X-Stream: Accelerating Streaming Segments on MPSoCs for Real-time Applications

ROHAN TABISH, University of Illinois at Urbana-Champaign, USA
RODOLFO PELLIZZONI, University of Waterloo, Canada
RENATO MANCUSO, Boston University, USA
GIOVANI GRACIOLI, Federal University of Santa Catarina, Brazil
REZA MIROSANLOU, University of Waterloo, Canada
MARCO CACCAMO, Technical University of Munich, Germany

Authors' addresses: Rohan Tabish, University of Illinois at Urbana-Champaign, USA, rtabish@illinois.edu; Rodolfo Pellizzoni, University of Waterloo, Canada; Renato Mancuso, Boston University, USA; Giovani Gracioli, Federal University of Santa Catarina, Brazil; Reza Mirosanlou, University of Waterloo, Canada; Marco Caccamo, Technical University of Munich, Germany, cpalmer@prl.com.

We are witnessing a race to meet the ever-growing computation requirements of emerging AI applications to provide perception and control in autonomous vehicles — e.g., self-driving cars and UAVs. To remain competitive, vendors are packing more processing units (CPUs, programmable logic, GPUs, and hardware accelerators) into next-generation multiprocessor systems-on-a-chip (MPSoC). As a result, modern embedded platforms are achieving new heights in peak computational capacity. Unfortunately, however, the collateral and inevitable increase in complexity represents a major obstacle for the development of correct-by-design safety-critical real-time applications. Due to the ever-growing gap between fast-paced hardware evolution and comparatively slower evolution of real-time operating systems (RTOS), there is a need for real-time oriented full-platform management frameworks to complement traditional RTOS designs.

In this work, we propose one such framework, namely the *X-Stream framework*, for the definition, synthesis, and analysis of real-time workloads targeting state-of-the-art accelerator-augmented embedded platforms. Our X-Stream framework is designed around two cardinal principles. First, computation and data movements are orchestrated to achieve predictability by design. For this purpose, iterative computation over large data chunks is divided into subsequent *segments*. These segments are then streamed leveraging the three-phase execution model (load, execute and unload). Second, the framework is workflow-centric: system designers can specify their workflow and the necessary code for workflow orchestration is automatically generated.

In addition to automating the deployment of user-defined hardware-accelerated workloads, X-Stream supports the deployment of some computation segments on traditional CPUs. Finally, X-Stream allows the definition of real-time partitions. Each partition groups applications belonging to the same criticality level and that share the same set of hardware resources, with support for preemptive priority-driven scheduling. Conversely, freedom from interference for applications deployed in different partitions is guaranteed by design. We provide a full-system implementation that includes RTOS integration and showcase the proposed X-Stream framework on a Xilinx Ultrascale+ platform by focusing on a matrix-multiplication and addition kernel use-case.

Additional Key Words and Phrases: MPSoC, Segment Streaming, Heterogeneous Computing

ACM Reference Format:
Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Giovani Gracioli, Reza Mirosanlou, and Marco Caccamo. 2018. X-Stream: Accelerating Streaming Segments on MPSoCs for Real-time Applications. *J. ACM* 37, 4, Article 111 (August 2018), 27 pages. 10.1145/1122445.1122456

1 INTRODUCTION

An important step in the evolution of embedded computing has been — and still is in many ways — the transition from single-core processors to multicore systems. In their first generations, these provided I/O interfaces (e.g., PCI-e, USB) to interact with external hardware accelerators such

as FPGA and/or GPUs. Following important hardware advancements and as new challenging applications emerged — especially in response to the increase in popularity of autonomous vehicles — there has been an spike in the demand for real-time, low-power data-intensive computing. To address such requirements, hardware vendors are packing multiple processing elements, such as CPU clusters, GPUs, programmable logic (PL), and AI accelerators, into integrated multiprocessor systems-on-a-chip (MPSoC).

One of the common attributes of these high-performance MPSoCs is the existence of a shared memory subsystem. This is well suited for systems where the goal is high average-case performance. But unfortunately, this is fundamentally ill-suited for safety-critical systems. Indeed, in the latter, worst-case execution times (WCET) are crucial for safety determination. If no measures are taken to mitigate performance interference at the level of shared memory components, the WCET of a task running in one of the processing elements can vary significantly as we activate more processing elements [20]. The three-phase execution model, where an application task is first (1) loaded into a local and private memory, (2) locally executed and then (3) written back to main memory[1] (load, execute, unload) has been proposed to address multicore contention *by design* [2, 25].

In light of its design principles, the three-phase model represents a highly attractive workloads management strategy especially in safety-critical systems. Therefore, it seems natural to investigate possible adaptations to handle real-time hardware-accelerated workloads. Doing so involves solving four main challenges. First, data movements between main memory and accelerators, and between local memories of accelerators need to be carefully orchestrated to prevent contention. Second, appropriate (double– or triple–) buffering mechanisms must be integrated depending on the user-defined workflow to properly implement pipelining. Third, workflow orchestration code needs to be auto-generated and appropriate integration with the underlying real-time operating system (RTOS) shall be provided to support priority-driven preemption within each partition. Fourth, a suitable schedulability analysis must be provided to aid system verification and validation.

In this work, we aim to tackle the four challenges mentioned above and propose our X-Stream framework as a solution. We hereby describe our framework that extends the use of the three-phase execution model to accelerator-enabled MPSoCs and offers following contributions:

- First, we provide a generic DAG formalization for parallel application tasks.
- Second, we offer a strategy to convert DAG tasks into a series of *segments*, each containing all the required data transfers between local (i.e. internal to the accelerator) and global (i.e. shared at the platform level) memory resources to translate the DAG-imposed precedence constraints.
- Third, we describe an OS-level runtime environment to deploy the generated application code onto a RTOS.
- Finally, we derive schedulability results and evaluate our design with a full system implementation.

2 RELATED WORK AND BACKGROUND

PRedictable Execution Model (PREM): Contention over shared memory resources such as last-level cache (LLC), main memory, and interconnect is known to be a major source of unpredictability in multicore systems and hence an important roadblock for the consolidation of safety-critical applications with strict timing requirements. Researchers over the last decade have proposed various methodologies to attack this problem from multiple angles. Solutions such as cache partitioning [8, 10, 13] have gained significant traction as a mitigation strategy for contention at the LLC. Similarly,

[1]After execution, what needs to be preserved is any (partial) output and state accumulated by the application during execution that is required for successive invocations of the same or other applications.

methods such as partitioning of DRAM banks and of the sustainable main memory bandwidth have been proposed and analyzed in [9, 11, 32, 33]. Another class of approaches followed by the researchers in the community pivots around the PRedictable Execution Model (PREM) originally proposed in [15] and later extended to multicore systems [17, 31]. The first work to consider SPM over cache for predictability include PRET [5]. In PREM, tasks execution is divided into memory and computation phases. Main memory can only be accessed during memory phases. Hence, contention over main memory is explicitly managed by making memory phases of different cores mutually exclusive. To ensure fairness, TDMA or round-robin arbitration is used to decide which core is allowed to perform a memory phase.

Three-phase Model: Given the inherently unpredictable nature of caches, scratchpad-based multicore platforms represent alternative popular architectural designs for safety-critical systems. These platforms provide software-managed (i.e., explicitly addressable) per-core fast memories that are located in close proximity of each CPU. These go under the name of scratchpad memories, or SPM for short, and are limited in size anywhere from tens to thousands of kilobytes. Because data need to be explicitly moved in and out of an SPM, the original PREM model was extended [2] into a *three-phase execution model* that involves load, execute, and unload phases for a given chunk of computation — e.g., a job of a periodic task. The three-phase model has close similarities with the Acquisition Execution Restitution (AER) model considered in [4]. Notably, the Scratchpad-centric OS proposed in [25] demonstrated the concept of an operating system designed around the concept of tasks scheduled and executed according to the three-phase model.

A key advantage of the three-phase model is the ability to offload memory load and unload phases to a data engine such as a direct memory access (DMA). Doing so allows performing execution phases on the CPU and load/unload phases carried out by the DMA in parallel [4, 12, 18, 24, 25, 27]. However, parallel execution of task on CPU and load/unload using a DMA requires splitting each local SPM into two memory regions. One region is used to execute the current job, while the other is used to load (resp., unload) the memory required (resp., produced) by the next (resp., previous) job. To serve multiple cores, a TDMA arbitration of the DMA is implemented.

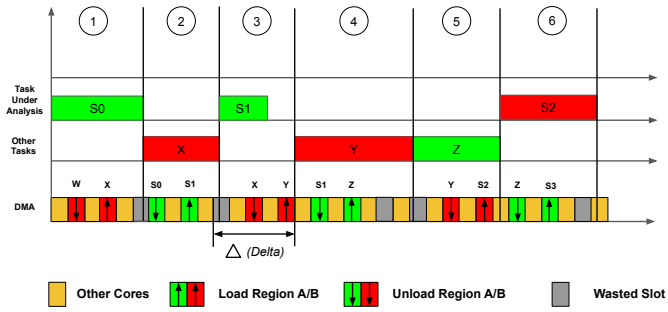

Fig. 1. Three-Phase Execution Model TDMA with M = 2 Cores

To better understand the core principles governing the execution and analysis of applications according to the three-phase model, consider the example provided in Figure 1. Here, we focus on one core and a task under analysis (first row) preempted by other tasks (second row). The SPM is divided into two regions, namely A (green) and B (red). Following [3, 23–25], we assume that the code of each task can be divided into a sequence of $\mathcal{S}$ segments, which we denote as S0, S1, ..., S$\mathcal{S}$ as discussed in [24]. A typical technique to analyze task scheduling in this setup is to split the timeline into a sequence of *scheduling intervals*. A scheduling interval is delimited by either the completion

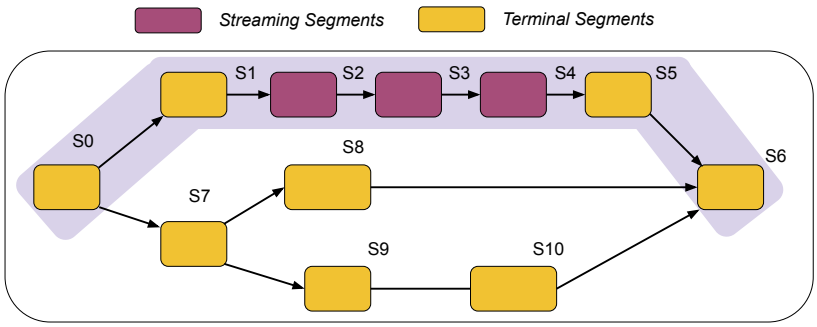

Fig. 2. Example control-flow DAG with highlighted streaming segments (magenta).

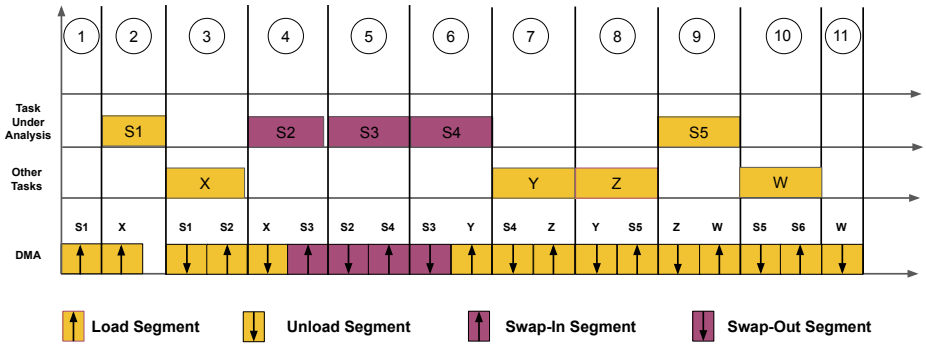

Fig. 3. Streaming execution timeline corresponding to the control-flow path highlighted in Figure 2

.

of an execution phase of a segment of the current job or the completion of a load operation for the segment of the next job, whatever occurs later. In Figure 1, the intervals are delimited by vertical lines and numbered Interval ① through Interval ⑥. The corresponding payloads to be loaded/unloaded and executed are indicated based on the segment name - S0 through S2 for the task under analysis, X, Y, and Z for other tasks. During Interval ①, Interval ② and Interval ⑥ the task under analysis is executed. Whereas, during Interval ②, Interval ④ and Interval ⑤ other tasks are executed. During each interval, (i) a segment of a job is executed from one of the SPM regions (e.g., X in Region B during Interval ②) while (ii) the data produced by the previous segment (e.g., S0 in Region A) is unloaded from the other region and (iii) the data of next segment is loaded (e.g., S1 in Region A) using the DMA.

