Composable and Predictable Dynamic Loading for Time-Critical Partitioned Systems

Shubhendu Sinha[1], Martijn Koedam[1], Rob van Wijk[1], Andrew Nelson[1],
Ashkan Beyranvand Nejad[2], Marc Geilen[1], Kees Goossens[1]
[1]Eindhoven University of Technology, Netherlands, [2]Delft University of Technology, Netherlands
[1]s.sinha@tue.nl

***Abstract*—In time-critical systems such as in avionics, for safety and timing guarantees, applications are isolated from each other. Resources are partitioned in time and space creating a *partition* per application. Such isolation allows 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. In this paper we propose a software architecture to dynamically create and manage partitions and a method for composable dynamic loading 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 partition 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.**

I. Introduction

Time-critical systems such as in the avionics domain have strict safety, real-time and fault tolerance constraints. To ensure these constraints and for fault-containment, applications are isolated from each other at the cycle-level. That is applications cannot affect each other even by a single cycle. With temporal and spatial partitioning, faults in an application cannot affect the rest of the system and 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. 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)* [1]. 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 [2] and AIR (ARINC in Space RTOS) [3] 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 [4]. They do not cope with dynamic aspects such as adding new applications at run-time or on-board application software updates or 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 add new applications or new functionality to the spacecraft system. 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 [5]. In such scenarios, dynamic loading is necessary. For these reasons dynamic loading is also a feature proposed for the future avionic architectures as argued in [6]. 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 should be composable. Composable dynamic loading for partitioned systems is challenging because a) the existing state-of-art partitioned systems [3,4,7,8] do not support dynamically creating and managing partitions at run-time and b) existing loading methods [9]–[12] do not provide timing isolation between the running applications and the loading process and vice versa.

The proposed state-of-art solutions for dynamic loading for partitioned systems in literature have only partially addressed the problem, either by lacking timing isolation completely [5] or partially [13]. Moreover the existing solutions only consider loading of one application at a time.

To address composable dynamic loading, 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 method for dynamic composable loading. We ensure that multiple simultaneously loading applications do not affect each other or the running applications and vice versa. Furthermore, our composable dynamic loading technique is also predictable, i.e., the loading process completes in a known bounded time.

We achieve composable dynamic loading by splitting the loading process into three steps. The first step 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. DMA unit is necessary to fetch application code and data. The second step expands the boot-strap partition by reserving the rest of the resources required by the application and the third step loads the application code and data. The second and third steps are executed strictly in the reserved application's time slots, which

ensures isolation from running applications and vice versa. To validate our design we implement our software architecture for a SoC prototype on an FPGA board and demonstrate composability and predictability with experiments.

In the following section we elaborate the related work. In Section III we explain the platform on top of which we develop our software architecture. In Section IV we introduce the software architecture and in Section V we introduce the composable and predictable loading method. Sections VI and VII contain the experiments and conclusion respectively.

II. Related work

We first present a brief overview of solutions in literature for dynamically updating software at run-time in real-time systems. In [9] 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 [11] 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 [10], the dynamic update is performed in idle time in a system which runs an RTOS. By performing the update in slack time, the real-time guarantees of existing applications are not affected, however no timing guarantees are provided with regards to loading. In the above mentioned works, although the loading process has some bounds, timing isolation at the cycle-level in between the running applications and the loading, is not guaranteed. Work in [12] targets dynamic loading in networked embedded systems and presents techniques for efficient loading such as generating minimal application code size, reducing linking overhead and de-coupling OS and application development.

We now give an overview of proposed solutions for dynamic loading for time-critical partitioned systems. A software layer proposed in [5] manages dynamic reconfiguration in ARINC based systems for fault tolerance. The proposed layer lies between RTOS and ARINC API and it reconfigures the system to reload partitions on a redundant hardware module when a fault is detected. Although [5] details the functional steps involved in reconfiguration, they however do not provide any timing isolation guarantees. The work in [13] proposes a software update methodology for AIR based platforms. The loading process is carried out on a best-effort basis in the system partition, where the loading process is executed in the slack time available. Their loading architecture isolates running applications from the loading process, however the running applications may affect the loading process, thus the loading process is not composable. Moreover, with best-effort loading, the loading time is not predictable. A design-flow to generate a multi-tile partitioned system prototyped on an FPGA is proposed in [8], where applications share resources composably, i.e. they do not affect each other even by a single cycle. Their 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 in [8] 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 in [8]. However, our method is applicable to any composable platform, e.g. the TTSOC platform [14].

