Composable and Predictable Dynamic Loading for Time-Critical Partitioned Systems on Multiprocessor Architectures

Shubhendu Sinha[a], Martijn Koedam[a], Gabriela Breaban[a], Andrew Nelson[a], Ashkan Beyranvand Nejad[b], Marc Geilen[a], Kees Goossens[a]

[a] *Eindhoven University of Technology, Eindhoven, The Netherlands*
[b] *Delft University of Technology, The Netherlands*

Abstract

Time-critical systems for instance in avionics, isolate applications from each other to provide safety and timing guarantees. Resources are partitioned in time and space to create an isolated partition per application which facilitates fault containment and independent development, testing and verification of applications. Current partitioned systems do not allow dynamically adding applications. Applications are statically loaded in their respective partitions. However, dynamic loading can be useful or even necessary for scenarios such as on-board software updates, dynamic reconfiguration or re-loading applications in case of a fault. Multiprocessors offer higher performance and by integrating applications on different single-core chips onto a single multiprocessor chip, power consumption and weight of the system can be reduced. For these reasons, interest in using multiprocessor platforms for time-critical systems has recently increased.

In this paper we propose a software architecture to dynamically create and manage multiprocessor partitions. We also propose a method for composable dynamic loading in uniprocessor and multiprocessor platforms which ensures that loading applications do not affect the running applications and vice versa. Furthermore the loading time is also predictable i.e. the loading time can be bounded a priori. We achieve this by splitting the loading process into parts, wherein only a small part which reserves minimum required resources is executed in the system partition and the other parts are executed in the allocated application partitions which ensures isolation from other applications. We implement the software architecture for a SoC prototype on an FPGA board and demonstrate its composability and predictability properties.

Keywords: Composable, Predictable, Loading, MPSOC

1. Introduction

Time-critical systems for instance in avionics, have strict safety, real-time and fault tolerance constraints. To ensure these constraints and for fault-containment, applications are isolated from each other. For isolation, resources are partitioned temporally and spatially. With temporal partitioning each application can be analysed individually, making it easier to give real-time guarantees. Spatial partitioning ensures that if an application fails due to a hardware or software fault, it cannot affect the other applications or the system, thus improving fault tolerance. With temporal and spatial partitioning applications can be developed, tested and verified independently. Without isolation, each small change to an application, requires the system and all applications to be re-verified [1, 2]. Allocating dedicated hardware to achieve isolation has disadvantages of increased cost and weight, therefore integrated architectures with shared resources are proposed in avionics with *Integrated Modular Avionics (IMA)* [3]. In IMA, resources are partitioned in time and space to create logical containers for each application. For commercial deployment, avionics industry has standardized partitioning with the ARINC 653 API [4] and AIR (ARINC in Space RTOS) [5] standards.

The aforementioned partitioned systems are static systems, wherein all applications are integrated at design-time. In standards such as ARINC653 and AIR, applications are statically loaded in their respective partitions and the system iterates through a static Time Division Multiplexed (TDM) schedule at run time [6]. Dynamic loading is not supported in such systems. However dynamic loading is essential for

scenarios such as 1)adding new applications at run-time 2)use-case switching, wherein some applications are removed and some new applications are added and 3)dynamic reconfiguration in case of a fault. Consider the example of a spacecraft. The operation plan of a spacecraft can be changed to deal with unexpected events. To adapt to these changes, it may be necessary to switch to a different mode of operation which may consist of applications with special functionalities. Furthermore it may be necessary to replace or modify an existing malfunctioning software module which requires individual applications in the spacecraft software to be dynamically updated. Consider a second example of fault tolerance. After discovery of a fault in a hardware module, it may be necessary to re-load all active applications, or in case of lack of resources, the set of priority applications, on a back-up platform [7]. In such scenarios, dynamic loading is necessary. For these reasons dynamic loading is also a feature proposed for future avionic architectures as argued in [8]. In the scenario of reloading on occurrence of a fault, it may be necessary to load multiple applications simultaneously.

In the above example scenarios, for safe dynamic updating or re-loading, it is essential to execute the dynamic loading without affecting the other running applications. For fault containment and timing isolation it is also necessary to ensure that the existing running applications do not affect the loading process. That is, dynamic loading in time-critical systems should be composable. We define loading as *composable*, if the loading process and the running applications do not affect each other. A strict cycle level isolation is necessary to allow independent development, testing and verification of applications. Otherwise, for every small change in an application, the entire system needs to be reverified, incurring heavy development costs. In case of reloading on occurrence of a fault, the reloading should finish in a known bounded time, otherwise it may lead to catastrophic situations. Hence the dynamic loading in time-critical systems should be predictable. We define loading as *predictable*, if a bound on the loading time can be computed at design-time.

The interest in partitioned multiprocessor systems has recently increased [9], [10], [11]. There are several reasons for moving towards multiprocessor architectures. Multiprocessor platforms provide high performance with parallel processing. Additionally, multiple applications that run on different single core chips can be integrated on a single multiprocessor chip, thus reducing the power, space and weight requirements of a system, which are crucial in avionics. Distributed memory multiprocessors, also known as NUMA (Non-Uniform Memory Access) are more attractive for partitioned systems for two reasons. First, tile-based architectures connected by a Network-on-Chip (NOC), do not face scalability issues of shared memory symmetric multi-processors (SMP). Second, distributed memory NUMA architectures which are non-cache coherent, where each core has its own private memory, suit the isolation requirements for partitioning.

This work presents a solution to the problem of composable and predicable dynamic loading for time-critical partitioned systems on multiprocessor architectures. The challenges in achieving this goal are multi-fold. A primary requirement for dynamic loading in partitioned systems is the ability to create and manage partitions at run-time. The existing state-of-the-art partitioned systems either do not support dynamically creating and managing partitions [5, 6, 12, 13] or only support dynamically updating the schedule of the partitions [14]. With regards to composability, existing non-partitioned loading methods [15–18] do not provide timing isolation between the running applications and the loading process. Extending existing single processor loading methods to multiprocessors is not trivial. An additional entity is necessary to coordinate loading on the allocated processor tiles. This coordination should guarantee isolation of the running applications and the loading process. Furthermore all of the existing methods only load one application at a time.

To provide composable and predictable dynamic loading for partitioned systems, this paper has two contributions. The first contribution is a software architecture with which new partitions can be dynamically created and managed at run-time. The second contribution is a design of a composable and predictable loading method for partitioned multiprocessor architectures. With composable loading, we ensure that multiple simultaneously loading applications do not affect each other or the running applications and vice versa. Predictable loading ensures that the loading process completes in a known bounded time. In our previous work [19] we addressed the problem of composable and predictable loading for partitioned uniprocessor platforms. In this work we extend these results for loading multiprocessor applications on partitioned multiprocessor platforms.

We achieve composable dynamic loading on a uniprocessor platform by splitting the loading process into

two parts. The first part is executed by the system partition, it creates a boot-strap partition by reserving the minimum required resources, i.e. processor time slots, memory and a Direct Memory Access(DMA) unit. The DMA unit is necessary to fetch the application code and data. In the second part, we complete the loading process by reserving the rest of the resources required by the application and loading the application code and data. The second part is executed strictly in the reserved application's time slots, which ensures isolation from running applications. For composable dynamic loading of multiprocessor applications on a multiprocessor platform, we propose a master-slave loading method. The master and slaves coordinate to create a minimal multi-tile platform for each of the applications that have to be loaded that allows them to boot-strap. The loading then continues strictly in the (isolated) allocated boot-strap platforms. To achieve composability, we ensure that the initial coordination between the master and the slave completes in a fixed time. In both the uniprocessor and the multiprocessor loading methods, the system partition creates boot-strap platforms for all the applications that have to be loaded within a fixed amount of time. This is achieved by ensuring the processor TDM slots are wide enough to accommodate the necessary work. To facilitate this, we derive constraints for the duration of the TDM slots which have to be satisfied while designing the system. To validate our design we implement our software architecture for a SoC prototype on an FPGA board and experimentally demonstrate composability and predictability. There are certain requirements of the proposed work which are discussed in the following sub-section.

1.1. Requirements

There are two requirements for composable loading. Among partitioned systems, the proposed composable loading method relies on a composable platform i.e. partitions do not affect each other at the cycle-level and the predictability of the loading method relies on a predictable platform, i.e. the platform resources give guaranteed performance. Possible ways of designing a composable and predictable platform are discussed in Section 3. Secondly, since the loading of application code and data is done in the TDM slots allocated to the application, for composable loading it is crucial that the allocated boot-strap platform starts independently of running applications and other simultaneous loadings. With dynamic allocation of processor TDM slots, the loading time would depend on the occupied slots and thus would violate composability. Hence we require static allocation of the processor TDM slots. This requirement however does not diminish the advantages of dynamic loading. In a statically loaded system, to add or remove an application, the entire system needs to be stopped. However in the proposed system, applications can be added(loaded), removed, upscaled or rebooted without interrupting the system and in a composable manner, i.e. without affecting the other applications or other simultaneous loadings.

With static allocation of the TDM slots, the system designer has broadly two options to design the system. In the first option, the system designer can identify the static use-cases which describe the set of applications that may run together and allocate non-conflicting slots to applications across all use-cases. The TDM can then be dimensioned based on the maximum number of required slots across all the use-cases. In this method, there is no possibility of failure of loading due to unavailable TDM slots. However, if the maximum number of applications that run simultaneously is not known at design-time, then, the second option is that the system designer can over-dimension the system, by designing the TDM with a large number of slots. If multiple applications are mapped to the same slot(s), there is a chance that the required slot(s) for the new application to be loaded may be occupied by the running applications and consequently, the application will not be loaded. However if the required resources are available, the application will load composably. There can be a hybrid approach, wherein some of the slots in the over-dimensioned TDM are allocated to the critical applications without any overlap in their allocations and the rest of the TDM slots are allocated to other applications. In this way, it can be ensured that the loading of the critical applications never fail, but the loading of other applications is possible, depending on available resources. In the following subsection we describe the key aspects of the system design for composable and predictable loading.

1.2. Overview of System Design

In Figure 1 we have shown an overview of the work-flow for designing a system for composable and predictable loading with the necessary inputs and the involved steps. For composable loading, we ensure that

the system partition completes the necessary work for loading all the applications in a fixed amount of time. This is achieved by dimensioning the processor TDM slots to accommodate the necessary work. We derive constraints for the duration of the TDM slots which have to be satisfied while designing the system. These constraints are explained in depth in Sections 6 and 7 for the uniprocessor and the multiprocessor loading methods respectively. In step (1) shown in Figure 1, the constraints for the TDM slot are evaluated. The required input for this step depends on the maximum number of applications that load simultaneously, which is derived from the application mappings, i.e. the processor TDM slot allocations. To decide the application mappings, the design options with static allocation of slots are mentioned in the previous subsection. We assume for the purpose of this work that the application mappings are available as input. In step (2) the TDM slot size is dimensioned such that it satisfies all the constraints evaluated in step (1). In step (3), we compile the application sources along with its resource requirements so that the required resources can be reserved during loading. The details of this step are explained in Section 5. The bound on the worst-case (WC) loading time of an application is computed in step (4). The necessary inputs for this step are the application binary size, the model of the hardware platform and the TDM schedule. This step is detailed in Section 8.

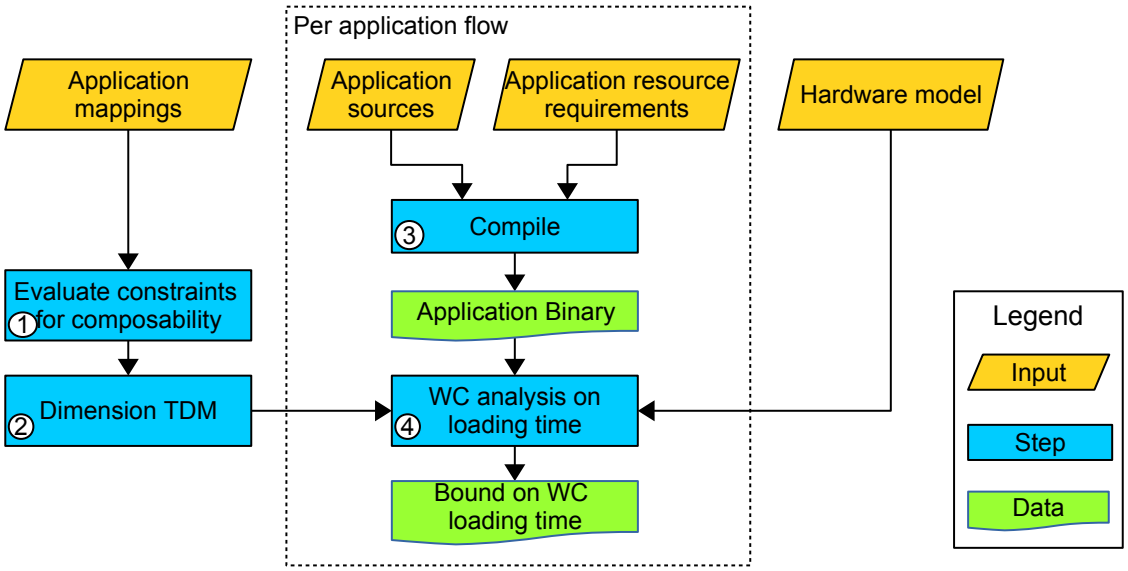

Figure 1: An overview of the work-flow for designing the system for composable and predictable loading.

In the following section we discuss the related work. In Section 3, we explain the multiprocessor hardware platform on top of which we build our software architecture. In Section 4, we describe the software architecture. In Section 5, we explain the compilation flow, which makes dynamically loadable application binaries for uniprocessor and multiprocessor applications. Sections 6 and 7 explain the uniprocessor and multiprocessor loading method respectively. In Section 8 we explain the predictable natures of our loading methods. In Section 10 we address the future work and we conclude with experiments in Section 9 and conclusion in Section 11.

2. Related work

We first present a brief overview of solutions in literature for dynamically updating software at run-time in real-time systems. Secondly, we discuss existing works relevant to dynamic loading in partitioned systems. Then we review existing loading methods for multiprocessor architectures and finally we present the references to the architecture that we used in our work.