The time required to complete all the necessary DMA operations (1 load + 1 unload) in each interval depends on the number of cores and the employed TDMA discipline. Under the coarse-grained TDMA approach utilized in [25], the TDMA slot size σ assigned to a core is sufficient to perform the load or unload operation in a single slot. With M cores and the same slot size for all cores, the worst-case memory time is $\Delta = \sigma \cdot (2M + 1)$, as also highlighted in Interval ③ in Figure 1. Note that the first slot in Interval ③ is wasted because the previous interval has not yet been completed at the beginning of the slot, hence the unload operation cannot be guaranteed to complete within it.

Streaming Segments: In the aforementioned works [12, 24–27] the parallelism that can be achieved has important limitations. Specifically, an execution can overlap with load/unload phases

only if they belong to different jobs. The first work to relax this limitation was [23]. In this case, a single job might be executed over a sequence of consecutive intervals by defining the concept of *streaming* segments. Streaming segments are produced through a tiling compilation pass where a loop processing over a large chunk of data is broken down into smaller subsets of iterations (a *loop tile*), each of which is encapsulated in a segment. With the exception of the last tile in the loop, which comprises a *terminal* segment, while a tile executes it is possible to load the data for the next segment in parallel. Hence, each such streaming segment can be immediately followed by another segment of the same task. The application logic that is not part of a loop and cannot be converted into a sequence of streaming segments is also encapsulated into one or more terminal segments, with the same parallelism limitations as depicted in Figure 1.

In [23] a task is characterized by a Directed Acyclic Graph (DAG) that represents its control flow. A simplified example is provided in Figure 2. When the source node (S0) is executed, it is uniquely determined which branch the current job will follow to reach the sink node (S6). In other words, the possible paths in the DAG represent *conditional alternatives* in the execution of the job. Some of these paths might include one or more sequences of streaming segments, as it is the case for the S0→S1. . .S5→S6 path highlighted in Figure 2. Source and sink nodes are always terminal segments. Differently from [23], in this paper we do not consider tasks with alternative paths for the sake of simplicity. Instead, we focus on jobs that consist of a sequence of processing segments, like the sub-graph comprising segments S1-S5 in the top branch. Furthermore, we impose that each segment, with the exception of the first one that is needed for job initialization, performs the same type of computation on a different chunk of data. Thus one can think of the sequence of streaming segments as multiple iterations of the same computational function invoked over different inputs.

To better understand how the three-phase model can be leveraged to pipeline computation and data movement in a sequence of streaming segments, consider Figure 3 where we depict the execution of segments S1-S5 from Figure 2 under the interference of other tasks on the same CPU (segments X, Y, Z and W). In the figure, we have 11 scheduling intervals and we color-code in magenta the data operations and segments where streaming is performed via an alternation of swap-in/swap-out operations. The latter correspond to the load of new operands for the next iteration and the unloading of generated partial outputs to/from the SPM region currently used by the task. Please note that for simplicity, the schedule shown in Figure 3 does not include the TDMA of other cores. In reality, there are other cores in the system.

Contribution: The key contribution of this paper is the proposal of a practical framework for the adoption of the three-phase model in safety-critical systems with application workloads deployed on user-defined hardware accelerators. Custom accelerators are a natural fit for the three-phase model, since they are typically designed to operate on local memories, i.e. either banked SPMs or FIFO queues. The challenge is to orchestrate accelerator execution and data movements in a way to guarantee strict real-time requirements. Additionally, in light of the research attention that the model has received on scratchpad-based platforms, we extend our framework to support traditional CPUs. The latter is achieved by defining dedicated per-CPU scratchpad memories to obtain a unified management strategy for accelerators as well as CPUs. The scratchpad-aided CPU management strategy borrows from [23] and the existing literature referenced above. However, what sets this work apart is that the proposed framework allows system designers to reason in terms of application-level logic and data movements. The framework then provides end-to-end system consolidation that also includes OS-level resource management and automatic code generation.

3 APPLICATION AND PLATFORM MODEL

3.1 Application Model

In this work, we target platforms that allow the definition of custom accelerators and that provide (potentially) multiple application CPUs, as described is Section 3.2. Multiple logically independent set of applications are allowed to share the same platforms. In this case, they are isolated from one another via the definition of a *spatio-temporal partition*, which we simply refer to as a *partition*. Each partition is assigned dedicated hardware resources both in time and space. Spatial partitioning is achieved by statically assigning hardware resources — e.g., accelerators, physical memory ranges, I/O devices — to the applications sharing that partition; temporal partitioning is achieved by ensuring that only one partition at a time has full access to shared hardware components. Most importantly, access to the shared main memory subsystem is strictly managed such that non-overlapping time slots are assigned to partitions during which they are allowed to access main memory. In other words, a time division multiple access (TDMA) scheme is employed to schedule access of partitions to main memory. Partitioning ensures freedom from interference, i.e., each partition is unaffected by the behaviors of other partitions.

Because partitions behave independently from one another, for the purpose of our analysis, it is enough to reason on the workload deployed on a single partition. To simplify the discussion, we further assume that the partition of reference contains one general purpose CPU and an arbitrary number of special-purpose accelerators. We then consider a set of sporadic real-time tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$ bound to the partition under analysis. T_i denotes the minimum inter-arrival time or period of task τ_i and D_i represents its relative deadline: each task releases a potentially infinite number of jobs, where the activation time of successive jobs is separated by at least T_i time units, and each job must complete at most D_i time units after its activation. We assume constrained deadlines, i.e., $D_i \leq T_i$. We omit the index i when referring to a generic task to prevent notation cluttering. The generic task τ is internally structured as a loop/sequence of I iterations of the same computational function $\mathcal{F}(\mathcal{X})$, where $\mathcal{X}$ represent the set of data operands required to compute $\mathcal{F}$. The data operands accessed during each iteration are a fraction of a larger input to be batch-processed. A typical example are vision kernels executed over a sequence of video frames. We assume no data dependencies between successive iterations, so that we can pipeline the execution over multiple processing elements — e.g., specialized accelerators and CPUs [2] We use $\mathcal{P}$ to denote the set of elements, where $CPU \in \mathcal{P}$ represents the CPU in the partition under analysis, while the other elements in $\mathcal{P}$ are accelerators.

We use set $\mathcal{A} = \{a_1, a_2, \ldots a_k\}$ to denote the data elements (structs, arrays, matrices, or corresponding tiled subarray/matrixes) used in each iteration. These k data elements include any intermediate data produced in the current iteration and hence it holds that $\mathcal{X} \subseteq \mathcal{A}$. The computation in each iteration is expressed by a DAG $\mathcal{F} = (V, E)$, where V is a set of vertices representing operations on data, and E is a set of edges representing data movements / dependencies. More in details, each vertex $v \in V$ is characterized by a processing element $v.PE$, where $v.PE \in \mathcal{P}$, meaning that v is bound to execute on a specific accelerator or on the general-purpose CPU.

Because each vertex corresponds to an intermediate computational block to compute $\mathcal{F}$, we use $v.func(\cdot)$ to denote the data processing function performed by v. When $v.PE = CPU$, $v.func(\cdot)$ corresponds to the semantics of a portion of binary code that is compiled to executed on the CPU. When $v.PE \neq CPU$, the function corresponds to the operation performed by an accelerator. The function takes a variable number of parameters depending on the operation-specific number of input and output data elements. Data elements are uniquely identified in terms of their location

[2]This assumption could be relaxed to only exclude dependencies that would stall the pipeline. In particular, read-after-read (RAR) dependencies are always acceptable.

in main memory. For simplicity, we impose the constraint that no two vertexes can use the same accelerator. Formally: $\forall v', v'' \in V : v' \neq v'' \Rightarrow v'.PE = v''.PE = CPU \vee v'.PE \neq v''.PE$.

Each edge $e \in E$ is characterized by source $e.s \in E \cup \{\perp\}$, destination $e.d \in E \cup \{\perp\}$, and data element $e.a \in \mathcal{A}$. An edge e with $e.s \in E \wedge e.d \in E$ represents a data dependency and precedence constraint between the two vertices, where $e.s.func(\cdot)$ (that is, the function performed by the source vertex) first writes to data element $e.a$ and then $e.d.func(\cdot)$ (the function of the destination vertex) reads $e.a$. An edge e with $e.s = \perp \wedge e.d \in E$ represents a read dependency for $e.d.func(\cdot)$ on input data $e.a \in \mathcal{X}$ in main memory; an edge e with $e.s \in E \wedge e.d = \perp$ represents a write dependency for $e.s.func(\cdot)$ on output data $e.a \in \mathcal{X}$ in main memory. We use function $use(v)$ to denote the set of data items used by v; formally, $a \in use(v) \Leftrightarrow \exists e \in E : (e.s = v \vee e.d = v) \wedge e.a = a$.

Given two vertices v', v'', we say that v' is an immediate precedecessor of v'' (and equivalently, v'' is an immediate successor of v') if there is an edge between v' and v'': $\exists e \in E : e.s = v' \wedge e.d = v''$; we write $v' \in pred(v'')$ and $v'' \in succ(v')$. We say that vertex v is a source if it has no predecessors. We say that vertex v is a sink if it has no successors. Finally, for computational function $\mathcal{F}$ to be valid in our model, it must be deterministic: this means that the result of the computation must be independent of the order in which individual processing functions are executed and data elements are read/written, as long as all precedence constraints encoded by the edges in E are respected. This requires two conditions. First, a vertex v cannot have two incoming edges $e', e'' : e'.d = e''.d = v$ for the same data element: $e'.a = e''.a$; otherwise, we would not know whether $v.func(\cdot)$ should read the value of a from e' or e''. Second, for any pair of vertexes v', v'', if v' has a write dependency to main memory for a data element a (i.e., $\exists e' \in E : e'.s = v' \wedge e'.d = \perp \wedge e'.a = a$), and v'' has either a read or write dependency to main memory for the same data element (i.e., $\exists e'' \in E : \big((e''.s = \perp \wedge e''.d = v'') \vee (e''.s = v'' \wedge e''.d = \perp)\big) \wedge e''.a = a$), then there must exist a directed path in the DAG to which both v' and v'' belong; this guarantees that the order in which the dependencies to main memory are handled is specified by the DAG precedence constraints.