III. Platform Overview

In this section we explain the hardware platform and the microkernel, on top of which we construct the software architecture for composable dynamic loading.

A. Hardware platform

We build the software architecture for composable dynamic loading on top of the partitioned system architecture proposed in [8]. The architecture is multi-tile, however the work in this paper is aimed for loading single-tile applications. A single-tile hardware platform instance that is used for implementation in this paper is shown in Figure 1.

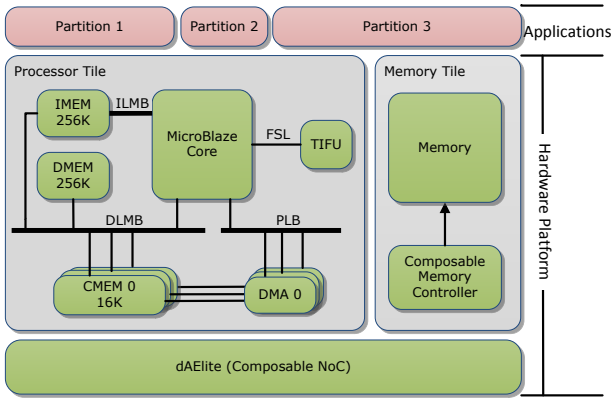

Fig. 1. A single tile composable hardware platform instance.

For each application, the design proposed in [8] creates partitions which are cycle-level composable, i.e. partitions do not affect each other, not even by a single cycle. This is achieved by sharing each resource composably, that is by partitioning it in space or time. 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 Network on Chip (NoC) [15] and a composable memory controller [16]. Shared memory shown in Figure 1 can be an off-chip DDR memory or an on-chip SRAM memory.

The processor tile in Figure 1 contains 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) by a Processor Local Bus (PLB). The processor is also connected to a Timer Interrupt Frequency Unit (TIFU) which enables generating interrupts at programmable intervals. TIFU assists the microkernel (explained in the next subsection) by maintaining strict timings in hardware.

Applications on this platform can be bare code or based on a *Model-of-Computation (MoC)* such as *SDF* [17] or *KPN* [18]. Applications can also be an RTOS which in turn hosts other applications. They are single threaded and have access to DRAM or SRAM via the NoC.

B. Microkernel

To create partitions on the processor, the Composable Microkernel (CoMiK) [19] is used. CoMik partitions the processor in time using Time-Division-Multiplexed (TDM) arbitration. With the help of the TIFU unit, CoMik enforces cycle-level partitioning. 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 in time and space.

IV. Software Architecture

An essential part of dynamic loading in partitioned system is creating new partitions and managing them at run-time. In this section we propose a software architecture that facilitates creating and managing partitions at run-time. A resource management system is necessary that allows reserving resources at run-time and associating them with a partition. Resources should be able to 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. Keeping in mind these requirements, we propose a **Resource Management (RM)** framework explained in the following subsection.

A. Resource Management Framework

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 III. A **Virtual execution Platform (VP)** is the set of all resource budgets required by an application. As a result a VP descriptor 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. BDs can be arranged in a hierarchy as shown in Figure 2, which shows some example BDs of resources. For instance, the processor BD describes the parameters necessary to create an application partition on the processor, namely the required number of TDM slots, required size of stack, heap and the space required to store application code and data. The NoC BD describes the NoC connections used by the application. Each connection requires source and destination node and the throughput and latency requirements of the connection.

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 3. 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

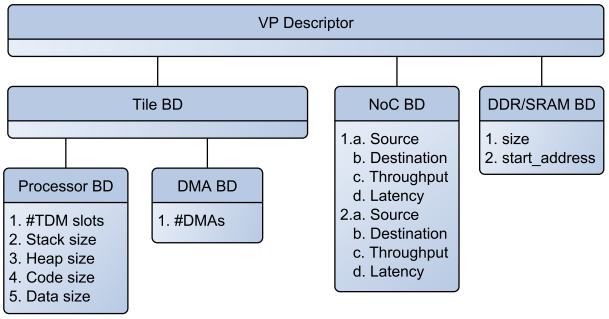

Fig. 2. Structure of a VP descriptor.