Work in [20] addresses periodic and sporadic tasks arbitrated with TDM. When a new task arrives, the TDM is re-dimensioned and the TDM slots are re-allocated, in such a way that all tasks get a share

of the TDM that is close to their requested share. To ensure that the task deadlines are met, the task periods are re-computed with *elastic scheduling* [21] for all running applications. In [15] an update task is responsible for dynamic loading in a system which follows priority based Rate-Monotonic (RM) scheduling. By making the period of the update task equal to the hyper-period of the existing tasks, they show that the dynamic update is guaranteed to finish by the second hyper-period of the system. Work in [17] uses two level hierarchical scheduling. Earliest Deadline First (EDF) scheduling is used to schedule multiple servers containing applications and each server can have a scheduler of its own. For each new application to be loaded, if it passes the EDF schedulability test, it is added to the system by creating a server for it. In this way they ensure that adding new application does not affect the real-time guarantees of existing applications. In [16], the dynamic update is performed in idle time in a system which runs an RTOS. By performing the update in slack time, they ensure that the real-time guarantees of existing applications are not affected. In the above mentioned works, the notion of composability is limited to schedulability of applications. We believe a stronger notion of composability (isolation at cycle-level) is required for partitioned time-critical systems and the existing solutions do not provide that. Furthermore, except for [15], the above mentioned works do not provide timing guarantees on the loading time.

We now give an overview of relevant works for dynamic loading in time-critical partitioned systems. The work in [7] proposes a software layer that manages dynamic reconfiguration in ARINC based uniprocessor systems for fault tolerance. The proposed layer lies between RTOS and ARINC API and it reconfigures the system to reallocate partitions on a redundant hardware module when a fault is detected. The reallocation process takes place in the application's own partition, therefore the reallocation of a partition and other existing partitions are isolated from each other. They detail the functional steps involved in reconfiguration, however they do not provide details with regards to loading of a partition. Additionally, no timing guarantees are provided in [7], hence their architecture is not predictable. The work in [14] proposes a methodology to update partition schedules for AIR based uniprocessor platforms. The update process is carried out on a best-effort basis in the system partition, where the update process is executed in the slack time available. Their proposed method isolates running applications from the update process, however the running applications may affect the update process. Moreover, with best-effort scheduling, the update time is not predictable. They encode the partition scheduling table in the application binary, which is read and processed at run-time. We extend this idea to describe all the required resources by an application in a multiprocessor system as detailed in Section 5.

In the context of dynamic loading on multiprocessor architectures, [22, 23] are relevant. In [22], for NUMA multiprocessor architectures, two strategies are proposed for task migration based on a master-slave architecture. In the first strategy, on master's commands, the slave on the original core removes the application from the original core and the slave on the target core loads the application on the target core. In the second strategy the application object code is compiled into all the necessary cores and the master and slave ensure that only one copy of the application is in *running* mode, and others are in *sleeping* mode. In both the strategies, the master and the slaves only guarantee isolation in space (with non-overlapping memory allocations) for each application. Temporally the loading of an application can affect the running applications and vice versa. Furthermore, there are no timing guarantees on the loading time. The work in [23] proposes a system managed master-slave architecture for task migration on a NOC based NUMA partitioned multiprocessor architecture. The master and the slaves co-operate to execute the task migration of an application. This execution is carried out only in the allocated application's partition. Hence their architecture is composable and predictable for migration of a single application. However, if multiple applications have to be migrated, the time to complete the migration of each the application will depend on the order in which the master processes the migration requests. Thus their architecture is not composable, if multiple applications have to be loaded. To overcome this drawback, we ensure that the cooperation between the master and the slaves completes in a fixed time, independent of the number of applications that have to be loaded. To achieve this, as described in Section 7, we define constraints to design the TDM system, based on the master and slave's work that is necessary to load the maximum number of applications in the system. Additionally, the work in [23] has the following drawbacks. It requires statically defined partitions. They lack a run-time resource manager, required for dynamically creating and managing a partition. As a consequence, their master-slave architecture does not handle failure scenarios in which resource allocation

fails on processor tile(s). Their architecture requires that the application code and data should exist in the private memories of the required processors for all the applications that may be loaded. Such a requirement limits the number of applications that can be loaded on a multiprocessor platform. With regards to their first drawback, we build a *Resource Management* framework that enables creating and managing partitions at run-time. The proposed loading architecture in this paper handles failure scenarios and loads applications from a shared memory that has a larger capacity than the local private memories.

The implementation of our solution is based on the platform described in [13]. It introduces a design-flow to generate a multi-tile partitioned system prototyped on an FPGA, where applications share resources composably, i.e. they do not affect each other even by a single cycle. This system is static, applications are compiled into the FPGA bitstream. Thus applications cannot be loaded at run-time. Furthermore, no run-time resource management is available, which is essential for dynamic loading of applications. We extend the partitioned system of [13] by developing a software architecture to facilitate run-time creation and management of partitions and a composable and predictable loading methodology on top of the hardware platform.

3. Platform Overview

The composability of the proposed loading method relies on a composable platform, i.e. partitions do not affect each other at the cycle-level and the predictability of the loading method relies on a predictable platform, i.e. the platform resources give guaranteed performance. The CompSOC platform architecture used in this work [13] satisfies these two requirements. It is discussed in detail in the following sub-section. However the proposed loading method is also applicable to other partitioned systems for example [24, 25]. A platform can also be designed to be composable by ensuring that each resource in the platform is either composably shared or by exclusively allocating resource units to an application [1, 2, 26]. The communication interconnect can be composably shared for instance by using a bus with TDM arbitration or TDM based NOCs [24, 25, 27, 28]. Hardware units such as the DMAs and the local memories can be exclusively allocated per application. Memory can be either physically partitioned per partition or can be composably shared with a composable memory controller [29, 30]. Memory can be statically or dynamically allocated as long as the allocations do not overlap in space. The processor can be composably shared with TDM arbitration as detailed in the last sub-section.

3.1. Hardware Platform

The proposed software architecture for composable and predictable dynamic loading is built on top of the partitioned system architecture proposed in [13]. A 4-tile platform instance is shown in Figure 2. For each application, the design proposed in [13] creates partitions which are cycle-level composable. This is achieved by sharing each resource composably, that is by partitioning it in space or time.

The processor tiles in Figure 2 contain a MicroBlaze core, a soft-core RISC processor. Caches are not used to simplify composability and predictability. Local memories such as Instruction Memory (IMEM) and Data Memory (DMEM) are used. Apart from the Instruction Local Memory Bus (ILMB), the IMEM is also connected to the Data Local Memory Bus (DLMB). This is essential, so as to be able to load application code in IMEM. The processor is connected to the DMA(s). The processor is also connected to a Timer Interrupt Frequency Unit (TIFU) which enables generating interrupts at programmable intervals. TIFU [31] assists the microkernel (explained in the next subsection) by maintaining strict timings in hardware. Each partition is exclusively allocated a set of DMA units and communication memories (CMEMs).

The communication architecture is composable by making use of a composable NOC [28] and a composable memory controller [29, 30]. The processor and the memory tiles are connected with the NOC through NOC ports. The NOC port connects to the network interface in the NOC. I/O peripherals can also be connected to the platform by implementing a hardware unit that translates the communication protocol used by the peripheral into the communication protocol used by the NOC. Multiple connections from a master (DMA) are connected to the NOC with a de-multiplexer that implements the memory map. Shared memory shown in Figure 2 can be an off-chip DDR memory or an on-chip SRAM memory. We make use of a software

library [32] to implement FIFO communication. The FIFO library is flexible, in that it allows storing the FIFO buffer either at the producer side (DMA MEM) or at the consumer side (CMEM) or in the shared memory. For local FIFOs, where the producer and consumer are on the same tile, the processor DMEM can be used to store the FIFO buffer. Both the producer and the consumer set up the FIFO administration, by allocating space for the necessary administration. The library API ensures the state of the FIFO is updated, both at the producer side and the consumer side, when the produced tokens are sent to the consumer and when the consumer consumes the received tokens.

Applications on this platform can be bare code or based on a *Model of Computation (MoC)* [33] such as *SDF* [34] or *KPN* [35]. Applications can also be an RTOS which in turn hosts other applications. Applications are single threaded and have access to DRAM or SRAM via the NOC. The contributions of this article make no assumptions on the type of applications. The applications may or may not have tasks and may use different types of scheduling within the application, however TDM is used between applications with a microkernel explained in the following sub-section.

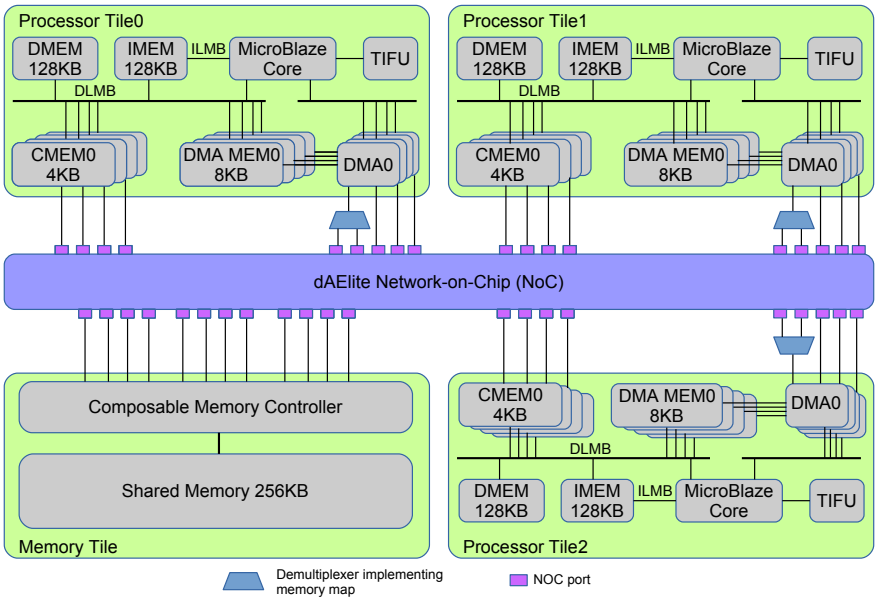

Figure 2: An example of a 4 tile (3 processor tiles, 1 memory tile) platform instance.

3.2. Microkernel

To create partitions on the processor, the Composable Microkernel (CoMiK) [36] is used. CoMik partitions the processor in time using TDM arbitration. IMEM and DMEM memories are partitioned in space for instruction, data, stack and heap memory per partition. CoMik runs for a small fixed slot duration between every partition. During this slot, it switches the context of the partition. Care is taken to prevent the jitter from critical regions and multi-cycle instructions from violating the defined duration of partitions. In this way each partition on the processor is isolated from other partitions.

4. Resource Management Framework

An essential part of dynamic loading in a partitioned system is creating new partitions and managing them at run-time. In this section we propose a **Resource Management (RM)** framework that facilitates creating and managing partitions at run-time. The resource management allows reserving resources at run-time and associating them with a partition. Resources can be allocated offline (pre-computed allocations) or online. After the usage, resource reservations can be released. The management of resources should be

done with a privileged Application Programming Interface (API), so that applications cannot change their own or other's resource allocations. The **Resource Management** framework is designed keeping in mind these requirements.

The RM framework is based on the following concepts. A resource **Budget Descriptor (BD)** describes the share of a resource usage, required by an application. A BD must be defined for each resource in the system required by the application such as the processor, NOC, and the memory. This share of resource is composably shared (partitioned in time or space) on the hardware platform as described in Section 3. A **Virtual execution Platform (VP)** is the set of all resource budgets required by an application. As a result a VP descriptor, i.e. a **VP BD**, is the collection of all the BDs for the resources required by an application. Since each VP consists of composably shared resources, each VP is isolated from other VPs in time and space during execution (hence the name *Virtual execution Platform*). BDs in the VP BD can be arranged in a hierarchy as shown in Figure 3. Figure 3 shows the structure of the VP BD as a SysML block diagram. It shows that a VP BD contains one or more Tile BDs and zero or more NOC connections and memory (DDR/SRAM) BDs. Each Tile BD contains exactly one Processor BD and zero or more DMA and CMEM BDs. Each BD specifies the allocation type, that is either offline or online in the *alloc_type* field. The Processor BD describes the parameters necessary to create an application partition on the processor. For offline allocation, the list of required TDM slots are described and for online allocation, the required number of TDM slots are described. Other necessary parameters in the Processor BD are the required size of stack, heap and the space required to store application code and data. Each NOC BD describes a NOC connection required by the application. For offline allocated connections, the request and the response channels are specified with the details of the TDM slot allocations in the NOC and the path for each channel. For online allocation, the source and the destination node and the throughput and latency requirements of the connection are described. In the DMA BD and CMEM BD, with offline allocation the ID of the specific unit is described and with online allocation, a unit is allocated online. The shown structure of the VP BD can be used to describe both uniprocessor and multiprocessor VPs.

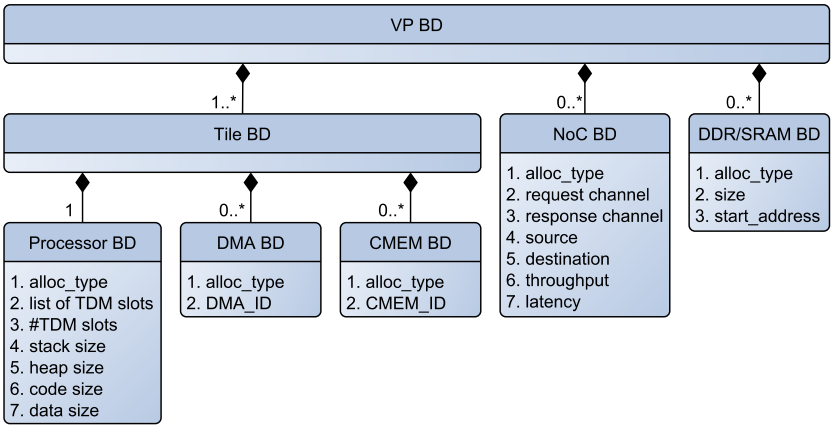

Figure 3: Structure of a Virtual Platform Budget Descriptor (VP BD) shown with a SysML block diagram.

We construct a run-time RM library with a generic API for all types of resources. Resources are first reserved, then allocated and when no longer needed, released. The *reserve* API requires a BD and returns a Budget Identifier (BID). This BID can be further used to *allocate* and *release* the resource. The run-time RM library has two components for each type of resource as shown in Figure 4. The first component is a *Budget Manager* which accepts reserve, allocate and release requests. It contains the online allocation algorithm for the resource, which is used to find an allocation at run-time. It also maintains the reservation state of the resource to ensure correct allocation. For each online or offline request, it first checks with the state management, if there are no collisions between the requested allocation and the existing allocations. If there are no collisions, the requested allocation is reserved in the software state, but not in the hardware. With the *allocate* API, the reserved allocation is programmed in the resource. The second component is

the software *Driver* which configures the hardware as per the requested allocation. The *release* API first configures the hardware to release the allocated resource and then accordingly updates the software state by freeing the allocations.

The run-time RM API is classified into two groups. *System API* allows to create, modify or destroy VPs. The System API is only accessible by a privileged application. The *User API* allows an application to use a resource within its VP. The User API validates if the requested resource usage is within the budget allocated to the application, if it is not, the request is rejected and an error is returned. The User API is resource dependent.