Example: To better understand the proposed DAG-based model, let us introduce a concrete example that will be used throughout the paper. We consider a set of data elements $\mathcal{X} = \mathcal{A} = \{A, B, C\}$, where A, B, and C are square matrices. The computation consists of a matrix multiplication and addition: $\mathcal{F}(A, B, C) = (A \times B) + C$. We use matrix-multiplication and addition kernel as an example because these operations are building blocks of current state-of-the-art neural networks and many signal processing applications such as convolution and others. Moreover, it provides a basic example of which part of the kernel needs to be accelerated and which part can execute on the CPU as we will show later in the paper. It should be noted that a user can always take a neural network and accelerate first few layers which are generally very computation intensive on the accelerator and last few layers on the CPU. However, exploring this a part of future work and is out of the scope of this paper. The corresponding DAG representation is depicted in Figure 4 (note that some of the DAG parameters will be computed in next Section 4). The DAG has two vertices, source v_1 and sink v_2. The first vertex corresponds to the $v_1.func(A, B, O) \equiv (O := A \times B)$ sub-operation and is bound to be executed on a matrix multiplication hardware accelerator (i.e., $v_1.PE = ACC$). The second vertex represents the $v_2.func(O, C) \equiv (O := O + C)$ sub-operation and is executed on the CPU ($v_2.PE$ = CPU). Note that edges e_1, e_2 and e_4 do not have a source vertex, meaning $e_1.s = e_2.s = e_4.s = \perp$; edge e_5 does not have a destination vertex, meaning $e_5.d = \perp$. To compute v_1, input data A and B needs to be moved from main memory to the SPM of the accelerator by the GDMA; hence, edges e_1 and e_2 represent global load operations. Similarly to compute v_2, the operand C needs to be moved from main memory to the SPM of the CPU by the GDMA (edge e_4). The operand O corresponds to data produced in output by $v_1.func(A, B, O)$ and thus it can be transferred from the accelerator's SPM directly into the target CPU's SPM by the LDMA; hence,

edge e_3 represents a local data transfer. Finally, the output of $v_2.func(O, C)$ needs to be written back from the CPU SPM to main memory through the GDMA; hence, edge e_5 represents a global unload operation.

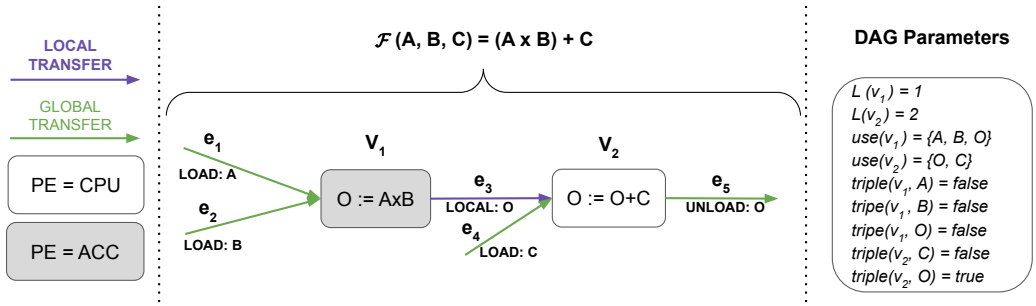

Fig. 4. DAG model for computational function $\mathcal{F}(A, B, C) = (A \times B) + C$

3.2 Reference Platform

We consider a multiprocessor system-on-chip (MPSoC) platform, composed of two main super-blocks: (1) an array of special-purpose user-defined accelerators, and (2) a set of M general-purpose application CPUs, divided into M partitions. An example is depicted in Figure 5. One key requirement in the proposed framework is the ability to move data close to the corresponding processing element before data items are consumed. This is in match with the typical processing model of specialized accelerators, where data is first moved to either accelerator's local memory or to an input FIFO, and then the accelerator accesses the local resource.

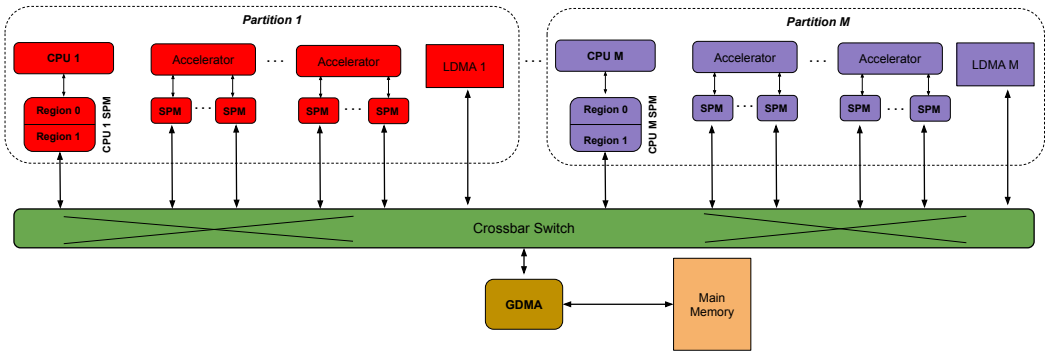

Fig. 5. Block Diagram of Reference Platform

Thus, we assume that each accelerator features one or more local SPMs. Each SPM stores a set of data buffers, which are used to receive data elements the accelerator reads from and store data elements it writes to. An accelerator can access a data buffer either randomly (through a memory interface) or sequentially (through a FIFO streaming interface). All the SPMs are byte-addressable, memory-mapped, and accessible by DMA engines. We also assume that the SPMs are dual-ported, so that the accelerator can read/write from/to a data buffer while a DMA engine accesses a different data buffer in the same SPM. While not required, we are particularly interested

in supporting development workflows where custom accelerators are designed through HLS. This is not a problem because two major FPGA manufacturers, Xilinx and Altera, provide such tools through their respective Vivado HLS and Intel HLS tools.

In order to incorporate CPU processing in our model, we also assume that each CPU is associated a private SPM. While general-purpose CPUs do not always have local SPMs, we consider platforms in which user-defined hardware blocks can be instantiated, for example on FPGA. Following the three-phase model, all instructions and data used by a CPU while executing a task segment must be contained in the CPU's SPM. Similarly to the accelerators' SPMs, the CPUs' SPMs are also dual-ported and accessible by DMA engines. In addition, each CPU's SPM is divided into two regions, so that the CPU can execute a task from one region while a different task is loaded in the other region. Each region contains multiple data buffers for the data elements accessed by the CPU.

Access to main memory is performed through a Global DMA engine (GDMA). The GDMA is responsible for moving data (i) from/to main memory (ii) to/from an accelerator's local SPM or a CPU's local SPM. Furthermore, we assume that each partition is associated with a Local DMA (LDMA) engine that can be used to move data directly between two SPMs within that partition, either the CPU's SPM and an accelerator's SPM or the SPMs of two different accelerators. Only one GDMA transfer can be carried out at a time. However, since each processing element is statically assigned to one partition, multiple LDMAs can be operated in parallel with the GDMA as long as the LDMA transfers target different SPMs than the GDMA. We enforce such constraints by implementing a TDMA-based schedule of the DMAs, where the partition under analysis is assigned a slot of length σ every Σ time units. Specifically, a partition uses the GDMA during its assigned TDMA slot, while it uses its LDMA during the interval of duration $\Sigma - \sigma$ corresponding to slots assigned to other partitions. Finally, an interconnection is needed to connect the various components. We do not pose restrictions on the architecture of the interconnection, e.g. a crossbar, multi-bus, NoC, etc, as long as it does not restrict parallelism between GDMA and LDMA transfers.

4 TASK TRANSFORMATION AND AUTOMATIC THREE-PHASES TASK SYNTHESIS

In this section we describe how applications that follow the generic DAG-based model described in Section 3.1 can be transformed to execute on the platform described in Section 3.2 following the three-phase model [23, 25]. Recall that a task τ is expressed as a sequence of I iterations of the DAG processing blocks that correspond to the aforementioned $\mathcal{F}(X)$ function. The output of the transformation needs to be a sequence of streaming segments (see Section 2), each where some processing and the necessary data transfers are performed. The key idea is that we can treat vertexes and edges in the DAG as stages in a pipeline, so that each segment can execute multiple stages in parallel on different iterations.

Figure 6provides the result of the transformation into the three-phase model of the DAG in Figure 4, where the $\mathcal{F}(A, B, C) = (A \times B) + C$ function is repeated for a number of iterations $I = 4$. First note that when translating a generic task τ to execute on the considered platform (see Section 3.2), a first special segment, namely S0, needs to be introduced. Indeed, when a new job of τ is released, the very first step must be dedicated to loading the code itself of the task—Interval ①. Only once the code starts executing (Interval ②), the actual streaming—and hence the remaining segments—can be initialized and set in motion. Following our example, the first load is for operands $A^{(1)}$ and $B^{(1)}$ during Interval ③, where the superscript notation refers to the iteration number, from 1 to 4. This load allows the accelerator to perform the first matrix multiplication during segment S1—Interval ④. At the same time, during Interval ④, the operands $A^{(2)}$ and $B^{(2)}$ for the next iteration are loaded. Next, in Interval ⑤, the operation $(O^{(2)} := A^{(2)} \times B^{(2)})$ is performed on the accelerator. Simultaneously, a local transfer is performed to pass $O^{(1)}$ from the SPM of the

accelerator to the SPM of the CPU. Because there is no guarantee about when the local transfer will exactly occur during Interval ⑤, CPU processing can only start in the successive Interval ⑥. To allow the operation ($O^{(1)} := O^{(1)} + C^{(1)}$) to be carried out on the CPU during Interval ⑥, the operand $C^{(1)}$ is also loaded in Interval ⑤. The final output, i.e. the result of $\mathcal{F}(A^{(1)}, B^{(1)}, C^{(1)})$ is written back to main memory during the unload operation depicted in Interval ⑦. The same sequence of operations applies until Interval ⑨ is reached where the last ($O^{(4)} := O^{(4)} + C^{(4)}$) operation is performed on the CPU and its result is moved to main memory in Interval ⑩. Note that we assume that all load operations for a segment can complete within one TDMA slot of size σ; the same applies for the unload operations. Similarly, all local transfers must complete within one window of length $\Sigma - \sigma$.

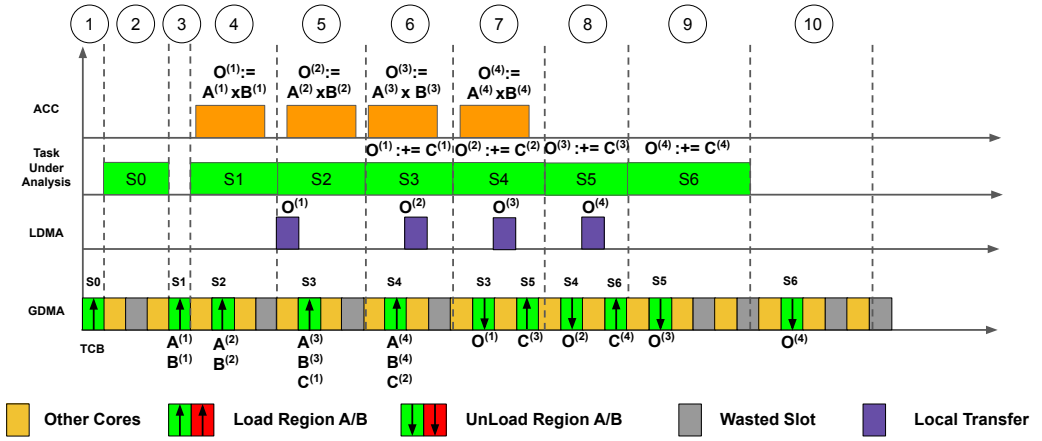

Fig. 6. Pipelined Streaming Execution Example.

In the following, we provide a general formulation for the transformation described above. We first determine the number of segments $\mathcal{S}$ for the task, and then specify the operations carried out in each segment. For ease of explanation, in this section we adopt an abstract model, where each segment is associated with a list of operations, and we use function $append(s, op)$ to denote adding operation op to the end of the list for segment index s. However, in Section 5 we will then show how operations are directly translated to actual API calls invoked by the CPU based on our OS-level support layer. Therefore, our proposed algorithm can be used to automatically generate the CPU code for each task, including allocating buffer, invoking the execution of each processing function, and scheduling data movements.

