ then
        // reduce to minimal-oversized
        for $buffer \in clique$ do
            if $size(clique \setminus buffer) > mem\_size$ then
                $clique \leftarrow clique \setminus buffer$;
            end
        end
        $OC \leftarrow clique$;
        break;
    end
end
```
Algorithm 2: Find minimal-oversized clique in MEG

Our anti-deadlock mechanism is based on finding minimal oversized cliques in an unscheduled MEG, as described in Algorithm 2. To find a minimal oversized clique we first look for a maximal clique[4] of the MEG that is oversized, and then remove vertices from it until we get a minimal oversized clique. If there is an oversized clique in the MEG, a scheduling constraint is added to an application's AHSDF graph, from the consumer task of a buffer in the clique to the producer task of another buffer of the clique. The MEG is then updated. We look for minimal oversized cliques, because removing a minimal oversized clique also removes any larger clique containing all buffers of that minimal oversized clique. The additional scheduling constraint corresponds to the dashed



(a) Cluster of partitioned AHSDF graphs, updated with a scheduling constraint (b) Updated MEG

Fig. 4. Updated applications, deadlock-free

[4]A maximal clique is a clique to which no vertices can be added without resulting in a graph that is not a clique.

link in Fig. 4a, and the updated MEG is shown in Fig. 4b. Adding scheduling constraints prevents from having all the producer tasks of buffers which are part of the clique from being executed while none of the corresponding consumer has. This avoids having to fit all the oversized clique's buffers in memory at the same time, thus preventing deadlocks. In our example, the deadlock prevention is visible in the updated MEG in Fig. 4b as the oversized clique $\{AB, BC, XY\}$ is no longer present. The remaining maximal cliques $\{AB, BC\}$ and $\{BC, XY, YZ\}$ are not oversized, meaning that there is no oversized clique left in the MEG.
Since finding the cliques in a MEG is computationally expensive (the clique decision problem is NP-complete [20]), we propose to perform anti-deadlock analyses statically. This analysis is made on the worst-case scenario, where we consider that all applications run simultaneously.

IV. Experimental results

We evaluated Odyn on the scheduling of a workload of ten application graphs. The graphs were generated using SDF^3 [21] and have ten tasks (nodes) each. The buffer sizes were generated to be in the range $[32, 8192]$ bytes, with an average size of 2048 bytes. Our evaluation platform is an instance of the one in Fig. III-A3 with three PEs and three memory nodes, each PE being statically associated to a different memory node.

We remind to the reader that we consider a workload of applications with identical periods (hypothesis H_1). Scheduling decisions are locally taken by each PE. To validate these decisions, we run a simulation that ensures the overall correctness of the scheduling of all applications, i.e. dependencies are respected and memory can be safely allocated when requested.

A. Anti-deadlock analysis

The number of anti-deadlock scheduling constraints to add to an application's AHSDF graph $H = <T, D>$ depends on the number of oversized cliques in the application's MEG. The number of oversized cliques depends itself on the number and size of a MEG's buffers, and on a target PE's memory capacity. Fig. IV-A shows the impact of the number of parallel applications on the number of scheduling constraints (links in D) that are added to prevent deadlocks in H. In this configuration, no anti-deadlock links are required when running 4 applications or fewer. As expected, the number of links increases exponentially with the number of applications that run simultaneously, whereas the number of nodes per MEG grows linearly. The reason is that the number of cliques in a MEG grows exponentially with the number of nodes and the parallelism induced by the number of applications.

Fig. IV-A shows the relation between the PEs' memory size and the number of anti-deadlock links in a partitioned AHSDF graph. In our target architecture, the size of memory nodes is fixed by the platform's hardware design and cannot be changed at run-time. Using very large memories can ensure that no deadlock will occur, but at a very high cost since memory nodes will be greatly oversized in comparison to the average need. On the contrary, very small memory size will greatly reduce the overall performance of the system. The lesser the size of memories, the greater the number of links added to prevent deadlocks, and each additional anti-deadlock scheduling constraint can have an effect on the performance.