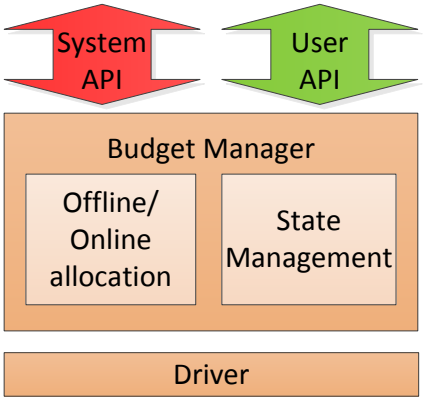

Figure 4: Structure of the run-time resource management library (from [19]).

5. Compilation flow

In this section we describe the compilation flow to generate dynamically loadable application binaries for uniprocessor and multiprocessor applications. In order to make use of the run-time RM library API, the description of the required platform should be locatable in the application binary. We describe our solution to address this aspect below.

A multiprocessor application may use multiple tiles. The application object code for each tile, after loading, resides in separate memory spaces that are independent from other tiles. The resource requirements for the application on each tile are also independent from other tiles. To combine the resource requirements on each tile and the object code on the respective tiles, into a *single* application binary, we introduce the concept of *bundle*. A **Bundle** consists of the *Budget Descriptors* and the *Application Object Code* that is necessary to execute the application. The exact structure of the bundle is shown with a SysML block diagram in Figure 5. Owing to the hierarchical structure of the budget descriptors, exactly one BD is necessary in the bundle. However, to be able to execute either the BD in the bundle should be a Processor BD or it should contain a Processor BD. A bundle may also contain zero or more bundles within itself. As a result, for a uniprocessor application, the bundle may contain a Tile BD or a Processor BD and the application object code. For a multiprocessor application, the bundle may contain a VP BD and one or more *Tile Bundles*. Each *Tile Bundle* contains one Tile BD or Processor BD and the application object code for that tile. In the following subsections we explain the compilation flow to generate application bundles for the uniprocessor and the multiprocessor applications.

5.1. Compilation flow for Uniprocessor applications

We require that alongside the source code of the application, the budget description of the required VP should be provided. The compiler compiles the application into the **Executable Linkable Format (ELF)** binary. The VP BD is stored in a dedicated section in the ELF binary. For dynamic loading the application code needs to be independent of the location where it is placed in memory. With absence of Memory Management Unit (MMU) in the hardware platform, using virtual memory is not possible.

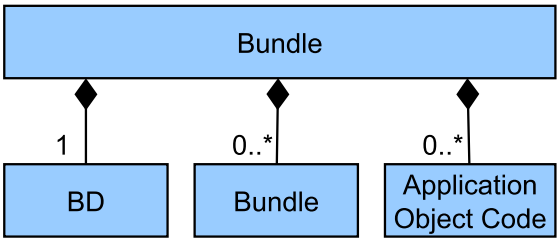

Figure 5: The structure of a *bundle* shown as a SysML block diagram.

Therefore compilation is done with *Position Independent Code(PIC)*. The compilation flow is illustrated in Figure 6.

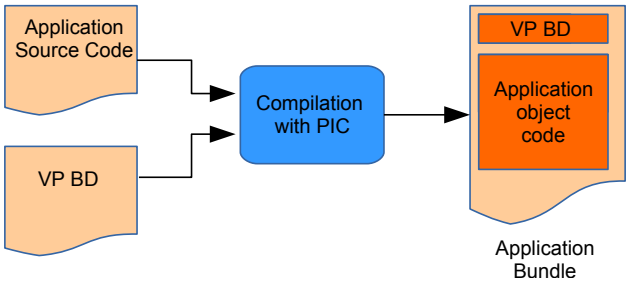

Figure 6: The compilation flow to generate an application bundle for a uniprocessor application.

5.2. Compilation flow for Multiprocessor applications

The compilation flow based on the described *bundle* structure, is illustrated in Figure 7. For each processor used by the application, we require the source code and the *Tile BD* for that tile. Similar to the uniprocessor compilation process, these are compiled with *Position Independent Code* to generate an ELF binary. This binary is called a *Tile Bundle*. In this work we assume the tile bundle to tile mapping is done at design-time. This mapping information, other BDs (such as NOC BD, DRAM BD) and the compiled tile bundles, are then given to an *Application Bundle Generator* tool. This tool has two tasks, it generates the VP BD and it creates the *Application Bundle* as an ELF binary. The tool places each tile bundle as an ELF section in the Application Bundle. Based on the number of the tile bundles, the input of other BDs, and the tile bundle to tile mappings, this tool constructs the VP BD. The section names of the stored tile bundles and their addresses are also stored as part of the VP BD. The VP BD is stored in the application bundle in a separate section.

6. Uniprocessor Loading Architecture

In the following subsection, we first explain the loading architecture for loading a uniprocessor application and at the end of the section we argue the composability property of the loading architecture.

6.1. Loading Architecture

To execute the dynamic loading process, we propose a privileged application called as the *System Application (SA)*. The SA is an application with access to the RM *System API*, thus it can create, modify or destroy VPs. At the system start-up, a VP is allocated for SA with one processor TDM slot and one DMA connection to the shared memory via the NoC. At the start of every SA slot, the SA checks for ELF binaries

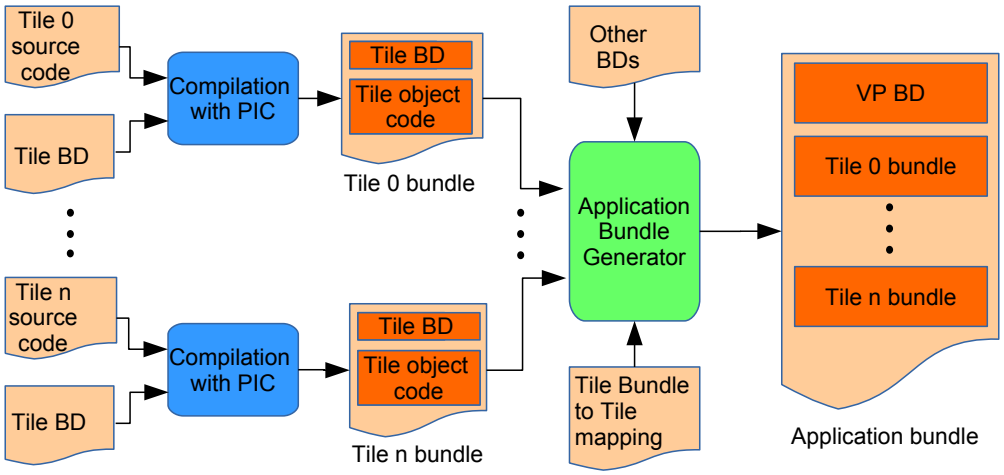

Figure 7: The compilation flow to generate an application bundle for a multiprocessor application.

in pre-defined locations in the shared memory, which indicate they need to be loaded. If an ELF binary arrives after the checking routine of SA, it is detected at the start of the next SA slot.

We define ***Loading Time (LT)*** *as the time from the beginning of the SA slot in which an ELF binary is detected to the moment when the application contained in the ELF begins execution.* We split the loading process in three steps. For each detected ELF binary the SA initiates the first step. In the first step, the SA creates a boot-strap VP by reading the VP BD section in the application ELF binary and by reserving the processor and memory budgets. This step is called *Boot-strap VP Creation*. The first step is executed in the SA's time slot. In the second step, called *Boot-strap VP Expansion*, the *Boot-strap VP* is expanded by reserving the rest of the required resources as described by the VP BD in the application ELF binary. In the final step, called *Loading Code & Data*, the application code and data are loaded. The second and third steps are entirely executed only in the application's VP. All resource reservations and allocations are done using the RM *System API* before the loading process completes. The details of the loading steps are as follows:

Boot-strap VP Creation. The SA, using its own DMA, fetches the ELF header and parses it to locate the section containing the VP BD and reads the processor BD. The processor BD describes the required processor TDM slots, stack size, heap size and memory required in IMEM and DMEM. The requested processor TDM slots are reserved and allocated and are registered with the microkernel. Stack and heap memory are allocated in DMEM and are associated with the newly created VP. A DMA is also reserved, as it is essential to pull in the code and data from the shared memory into the instruction and data memories in the next step. In this step, if the reservation of the processor BD or DMA BD fails due to insufficient available resources, the loading process is aborted by deallocating and releasing already allocated resources. Since no TDM slots remain allocated in case of reservation failure, the subsequent steps do not take place.

Boot-strap VP Expansion. When the microkernel switches to the allocated TDM slot of the newly created VP, the VP BD is fetched by the boot-strap code, using the allocated DMA. The boot-strap code is privileged code that reserves the rest of the required resources such as NoC connections and space in the shared memory as described by the VP BD using the RM *System API*. In this way, the *Boot-strap VP* is expanded to the full size of the VP as required by the application. If the reservation of the resources fail, the boot-strap code deallocates and releases the resources that it allocated and exits. In the next iteration of the SA, it detects that the boot-strap code has exited. The SA then de-allocates the BDs associated with the failed VP, thus completely removing the Boot-strap VP.

Loading Code & Data. After the completion of VP expansion, application code and data are fetched from the application binary and loaded in IMEM and DMEM in the allocated ranges, respectively. *Position Independent Code (PIC)* is used so that the application code only has relative jumps. Therefore the

application code is independent of its location in IMEM. The function calls, that point outside the scope of the application, such as OS and debug APIs, are accessed using a *Virtual Function Table*. It holds an array of pointers to (virtual) functions. In this step, these pointers are set to point to the right functions. To correctly access *data*, independent of its storage location, the compiler creates a *Global Offset Table (GOT)*, which contains the relative offsets to all global variables. The global variables are then accessed through indirect addressing using this table. The GOT is accessed using the *GOT* pointer (register r20 in Microblaze). After loading the application data in the allocated memory in DMEM, the *GOT* pointer and the table entries are updated to the new addresses.

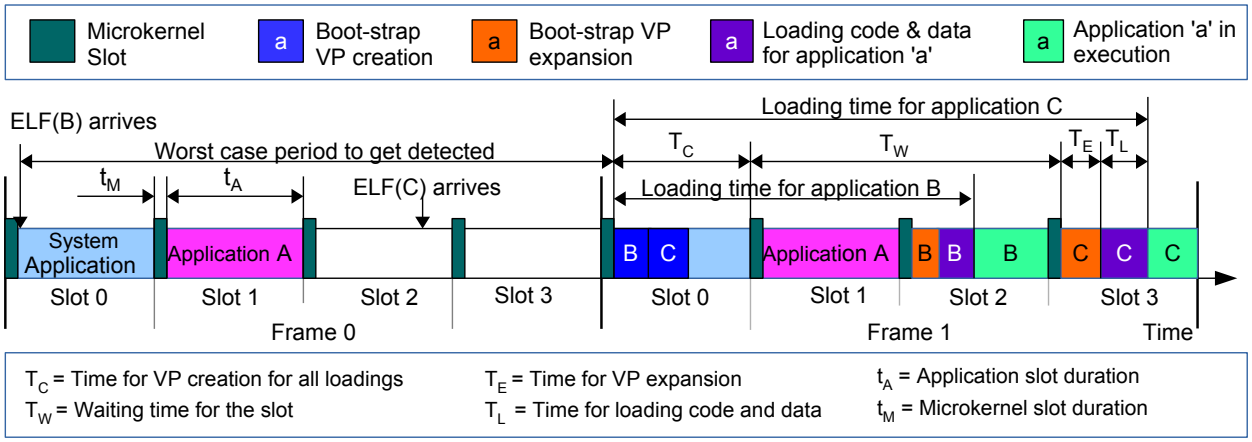

Figure 8: An illustration of the uniprocessor loading architecture.

The above loading steps are illustrated in an example shown in Figure 8. The figure shows the time line of a processor during loading. In the example, the processor is partitioned into TDM frames of four application slots. Between every application slot, a microkernel slot runs. The durations of the application slot (t_A) and the microkernel slot (t_M) are fixed and constant. Application A is an application that runs in slot 1. The ELF binary of application B happens to arrive just after the completion of the SA routine which checks ELF binaries. Therefore the ELF binary of application B will be detected in the SA slot of the next TDM frame. In fact this corresponds to the worst case period for an ELF binary to be detected and it is equal to the length of one TDM frame. At the start of the next frame, binaries of applications B and C are detected. Their *Boot-strap VP creation* steps are executed in slot 0. Slot 2 is allocated to application B and slot 3 to application C. In their respective application slots, the *Boot-strap VPs* are expanded by reserving the requested resources. After that the application code and data is loaded. The loading times for both applications are indicated.

The proposed loading process divides the loading time into four parts. These four parts are indicated in Figure 8. T_C is the SA's slot duration during which the first step of loading, i.e. *Boot-strap VP Creation* is executed for all the detected ELF binaries. T_W is the waiting time, after T_C, until the first allocated slot arrives. T_E is the time it takes to execute the second step of loading, i.e. *Boot-strap VP Expansion* and T_L is the time it takes to load application code and data. Thus the loading time (LT) for an application is

$$LT = T_C + T_W + T_E + T_L \tag{1}$$

In the following sub-section we shall explain how LT is composable and the TDM design constraints for it to be composable.

6.2. Composability

To ensure composable loading, each of the four parts of the loading time (LT), viz. T_C, T_W, T_E and T_L, should be independent from execution of other running applications or other simultaneous loadings.

T_C is made composable by always spending a fixed time on it, for all possible loading scenarios. This fixed duration is set to the duration of the SA's time slot and the SA is assigned the first TDM slot. In this way all application binaries that have to be loaded, are handled in the same TDM frame, in which they are detected. This defines the minimum duration of the SA TDM slot. For simplicity, we assume the TDM schedule consists of TDM slots of equal duration. Thus, the duration of the TDM slot in the system should be at least equal to the total time required to create the *Boot-strap VPs* for maximum number of applications that can be loaded simultaneously. In a processor partitioned in n TDM slots, the maximum number of applications that can be loaded is $(n-1)$, since the SA occupies one TDM slot. We can divide the *Boot-strap VP creation* step in three parts. Let T_α be the worst-case time taken to parse the ELF binary to load the VP BD, T_β be the worst-case time it takes to allocate all the memory sections (stack, heap, space in IMEM and DMEM) and T_γ be the worst-case overhead spent in the RM API to reserve and allocate the *Boot-strap VP* in the system. The worst case time bounds for T_α and T_β exists due to the predictable nature of the loading process as explained in the next subsection. The RM API is designed to have a bounded overhead. Hence for composability the TDM time slot duration t_A has to meet the following constraint.

$$t_A \geq (n-1) \times (T_\alpha + T_\beta + T_\gamma) \tag{2}$$

T_W is the waiting time, until the first allocated processor TDM slot arrives, hence it depends on the allocated TDM slot(s). For TDM slot allocations which are decided at design-time, the duration of T_W can be computed and it will be the same for every possible loading. Thus it is independent of all the running applications or other simultaneous loadings, therefore it is composable. If however, the processor TDM slot allocation is decided at run-time, then the SA can allocate the requested number of TDM slots, as per the availability of free TDM slots. In this case, the T_W is not of a fixed duration, thus it would not be composable, however it has an upper bound which can be computed. Therefore it will be predictable. The maximum duration of T_W in a system with n slots will be when the application is assigned the last TDM slot in the TDM frame. Therefore, after completion of the SA TDM slot, the waiting time until the last TDM slot arrives is $(n-2)$ TDM slots. During this waiting period, $(n-1)$ microkernel slots occur. Hence the maximum duration of T_W is given by $T_{Wmax} = (n-2)t_A + (n-1)t_M$. In this paper we assume TDM allocations are done at design-time.