More in details, we consider the following operations: (i) $v.func(params)$ denotes the execution of processing function $v.func(\cdot)$ of CPU-bound vertex v. Here, $params$ is a set of pointers to buffers allocated in the CPU SPM that correspond to the function parameters. (ii) $execute(v.PE, params)$, where $v.PE$ is an accelerator. In this case, $params$ is a set of IDs of buffers allocated in the accelerator's SPM(s) to hold the function parameters. (iii) $transfer\_local(spm\_source, spm\_dest)$ denotes a local transfer between source SPM buffer spm_source and destination SPM buffer spm_dest. (iv) $transfer\_load(mem\_source, spm\_dest)$ denotes a global load between main memory address mem_source and SPM buffer spm_dest; and (v) $transfer\_unload(spm\_source, mem\_dest)$ denotes a global unload operation.

Finally, the placement of the $transfer\_$ operations must be carefully considered. CPU and accelerator execution are under the control of the task, but the same is not true for GDMA and

LDMA transfers: both DMAs must respect the TDMA schedule, and furthermore as we will detail in Section 5, the schedule of DMA transfers must be modified in case of task preemption. Hence, in our system memory transfers are controlled by the scheduler in the OS. As a consequence, for the OS to know which transfers must be performed during a scheduling interval, the task must issue transfer operations during the segment executed in the *previous* interval. Referring to Figure 6 as an example, the transfers of $C^{(1)}, A^{(3)}$ and $B^{(3)}$ performed in Interval ⑤ must be programmed in segment S1 during Interval ④. Note that no segment of the task under analysis is executed in Interval ③; hence, the transfers performed during both Interval ③ and Interval ④ must be programmed in S0. For this reason, the special segment S0 supports two lists of transfer operations: we use index $s = -1$ to refer to the list of transfers performed during Interval ③, and $s = 0$ for the list of transfers performed in Interval ④.

4.1 Pipelining and Number of Segments

We begin by determining the number of segments $\mathcal{S}$ required by the task. As shown in the example in Figure 6, every data dependency between two vertices in the DAG adds two stages to the processing pipelining: one stage for the local data transfer, and another stage for the execution on the dependant vertex. More in general, the number of stages depends on the maximum length of any path in the DAG. Hence, let us use $L(v)$ to denote the level of vertex v, that is, the length of the longest path in number of vertexes between any source and v included; formally, $L(v) = 1$ if v is a source, otherwise $L(v) = 1 + \max_{v' \in pred(v)}\{L(v')\}$. The number of segments is then equal to one (for Segment S0), plus the number of iterations I, plus 2 additional segments for each vertex level past the first:

$$\mathcal{S} := 1 + I + 2 \cdot (\max_{v \in V}\{L(v) - 1\}). \tag{1}$$

Following the same logic, the i-th instance of processing function $v.func(\cdot)$ is executed in segment index s, with $s = i + 2 \cdot (L(v) - 1)$.

Example: note that in Figure 6, where $L(v_1) = 1$ and $L(v_2) = 2$, $v_1.func(A, B, O)$ is executed in segments S1-S4, while $v_2.func(O, C)$ is executed in segments S3-S6. The total number of segments is $1 + 4 + 2 \cdot (2 - 1) = 7$.

4.2 SPM data buffering

At compile time and for each vertex v, buffers are allocated at static addresses in the SPM of $v.PE$ to hold the data elements used by processing function $v.func(\cdot)$ [24]. Each buffer is associated with a numeric ID. As discussed in Section 2, previous work used a double-buffering approach, where the current segment executes using the data in one buffer, while the other buffer is used for transfer operations.

More in general in our model, the number of buffers required by a data element $a \in use(v)$ depends on the number of transfers for a. If v has only either one incoming edge or one or more outgoing edges for a, then double buffering is always sufficient. However, if v has both one incoming and one or more outgoing edges for a, then the order of data transfers must be carefully considered. First, consider the case where the edges represent global transfers. Then, double buffering can still be used as long as a is unloaded from v, thus leaving the transfer buffer free, before it is loaded in the same buffer. The same consideration applies when all edges represent local transfers, as long as outgoing transfer(s) from v to the successor vertex(es) are performed before the incoming transfer from the predecessor vertex to v. As we will show in Section 5.3, our scheduling logic can indeed guarantee that such transfer order is respected, with the exception of any vertex v bound to an accelerator that receives an incoming local transfer from a CPU-bound predecessor. In such a case, triple buffering is required to avoid overwriting the current transfer

buffer. Similarly, note that the relative order of global and local transfers cannot be guaranteed: this is because they are performed in different TDMA slots which are not synchronized with the interval start/end times. Hence, if an incoming edge calls for a local transfer while an outgoing edge calls for global transfer or vice-versa, then triple buffering is also required. Formally, we use predicate $triple(v, a)$ to determine whether vertex v requires triple buffering for data element $a \in use(v)$: $triple(v, a) \Leftrightarrow \exists e', e'' \in E : e'.d = e''.s = v \wedge e'.a = e''.a = a \wedge \big((v.PE \neq CPU \wedge e'.s \in E \wedge e'.s.PE = CPU) \vee (e'.s = \perp \wedge e''.d \in E) \vee (e'.s \in E \wedge e''.d = \perp)\big)$.

Example: for the example in Figure 4, it holds $triple(v_2, O) = true$, because $v2$ has both an incoming local transfer for O, and an outgoing unload. All other data elements and vertexes can use double-buffering; in particular, note that for the same data element O, it holds $triple(v_1, O) = false$.

We use function $mem(a)$ to denote the address of data element a in main memory; we use function $spm\_id(v, a, k)$ and $spm\_addr(v, a, k)$ to denote the ID and the SPM address, respectively, of the k-th SPM buffer of a for v. Following the discussion above, if $triple(v, a) = true$, then the index k ranges from 1 to 3, otherwise k ranges from 1 to 2. For the first iteration, i.e. $a^{(1)}$, buffer index $k = 1$ is used; for the second iteration $a^{(2)}$, buffer index $k = 2$ is used; while in general for the i-th iteration $a^{(i)}$, buffer $k = index(v, a, i)$ is used, where $index(v, a, i) \equiv (i - 1)\%3 + 1$ if $triple(v, a) = true$, and $index(v, a, i) \equiv (i - 1)\%2 + 1$ otherwise.

4.3 Graph Transformation

Before we list the operations in each segment, a final graph processing step might be required to avoid increasing buffer space. Specifically, consider Figure 7(i), which depicts the DAG for an example computational function with three vertices. Here, vertex v_1 transfers data element a_3 to v_3; however, we have $L(v_1) = 1$ and $L(v_3) = 3$, meaning that v_1 and v_3 are not consecutive processing stages in the pipeline. This creates a problem with the local transfer of a_3 between v_1 and v_3. Consider for example the first iteration $a_3^{(1)}$: here, v_1 executes on $a_3^{(1)}$ in segment S1, while v_3 executes on $a_3^{(1)}$ in segment S5. If we perform the local transfer of $a_3^{(1)}$ in parallel with segment S2, v_3 needs two extra buffers for a_3; if we perform it during S4, instead v_1 needs two extra buffers; and if we perform it during S3, both v_1 and v_3 require one extra buffer.

To avoid such complexity, for each local transfer edge $e : e.s, e.d \in E$ such that $L(e.d) > L(e.s) + 1$, we perform a graph transformation. Specifically, we remove e and substitute it with two edges e', e'' with $e'.s = e.s, e'.d = \perp, e''.s = \perp, e''.d = e.d, e'.a = e''.a = e.a$; basically, we change the local transfer with a unload and a load to main memory. Figure 7(ii) shows the transformed graph from Figure 7(i). Note that to remain consistent, the definition of $triple(v, a)$ must be applied to the transformed graph.

4.4 Three-phase Task Synthesis

We can now present Algorithm 1 that generates the segments for computational function $\mathcal{F}$ given a number of iterations I. The key idea is to process the vertices of $\mathcal{F}$ in reverse topological order (i.e., starting from the sinks); then for each vertex we add operations for executions, outgoing local transfers, unloads, and then loads in this order to each corresponding segment list. There is no need to add calls for incoming local transfers because the data transfer will be added as an outgoing transfer when processing one of the immediate predecessors of the vertex later on. The reverse topological order ensures that for each vertex, outgoing local transfers are performed before incoming ones; as discussed in Section 4.2, this is required to support double buffering.

For each vertex v, the algorithm enumerates all I iterations. For the i-th iteration, the segment index s where $v.func(\cdot)$ executes is computed on Line 4 as discussed in Section 4.1. On Lines 5-14, the execution of $v.func(\cdot)$ is added to the operation list for segment s. Note that the *params*

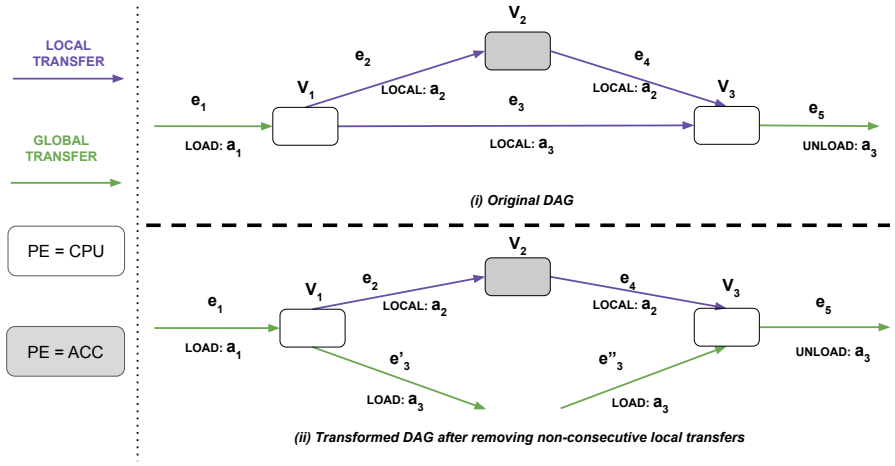

Fig. 7. Example DAG Transformation. (i): Original DAG. (ii): Transformed DAG after removing non-consecutive local transfers.

set is constructed by listing all data elements in $use(v)$, and employing functions $index(v, a, i)$ and $spm\_id(v, a, k_a)$ (resp., $spm\_addr(v, a, k_a)$) to determine the buffer ID (resp., buffer address) where $a^{(i)}$ is located in the accelerator's or CPU's SPM. In the same manner, Lines 15-27 add to the operation list outgoing local transfers, unloads and loads. As discussed at the beginning of this section, transfers must be programmed in the segment before the interval in which they are performed. Hence, outgoing transfers and unloads are appended to segment s at Lines 18 and 22, while loads to segment $s-2$ at Line 26. Specifically, note that for $L(v) = 1, i = 1$, we obtain $s = -1$ (**example:** loads of $A^{(1)}$ and $B^{(1)}$ in Interval ③ in Figure 6), while for $L(v) = 1, i = 2$, we obtain $s = 0$ (loads of $A^{(2)}$ and $B^{(2)}$ in Interval ④): these are the loads operations programmed during S0 to start the pipeline.

5 OS SUPPORT AND SCHEDULING

In this section, we discuss how the proposed pipelined, streaming model detailed in Section 4 can be realized at the OS level by modifying the programming interface and scheduling logic introduced in [23]. We then show how to test the schedulability of the resulting system on a given core under analysis.

5.1 Streaming API

Table 1 summarizes the proposed API to handle the segment streaming on both CPU and hardware accelerators, together with the corresponding operations used in Section 4. Note that there is no equivalent API call for the execution of $v.func(\cdot)$ on a CPU, since it simply corresponds to a user-level function call. The RTOS tracks the buffers used by each task through the use of a streaming table (ST). An entry in ST is generated when `allocate_buffer` is called in segment S0 and a buffer ID is returned.