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.

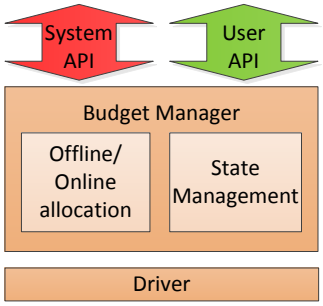

Fig. 3. Structure of the run-time resource management library.

The following sub-section describes the compilation flow which allows making use of the run-time RM library to generate dynamically loadable application binaries.

B. Compilation flow

For partitions described using the VP descriptor, the run-time RM library API can be used to create and manage new partitions at run-time. Hence we require that alongside the source code of the application, the description of the required VP should be provided. The compiler compiles the application into the **Executable Linkable Format (ELF)** binary. The VP descriptor 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. Therefore compilation is done with *Position Independent Code(PIC)*.

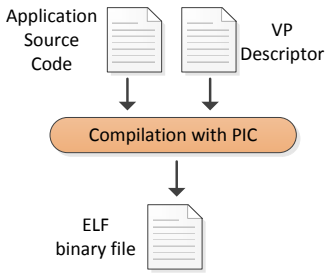

Fig. 4. Compilation flow for dynamic loading.

We assume the application binary is sent through a dedicated communication link into the shared memory of the System on Chip (SoC) at a pre-defined location. After detection of the application binary the loading process is executed. The compilation flow is shown in Figure 4.

V. Loading Architecture

In this section we first explain the loading process and in the following sub-sections we argue the composability and predictability properties of the loading process.

A. System Application

The *System Application (SA)* is an application with access to the RM *System API*, thus it can create, modify or destroy VPs. It is also responsible for executing the dynamic loading process. 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 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 period from the time an ELF binary is detected by the SA to the time 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 descriptor 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 descriptor 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.

B. Loading Steps

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 descriptor 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 descriptor 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 descriptor 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.

The above loading steps are illustrated in an example shown in Figure 5. 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 5. 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 until the first allocated slot

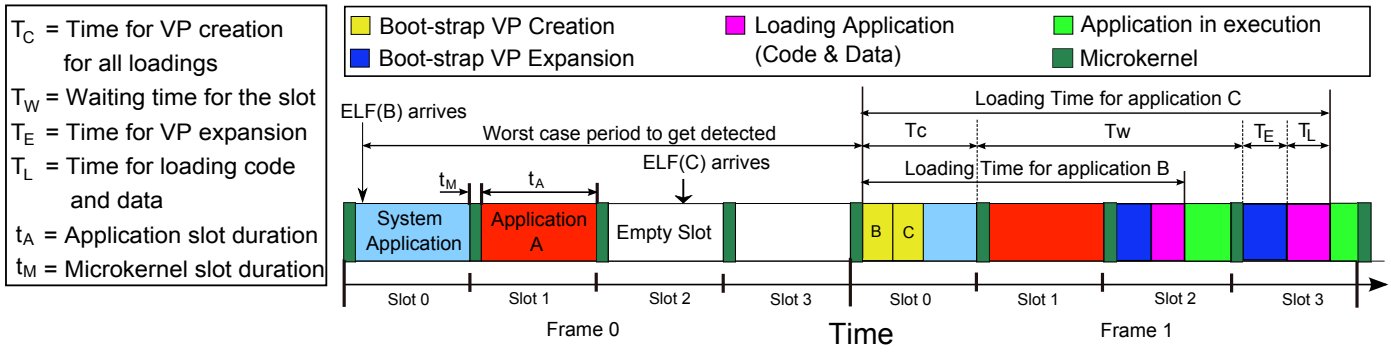

Fig. 5. An example of application loading with the proposed loading process.

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-sections we shall explain how *LT* is composable and predictable and the TDM design constraints for it to be composable.