T_E indicates the *Boot-strap VP Expansion* step. Since it is entirely executed in the application's time slot, it is free from interference from other applications or other simultaneous loadings and vice versa. However, during reservation, resources are locked and if the resource reservations for a VP do not finish within the same application slot, then other applications that try to reserve resources may experience interference. Therefore for the *Boot-strap VP Expansion* step to be composable, reservation for all the resources required to expand the application VP should finish within one application slot. Hence for all applications that have to be loaded, constraint 3 should be satisfied, where R is the set of BDs that are reserved and allocated in the *Boot-strap VP Expansion* step (i.e. all BDs required by the application, except the BDs of the processor and the DMA) and t_r is the time taken to reserve and allocate the BD r.

$$t_A \geq \sum_{r \in R} t_r \tag{3}$$

T_L involves loading the application code and data which is also done entirely in the application's time slots. Additionally the loading is done using the composable communication architecture(composable NoC with composable Memory controller), as a result the communication is composable. After VP expansion, DMA and allocated memory ranges in CMEM, IMEM and DMEM are exclusively owned by the application VP. This ensures independence from other applications and also this loading step does not affect other VPs. In this way, T_L is composable.

Therefore we may conclude that the loading process is composable provided that the TDM slot duration meets Constraints 2 and 3.

7. Multiprocessor Loading Architecture

The following subsection explains the multiprocessor loading architecture to load multiprocessor application binaries and the subsection after that explains the composability property of our multiprocessor loading architecture.

7.1. Loading Architecture

To load an MPSOC application that spans multiple tiles, we need to create the VP required by the application. The VP for the application can be created by reserving and allocating resources on the required tiles. On successful reservation of the required resources on all the required tiles, the application object code on the respective tiles can be loaded. In the uniprocessor loading architecture, the **System Application (SA)** is able to reserve, allocate and release all the local resources by using the local RM library. However an SA on a tile cannot reserve resources that belong to other remote tiles. Furthermore, an additional entity is necessary to 1) command the SAs on the tiles to initiate the partial platform reservation, 2) to check the reservation status of all the required tiles and 3) on successful reservation of VP, to command the required SAs to allocate the VP and load the respective application object code on their tiles. And in case of failure on any of the required tiles, to command all the SAs to abort loading and release the reserved resources. We propose a master-slave software architecture that satisfy these requirements.

As a master, we have the **System Manager (SM)** application. For an MPSOC platform, there is precisely one SM application, running on one of processor tiles. As slaves, on each of the processor tiles in the platform, including the one which hosts the SM, there is one SA. Both the SM and the SAs require a VP with at least 1 TDM slot in the TDM schedule of their respective processors, 1 DMA, 1 CMEM and a connection to the shared memory. Additionally there are NOC connections from the SM to every SA and vice versa. At the system start-up, the required VPs for the SM and the SAs are allocated. Then, by utilizing the NOC connections, the allocated DMA and CMEM, FIFO channels are setup from the SM to every SA and vice versa. The SM and the SAs use these FIFO channels to send messages to each other. Both the SM and SA have the privileges to access the RM *System API*. The loading architecture works as follows.

The loading architecture consists of 2 parts. In the first part, the SM and SA work together to create *Boot-strap VPs* for all the detected application bundles, by reserving minimum required resources (processor TDM slots, memory space in IMEM, DMEM and a DMA). In the second part, for each application, in their respective allocated slots, the *Boot-strap VP* is expanded to the full required platform by reserving the rest of the required resources and the application code and data of the detected applications are loaded in the IMEM and DMEM of the respective allocated processors. The first part consists of 4 phases, explained below. Phases 1 and 3 are carried out by SM and phases 2 and 4 are carried out by SA.

Phase 1. In phase 1, the SM executes 3 tasks. In task 1, the SM at the start of its slot, checks for application bundles in the shared memory in predefined locations and fetches the VP BD for each of them. In task 2, for each detected application bundle, the SM checks to which tiles the tile bundles are mapped. Based on this information, the SM computes the number of tile bundles that each SA has to load. The SM then sends the respective number to the respective SAs. Finally in task 3, for each application, by reading it's VP BD, it acquires the addresses of the tile bundles and sends these addresses to the respective SAs to which this tile bundles are mapped.

Phase 2. In phase 2, the SAs also have 3 tasks. In task 1, they receive the message from SM which contains the number of application tile bundles they are expected to load. Based on this number the SAs repeat tasks 2 and 3. In task 2, for an application, by using the tile bundle pointer, the SAs read the Processor BD, which is contained in the Tile BD. The Processor BD specifies the requirements of the processor TDM slots and the space in the IMEM and DMEM. The Processor BD is then reserved to create a *Boot-strap VP*. Additionally a DMA is reserved, to load the application code and data, in the application's own partition. The status of the reservation (`SUCCESS/FAIL`) per application is then sent to the SM in task 3.

Phase 3. In phase 3, for each detected application, the SM receives the reservation status from all the mapped tiles. If the reservation succeeded on all the tiles, the SM sends a `LOAD` message to the respective SAs. Otherwise an `ABORT` message is sent to all the mapped tiles.

Phase 4. In phase 4, for each detected application, the SAs check the message from the SM. For a `LOAD` message, the SAs allocate and enable the reserved *Boot-strap VP*. For an `ABORT` message, the reserved *Boot-strap VP* is deallocated and the resources are released.

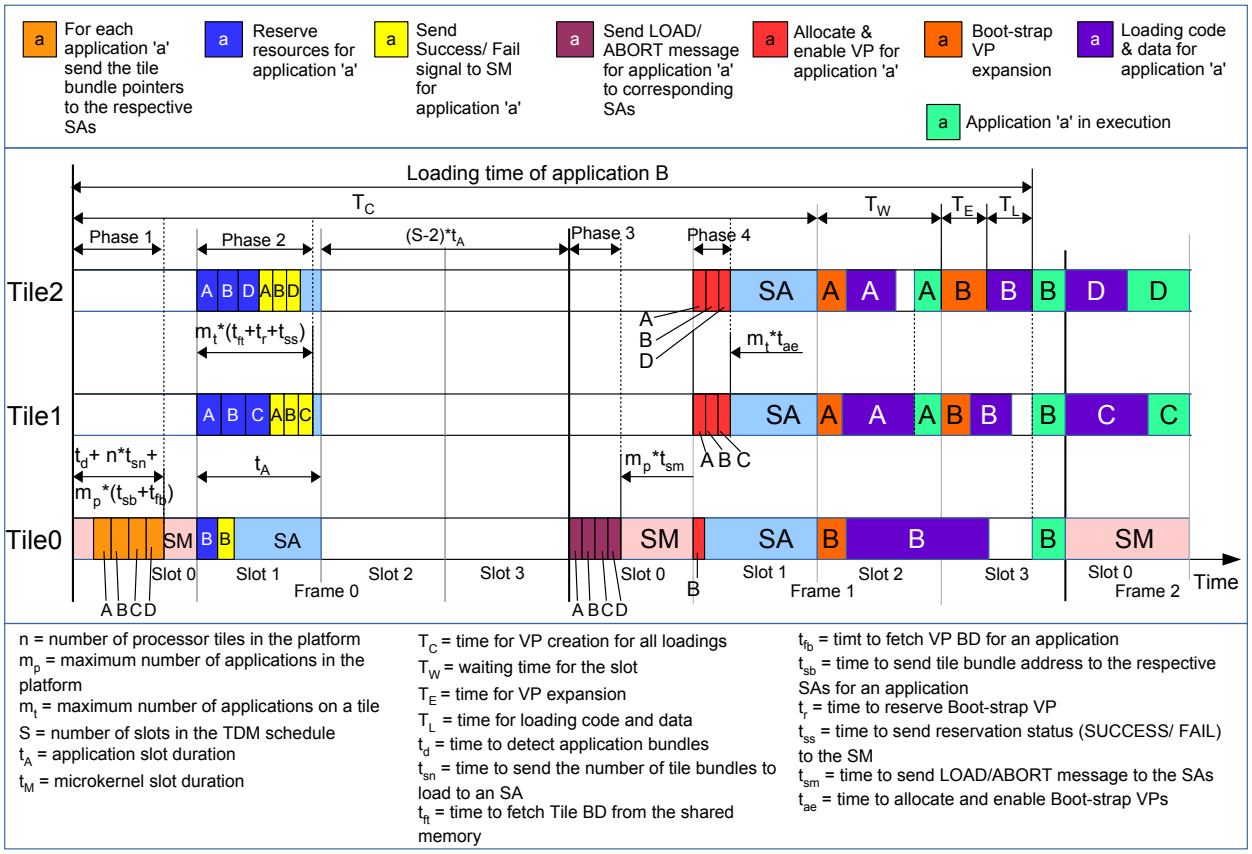

Figure 9: An illustration of multi-tile loading architecture on an MPSOC platform with 3 processor tiles.

It should be noted that since all the messages between SM and SA are sent via lossless FIFO channels, the order of messages sent in phase 1 match with the order of messages received in phase 3. Similarly the order of messages sent in phase 2 match with the order of messages received in phase 4. Therefore the messages do not require a header that specifies for which application is the sent message meant.

After the completion of the 4th phase, the second part of the loading begins. In the second part, there are two steps. In the first step, the allocated boot-strap platform is expanded to the required full platform by reserving the rest of the required resources. This step is called the *Boot-strap VP expansion* step. In the second step, application code and data are loaded in the allocated IMEM and DMEM regions respectively. This step is called *Loading code and data*. In case a tile fails to reserve resources in the *Boot-strap VP expansion* step, the boot-strap code exits. This is detected by the local SA and accordingly it sends a failure message to the SM. The SM on receiving this failure message, signals the SAs on all the tiles of the application to deallocate and remove their partial VPs. The SAs on the other tiles of the applications, on receiving the abort message, deallocate all the resources reserved for the application and remove the partial VP by deallocating the reserved slots in the processor TDM schedule.

The resource requirements for an application on different tiles may be different. Consequently the *Boot-strap VP expansion* step may take different amounts of time on different tiles. Similarly the amount of

code and data on different tiles may vary, and therefore loading of the application code and data may finish at different times on different tiles. Accordingly the application code on different tiles would start their execution at different times. This would lead to race conditions and therefore the application may deadlock during execution. For example, consider a case of FIFO set-up. FIFO administration needs to be set-up by both the producer and the consumer tiles. If the producer tile has completed its loading before the consumer tile, it initializes its FIFO administration and sends a token into the consumer tile. If the consumer tile is delayed in its loading, when it begins execution, it will initialize it's own local FIFO administration, thus overwriting the sent token by the producer. Hence, the consumer would either miss a data token or would block forever. To avoid such a situation, each tile does not begin the execution of the application, until all the required tiles by the application have finished the loading of code and data on their respective tiles. To implement this behavior, we have built **Barrier Synchronization**. The implementation of the barrier synchronization is presented in Appendix A.

With barrier synchronization, all the tiles of the application begin execution, after all of them have composably reached the barrier. We define the **Loading Time** for loading multiprocessor applications as *the time from the beginning of the slot in which the SM detects the application bundle until the time when the application begins execution.* The application begins execution when all the tiles of the application have finished the second part of the loading. The *Loading Time* is therefore T_C plus the maximum of the time it takes to complete the second part of loading on each tile on which the application is mapped. The loading time LT_a for a multiprocessor application a is thus represented as follows.

$$LT_a = T_C + max_{t \in M_a}(T_W^t + T_E^t + T_L^t) \tag{4}$$

where M_a is the set of tiles on which application a is mapped. T_C, is the duration in which the *Boot-strap VP* for application a is created. T_W^t, T_E^t, and T_L^t are the waiting time for the allocated slot, the time for *Boot-strap VP expansion* and the time for *Loading code and data* for application a on tile t respectively.

The slot allocations of SM and SA affects the loading time. To detect the application bundles as early as possible, we allocate 1 slot, slot 0 to the SM. It is also possible to allocate slot 0 to SAs (except for the SA on the processor which hosts SM). However, it is more efficient to allocate slot 1 as explained in Section 7.2.

The loading architecture is illustrated in Figure 9. In this figure we show simultaneous loading of 4 applications (applications A,B,C and D) on a platform with 3 processor tiles. Each processor is partitioned in a TDM frame of 4 slots. We assume all the processors are running on synchronized clocks and the TDM slots are aligned. Application A is mapped to 2 tiles (tiles 1 & 2) and has been allocated slot 2 on each tile. Application B is mapped to 3 tiles (tiles 0,1 & 2). On tile 0 it has been allocated slots 2 & 3 and on the other tiles, it has been allocated slot 3. Applications C & D are mapped to tiles 1 and 2 respectively and have been allocated slot 0 on the respective tiles. As an example the loading time of application B is illustrated. The loading time of application B is accounted from the moment it's application bundle is detected until the moment, when all the required tiles have completed the second part of loading. In the shown example, tile 2 happens to be the last tile to finish the second part of the loading, and after it's loading, application B begins execution on all tiles. Hence the loading time for application B is accounted until tile 2 finishes it's loading.

7.2. Composability

For the multiprocessor loading architecture to be composable, the loading time given by equation 4 should be composable, that is these parts should be independent of the running applications and other simultaneous loadings. Equation 4 has two parts, the first part is T_C and the second part depends on T_W, T_E and T_L on each tile.

The second part is composable due to the same reasons as in the uniprocessor loading. Namely, the waiting time for the slot (T_W) depends only on slot allocation. In this work we assume static processor slot allocation, therefore the waiting time is fixed and does not depend on running applications or other simultaneous loadings and vice versa. And T_W and T_L, representing the *Boot-strap VP expansion* step and

the *Loading code and data* step, take place only in the application's own partition. Thus T_W, T_E and T_L are also independent from running applications or other simultaneous loadings.