Similar to [23], the RTOS manages DMA transfers of behalf of a task through the use of two different queues: *Streaming Wait Queue (SWQ)* and *Streaming Dispatch Queue (SDQ)*. We use two separate queues so that transfer requests made by the current segment can be enqueued in the SWQ, while the OS processes requests from the SDQ. The scheduler, as will be shown in Section 5.3, and the `dispatch` function move the requests from the SWQ to the SDQ. Specifically, the `dispatch` function informs the OS that the transfers issued so far are needed by the next segment; it is

Algorithm 1 Automatic Segments Code Generation

```
1: procedure GenerateSegments(F, I)
2:     for v ∈ V in reverse topological order do
3:         for i = 1; i ≤ I; i ← i + 1 do
4:             s ← i + 2 · (L(v) − 1)
5:             params_acc ← ∅
6:             params_cpu ← ∅
7:             for a ∈ use(v) do
8:                 k_a ← index(v, a, i)
9:                 params_acc ← params_acc ∪ spm_id(v, a, k_a)
10:                params_cpu ← params_cpu ∪ spm_addr(v, a, k_a)
11:            end for
12:            if v.PE = CPU then                                   ▷ Execution
13:                append(s, v.func(params_acc))
14:            else
15:                append(s, execute(v.PE, params_cpu))
16:            end if
17:            for e ∈ E : e.s = v ∧ e.d ∈ E do                     ▷ Local Transfer
18:                k_s = index(v, e.a, i)
19:                k_d = index(e.d, e.a, i)
20:                append(s, transfer_local(spm_id(v, e.a, k_s), spm_id(e.d, e.a, k_d)))
21:            end for
22:            for e ∈ E : e.s = v ∧ e.d =⊥ do                      ▷ Unload
23:                k_s = index(v, e.a, i)
24:                append(s, transfer_unload(spm_id(v, e.a, k_s), mem(e.a)))
25:            end for
26:            for e ∈ E : e.s =⊥ ∧e.d = v do                       ▷ Load
27:                k_d = index(v, e.a, i)
28:                append(s − 2, transfer_load(mem(e.a), spm_id(v, e.a, k_d)))
29:            end for
30:        end for
31:    end for
32: end procedure
```

used to separate the load operations in the first list of S0 ($s = -1$ in Algorithm 1), which are needed in S1, from the load operations in the second list of S0 ($s = 0$), which are needed in S3. The `end_segment` function informs the OS that the current segment has completed execution; if there is still any pending GDMA or LDMA transfer in the current interval, or any accelerator has not finished executing, the CPU is suspended until all GDMA and LDMA transfers complete and all accelerators finish executing. We assume that the GDMA and LDMA generate an interrupt when they finish performing a transfer; similarly, each accelerator generates an interrupt when it finishes executing. Then, the OS scheduler is invoked. Finally, the `wait` function marks the end of the last segment of the task. In our design, we also support polling to determine the completion status of the accelerators.

As it will become clear in Section 5.3, an accelerator cannot be used concurrently by multiple tasks. Therefore, for simplicity we assume that each accelerator is accessed by a single task. Such assumption could be relaxed to allow multiple tasks within the same partition to share an accelerator by implementing a locking API, so that a job acquires all required accelerators upon starting and releases them upon completion. The schedulability analysis would then need to be modified to

incorporate the blocking time induced by locking. We discuss how the locking protocol could be implemented and its impact on the analysis in Section 5.4.

Table 1. Proposed streaming API.

API	Operation	Description
buffer_id = allocate_buffer(uint64_t *address);	-	address is a static address in the SPM of either the CPU or an accelerator
execute_acc(int acc_id, int id1, ...)	*execute*	acc_id is the unique ID of the accelerator id1, ... is a list of buffer IDs on which to execute the accelerator
load_buffer(int id, uint64_t *src, int size);	*transfer_load*	Reads size in bytes from the src address in main memory and writes at ID buffer in the SPM of CPU or accelerator
unload_buffer(int id, uint64_t *dst, int size) ;	*transfer_unload*	Writes from the ID buffer in the SPM to the dst address in main memory
transfer_local(int src_id, int dst_id, int size);	*transfer_local*	Transfer from src_id buffer to dst_id buffer
dispatch();	-	Force all buffer DMA requests to move from waiting queue to dispatch queue
end_segment();	-	End segment execution
wait();	-	End job and wait until the next task activation

5.2 Example: $\mathcal{F}(A, B, C) = (A \times B) + C$

For added clarity, Listing 3 details the generated code for the example introduced in Section 4. We show the original code of a software-only, non-streaming implementation of the task in Listing 2. Here, `matrix_multiply` implements the $O := A \times B$ processing function in v_1, while `matrix_sum_inplace` implements the $O := O + C$ processing function in v_2. In the code in Listing 3, constants `ACC_A1` to `CPU_O3` represent the static address of each buffer in the SPM.

Listing 1. Preamble code for $\mathcal{F}(A, B, C) = (A \times B) + C$ operation.

```
#define I 4
#define ROWS 64
#define COLS 64
#define SZ (sizeof(float))

float A[I][ROWS][COLS];  /* Input Matrix */
float B[I][ROWS][COLS];  /* Input Matrix */
float C[I][ROWS][COLS];  /* Input Matrix */
float O[I][ROWS][COLS];   /* Output Matrix */
```

Listing 2. CPU code for task implementing $\mathcal{F}(A, B, C) = (A \times B) + C$ operation.

```
/* Preamble code. See Listing @\ref{lst:preamble}@. */
int main() {
    while(true) {
        for (i = 0; i < I; ++i) {
            matrix_multiply(&A[i][0][0], &B[i][0][0], &O[i][0][0]);
            matrix_sum_inplace(&O[i][0][0], &C[i][0][0]);
        }
        wait();
    }
    return EXIT_SUCCESS;
}
```

Listing 3. CPU + Accelerator streaming code for task implementing $\mathcal{F}(A, B, C) = (A \times B) + C$ operation.

```
/* Preamble code. See Listing @\ref{lst:preamble}@. */
int main() {
    while(true) {
        /* Segment S0 Begin */
        acc_A1_id = allocate_buffer(ACC_A1); acc_A2_id = allocate_buffer(ACC_A2);
        acc_B1_id = allocate_buffer(ACC_B1); acc_B2_id = allocate_buffer(ACC_B2);
        acc_O1_id = allocate(ACC_O1);        acc_O2_id = allocate (ACC_O2);
        cpu_C1_id = allocate_buffer(C1);     cpu_C2_id = allocate_buffer(C2);
        cpu_O1_id = allocate_buffer(CPU_O1);
        cpu_O2_id = allocate_buffer(CPU_O2);
        cpu_O3_id = allocate_buffer(CPU_O3);
        load_buffer(acc_A1_id, &((uint64_t)(A[0][0][0])), ROWS*COLS*SZ);
        load_buffer(acc_B1_id, &((uint64_t)(B[0][0][0])), ROWS*COLS*SZ);
        dispatch();
        load_buffer(acc_A2_id, &((uint64_t)(A[1][0][0])), ROWS*COLS*SZ);
```

```
        load_buffer(acc_B2_id, &((uint64_t)(B[1][0][0])), ROWS*COLS*SZ);
        end_segment();

        /* Segment S1 Begin */
        load_buffer(cpu_C1_id, &((uint64_t)(C[0][0][0])), ROWS*COLS*SZ);
        execute_acc(mm_id_accelerator, acc_A1_id, acc_B1_id, acc_O1_id);
        transfer_local(acc_O1_id, cpu_O1_id, ROWS*COLS*SZ);
        load_buffer(acc_A1_id, &((uint64_t)(A[2][0][0])), ROWS*COLS*SZ);
        load_buffer(acc_B1_id, &((uint64_t)(B[2][0][0])), ROWS*COLS*SZ);
        end_segment();

        /* Segment S2 Begin */
        load_buffer(cpu_C2_id, &((uint64_t)(C[1][0][0])), ROWS*COLS*SZ);
        execute_acc(mm_id_accelerator, acc_A2_id, acc_B2_id, acc_O2_id);
        transfer_local(acc_O2_id, cpu_O2_id, ROWS*COLS*SZ);
        load_buffer(acc_A2_id, &((uint64_t)(A[3][0][0])), ROWS*COLS*SZ);
        load_buffer(acc_B2_id, &((uint64_t)(B[3][0][0])), ROWS*COLS*SZ);
        end_segment();

        /* Segment S3 Begin */
        matrix_sum_inplace(CPU_O1, CPU_C1);
        unload_buffer(cpu_O1_id, &((uint64_t)(O[0][0][0])), ROWS*COLS*SZ);
        load_buffer(cpu_C1_id, &((uint64_t)(C[2][0][0])), ROWS*COLS*SZ);
        execute_acc(mm_id_accelerator, acc_A1_id, acc_B1_id, acc_O1_id);
        transfer_local(acc_O1_id, cpu_O3_id, ROWS*COLS*SZ);
        end_segment();

        /* Segment S4 Begin */
        matrix_sum_inplace(CPU_O2, CPU_C2);
        unload_buffer(cpu_O2_id, &((uint64_t)(O[1][0][0])), ROWS*COLS*SZ);
        load_buffer(cpu_C2_id, &((uint64_t)(C[3][0][0])), ROWS*COLS*SZ);
        execute_acc(mm_id_accelerator, acc_A2_id, acc_B2_id, acc_O2_id);
        transfer_local(acc_O2_id, cpu_O1_id, ROWS*COLS*SZ);
        end_segment();

        /* Segment S5 Begin */
        matrix_sum_inplace(CPU_O3, CPU_C1);
        unload_buffer(cpu_O3_id, &((uint64_t)(O[2][0][0])), ROWS*COLS*SZ);
        end_segment();

        /* Segment S6 Begin */
        matrix_sum_inplace(CPU_O1, CPU_C2);
        unload_buffer(cpu_O1_id, &((uint64_t)(O[3][0][0])), ROWS*COLS*SZ);
        wait();
    }
    return EXIT_SUCCESS;
}
```

A potential downside of the presented code generation process is that it creates one code block for each segment, as shown in Listing 3. If the number of iterations I, and therefore the number of segments, is high, this can result in a large code footprint. In this case, it is possible to adopt an alternative code generation approach, where first the code of S0 is generated, and then each further segment executes the code of one iteration in a loop over $s = 1...\mathcal{S} - 1$. The same logic as in Algorithm 1 can be used, except that the expressions must be evaluated at run-time rather than at compile time. Specifically, at each iteration over s and for each vertex, we can compute the value of i by inverting the expression at Line 4 to obtain $i = s - 2 \cdot (L(v) - 1)$; then if $1 \leq i \leq I$, we compute the buffer indexes and issue the corresponding execution or transfer. We decided to describe Algorithm 1 and the example based on the loop-unrolled version of the generated code because we believe it is easier to understand.

5.3 Scheduling logic

We next discuss how the scheduling logic proposed in [23] must be modified to handle both GDMA and LDMA transfers for buffered data elements [3]. In the three-phase model, scheduling decision are made at the beginning of each interval. Specifically, the scheduler must decide which task to execute (if any is active) in the *next* interval, so that it can load the required data and code (if it is a different task than the one currently executed) in the current interval.