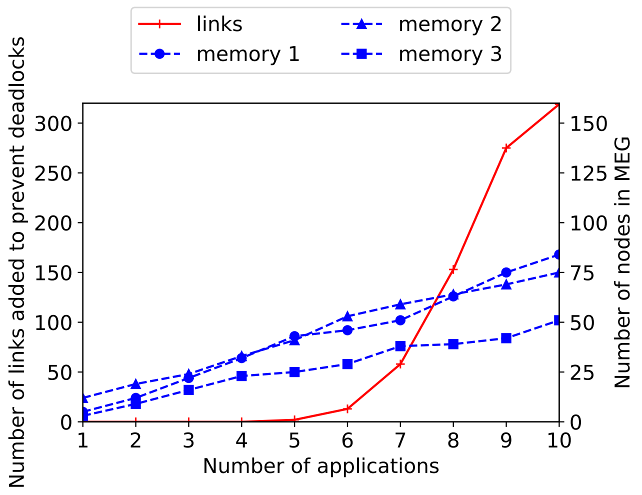


Fig. 5. In red (continuous line), it is shown the number of links that are added to prevent deadlocks as a function of the number of applications (memory size 64 KB per PE). Blue (dashed) curves show the size of MEGs for three memories, as a function of the number of applications.

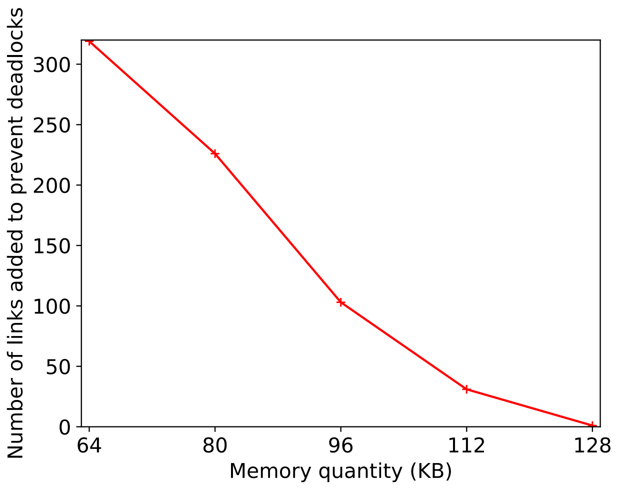


Fig. 6. Number of links added to prevent deadlocks, as a function of the memory size per PE (10 applications)

B. Evaluation of anti-deadlock impact on performance

We evaluated the performance impact of the anti-deadlock mechanism by running simulations on multiple sets of ten applications each. Using SDF^3 we generated 100 random AHSDF graphs, each composed of 10 tasks. We then selected randomly 2000 sets of 10 AHSDF graphs. For each of these sets, we ran two simulations, with a memory size of 64KiB per PE. For the first simulation, the deadlock prevention

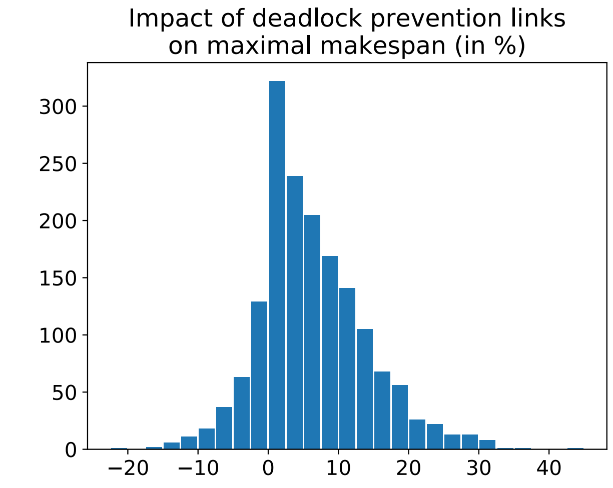


Fig. 7. Histogram of the impact of the deadlock prevention mechanism on the maximal makespan.