Similar to the uniprocessor loading architecture, the first part, T_C, is made composable, by always spending a fixed time on it, for all possible loading scenarios. This fixed duration should be as small as possible to shorten the loading time, but long enough to finish creating the *Boot-strap VPs* for the maximum possible applications that can be loaded on a platform. The following gives an explanation on how we derive this fixed duration of T_C.

T_C in the multiprocessor loading architecture consists of 4 phases. Finishing the tasks of a phase or phases in 1 TDM slot duration, helps prevents elongating the duration of T_C by 1 TDM frame, since otherwise the work in that phase(s) continues only in the next TDM frame. The execution of these 4 phases depends on the slot allocations of the SM and the SAs. As mentioned in the previous subsection, we allocate one slot, slot 0 to the SM, to detect application bundles at the earliest. And therefore the SA on the tile containing the SM can only have slot(s) allocated ≥ 1. To shorten the duration of T_C, the SAs on the other tiles can also be allocated slot 0. However such an allocation has two drawbacks. The first drawback is that the SAs shall remain idle until the SM completes part of the phase 1 on which they are blocking, thus wasting time. The tasks in phase 1 and phase 2 should finish within the duration of slot 0, otherwise work of phase 1 and phase 2 will continue in the next TDM frame. Therefore the second drawback of allocating slot 0 to the SAs, is that the TDM slot duration will have to be wide enough to finish the phases 1 and 2 for the maximum number of applications that can be loaded on the platform. Therefore we decide to allocate all the SAs the next earliest possible slot, slot 1. Note that a later slot would delay the completion of T_C. To avoid elongating T_C and therefore lengthening the loading time, only a single slot is allocated to the SAs.

With the derived slot allocations for the SM and the SAs and with the constraint that the work in each phase should finish within 1 TDM duration, the time gap between phases 2 and 3 is equal to the duration of $S-2$ TDM slots, where S is the number of TDM slots in the TDM frame. As a result the fixed duration of T_C is equal to the length of 1 TDM frame plus 2 TDM slots. The structure of T_C is illustrated in Figure 9. For T_C to be composable, each phase should finish within 1 TDM slot duration, that is the TDM slot duration should be designed in such a way that the worst-case work for each phase finishes within the duration of 1 TDM slot. This constraint helps define the TDM design constraints to make the multiprocessor loading architecture composable. We elaborate the TDM design constraints for each phase below.

To help frame the TDM design constraints, we first define some useful symbols described in Table 1. m_t is bounded by $S-1$, since on each tile, the SA occupies 1 slot. And m_p is bounded by $(n-1) \times m_t + (S-2)$ as on the tile hosting the SM, we have the SM and the SA, each occupying one TDM slot. The bound on m_p is large and designing a system to support the bounded value of m_p will lead to large slot sizes. The system can be also be designed for an m_p that is smaller than the maximum bound of m_p based on the knowledge of the number of maximum applications that will running simultaneously.

t_M	Microkernel slot duration.
t_A	Application slot duration.
S	The number of TDM slots in the TDM frame.
n	The number of processor tiles in the platform.
m_t	The maximum number of applications that can be loaded on a tile.
m_p	The maximum number of applications that can be loaded on the platform.

Table 1: Symbols to define the TDM design constraints

Phase 1. We define t_d to be the worst-case time it takes on a platform to check the pre-defined locations for the maximum possible loadable application bundles. Let t_{fb} be the worst-case time it takes to fetch the VP BD from the shared memory, for an application. We define t_{sn} to be the worst-case time it takes for the SM to send the largest number of tile bundles that each SA has to load. Let t_{sb} be the worst-case time it takes on a platform for the SM to send the tile bundle addresses to all the tiles for an application. Thus the

worst-case time to complete work in phase 1 is $t_d + n \times t_{sn} + m_p \times (t_{fb} + t_{sb})$. Accordingly the constraint for phase 1 is as follows.

$$t_A \geq t_d + n \times t_{sn} + m_p \times (t_{fb} + t_{sb}) \tag{5}$$

Phase 2. In phase 2, for each application, the SA reserves the *Boot-strap VP* and sends the reservation result (`SUCCESS/FAIL`) to the SM. After receiving the tile bundle pointer, the SA fetches the Tile BD to read the Processor BD. Let t_{ft} be the worst-case time it takes to fetch the Tile BD from the tile bundle in the shared memory. Let t_r be the worst-case time it takes to reserve the *Boot-strap VP* for an application. Let t_{ss} be the worst-case time it takes for the SA to send the reservation result (`SUCCESS/FAIL`) for an application to the SM. Then the worst-case time to complete work in phase 2 is $m_t \times ((t_{ft} + t_r + t_{ss})$. Thus the constraint for phase 2 is as follows.

$$t_A \geq m_t \times (t_{ft} + t_r + t_{ss}) \tag{6}$$

Phase 3. In phase 3, if all the SAs on the tiles required by an application succeeded in creating the *Boot-strap VP* on their respective tiles, then the SM sends a `LOAD` message to the respective SAs, else the SM sends an `ABORT` message. Let t_{sm} be the worst-case time it takes to send `LOAD/ABORT` message to the SAs on all the tiles, for an application. As a result, the worst-case time to complete work in phase 3 is $m_p \times t_{sm}$. Consequently the constraint for phase 3 is as follows.

$$t_A \geq m_p \times t_{sm} \tag{7}$$

Phase 4. In phase 4, the SAs receive the `LOAD` or `ABORT` message for each application to be loaded. On receiving the `LOAD` message for an application, the SAs allocate and enable the *Boot-strap VP* for that application. Allocating and enabling the *Boot-strap VP* is done with a predictable code. Let t_{ae} be the worst-case time it takes to allocate and enable the *Boot-strap VP*. Then the worst-case time to finish work in phase 4 is $m_t \times t_{ae}$. Hence the constraint for phase 4 is as follows.

$$t_A \geq m_t \times t_{ae} \tag{8}$$

For an `ABORT` message, the SAs deallocate the *Boot-strap VP* and release the reserved resources. Deallocating and releasing the resources is also done with a predictable code. Let t_{dr} represent the worst-case time to deallocate and release resources. Then an additional constraint for phase 4 is as follows.

$$t_A \geq m_t \times t_{dr} \tag{9}$$

The above constraints for each phase are also illustrated in Figure 9.

8. Predictability

In this section we first discuss the requirements for the single and multiprocessor loading architectures to be predictable and how these requirements are met. Owing to the predictable nature of the loading architecture, it is possible to compute a bound on the loading time for both the single processor and multiprocessor loadings. In the following subsection we explain how to compute this bound.

8.1. Requirements

For the uniprocessor and multiprocessor loading architectures to be predictable, the loading time given by equations 1 and 4 should have a known bound. Both the equations depend on T_C, T_W, T_E, and T_L. In the multiprocessor loading architecture except for T_C, the three parts are the same as in the uniprocessor loading architecture and have the same requirements as the uniprocessor case. To have a bound for T_C in multiprocessor loading, in addition to the requirements of the uniprocessor loading, an extra requirement is that the coordination between the SM and SA completes in a known time bound for each loading. With the proposed multiprocessor loading architecture, T_C takes a fixed duration by design as explained in Section 7. The fixed duration depends on the slot allocation of the SM and the SAs and the duration of the TDM

slot. With fixed TDM slot allocations at design-time, the waiting time for the first allocated slot in the application VP, T_W, is known. To have a computable bound for T_C, T_E, and T_L the necessary requirements for both the loading architectures are, 1) the application binary should be of known size, 2) the resource allocation algorithms should have a known worst-case execution time, and 3) the involved platform resources (processor, DMA, NOC, and the shared memory) should provide guaranteed performance.

With regards to requirement 2, the TDM slots are allocated at design time. For memory allocation, we employ a naive predictable memory allocator. The memory is divided into blocks of a fixed size. The allocator iterates through the list of memory blocks and allocates the first free contiguous blocks of memory that fit the requirement. The worst-case for the memory allocator occurs when the allocator has to iterate through the complete list of blocks. During allocation, there are only accesses to the local memory (IMEM,DMEM). Since the processor execution time and the time to access local memories are predictable, the worst-case execution time to iterate through the complete list of blocks can be computed. For resource allocations of NOC connections, a predictable allocation algorithm described in [37] is used.

With regards to requirement 3, we use the platform in [13], which is predictable. The processor is shared using TDM, which ensures a guaranteed share. Since each application is exclusively allocated a DMA, it gets its full share. The NOC is also predictable and a bound on the worst-case communication time for a given amount of data can be computed with the dataflow model presented in [38]. The shared memory has a predictable front-end and a bound on the time to serve memory read or write requests can be computed using the *Latency-rate Server* model as described in [30].

8.2. Computing a Bound on the Loading Time

To compute a bound on the loading time for both the loading architectures, a bound for each term in equation 1 and 4 has to be computed. In equations 1 and 4, T_C is of fixed duration, by design. For uniprocessor loading, T_C is equal to the duration of one TDM slot. For multiprocessor loading, T_C is equal to the duration of one TDM frame plus 2 TDM slots, as explained in Section 7. For design-time TDM slot allocations, T_W is known. T_E depends on the worst-case time for resource allocation for all the resources allocated in the *Boot-strap VP expansion* step. On our platform, NOC connections are allocated in this part of the loading using the predictable allocation algorithm in [37] and thus the bounds provided in [37] can be used. T_L depends on the communication time between the processor and the memory, the implementation of the loading code, and the TDM schedule. Each of these factors are analysed below.

8.2.1. Computing a Bound on communication time

In the uniprocessor and the multiprocessor loading methods, during the loading step the boot-strap code loads the application code and data from the shared memory. To compute a bound on the loading time, the communication between the processor and the shared memory is modelled. The design of such a model depends on the communication architecture in a platform. We explain below the key elements of such a model for the platform in [13]. For other platforms, the model needs to be adapted based on the communication architecture of the target platform.

To load the application code and data, the boot-strap code issues a DMA read request. The DMA read transaction request is released from the 1) DMA and it goes through the 2) de-multiplexer, 3) the NOC, and finally arrives at 4) the memory controller of the shared memory. The response packets follow the same route back to the DMA which stores them in the DMA memory. To model the communication to the shared memory, the listed four hardware components are modelled. We make use of the Cyclo-static dataflow (CSDF) model [39] to model each of these hardware components. The DMA and the de-multiplexer have simple state machines. To model these two hardware components, we have an actor representing each of the states of these components. For a DMA read transaction, the resulting state transitions in these hardware components are known. The rates of the CSDF actors are set to the corresponding number of output tokens for the respective state transitions. The execution time of each of the actors is derived from the execution time of the respective hardware component in the respective state. We make use of the single actor model from [38] to model the NOC. The single actor captures the worst-case latency for each packet between the processor tile and the memory tile. This latency can be computed based on the NOC topology and the size

of the TDM schedule in the NOC. The worst-case performance of the composable memory controller can be computed with the *Latency-rate server* model described in [30]. We construct a dataflow model that models the *Latency-rate server* in [30] of the shared memory with the technique described in [40]. In this model there are two actors, a latency actor which captures the worst-case latency to serve a memory read request and the rate actor which captures the worst-case throughput provided by the memory. Based on the hardware structure of the memory controller we derive the latency and rate of the memory controller used in the Compsoc platform [13]. The dataflow models of each of these hardware components are connected based on the structure of the communication in the platform to form a complete DMA *read* model. The DMA splits the read transaction request into equivalent requests of a smaller chunk size. Based on the size of this chunk, we compute the number of the resulting service units in the rest of the hardware components. These resulting number of service units are used to derive the token rates of the respective connecting edges in the DMA *read* model. In the constructed DMA *read* model, other than the read transaction size, all the rest of the parameters are based on the hardware components. In this way we construct a parametric dataflow model that models a DMA read transaction of arbitrary size. This model is used to compute a bound on the time it takes to read application code and data from the shared memory as explained in the next subsection.

8.2.2. Analysis of the Loading Code

We analyze our implementation of the loading code to compute the bound on T_L. A brief outline of the implementation of the loading code is listed in Algorithm 1. The loading API is composed of four parts. In part 1, the ELF header is parsed to fetch and store the section headers and the table with section names. The duration of executing this part depends on the size of the section headers and the number of sections. The number of ELF sections is fixed during generation of application binary. As a result, the time to execute this part is independent of the application binaries, but depends only on the execution time of the processor, the DMA and the NOC connection to the shared memory. The processor and the DMA have predictable execution time. The NOC is also predictable. Thus the bound to execute this part can be computed for a platform, based on the loading code and the DMA *read* transaction dataflow model. This computed time will be the same for all application binaries on that platform. Let δ_{header} represent the time in cycles to execute part 1.

Input : Address of the tile bundle.

```
1  Define TO_LOAD_SECTIONS = {.ctors, .dtors, .rodata, .bss, .patchdata,.data,.text};
2  // Part 1
3  Fetch the ELF header;
4  Parse the ELF header to get the section table, section headers;
5  Fetch and section names ;
6  // Part 2
7  foreach section in tile bundle in TO_LOAD_SECTIONS do
8  |  DMAReceive() ;
9  |  CopyDMAmemtoLocalmem();
10 end
11 // Part 3
12 Patch the instructions in IMEM at addresses given in .patchtable ;
13 // Part 4
14 Signal the barrier ;
15 while other tiles have not reached barrier do wait ;
16 Jump to main() of the application ;
```

Algorithm 1: Pseudocode of the loading code.

In part 2, every section in the application tile bundle that needs to be loaded (defined by `TO_LOAD_SECTIONS`) is loaded. To load a section two functions are used. The first function is the `DMAReceive()` function, which

programs the DMA to read a section from the provided shared memory address into the provided DMA memory address. The worst-case time to complete this memory read request is computed using the dataflow model for a DMA *read* transaction, explained in Section 8.2.1. For reading B bytes from the shared memory into the DMA memory, we let the function $\Delta(B)$ return the worst-case time in cycles to read B bytes by evaluating DMA read dataflow model for a read request of B bytes.

The `CopyDMAmemtoLocalmem()` function copies a section from the DMA memory to the allocated region in the IMEM or the DMEM. Only the `.text` section is loaded into the IMEM, the other sections are loaded into the DMEM. In the implementation of `CopyDMAmemtoLocalmem()`, to copy each word of 4 bytes, it takes 8 cycles. Additionally, the `CopyDMAmemtoLocalmem()` has a fixed overhead each time it is used. We define a total overhead as δ_{copy} cycles, such that, to copy a section of B bytes, `CopyDMAmemtoLocalmem()` takes no more than $\lceil \frac{B}{4} \rceil \times 8 + \delta_{copy}$ cycles. In this way the worst-case time to load a section s is $\Delta(size(s)) + \delta_{copy} + \left\lceil \frac{size(s)}{4} \right\rceil \times 8$, where $size(s)$ returns the size of a section in bytes.