To understand the behavior of the DMA scheduler, we first present an example where other tasks co-execute together with a task under analysis. Figure 8 depicts the same schedule as in Figure 6, except that in this case, the scheduler decides to execute segments X, Y of some other task(s) during intervals Interval ⑦ and Interval ⑧ (we assume that such task(s) do not require local transfers). Note that again for simplicity, the figure does not include the TDMA of other cores. During Interval ⑥, the GDMA is used to load the code and data of X instead of the data for S4; the data of S4, namely $A^{(4)}$, $B^{(4)}$ and $C^{(2)}$, is instead loaded in Interval ⑧, just before S4 is executed in Interval ⑨. We also do not perform the local transfer for $O^{(2)}$ in Interval ⑥, because such transfer would move $O^{(2)}$ to the SPM of the CPU, which is needed by other task(s); similarly to load data, the transfer is moved to Interval ⑧. Note this means that the accelerator holds a buffer on behalf of the task under analysis during Interval ⑦ and Interval ⑧ while the task is preempted; hence, the accelerator cannot be used by other tasks. Finally, note that no unload operation takes place in Interval ⑨, because the data of S3 was already unloaded in Interval ⑦. If the task needed to perform a local transfer from CPU to accelerator, than the local transfer for S3 would similarly have to be performed in Interval ⑦ to free the CPU SPM.

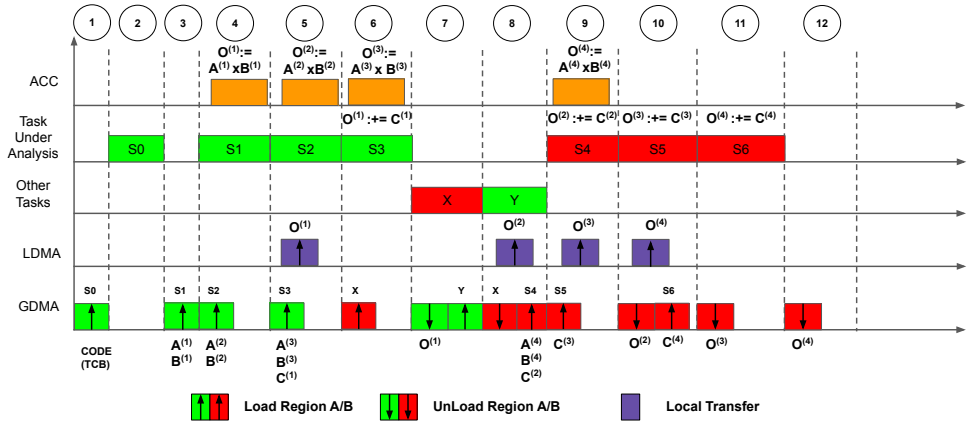

Fig. 8. Example: Task Preemption.

Algorithm 2 shows the resulting steps taken by the scheduler at the beginning of an interval, where we use S_i to denote the segment executed in Interval ⓘ, and $\tau(S)$ to denote the task to which segment S belongs. The scheduler starts by determining the segment S_{i+1} to be executed in the next Interval ⓘ+1 at Line 1. If a segment S_i is scheduled in Interval ⓘ (Line 2), then it moves all transfer requests in SWQ (if any) to SDQ of task $\tau(S_i)$ (Line 3); this ensures that S_i can place new transfer

[3]Note that apart from streaming buffers, the GDMA is also used to load the code and static data of the task, and to unload modified static data. For ease of presentation, we do not detail such operations since they follow the same logic presented in Figures 1, 3 for previous work.

requests in SWQ. The scheduler than processes all unload and local transfers from the CPU SPM for segment S_{i-1}, and all loads and transfers to the CPU SPM and between accelerators for segment S_{i+1}. Note that for the former case, the transfer requests are found in the SDQ if $\tau(S_{i-1}) = \tau(S_i)$, since in this case Line 3 moved the transfers in the SWQ of $\tau(S_{i-1})$ to the SDQ; otherwise, the transfer requests are still in the SWQ.

We assume that the send operation removes the selected transfers from the queue, and that it is guaranteed to perform the transfers in the same order in which they appear in the queue. Recalling the discussion in Section 4.2, based on the send order in the algorithm, it guarantees that unloads are performed before loads, and furthermore because of the inverse topological order, outgoing local transfers are performed before incoming local transfers, with the exception of accelerators receiving local transfers from the CPU given that CPU $\rightarrow$ ACC local transfers are sent first.

A final pair of notes regard the TDMA parameters, and the SPM utilization. Based on the described logic, the slot size σ must be large enough to perform all load transfers for largest task in Γ, which include at most one buffer for each data element that must be loaded from main memory for that task. σ must also be large enough to perform all unload transfers for any task, again including a single buffer per data element. However, we also require that the window $\Sigma - \sigma$ be large enough to perform all local transfers (again, on at most one buffer per data element) CPU $\rightarrow$ ACC for any task, plus all ACC $\rightarrow$ ACC and ACC $\rightarrow$ CPU local transfer for any task, either the same or a different one. Since most tasks have more code/data to load than data to unload, and $\Sigma - \sigma$ is generally larger than σ with a number of cores $M \geq 3$, the DMA schedule is typically constrained by load transfers.

Finally, we consider the SPM utilization. Since each accelerator is only used by one task at a time, an accelerator SPM only needs space for the buffers of any one task. However, the CPU SPM must be divided into two regions, one for the task executed in the current interval, and one for the task(s) executed in the previous/next interval. To ensure that each task can be executed from either region, relocation support is needed either at the compiler level [25], or in the form of a dedicated hardware component [27], or simply through virtual memory [7]; we assume the latter. In the simplest case, each region must be sized so that it contains all the code, static data and CPU-bound buffers for any one task. As discussed in [23], an improved solution can leverage the observation that the CPU SPM never contains the code/data for more than two consecutive segments. Hence, a region could be sized to contain only one buffer for data element, if it uses double buffering, or two buffers for data element, if it uses triple buffering; the remaining buffer could be allocated in the other region.

5.4 Schedulability Analysis

A schedulability analysis for the streaming model is presented in [23], under the assumption that tasks are scheduled according to fixed priorities; without loss of generality, let them be indexed based on priority, with τ_1 having the highest priority and τ_N the lowest. Since our model does not change the way terminal and streaming segments are scheduled, the same analysis can be employed; we summarize it below.

The analysis takes as inputs the length of each segment, which is computed as the maximum of its execution time and the worst-case memory time Δ in any interval. For our model, the execution time of each segment must be computed as the maximum between its CPU execution time and the execution time of any accelerator used during the segment. The worst-case scenario for Δ is when an interval starts right after the beginning of the TDMA slot, resulting in a GDMA time of $\sigma + 2 \cdot \Sigma$; note that since the LDMA only needs one consecutive time window to perform the local transfers, its worst-case time is $\Sigma - \sigma + \Sigma$, which is lower than the GDMA. If all cores use the same TDMA slot, i.e. $\Sigma = M \cdot \sigma$, this results in $\Delta = \sigma \cdot (2M + 1)$, as discussed in Section 2. We also compute Δ^{single} to be the memory time required for only load transfers or only unload transfers; based on

Algorithm 2 Scheduler logic in Interval i

```
1: Determine segment S_{i+1} (if any)
2: if a segment S_i is scheduled in Interval (i) then
3:     τ(S_i): Move transfers from SWQ to SDQ.
4: end if
5: if a segment S_{i-1} was scheduled in Interval (i-1) then
6:     if τ(S_{i-1}) = τ(S_i) then
7:         τ(S_{i-1}): Send unload transfers in SDQ to GDMA.
8:         τ(S_{i-1}): Send CPU → ACC local transfers in SDQ to LDMA.
9:     else
10:        τ(S_{i-1}): Send unload transfers in SWQ to GDMA.
11:        τ(S_{i-1}): Send CPU → ACC local transfers in SWQ to LDMA.
12:    end if
13: end if
14: if a segment S_{i+1} is scheduled in Interval (i+1) then
15:    τ(S_{i+1}): Send load transfers in SDQ to GDMA.
16:    τ(S_{i-1}): Send ACC → ACC and ACC → CPU local transfers in SDQ to LDMA.
17: end if
```

the same logic, we have $\Delta^{single} = \sigma \cdot (M+1)$. In the rest of the analysis, we use L_i to denote the sum of the lengths of all segments of task τ_i; l_i^{first}, l_i^{last} for the lengths of its first and last segment; and $l_i^{l\max}$ for the maximum of Δ and the length of any segment of a task with lower priority than τ_i.

The analysis computes an upper bound R_i to the time between the release of any job of task τ_i (the job under analysis) and the time when its last segment starts executing. R_i is computed using the following response time iteration:

$$R_i = L_i - l_i^{last} + \text{Inter}_i(R_i) + B_i^{first} + l_i^{l\max}, \tag{2}$$

where: (i) $L_i - l_i^{last}$ represents the time required to execute all segments of τ_i except the last. (ii) $\text{Inter}_i(R_i) = \sum_{j=1}^{i-1} \lceil R_i/T_j \rceil \cdot L_j$ bounds the interference caused by higher priority tasks. (iii) $B_i^{first} = 2 \cdot l_i^{l\max}$ bounds the blocking time suffered by the first segment of either τ_i or a higher priority task executed after the arrival time of the job under analysis. In the worst case, the job under analysis arrives just after the beginning of an interval where the segment of a lower priority task executes; then, since the segment executed in an interval is decided at the beginning of the previous interval, another lower priority segment can execute in the following interval, resulting in a blocking time of $2 \cdot l_i^{l\max}$. (iv) The last term $l_i^{l\max}$ represents the blocking time suffered by segments of the job under analysis that follow a terminal segment, which in our model is only S1. This is because S1 cannot be executed directly after S0; hence, in the worst case a segment of a lower priority task could instead be executed in between the two segments of the job under analysis (note that this can happen in Interval ③ in Figure 8). The value of R_i can then be used to check the schedulability of the task set. Specifically, if the last segment of τ_i outputs data, then such data will be unloaded to main memory no later than $l_i^{end} + \Delta^{single}$ time after the last segment of the job under analysis starts executing. Therefore, assuming we require such operation to complete by the deadline, the following is a sufficient schedulability condition: $\forall i = 1...N$, the iteration in Equation 2 converges to fixed point R_i such that $R_i + l_i^{end} + \Delta^{single} \leq D_i$.

As noted in Section 5.1, to allow an accelerator to be used by multiple tasks, we would need to implement a suitable locking protocols. For example, we could employ the well-known priority ceiling protocol [21], which for non-self-suspending tasks guarantees that the maximum blocking time is equal to the length of a single lower-priority critical section. Specifically, we propose to

treat each accelerator as a shared resource protected by an individual mutex; to avoid deadlock in the case a task uses multiple accelerators, we require the task to lock all employed accelerators at the same time in a fixed order. We discuss two possible ways in which the protocol could be implemented, depending on when the lock(s) is acquired.

Lock before executing S1; the mutex is then unlocked once the last segment finishes executing. Since the lock is acquired after S0, the length of the critical section of a lower priority task τ_j is equal to $L_j - l_j^{first}$. The blocking time B_i^{lock} for the job under analysis is then equal to the maximum value of $L_j - l_j^{first}$ over all lower priority tasks that lock an accelerator with ceiling higher or equal to the priority of τ_i. The drawback with this approach is that the job under analysis effectively self-suspends between the execution of its segments S0 and S1; hence, a lower priority job executed in between those segments could re-acquire a lock and again block the job under analysis. Hence, τ_i suffers a maximum locking-induced blocking time equal to $2 \cdot B_i^{lock}$.