mechanism was *not* enabled. This allowed us to discover the number of applications sets that contained deadlocks; it amounted to 343 out of 2000 sets of applications. In the second simulation, the deadlock prevention mechanism was enabled. As expected, no deadlocks have been encountered, illustrating that our mechanism effectively prevents deadlocks. For the remaining 1657 out of 2000 application sets that did not present deadlocks[5] we evaluated the impact of our anti-deadlock mechanism on the system's performance. More precisely, we computed the difference in the makespan (i.e., the total length of a schedule) observed on these sets in the two simulations. The distribution of the impact on performance can be seen in Fig. IV-B. It resembles a biased Gaussian distribution. The average difference in performance is equal to 6.15%, the median difference is equal to 5.00%, and differences range from −20.30% (i.e. a reduction of the makespan by 20.30%) in the best case, to +43.25% in the worst case.

V. Conclusions and Future Work

In this paper we introduced Odyn, a hybrid approach for scheduling periodic dataflow applications on parallel, heterogeneous, Non-Uniform Memory Architecture platforms. We also developed a deadlock prevention mechanism based on the *static* computation of cliques in Memory Exclusion Graphs. The additional cost of these computations is mitigated in Computer Aided Design tools in the Design Space Exploration while computing a mapping for the input applications.
In terms of future directions, we believe that Odyn's anti-deadlock analysis could be used offline at design time to determine a compromise between the size of memory regions that are assigned to PEs, the set of concurrent applications that a platform can efficiently support and the desired performance. An additional future work is to automatically compute the optimal memory regions that must be assigned to PEs for a given performance requirement.

[5]At least during the simulation without deadlock prevention that we ran.

References

[1] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[2] J. Mitola III, "Software radios-survey, critical evaluation and future directions," in *National Telesystems Conference*, 1992, pp. 13/15–13/23.

[3] Y. Sorel et al., "Syndex," http://www.syndex.org, last visited on September 2018.

[4] S. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Static mapping of mixed-critical applications for fault-tolerant mpsocs," in *DAC*, 2014, pp. 1–6.

[5] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in mpsocs," in *DATE*, 2007, pp. 1–6.

[6] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg, and G. Fettweis, "Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3, pp. 107:1–107:24, 2014.

[7] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *IPDPS*, 2013.

[8] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi, "Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps," in *EDERC*, 2014, pp. 167–171.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[10] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya, "PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms," *IEEE Trans. on Sig. Proc.*, vol. 66, no. 3, pp. 654–665, 2018.

[11] A. K. Singh, A. Kumar, and T. Srikanthan, "A hybrid strategy for mapping multiple throughput-constrained applications on mpsocs," in *CASES*, 2011, pp. 175–184.

[12] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins, "Energy-aware runtime scheduling for embedded-multiprocessor socs," *IEEE Design Test of Computers*, vol. 18, no. 5, pp. 46–58, 2001.

[13] A. Abdi and H. R. Zarandi, "Hystery: a hybrid scheduling and mapping approach to optimize temperature, energy consumption and lifetime reliability of heterogeneous multiprocessor systems," *The Journal of Supercomputing*, vol. 74, no. 5, pp. 2213–2238, 2018.

[14] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *ECRTS*, 2007, pp. 247–258.

[15] Z. Li, M. Zhou, and N. Wu, "A survey and comparison of petri net-based deadlock prevention policies for flexible manufacturing systems," *IEEE Trans. Syst., Man, Cyber. C*, pp. 173–188, 2008.

[16] K. Rosvall and I. Sander, "A constraint-based design space exploration framework for real-time applications on mpsocs," in *DATE*, 2014, pp. 1–6.

[17] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten, "Fast Multiprocessor Scheduling with Fixed Task Binding of Large Scale Industrial Cyber Physical Systems," in *EUROMICRO DSD*, 2013, pp. 979–988.

[18] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[19] K. Desnos, M. Pelcat, J. Nezan, and S. Aridhi, "Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs," *Jour. of Sig. Proc. Syst.*, pp. 1–19, 2014.

[20] R. M. Karp, *Reducibility among Combinatorial Problems*. Springer US, 1972, pp. 85–103.

[21] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *ACSD*, 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3