In part 2, the code that searches and matches the section names and provides the DMA memory address, shared memory address and the local memory address has the worst-case execution time, when the maximum number of sections are found in the tile bundle. Let δ_{search} represent the worst-case time to execute this code. Thus the total worst-case time in cycles to complete **Part 2** is $\delta_{search} + \sum_{s \in SECT} \Delta(size(s)) + \delta_{copy} + \left\lceil \frac{size(s)}{4} \right\rceil \times 8$, where $SECT$ is the set of sections in the tile bundle that have to be loaded.

Part 3 of the loading code is an extra part that is necessary to add patches to the position independent code during the loading process. This is done in two steps. In the first step, at compile time, when generating the application binary, the generated assembly is scanned for instructions that access global variables and the location of these instructions are stored in a table. This table is stored in the `.patchdata` section. The second step is executed in part 3, where the table in `.patchdata` section is read and the listed instructions are updated to point to the correct locations. In our implementation it takes $262 + 156 \times P/4$ cycles to execute part 3, where P is the size of `.patchdata` section in bytes.

In part 4, the loading code signals the barrier indicating that it has completed the loading on this tile and waits until other required tiles reach their barrier using **Barrier Synchronization** (more details can be found in Appendix A). After completion of the 4 parts, the loading code jumps to the `main()` function of the application, thus beginning the execution of the application.

To compute the bound for T_L, the first three parts of the loading API have to be accounted. Let T'_L be the time in cycles to load the application code and data assuming the full share of the processor time is allocated for executing the loading API. Then by combining the worst-case time for the first three parts, $T_{L'}$ is as follows.

$$T'_L = \delta_{header} + \delta_{search} + \sum_{s \in SECT} \Delta(size(s)) + \delta_{copy} + \left\lceil \frac{size(s)}{4} \right\rceil \times 8 + 262 + 156 \times \frac{P}{4} \tag{10}$$

8.2.3. TDM Arbitration on the Processor

The processor is shared with different applications with TDM arbitration. For known TDM slot allocations to an application on a given TDM schedule, it is possible to compute the exact time it takes to complete a given amount of work. However such computation is not trivial if the allocated slots are discontiguous. Therefore we compute the worst-case time it takes to complete a given amount of work on a TDM schedule when the total number of slots allocated to an application is known. This worst-case bound can be computed by calculating the number of TDM frames required to complete the given amount of work, when a known amount of slots are allocated. If the application is allocated a total of p slots in a TDM schedule with S slots, then the worst-case time in cycles to complete T'_L amount of work on a processor tile t is as follows.

$$T^t_L = \left\lceil \frac{T'_L}{p \times T_A} \right\rceil \times S \times (T_A + T_M) \tag{11}$$

Where T_A is the application slot duration in cycles, T_M is the microkernel slot duration in cycles and $S \times (T_A + T_M)$ is the TDM frame duration.

9. Experiments

In the following subsection we detail the experiments which verify the composability of the uniprocessor loading architecture. Next we illustrate the experiments to verify the composability of the multiprocessor loading architecture. In the last subsection we describe the experiments which demonstrate the predictability of both loading architectures.

9.1. Composability of the Uniprocessor Loading Architecture

In this subsection we first give a brief description of the experimental setup and the applications chosen for experiments. Next we evaluate the TDM constraints for the uniprocessor loading architecture. Then we conduct the composability experiments with the designed TDM slot duration.

9.1.1. Experimental Setup

The hardware platform instance used in our experimental setup is shown in Figure 10. We implemented the hardware platform on a Xilinx ML605 FPGA board. We use a MicroBlaze soft-core processor has an instruction memory (IMEM) and a data memory (DMEM) of 256KB each. The platform consists of 3 sets of DMAs with their respective DMA memories (DMA MEMs) and communication memories (CMEMs). The DMA MEMS and the CMEMs are of 16KB each. The platform runs on a clock frequency of 120MHz.

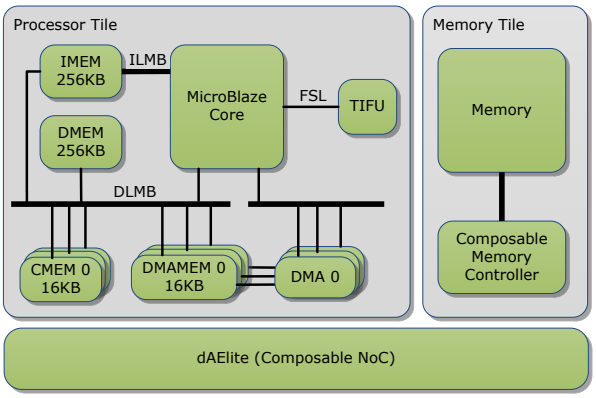

Figure 10: A single tile composable hardware platform instance (from [19]) .

9.1.2. Applications

For the experiments we choose three applications with different criticalities. *Tick* is a hard real-time application, which produces ticks at defined periodic intervals which are used to monitor various sensor and telemetry data. *MP3-decoder* is an example of a soft real-time application. *Susan* [41] is an example of a non real-time image processing application used for edge detection. We implemented these applications in C and compiled them using the MicroBlaze GCC toolchain. The details of the resulting application binaries are shown in Table 2. Each application has fixed processor TDM slot assignment encoded in the application binaries. *Tick* is allocated slot 1, *Susan* is allocated slot 2 and *MP3-decoder* slot 3. SA runs in slot 0.

9.1.3. TDM Design

For executing the considered applications we need 3 TDM slots and for executing the SA an additional slot is needed. Hence we partition the processor in a TDM frame of four slots. We derive the TDM slot duration which satisfy Constraints 2 and 3 below.

Constraint 2 requires us to know the timings of T_α, T_β and T_γ for the largest VP in the system. The largest VP in the considered platform consists of 3 DMAs with 3 corresponding NoC connections and 1 tile with 1 processor. Fetching and parsing the ELF binary to load the VP descriptor of the largest platform takes 7806 cycles in our system. Thus $T_\alpha = 7806$. We emulate the worst case condition to allocate memory, using the allocator explained in Section 8 on our platform to determine the worst case time to allocate memory. It takes 1673 cycles. Assuming allocating stack, heap, space in IMEM and DMEM, each take the worst-case time, $T_\beta = 4 \times 1673 = 6692$. The RM API overhead in our setup to reserve, allocate and enable the largest VP is 1926 cycles, thus, $T_\gamma = 1926$. Therefore according to constraint 2:

$$\begin{aligned} t_A &\geq (4-1)(7806 + 6692 + 1926) \\ &\geq 49272 \end{aligned}$$

The SA needs some additional time for book-keeping. We choose the duration of application slot to be 65536 cycles which satisfies constraint 1. The microkernel slot is 12288 cycles.

In the *Boot-strap expansion* step, required resources other than the processor and memory are reserved and allocated. In the considered hardware platform, other than the NOC, every other resource (DMAs, CMEMs) takes a short time(few hundred cycles) for reservation and allocation. For the NOC, we employ an online allocation algorithm presented in [37]. The worst case time for finding an allocation in a 4×4 mesh with a NOC TDM table of 16 slots, is 6394 cycles as reported in [37]. The NOC in the considered platform is 2×1 mesh with the NOC TDM table of 7 slots. The algorithm in [37] performs an exhaustive search of minimal path length, based on the backtracking method on a table whose dimension is ($NumberOfLinks \times LengthofNOCTDMtable$) (for details, the reader is referred to [37]). Since the table dimensions of the NOC in the experiment platform is smaller than the dimensions of the table for the NOC in [37], the NOC allocation algorithm will have even smaller worst-case allocation time than the 4×4 mesh. Therefore the chosen t_A also satisfies Constraint 3.

9.1.4. Composability Experiments

For verifying composability seven independent experiments were conducted and the loading times were compared. These seven experiments do not represent all the combinations of loadings for the considered three applications, however the chosen combinations are sufficient to demonstrate that the loading process is isolated from the existing applications and other simultaneous loadings and vice versa. Figure 11 shows the processor timeline for each of these experiments. In each experiment application binaries are sent via the serial connection into the shared memory of FPGA. To simplify Figure 11, the timer is started at the detection of the first ELF binary.

In experiment A, only the *Tick* application is run by pre-loading it, which serves as an existing application. The periodic interval of tick is set to the duration of one TDM frame, so that in every frame we have one tick. After start-up code, the first tick arrives at 4198 cycles, from then onwards we get periodic ticks every 335872 cycles. In experiment *B*, while the *Tick* application is running, the *Susan* application is loaded, and in experiment *E*, the MP3-Decoder application is loaded while the *Tick* application is running. In experiment *C* only the *Susan* application is loaded with no other applications running in other slots. Similarly, in experiment *D*, only the *MP3-decoder* application is loaded with no other running applications in other slots. To emulate simultaneous loadings, in experiment *F*, both the *MP3-decoder* and the *Susan* applications are loaded, with the *MP3* application sent before the *Susan* application. In experiment *G*, the order is reversed, that is *Susan* followed by *MP3*.

In the experiments *A,B,E,F* and *G* the ticks occur at the defined periodic intervals even in the presence of loading applications. This shows that the loading process does not affect the existing running applications.

The loading time of the *Susan* application in isolation (experiment *C*), is the same as the loading time in experiment *B* (loading in presence of the *Tick* application). Similarly experiments *D* and *E* demonstrate

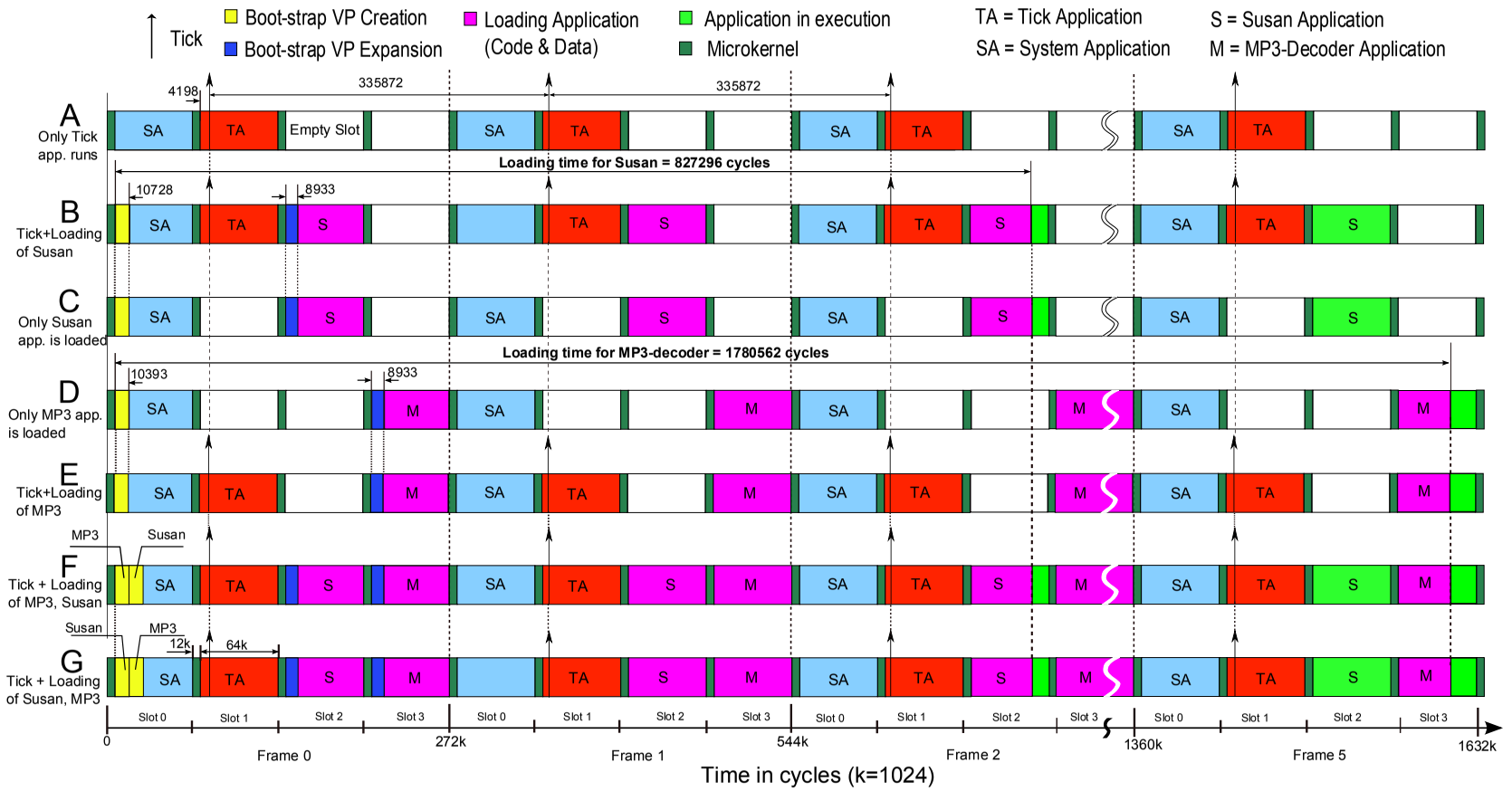

Figure 11: Processor timeline for single processor composability experiments.

that the loading time of *MP3-decoder* application is unaffected by the existing running application. This demonstrates that the loading time is unaffected by the existing running application. Thus we have shown that the loading process do not affect the existing applications and the existing applications do not affect the loading process.

The loading time of *Susan* application in isolation (experiment C) is the same as in the experiments F and G, wherein simultaneously the *Susan* and the *MP3-decoder* applications are being loaded. Similarly the loading time of the *MP3-decoder* application in isolation (experiment D) is the same as in the experiments F and G. In all of these experiments, we observe periodic *Ticks* at the defined interval. The exact same loading times for the respective applications and the occurrence of Ticks at defined intervals, in these different experiments, demonstrate that multiple simultaneous loadings do not affect each other or the existing application.

Due to the relatively larger size of *MP3-decoder* application, it completes loading in the fifth TDM frame. Therefore in Figure 11 the processor timeline jumps to the 5^{th} frame. The durations of the *Boot-strap VP Creation* steps for each of these applications are indicated. In the *Boot-strap Expansion* step for all the applications, no additional resources were necessary. However the VP BD section in the ELF binary is still fetched and checked and that takes the time as indicated in the Figure 11 for the respective applications.

9.2. Composability of the Multiprocessor Loading Architecture

In this subsection, we investigate the composability of the multiprocessor loading architecture.