Lock before executing S0. This ensures that no lower priority task can execute between segments S0 and S1 of the job under analysis, hence τ_i can only be blocked by one critical section. In addition, term $l_i^{l\max}$ in Equation 2 can be replaced by Δ^{single}, since in the worst case the interval between S0 and S1 is occupied by load transfers (see again Figure 8). However, for the same reason, in the worst case no segment can execute between segments S0 and S1 of all other jobs; hence, the blocking time B_i^{lock} must now be computed as $L_j + \Delta^{single}$, and similarly in the computation of $Inter_i(R_i)$, term L_j must be replaced with $L_j + \Delta^{single}$. The resulting iteration for R_i is thus:

$$R_i = L_i - l_i^{last} + \sum_{j=1}^{i-1} \lceil R_i/T_j \rceil \cdot \left(L_j + \Delta^{single}\right) + \max\left(B_i^{first}, l_i^{l\max} + B_i^{lock}\right) + \Delta^{single}. \tag{3}$$

Note that the initial blocking term is now computed as the maximum of B_i^{first} (in case no locking-induced blocking occurs) and $l_i^{l\max} + B_i^{lock}$: in the worst case, the job under analysis can again arrive just after the beginning of an interval where the segment of a lower priority task executes and segment S0 of the job causing lock-induced blocking is loaded.

6 IMPLEMENTATION

We implemented an instance of the reference platform discussed in Section 3.2 based on a Xilinx Zynq UltraScale+ MPSoC, specifically using the ZCU102 development board. The board is based on a XCZU9EG SoC featuring a processing system (PS) that includes all the hard logic and a block of tightly-coupled Programmable Logic (PL) implemented with FPGA technology. Figure 9 shows a block diagram of the platform, configured to execute the example $\mathcal{F}(A, B, C) = (A \times B) + C$ task on each core.

The PS includes a four-core ARM Cortex A53, a dual-core ARM R5, as well as a Global direct memory access (GDMA) engine and main memory (PS DRAM). We assume that one Cortex A53 application core is dedicated to non-real-time operations. Therefore, in our instantiation we consider $M = 3$ cores using the remaining A53. GDMA and local direct memory access (LDMA) engines scheduling responsible for global/local load/unload operations is implemented in one of the R5 processors. The PL is used to implement the LDMAs, the accelerators, and all SPMs; specifically, it includes about 3 MB of block ram (BRAM) cells that can be used to synthesize SPMs. All SPMs are double-ported, where each port is accessed through a dedicated SRAM controller. The A53 cores share a Last-Level Cache (LLC), but otherwise execute using code and data in their private SPMs. The R5 core has its own local Tightly-Coupled Memory (TCM).

Multiple interfaces exist between the processing system (PS) and the PL. In our implementation, we use two High-Performance Master (HPM) interfaces in the Full-Power Domain (FPD), namely

HPM0 and HPM1, and a single HPM port in the Low-Power Domain (LPD), referred to as LPD for simplicity. The LPD interface is used by the GDMA to transfer data from/to main memory to/from any SPM implemented in the PL. Conversely, the LPD port is used for configuration commands towards any memory-mapped PL IP, i.e., LDMAs and accelerators. It is also possible to program the LDMA and configure the IP using the HPM port. On the PL, we employ an Advanced eXtensible Interface (AXI) interconnection [29] to which all DMAs and SPMs are connected. Such interconnection does not limit parallelism between GDMA and LDMA transfers since it employs a crossbar architecture.

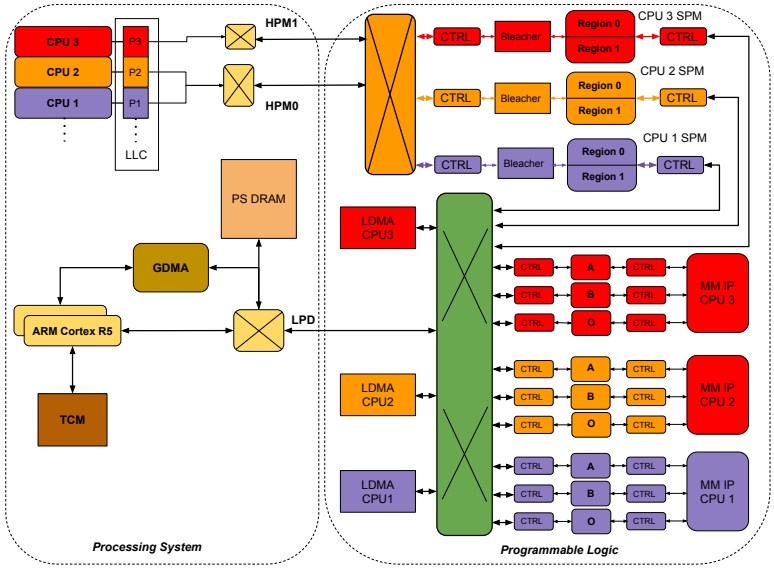

Fig. 9. Block Diagram of the proposed hardware design.

Following the considered use-case, we have implemented a Matrix-Multiplication (MM_IP) accelerator through High-Level Synthesis (HLS) and have generated an IP that provides AXI buses to read input data from SPM and write output results. As shown in Figure 9, each MM_IP is provided with three dual-ported SPMs, one for each data elements A, B, and O. This allows simultaneous access to all data elements. Since double-buffering is sufficient for all data elements, each SPM includes two buffers. Each IP has a control interface that is used for configuring and starting the IP.

In the software stack, we use the Jailhouse hypervisor to partition the shared resources [16]. Jailhouse provides cache partitioning through page coloring to the guest RTOSes running on top of it, performs code/data relocation for the load and unload phases of the three-phase model. Coloring results in non-contiguous physical addresses being assigned to the guest RTOSes and would normally force accessing SPM memories with a stride. To prevent wasting already limited SPM space, static cache bleaching [7, 19] is performed on accessed performed by the CPUs towards SPM addresses. The API described in Section 5.1 and related scheduling techniques were implemented in the Erika Enterprise RTOS version 3. The Erika RTOS is open-source and OSEK/VDX certified [6]. Section 7.1 shows the evaluation of the API implementation in terms of time and memory footprint.

Table 2. API OS Overhead.

API	WCET (ns)	Code (Bytes)	AVG	API	WCET (ns)	Code (Bytes)	AVG
allocate_buffer	949	76	46.24	*execute_acc*	210	100	200.05
load_buffer	798	400	178.58	*unload_buffer*	798	400	179.03
transfer_local	798	400	178.58	*dispatch*	767	112	197.32
end_segment	1565	284	79.11	*Streaming Wait Queue (SWQ)*	-	16	-
Streaming Table (ST)	-	248	-	*Streaming Dispatch Queue (SDQ)*	-	16	-

7 EVALUATION

7.1 OS API Overhead

We have compiled and measured the worst-case execution times of the proposed API functions in Erika Enterprise RTOS [6] version 3 release 55 on top of the Jailhouse hypervisor running on the Xilinx UltraScale+ ZCU102 MPSoC platform [28]. The segmented code of the application tasks that includes the API calls is produced following Algorithm 1. It is then compiled and linked together with the rest of Erika RTOS using the Gcc compiler version 7.2.1 for ARM64 architecture with the `-Os` flag.

Table 2 shows the obtained worst-case execution times of the proposed API functions and tables and the respective memory footprint. Erika does not support dynamic memory allocation, so we must statically define the number of DMA requests and consequently the queue elements that can be added into the queues. This number was defined to be 5 per task. We also consider that each task has a maximum of 5 streaming buffers (one streaming table entry per each buffer). At compile time, from the segmentation, we can retrieve the exact number of DMA requests and table entries, so this can be defined statically without compromising the system. In total, the memory footprint of the proposed API functions and tables is around 4096 bytes.

We ran each function 1000 times on the target platform, and collected the worst-case execution time (in nanoseconds) from these repetitions. Time was measured using a timer from Erika (`osEE_aarch64_gtimer_get_ticks` function). We used the obtained WCET to inflate the CPU execution time of each benchmark in the schedulability evaluation of Section 7.3. The API overhead has minimal impact on the schedulability ratio of the system; with respect to the results in Figure 10, removing the overhead improves the percentage of schedulable task sets by at most 0.4%.

7.2 Benchmarking $(A \times B) + C$

For our benchmarks we chose the same $(A \times B) + C$ kernel considered throughout the paper with matrix sizes of $64x64$, $80x80$, $96x96$, $112x112$, and $128x128$. To construct a comparison baseline, we first measure the execution time of the kernel executing from the SPM on the CPU without acceleration. Next, we compare to the case where matrix multiplications MM (i.e., sub-operation $O := A \times B$) are performed on hardware accelerator implemented in the PL via HLS. Conversely, matrix addition MAdd (i.e., sub-operation $O := O + C$) is always performed on the CPU. For our CPU-only implementation of MM (baseline), we use a tiling implementation for better cache locality since the considered matrix sizes are larger than the cache size in our platform [14]. For our hardware accelerators, we employ the matrix multiplication IPs generated by Vivado HLS [22]. For MM, Table 3 reports the observed runtimes for transfers and computational block for both CPU-only (Column 6) and accelerator (Column 7) implementations. We also report the total size of input (Column 2) and output (Column 4) operands in bytes. Whereas, Column 3 (resp., Column 5) shows the time required to move the matrices from main memory to the SPM using the GDMA (resp., from SPM to SPM using a LDMA). To run each of these kernels, (i) in the CPU-only case all the three matrices are loaded from main memory to the CPU's SPM; (ii) in the CPU + accelerator

case two matrices (A, B) are loaded from main memory to the accelerator's SPM while a third matrix (C) is loaded from main memory to the CPU's SPM. As such, regardless of the implementation, each time the kernel is invoked, three matrices need to be loaded from main memory. As highlighted in green in Table 3, the time to load a single matrix with the largest size is $146.02/2 = 73.01$µs. Thus we consider a slot size $\sigma = 3 \cdot 73.01 = 219.03$µs to be large enough to fully load all the necessary operands using the GDMA. The whole design on the PL was synthesized to run at 300 MHz.

Table 3. Run/Transfer Time of AxB + C with different Matrix Sizes

Benchmark	Input Size Bytes	Input Transfer Time (µs)	Output Size (Bytes)	Output Transfer Time (µs)	CPU Time (µs)	Accelerator Time (µs)	Ratio
MM Two (64x64) matrices	32768	LDMA = 28.98 GDMA = 38.5	16384	LDMA = 15.1 GDMA = 20.58	571.05	81.53	7.00
MAdd Two (64x64) matrices	32768	LDMA = 28.98 GDMA = 38.5	16384	LDMA = 15.1 GDMA = 20.58	36.91	-	-
MM Two (80x80) matrices	51200	LDMA = 45.15 GDMA = 60.15	25600	LDMA = 22.58 GDMA = 30.08	1133.12	128.98	8.79
MAdd Two (80x80) matrices	51200	LDMA = 45.15 GDMA = 60.15	25600	LDMA = 22.58 GDMA = 30.08	51.45	-	-
MM Two (96x96) matrices	73728	LDMA = 63.55 GDMA = 83.63	36864	LDMA = 31.78 GDMA = 41.82	1963.27	185.28	10.60
MAdd Two (96x96) matrices	73728	LDMA = 63.55 GDMA = 83.63	36864	LDMA = 31.78 GDMA = 41.82	74.92	-	-
MM Two (112x112) matrices	100352	LDMA = 86.5 GDMA = 113.83	50176	LDMA = 43.25 GDMA = 56.92	3115.99	251.83	12.37
MAdd Two (112x112) matrices	100352	LDMA = 86.5 GDMA = 113.83	50176	LDMA = 43.25 GDMA = 56.92	99.46	-	-
MM Two (128x128) matrices	131072	LDMA = 111.81 GDMA = 146.02	65536	LDMA = 56.49 GDMA = 74.34	4669.64	315.11	14.82
MAdd Two (128x128) matrices	131072	LDMA = 111.81 GDMA = 146.02	65536	LDMA = 56.49 GDMA = 74.34	142.98	-	-

7.3 Schedulability

Finally, we evaluate the improvements in terms of schedulability by executing the discussed AxB + C kernel on either the CPU only (*cpu* in Figures 10-12, using the same approach as in [23]), or on the CPU + Accelerator (*acc*). In both cases, schedulability is assessed using the test in [23], as discussed in Section 5.4, considering the task set allocated on one application core. As in the example in Section 4, we assume that the kernel is repeated 4 times.