C. 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 descriptor, 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.

D. Predictability

For the loading process to be predictable, it should finish in known time bound. To have a computable bound for the

loading process the necessary requirements are, 1) the application binary should be of finite size, 2) the resource allocation algorithms should have a finite Worst-Case Execution Time (WCET), 3) platform resources such as processor, DMA, NoC, Memory should give guaranteed performance. We assume requirement 1 always holds. With regards to requirement 2, for the processor TDM slot allocation, in this paper, the TDM slots are allocated at design time. For memory allocation, we employ a naive predictable memory allocator. The memory is divided into a number of 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 WCET for the memory allocator is when the allocator has to iterate through the complete list of blocks.

With regards to requirement 3, the platform in [8] is predictable. The processor is shared using TDM schedule, which ensures guaranteed share of processor. Since each application is exclusively allocated a DMA, it gets the full share. The communication time in the NoC is also predictable and the worst case bound can be computed with the dataflow model presented in [20]. The shared memory has a composable and a predictable front-end and the time to serve memory read or write request can be computed using the *Latency-rate Server* model as described in [16]. By combining the dataflow model presented in [20] and the Latency-rate Server model in [16], it is possible to get a worst-case bound for the loading time. Presenting the detailed derivation of such a worst-case bound is out of the scope of this paper.

In equation 1, T_C is of known fixed duration. With fixed TDM slot allocations at design-time, T_W is known. Due to the predictable nature of the loading process, T_E and T_L take constant time. As a result, the loading process for an application finishes in a constant time and this constant loading time for the application, is the same irrespective of when the loading process is initiated. This is demonstrated by the experiments explained in Section VI-D.

VI. Experiments

We shall first explain the experimental setup in the next subsection. Then for composable dynamic loading we derive the TDM slot duration that will satisfy Constraints 2 and 3. Then we explain the experiments conducted to demonstrate composability and predictability.

A. Experimental Setup

The hardware platform instance used in our experimental setup is shown in Figure 1. We implemented the hardware platform instance on a Xilinx ML605 FPGA board. The 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 and communication memories (CMEMs). The CMEMs are of 16KB each. The platform runs on a clock of 120MHz frequency.

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* [21] 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 resulting memory specifications of these applications are shown in Table I. 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.

TABLE I. Memory specification of applications.

Application	ELF binary size	Program Code	Program data	Stack	Heap
Tick	2.7KB	0.5KB	44B	512B	0B
Susan	35KB	26KB	7KB	2KB	1KB
MP3-decoder	72KB	30KB	41KB	8KB	1KB

B. 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 next derive the TDM slot duration which satisfy Constraints 2 and 3.

Constraint 2 requires us to know the timings of T_α, T_β and T_γ for the largest VP in the system. The sizes of BD structures in our implementation are shown in Table II. The largest VP in the considered platform consists of 3 DMAs with 3 corresponding NoC connections and 1 tile with 1 processor. Using the sizes in Table II, we see that the largest VP descriptor amounts to 1240 bytes. Fetching and parsing the ELF binary to load the VP descriptor of 1240 bytes takes 5785 cycles in our system. Thus $T_\alpha = 5785$. We emulate the worst case condition to allocate memory, using the allocator explained in Section V-D 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 the largest VP is 1448 cycles, thus, $T_\gamma = 1448$. Therefore according to constraint 2:

$$\begin{aligned} t_A &\geq (4-1)(5785 + 6692 + 1448) \\ &\geq 41775 \end{aligned}$$

The SA requires some additional time to check for ELF binaries and to check for *Boot-strap VP Expansion* failures. Therefore we choose the duration of application slot to be 65536 cycles which satisfies constraint 1. The microkernel slot is 4096 cycles.

In *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 takes a short time(few hundred cycles) for reservation and allocation. For the NoC, we employ an online allocation algorithm presented in [22]. The worst case time for finding an allocation in a 4×4 mesh is 6394 cycles as reported in [22]. The NoC in the considered platform is 2×1 mesh with even smaller allocation time. Therefore the chosen t_A also satisfies Constraint 3.

TABLE II. Sizes of BD structures.