9.2.1. Experimental Setup

The multiprocessor hardware platform used for composability experiments is shown in Figure 2. We implemented the hardware platform on a Xilinx ML605 FPGA board. It consists of three processor tiles and one memory tile. These 4 tiles are connected by the *dAelite* NOC [28] with a 2×2 MESH topology. Each of the processor tiles is composed of 1 MicroBlaze soft-core RISC processor and 4 sets of DMA and CMEM units. The MicroBlaze processor is connected to an IMEM and DMEM of size 128KB. Each DMA has a DMAMEM of size 8KB and each CMEM is of size 4KB. The memory tile hosts a shared SRAM memory of size 128KB.

9.2.2. Applications

For the multiprocessor experiments we choose 4 applications. Tick is the same hard-real time uniprocessor application used in uniprocessor experiments as explained in Section 9.1.2, which produces ticks periodically at defined intervals. We implemented two tile versions of the *Susan*[41] and *JPEG Decoder* applications. The JPEG Decoder application decodes JPEG images into BMP format. We have a three-tile hard-real time Electro-Mechanical Braking application [42] (called as *Braking Application* hereinafter). *Braking Application* is a control application, which controls the braking lever and ensures it reaches a reference position in a desired time. For each application, we allocate one slot on each of its tiles. Considering the slot allocation for the 4 applications and the slot requirements for the SM and the SAs, the minimum number of slots required on each tile is 4. The allocated tile and slot mappings for each application and the SM and the SA is shown in Figure 12. These applications were implemented in C and compiled using the MicroBlaze GCC toolchain. The details of the resulting application binaries are shown in Table 2.

9.2.3. TDM Design

As explained in Section 7.2, the maximum bound on m_p is large, as it is in the order of $O(n \times S)$, where n is the number of tiles in the platform and S is the number of TDM slots on the processor tiles. Using the maximum bound in the TDM constraints will lead to large slot sizes. Therefore we compute m_p based on the maximum number of applications that can be loaded simultaneously in our experiments. We run the *Tick* application by pre-loading it, which serves as an existing application. The rest of the three applications, viz. *Susan*, *JPEG Decoder* and the *Braking application* are loaded at run-time. Thus the maximum number of applications that are dynamically loaded simultaneously is 3. Therefore $m_p = 3$ and with the application slot allocations (as shown in Figure 12), $m_t = 3$ as well. For the multiprocessor loading architecture to be

Table 2: Section sizes in application binaries and the stack and heap requirements of the respective applications. All sizes are in bytes and K=1024

	Tick	**Susan (single-tile)**	**MP3 (single-tile)**	**Susan (multi-tile)**		**JPEG Decoder (multi-tile)**		**Braking Application (multi-tile)**		
				Tile 0	**Tile 1**	**Tile 1**	**Tile 2**	**Tile 0**	**Tile 1**	**Tile 2**
.ctors	8	8	8	8	8	8	8	8	8	8
.dtors	8	8	8	8	8	8	8	8	8	8
.rodata	6	6	6	136	108	930	70	34	114	46
.bss	32	272	272	55848	596	1628	49196	272	272	272
.patchdata	44	164	164	260	284	244	228	216	236	240
.data	260	7428	42244	712	7952	1180	424	260	260	260
.text	1376	26840	30936	23696	16332	10664	5604	7832	8596	8340
ELF binary size	3.15K	36K	74K	81K	27K	17K	57K	9.7K	11K	11k
Stack	512	2K	8K	1K	1K	1K	1K	1.3K	1.3K	1.3K
Heap	0	1K	1K	9K	27K	7K	7K	1.3K	1.3K	1.3K

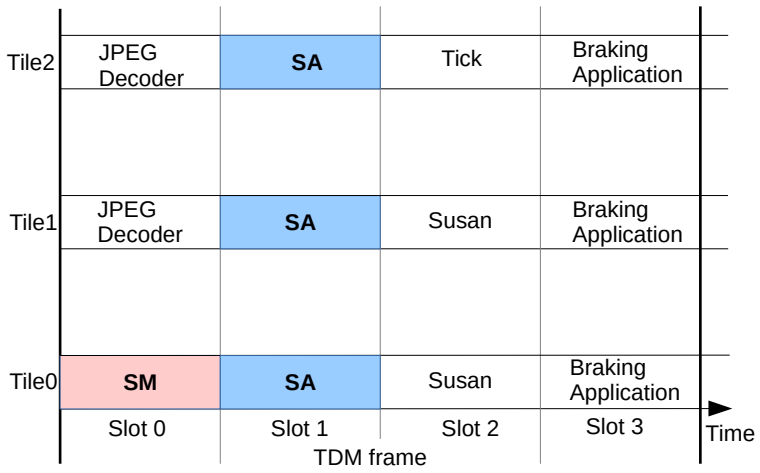

Figure 12: Allocated tile and slot mappings for the applications used in multiprocessor experiments.

composable, the TDM slot duration should satisfy constraints 5, 6, 7, 8, and 9. On the platform described in Section 9.2.1, we measure the parameters required for these constraints and evaluate the minimum duration of the TDM slot as per these constraints. The results are presented in Table 3. We choose the application TDM slot duration $T_A = 65536$ which satisfies all the constraints for the multiprocessor loading architecture to be composable. The duration of the microkernel slot is 12288 cycles.

Phase (Constraint)	Parameters	Value (cycles)	Evaluation of the constraint (Cycles)
Phase 1 (Constraint 5)	t_d	1005	61749
	t_{fb}	12361	
	t_{sn}	1060	
	t_{sb}	6827	
Phase 2 (Constraint 6)	t_{ft}	11044	41868
	t_r	1926	
	t_{ss}	986	
Phase 3 (Constraint 7)	t_{sm}	1060	3180
Phase 4 (Constraint 8)	t_{ae}	3978	11934
Phase 4 (Constraint 9)	t_{dr}	2442	7326

Table 3: Evaluation of TDM constraints for the multiprocessor loading architecture.

9.2.4. Composability Experiments

To demonstrate composability we conducted 4 independent experiments. These four experiments do not represent all the loading scenarios that are possible with the mentioned applications, however these four experiments are enough to demonstrate the composability of the multiprocessor loading architecture. In *experiment A*, we load the *Susan* application in isolation, i.e. neither with any existing application running, nor with other applications being loaded simultaneously. Similarly in *experiment B* and *experiment C* we load in isolation, the *JPEG Decoder* application and the *Braking Application* respectively. In *experiment D*, we have the *Tick* application pre-loaded, which serves as an existing application. Then we load the *Susan* application, the *JPEG Decoder* application and the *Braking Application* simultaneously. In experiment D, the duration of the tick interval was set to the length of one TDM frame ($S \times (T_A + T_M) = 4 \times (65536 + 12288) = 335872$ cycles), so that one tick is observed in every TDM frame.

The loading time for each application, on the respective tiles, in the above mentioned four experiments is shown in Figure 13. The loadings parts (T_C, T_W, T_E and T_L) for each loading is also shown in the Figure 13. Since none of these applications required additional resources, in the *Boot-strap VP expansion*, the duration of T_E only amounts to the time it takes to fetch the Tile BD and check the required resources. This time is very small compared to the other loading parts and hence T_E is not visible. The exact loading time for each application on each of it's tile is mentioned above the column.

In experiment D, a tick was observed periodically every 335872 cycles, while the other three applications were being loaded. This shows that multiple simultaneous loadings do not affect the running applications. In experiment D, the *Susan*, the *JPEG Decoder* application and the *Braking Application* were loaded while the *Tick* application was running. The loading time for each of these applications in isolation, i.e. in experiments A,B and C are the same as in experiment D. This demonstrates two things. First, the existing running application do not affect the loading process and second, multiple simultaneous loadings do not affect each other. As a result, we conclude that the running application and multiple simultaneous loadings are independent of each other. Moreover multiple simultaneous loadings do not affect each other.

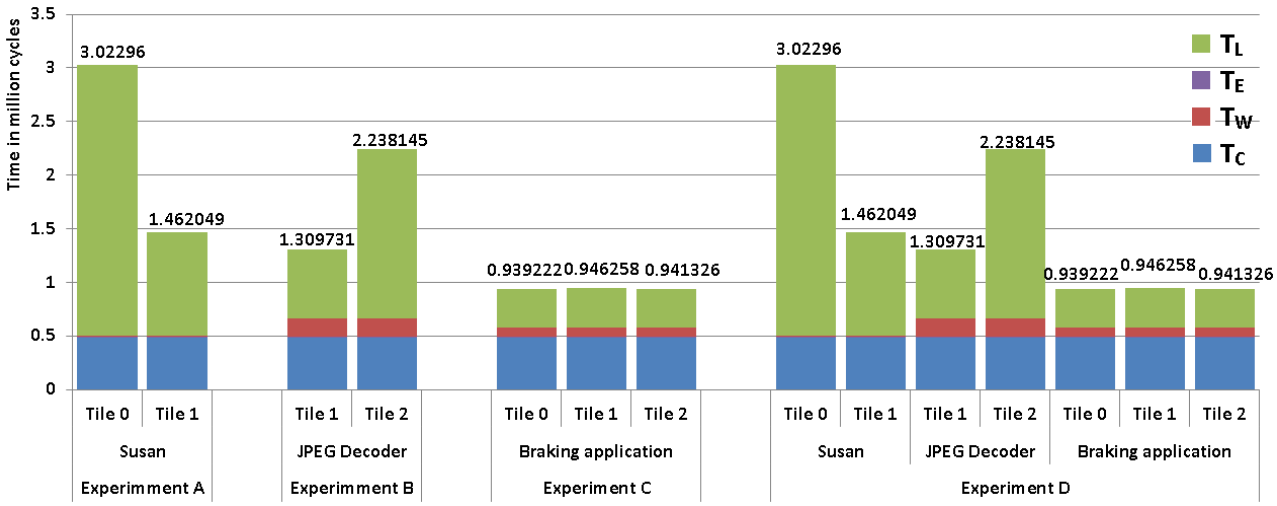

Figure 13: Loading time for each application in the experiments conducted for demonstrating composability.

9.3. Predictability of the loading architectures

For the loading architecture to be predictable, it should be possible at design-time to compute a bound on the loading time of an application. The details of computing such a bound are explained in Section 8.2. For computing this bound, the necessary parameters described in Section 8.2, for the uniprocessor loading hardware platform (see Section 9.1.1) and the multiprocessor loading platform (see Section 9.2.1) were measured and their values are presented in Table 4. The duration of the δ_{header} parameter is different for the uniprocessor and the multiprocessor platforms since the bandwidth and the latency of the NOC connection from the processor tile containing the SA, to the shared memory are different on these platforms. As a result it takes different time to fetch the ELF header. Consequently on the multiprocessor platform, the value of the δ_{header} parameter on the three processor tiles are also slightly different ($\sim$200 cycles). However to simplify the computation, we use only one value, the maximum of the values on the three tiles. This maximum value is presented in Table 4. The other two parameters ($\delta_{search}, \delta_{copy}$) are the same for the uniprocessor and the multiprocessor platforms since the loading code and the processor type for these two platforms are the same.

To verify the predictability of the uniprocessor loading architecture we load two uniprocessor applications namely the *Susan* application and the *JPEG Decoder* application. For the multiprocessor loading architecture we load three applications, viz. the two-tile *Susan* and *JPEG Decoder* applications and the three-tile

Parameter	Value on the Uniprocessor Platform (cycles)	Value on the Multiprocessor Platform (cycles)
δ_{header}	16149	21609
δ_{search}	12976	
δ_{copy}	95	

Table 4: Duration of parameters required for computing bound on loading time for applications.

Braking Application. The loading times for each of these applications on all of their tiles were measured. The bound on the loading time for each of these applications was computed as follows. For each application, on each of it's tile, using the section sizes in Table 2 and the parameters in Table 4, the worst-case time for the DMA *read* transaction (the Δ function, see Section 8.2) was evaluated for all the sections that are loaded. With these computed worst-case times for loading sections and the parameters in Table 4, equation 10 was evaluated for all the tiles of each application, giving the worst-case bound on the loading times when the full share of the processor is allocated. Using these loading times, equation 11 was evaluated to get the worst-case bound on the loading times when the processor is partitioned with TDM. A comparison of the computed worst-case bounds and the loading times measured on the respective hardware platforms is presented in Figure 14. Figure 14 also shows the percentage overestimation of the computed bound with respect to the loading time on the hardware platform. We see that for all the applications on all their tiles, the loading time on the hardware platform is within the computed worst-case bound for each application on the respective tiles. This verifies that the uniprocessor and the multiprocessor loading architectures are predictable.

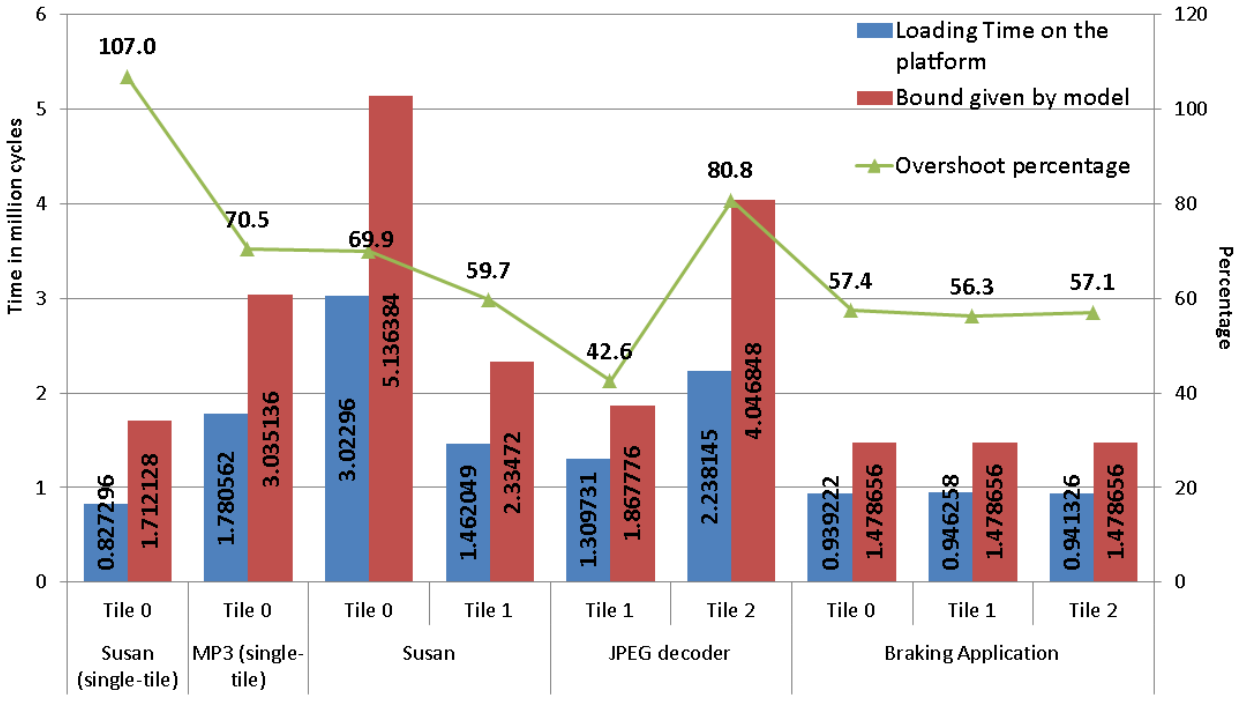

Figure 14: A comparison of loading times on the platform and the bounds given by the loading model for the uniprocessor and the multiprocessor applications.