For a given system utilization U (x-axis), we randomly generate 10,000 synthetic task sets as follows: first, we pick the number of tasks in the task set in the range $[5, 15]$. Each task is randomly assigned to one of the matrix sizes (from 64x64 to 128x128) in Table 3. Then, we uniformly generate the utilization u_i of each task in the task set [1], such that the sum of the tasks' utilizations is equal to U. Finally, each task is assigned a period $T_i = e_i/u_i$, where e_i is the sum of the execution times of all segments of the task when running on the CPU only. We define the utilization based on the execution time of segments, and not their length as used in the analysis, because the segment length depends on the memory time Δ, while the execution time is independent of such parameter and thus allows us to better compare schedulability when varying the transfer times based on the DMA speed.

Figure 10 shows the results for the ratio of schedulable task sets for both *cpu* and *acc* runs as we increase the system utilization, using the transfer times in Table 3. To evaluate multiple mixes of applications, we configure task generation to consider all the possible sizes measured in Table 3 (case "64–128") or only a subset of them, e.g. only matrices with sizes 96x96 through 128x128 for the curve labeled as "96–128". Following our implementation in Section 6, we consider a system

with $M = 3$ application cores, and for simplicity we assume that the same slot size $\sigma = 219.03\mu s$ is used for all cores. This results in a value $\Delta = \sigma \cdot (2M - 1) = 1.533ms$. We observe that schedulability for *cpu* improves when we consider only the larger matrix sizes. This is because for sizes 64x64 and 80x80, the CPU-only execution time is smaller than the value of Δ, meaning that such benchmarks are memory-bound rather than compute-bound. The improvement is even more significant in the case of *acc*, since here, all 5 benchmarks are memory-bound. Note that for certain utilization ranges and when all benchmarks are considered, *acc* actually performs worse than *cpu*. This is expected, as the number of segments is greater for the CPU + accelerator case compared to the CPU-only case (7 vs. 5); hence, when the benchmark is memory-bound, the sum of segment lengths under *acc* is larger than that under *cpu*.

In summary, the results in Figure 10 indicate that the considered ZCU102 platform does not provide sufficient memory throughput to support the bandwidth-hungry accelerators. Indeed, note that the GDMA transfer time for the largest size matrix in Table 3 equates to a throughput of only 897 MB/s, which is much lower than the theoretical DRAM bandwidth of 19.2 GB/s. We conducted a brief investigation into the issue. Our understanding is that the platform enacts throttling of memory transactions that cross the HPM ports between the PS and the PL. At the time of writing, we are not aware of an existing workaround for this problem.

The unexpectedly low PS-PL bandwidth appears to be a quirk of the considered platform. To have a more general idea of the potential of the proposed framework, we investigate the expected schedulability ratio in future platforms, such as the Xilinx Versal [30], where a better DMA throughput can be achieved. In this architecture, two to four memory controllers are directly connected to the PL fabric though a hardened Network-on-Chip (NoC); each NoC port supports 16 GB/s throughput, and the PL has access to multiple ports.

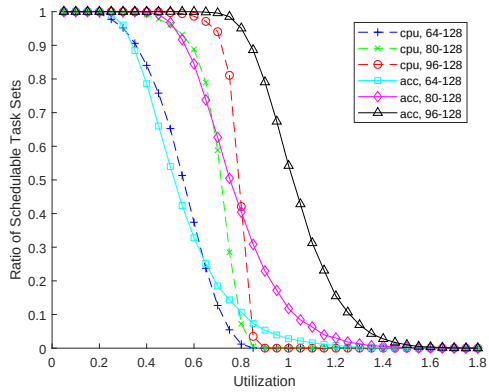

Fig. 10. Schedulable task sets for CPU only and CPU + Accelerator varying the matrix size

Figures 11 and 12 show the ratio of schedulable task sets as we vary the DMA throughput for *cpu* and *acc*, respectively. We consider all 5 matrix sizes in Table 3. Transfer times, and thus the value of Δ, are computed by dividing the data size by the specified throughput. We also show the theoretical *infinite* case where memory operations take zero time. Under this setting, *acc* supports 50% schedulability ratio for utilizations as high as 7.2, versus 0.8 for *cpu*, a 9x increase.

Finally, in Figure 13, the *no locking* case corresponds to the same scenario as in Figure 12 for a DMA throughput of 16 GB/s, equivalent to one NoC port in Versal. *Lock before S1* and *lock before S0* correspond to the case where a single matrix multiplication IP is shared among all tasks, and one of the two locking schemes discussed in Section 5.4 is employed. As intuitively expected, locking

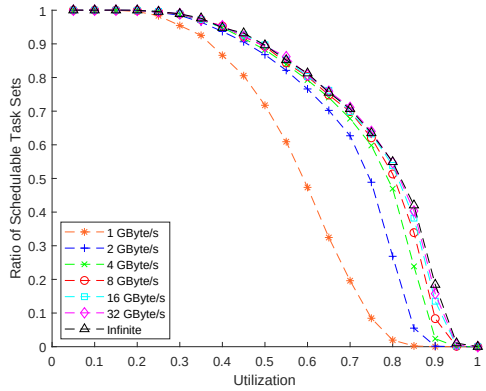

Fig. 11. Schedulable task sets for CPU only as a function of utilization, varying DMA throughput

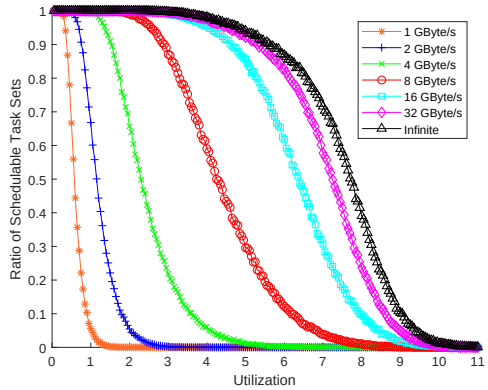

Fig. 12. Schedulable task sets for CPU + Accelerator as a function of utilization, varying DMA throughput

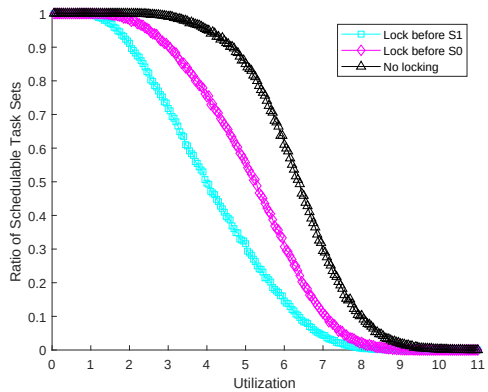

Fig. 13. Schedulable task sets for CPU + Accelerator as a function of utilization, 16 GB/s DMA throughput

before segment S0 leads to better results for the generated task sets, because it ensures that each job under analysis can suffer lock-induced blocking only once. Here, *lock before S1* supports 50%

schedulability ratio for utilizations up to 5.25, versus 6.35 for *no locking*, a 17% decrease, in exchange for saving FPGA area by sharing a single accelerator.

8 CONCLUSION AND FUTURE WORK

The framework proposed in this paper allows streaming of segments of tasks on CPU only or CPU + accelerators, depending on the nature of the task. This approach maximizes the use of hardware resources while maintaining predictability. The paper demonstrates that by breaking tasks into multiple CPUs and accelerators segments, both predictability and performance can be improved. Furthermore, the framework provides developers with a way to analyze the performance gain achieved by using accelerators. Overall, the proposed co-design framework provides developers with a way to optimize the performance and predictability of their systems by using a combination of CPUs and accelerators.

9 ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Giovani Gracioli was partially supported by Fundação de Desenvolvimento da Pesquisa - Fundep Rota 2030/Linha V 27192.02.01/2020.09-00 Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. We would like to thank Jayati Singh for help during experimental setup.

REFERENCES

[1] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the Performance of Schedulability Tests. *Real-Time Syst.* 30, 1–2 (May 2005), 129–154. https://doi.org/10.1007/s11241-005-0507-9

[2] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. 2012. Deterministic execution model on cots hardware. In *International Conference on Architecture of Computing Systems*. Springer, 98–110.

[3] Thomas Carle and Hugues Cassé. 2021. Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. *arXiv preprint arXiv:2103.17082* (2021).

[4] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*.

[5] Stephen A. Edwards and Edward A. Lee. 2007. The case for the precision timed (PRET) machine. In *Proceedings of the 44th annual Design Automation Conference (DAC '07)*. ACM, New York, NY, USA, 264–265.

[6] Evidence. 2018. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/ Online; accessed 16 October 2018.

[7] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. 2019. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Sophie Quinton (Ed.), Vol. 133. Dagstuhl, Germany, 27:1–27:25. https://doi.org/10.4230/LIPIcs.ECRTS.2019.27

[8] C. Kenna, J. Herman, B. Ward, and J. H. Anderson. 2013. Making shared caches more predictable on multicore platforms. In *ECRTS '13*. 157–167.

[9] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 145–154.

[10] H. Kim, A. Kandhalu, and R. Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *Proc. of the ECRTS 2013*. 80–89.

[11] N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith. 2016. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12.

[12] C. Maia, L. M. Nogueira, L. M. Pinho, and D. G. Pérez. 2016. A Closer Look into the AER Model. In *2016 IEEE International Conference on Emerging Technology and Factory Automation, (ETFA 2016)*.

[13] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 45–54.

[14] Stefano Markidis. 2021. Matrix Multiplication. https://kth.instructure.com/courses/17065. Accessed: 2021-02-02.
[15] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*. IEEE Computer Society, Washington, DC, USA, 269–279.
[16] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. 2017. Look Mum, no VM Exits! (Almost). In *Proc. of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2017)*. 13–18.
[17] Juan M Rivas, Joël Goossens, Xavier Poczekajlo, and Antonio Paolillo. 2019. Implementation of memory centric scheduling for COTS multi-core real-time systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[18] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. 2017. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–20.
[19] R. Mancuso S. Roozkhosh. 2020. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*. Sydney, Australia. https://doi.org/10.1109/RTAS48715.2020.00006
[20] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. 2016. Real-Time Computing on Multicore Processors. *Computer* 49, 9 (Sept 2016), 69–77.
[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (May 1990), 1175–1185.
[22] Sam Skalicky, Christopher Wood, Marcin Lukowiak, and Matthew Ryan. 2013. [IEEE 2013 International Conference on ReConFigurable Computing and FPGAs (ReConFig) - Cancun, Mexico (2013.12.9-2013.12.11)] 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig) - High level synthesis: Where are we? A case study on matrix multiplication. https://doi.org/10.1109/reconfig.2013.6732298
[23] M. R. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni, and M. Caccamo. 2019. Segment Streaming for the Three-Phase Execution Model: Design and Implementation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 260–273.
[24] M. R. Soliman and R. Pellizzoni. 2019. PREM-based optimal task segmentation under fixed priority scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[25] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. 2016. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–11.
[26] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni, and Marco Caccamo. 2019. A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems* 55, 4 (2019), 850–888.
[27] S. Wasly and R. Pellizzoni. 2014. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 75–86.
[28] Xilinx. [n.d.]. Zynq UltraScale+ Device - Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf
[29] Xilinx. 2017. AXI4 Refrence Guide. Accessed on 07.01.2020.
[30] Xilinx. 2021. Xilinx Versal. https://www.xilinx.com/products/silicon-devices/acap/versal.html Online; accessed 3 October 2021.
[31] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. 2011. Memory-Centric Scheduling for Multicore Hard Real-Time Systems. In *Real-Time Systems*. Springer.
[32] H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 155–166.
[33] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 55–64.