BD	Size (Bytes)
BD_dma	208
BD_noc	140
BD_proc	44
BD_tile	152

C. 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

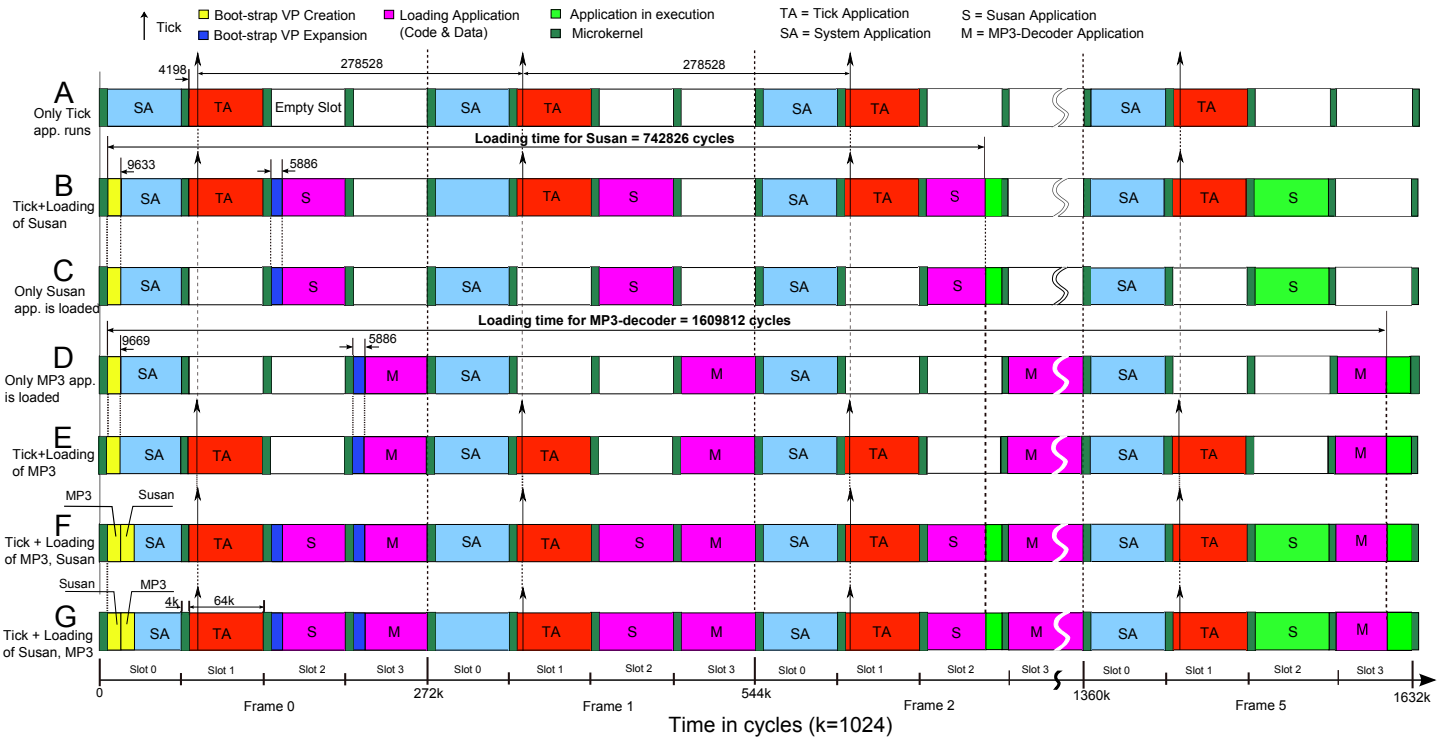

Fig. 6. Processor timeline for composability experiments.

chosen combinations are sufficient to demonstrate that the loading process is isolated from the existing applications and other simultaneous loadings and vice versa. Figure 6 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 6, 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 278528 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. 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 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, demonstrates that multiple simultaneous loadings do not affect each other or the existing application.