There are two reasons for the overestimation of these computed bounds. The first reason is that, in the DMA *read* transaction dataflow model we make use of the single actor dataflow model for the NOC

(described in [38]). The worst-case represented by the single-actor dataflow model is experienced by each flit(NOC packet) in the NOC in the DMA *read* transaction dataflow model. Whereas in case of the hardware, flits are pipelined in the NOC and therefore only the first flit may take the worst-case time, the rest of the following flits take a smaller duration that is proportional to the path length of the NOC connection. Therefore for applications with large sections, the overestimation is larger compared to applications with smaller section sizes (as in the case of the *Braking Application*). It is important to note that the shown loading times in Figure 14, apart from T_L is also composed of T_C, T_E and T_W. Therefore depending on the values of T_C, T_E and T_W, the overestimation in T_L due to the pessimistic NOC model is not visible in all the loading cases. The second reason is the pessimism of the equation 11, which computes the worst-case time it takes to complete a given amount of work on a TDM schedule when a given number of slots are allocated. Equation 11 always accounts the last entire TDM frame, regardless of when the work completes in the allocated slot(s). At the maximum equation 11 may add an overestimation of one TDM frame.

As explained above, the pessimism of the dataflow model which models a DMA read transaction increases with increase in the transaction size. For loading an application, the DMA reads one ELF section at a time. The Braking Application has small ELF sections (up to 8.5KB as seen in Table 2) hence the overestimation is lesser than the other applications with larger section sizes, and as a result a smaller absolute error (i.e. the difference between the bound given by the model and the loading time on the platform) and a smaller relative error (57%) as seen in Figure 14. The MP3, Susan (multi-tile) and JPEG Decoder (tile 2) applications have large ELF sections; hence they have a large overestimation and consequently a large absolute error. However with longer loading times, the relative error is not large. The single tile Susan application has a large text section (26.8KB, see Table 2) and hence has a large overestimation. Due to its relatively shorter loading time, we see in Figure 14 that the relative error is larger (107%) than the other applications with large ELF sections (59% to 80%).

The accuracy of the bound on the loading time can be improved by using a better NOC model, which models the pipelining in the hardware. Furthermore, instead of computing the worst-case time, a formulation to compute the exact time it takes to complete a given amount of work in a TDM schedule when the slot allocations are known, can be used for computing the bound on the loading time. These two improvements are future work.

10. Future Work

In this section we list possible future extensions to our work. The proposed work assumes that all the processors in the multiprocessor platform are running TDM with the same number of slots, with the same duration of the TDM slot and that the slots are synchronized and aligned. However in practice, the lengths of TDM schedule and the duration of the slot on each processor may be different. Moreover, the TDM schedules on different processors may not be aligned. If the clocks of procesor tiles are synchronized, there will be a constant misalignment, provided the duration of the slots remains the same at run-time. However, with unsynchronized clocks, the misalignment may vary, nonetheless the worst-case misalignment can be computed, if the durations of slots do not change at run-time. The TDM slot duration constraints for composable loading (equations 2, 5, 6, 7, 8, 9) need to be generalized to take into account different slot durations and the misalignment of the TDM schedules on different processors.

In systems, where the lengths of the TDM slots vary in length, loading can be composable, provided the length of the TDM frame does not change at run-time. Additionally, in the application bundle, apart from the minimum required length of the slot(s), the start time of the slot(s) relative to the beginning of the TDM frame is also specified. On a loading request, the resource manager can try to reserve the slot(s) with the dynamically dimensioned size from the specified relative start time. If the reservation succeeds, the application will be loaded composably, otherwise the loading will be aborted.

The proposed loading method is applicable to NOC based multi-core architectures. On many-core architectures, with hundreds of cores, the loading method may have scalability issues. There are mainly two bottlenecks. First, a NOC connection is needed from the master tile containing the System Manager to all other tiles and vice versa. And second, in our current prototype all tiles need to access the shared memory,

since it hosts the application bundles. Both the architecture of the platform and the loading method need to be reevaluated in the presence of large numbers of cores.

11. Conclusion

In the existing partitioned systems, it is not possible to dynamically add applications at run-time. However for scenarios such as on-board software updates and reloading for fault-tolerance, dynamic loading is essential. In this paper, a software architecture is presented which allows for dynamic creation and management of partitions. Furthermore a composable loading architecture was presented for uniprocessor and multiprocessor platforms, which ensure that the loading process and the running and applications are independent of each other and also simultaneous loading of multiple applications do not affect each other or the running applications. Additionally the loading architecture is predictable, which allows to compute a bound on the loading time a priori. The method to compute a worst-case bound on the loading time was also presented. The composability and the predictability properties of the uniprocessor and the multiprocessor loading architectures were verified with experiments on a SoC prototype on an FPGA board. The presented method to compute a bound on the loading time has an overestimation which depends on the size of the sections in the ELF application binaries and the formulation to compute the worst-case time it takes to finish a given work on a given TDM schedule. Solutions to decreases this overestimation were discussed.

Acknowledgements

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, CA505 BENEFIC, CA703 OpenES, ARTEMIS-2013-1 621429 EMC2 and 621353 DEWI.

Appendix A. Barrier Synchronization

A barrier synchronization helps multiple processes synchronize by waiting until all the involved processes have reached the barrier. In a centralized shared barrier, this is implemented by a single counter which is initialized to 0. When a process reaches the barrier, it signals the barrier by incrementing the counter value. The processes block on the barrier, until the counter reaches the value which is equal to the number of clients. In such an implementation, the counter needs to be atomically updated. However guaranteeing atomic writes to a shared location on a NoC based multiprocessor platform is not trivial, since the involved processes run in parallel and therefore a race condition may occur leading to incorrect functionality. Moreover the single counter location is a serious bottleneck.

In a Master-slave barrier [43], a master processor is needed that gathers the arrival notifications from all slaves and broadcasts a termination signal to all slaves. In the proposed work, the System Manager (SM) could serve as the master process; however there are two problems. Firstly, since the SM is running on a TDM schedule, the termination signal could be broadcast with a worst-case delay of one TDM frame after the actual completion of the barrier. Secondly, the timing of each broadcast depends on the number of barriers, since the SM can only deal with one barrier at a time. This dependency breaks the composability of the loading. A *Tree-barrier* [43] uses a hierarchical structure of the master-slave barrier for better performance on many-core architectures. The problems of the master-slave barrier are retained in the tree-barrier for our purpose of composable loading. The *Multi-level barrier* [44] relies on coherent view of the top memory level. The target architecture does not have hardware support for ensuring such coherency.

With lack of atomic writes and hardware support for coherency, we take a modified approach of the centralized shared barrier. To work around the problem of atomic writes, we construct an array of flags in the shared memory whose length is equal to the number of clients. Each client updates only it's own flag when it reaches the barrier. The barrier unblocks when all the flags in the have reached the same value. The coherent view is implemented in software with the barrier library API, which ensures the barrier is released, only when all the flags of the clients have the same value.

For each application to be loaded, when a boot-strap VP is created, a barrier is also created by the System Manager(SM). During the creation of the boot-strap VP, each application is allocated an exclusive DMA. Each DMA has a pre-allocated NOC connection to the shared memory with guaranteed bandwidth and latency. This is possible, since the NOC offers composable connections. The shared memory also offers a service with a dedicated bandwidth and latency with a composable memory controller. Thus with an exclusive DMA, a composable NOC connection to the shared memory and composable service from the shared memory, it is guaranteed that every application which has a reserved virtual platform, has access to the shared memory and as a result to the barrier structure with guaranteed bandwidth and latency. In this way, it is ensured that the execution and performance of the barrier is not affected by other applications.

[1] H. Kopetz, *Real-time systems: Design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[2] B. Akesson, A. Molnos, A. Hansson, J. Ambrose Angelo, and K. Goossens, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, ch. 2, Springer, 2010.

[3] Avionics Application Software Standard Interface, *Design Guidance for Integrated Modular Avionics*, 1997.

[4] Avionics Application Software Standard Interface, *ARINC Specification 653P1-2*, 2005.

[5] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor, "AIR technology: A step towards ARINC 653 in space," in *Proc. DASIA*, 2009.

[6] S. Samolej, "ARINC specification 653 based real-time software engineering," *e-Informatica Software Engineering Journal*, 2011.

[7] V. Lopez-Jaquero, F. Montero, E. Navarro, A. Esparcia, and J. Catal'n, "Supporting ARINC 653-based dynamic reconfiguration," in *IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2012.

[8] F. Boniol, "New challenges for future avionic architectures," in *Modeling Approaches and Algorithms for Advanced Computer Applications*, Springer, 2013.

[9] J. R. van Kampenhout and R. Hilbrich, "Model-based deployment of mission-critical spacecraft applications on multicore processors," in *Proc. of Reliable Software Technologies - Ada-Europe*, pp. 35–50, 2013.

[10] F. Kluge, M. Gerdes, and T. Ungerer, "An operating system for safety-critical applications on manycore processors," in *IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2014.

[11] R. Hilbrich and J. R. van Kampenhout, "Partitioning and task transfer on noc-based many-core processors in the avionics domain," *Softwaretechnik-Trends*, vol. 31, no. 3, 2011.

[12] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, 2009.

[13] S. Goossens, B. Akesson, M. Koedam, A. B. Nejad, A. Nelson, and K. Goossens, "The CompSOC design flow for virtual execution platforms," in *Proc. of the 10th FPGAworld Conference*, ACM, 2013.

[14] J. Rosa, J. Craveiro, and J. Rufino, "Safe online reconfiguration of time-and space-partitioned systems," in *9th IEEE International Conference on Industrial Informatics (INDIN)*, IEEE, 2011.

[15] H. Seifzadeh, A. Kazem, M. Kargahi, and A. Movaghar, "A method for dynamic software updating in real-time systems," in *IEEE/ACIS International Conference onComputer and Information Science*, IEEE, 2009.

[16] M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Proc. of the 2nd International Workshop on Hot Topics in Software Upgrades*, p. 2, ACM, 2009.

[17] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *Proc. of the Real-Time Systems Symposium*, IEEE, 1997.

[18] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, "Dynamic linking and loading in networked embedded systems," in *IEEE International Conference on Mobile Adhoc and Sensor Systems*, IEEE, 2009.

[19] S. Sinha, M. Koedam, R. van Wijk, A. Nelson, A. B. Nejad, M. Geilen, and K. Goossens, "Composable and predictable dynamic loading for time-critical partitioned systems," in *Proc. of the Euromicro Conference on Digital System Design (DSD)*, 2014.

[20] P. Burgio, M. Ruggiero, F. Esposito, M. Marinoni, G. Buttazzo, and L. Benini, "Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems," in *IEEE International Conference on Computer Design (ICCD)*, 2010.

[21] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, Mar 2002.

[22] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing task migration impact on embedded soft real-time streaming multimedia applications," *EURASIP J. Embedded Syst.*, vol. 2008.

[23] P. Tendulkar and S. Stuijk, "A case study into predictable and composable MPSoC reconfiguration," in *Proc. of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, IEEE Computer Society.

[24] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "The time-triggered system-on-a-chip architecture," in *IEEE International Symposium on Industrial Electronics*, 2008.

[25] R. Sorensen, M. Schoeberl, and J. Sparso, "A light-weight statically scheduled network-on-chip," in *NORCHIP*, Nov 2012.

[26] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos,

A. B. Nejad, A. Nelson, and S. Sinha, "Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow," *SIGBED Rev.*, vol. 10, 2013.
[27] K. Goossens and A. Hansson, "The aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proc. of Design Automation Conference (DAC)*, 2010.
[28] R. Stefan, A. Molnos, and K. Goossens, "dAElite: a TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up," *IEEE Transactions on Computers*, vol. 63, no. 3, 2014.
[29] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time sdram controller for mixed time-criticality systems," in *Proc. of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.
[30] B. Akesson, A. Hansson, and K. Goossens, "Composable resource sharing based on latency-rate servers," in *12th Euromicro Conference on Digital System Design*, IEEE, 2009.
[31] K. Goossens, D. She, A. Milutinovic, and A. Molnos, "Composable dynamic voltage and frequency scaling and power management for dataflow applications," in *Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, 2010.
[32] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, pp. 233–270, 2002.
[33] A. Beyranvand Nejad, A. Molnos, and K. Goossens, "A unified execution model for multiple computation models of streaming applications on a Composable MPSoC," *J. Syst. Archit.*, vol. 59, no. 10, 2013.
[34] E. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 100, no. 1, 1987.
[35] G. Kahn, "The semantics of a simple language for parallel programming," in *In Proc. of the IFIP Congress*, vol. 74, 1974.
[36] A. Nelson, A. Nejad, A. Molnos, M. Koedam, and K. Goossens, "CoMik: A predictable and cycle-accurately composable real-time microkernel," in *Proc. of the Conference on Design, Automation and Test in Europe*, March 2014.
[37] R. Stefan, A. B. Nejad, and K. Goossens, "Online allocation for contention-free-routing NOCs," in *Proc. of the Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, ACM, 2012.
[38] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *Computers & Digital Techniques, IET*, vol. 3, 2009.
[39] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow," in *ICASSP*, vol. 5, May 1995.
[40] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*, SCOPES '07, 2007.
[41] S. M. Smith and J. M. Brady, "Susan a new approach to low level image processing," *International journal of computer vision*, vol. 23, no. 1, 1997.
[42] D. Goswami, A. Masrur, R. Schneider, C. J. Xue, and S. Chakraborty, "Multirate controller design for resource- and schedule-constrained automotive ECUs," in *Proc. of the Conference on Design, Automation and Test in Europe*, DATE, 2013.
[43] A. Marongiu, P. Burgio, and L. Benini, "Supporting OpenMP on a multi-cluster embedded MPSoC," *Microprocess. Microsyst.*, vol. 35, no. 8, 2011.
[44] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *IEEE International Symposium on IPDPS*, 2009.