The duration of loading steps for each application is shown in Table III. Due to relatively larger size of *MP3-decoder* application, it loads in the fifth TDM frame. Therefore in Figure 6 the processor timeline jumps to the 5^{th} frame. Duration of *Boot-strap VP Creation* step for each of these applications takes approximately the same duration as the processor TDM slot allocation is fixed and the memory allocation takes small time as there is enough memory in IMEM and DMEM (256KB). The *Boot-strap Expansion* step for all the three applications happen to be the same, since after the creation of *Boot-strap VP*, no additional resources were necessary. However the BD section in the ELF binary is still fetched and checked and that consumes 5886 cycles.

D. Predictability Experiments

As explained in Section V-D, due to the predictable nature of the loading process, the loading time for an application takes a constant time, which is the same independent of when the loading process is initiated. To demonstrate this, the *MP3-decoder* and *Susan* applications were loaded in independent

TABLE III. DURATION OF LOADING STEPS FOR EACH APPLICATION.

Application	Boot-strap VP Creation	Boot-strap VP Expansion	Loading Code and Data	Loading Time
Tick	9640	5886	31226	106790
Susan	9633	5886	597630	742826
MP3-decoder	9669	5886	1394984	1609812

experiments. In each experiment, they were sent with a delay of one TDM frame. For each loading, the detection time, i.e. the absolute time when the SA detected the application ELF binary and the completion time, i.e. the absolute time when the application begins execution were observed for each experiment. In Figure 7, the numbers below the bar indicate the detection time of each loading and the numbers above the bar indicate the completion time of each loading. The loading time is the difference between the completion time and detection time. The loading time is shown inside the bar. As seen in Figure 7, for loadings at different start times, the loading time remains the same. This experiment verifies that the loading time for an application is constant and remains the same irrespective of when the loading process is initiated.

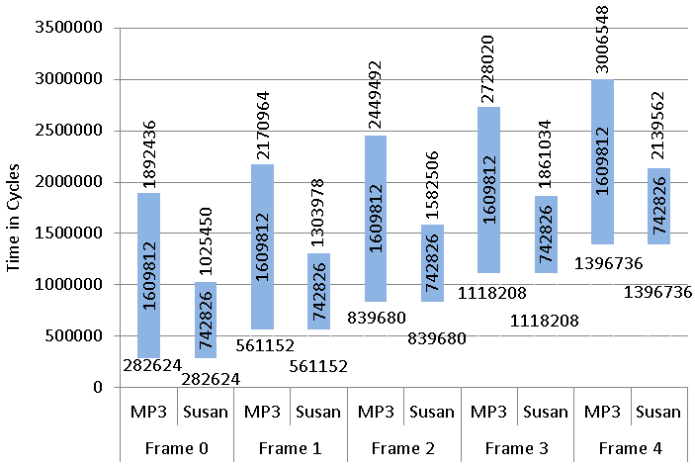

Fig. 7. Loading times for MP3-decoder and Susan applications when detected in different TDM Frames.

VII. CONCLUSION

Existing partitioned time-critical systems do not allow to dynamically add applications at run-time. However for various 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. This is essential for dynamic loading. Furthermore a loading method was proposed which ensures the loading process is composable and predictable. With composability, the loading process and the existing running applications are temporally and spatially independent of each other. With predictability the loading process completes in a known bounded time. For composability, design constraints for the TDM slot duration are discussed.

Results are achieved by splitting of the loading process into sub-steps, of which only the basic minimum step is executed in system time and the rest are executed in the allocated application time. Extending the current loading architecture for multi-tile applications is future work. A limitation of the proposed loading architecture is that the loading time is coupled with TDM slot allocation of the application. Therefore a big application with a small share of processor, will load slowly. This can be improved either by using slack time in the system application and/or the free available slots for loading, in which case, the loading will not be composable, however it can be predictable.

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.

REFERENCES

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

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

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

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

[5] V. López-Jaquero, F. Montero, *et al.*, "Supporting ARINC 653-based dynamic reconfiguration," in *IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2012.

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

[7] R. Obermaisser *et al.*, "From a federated to an integrated automotive architecture," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, 2009.

[8] S. Goossens, B. Akesson, *et al.*, "The CompSOC design flow for virtual execution platforms," in *Proc. of the 10th FPGAworld Conference*, ACM, 2013.

[9] H. Seifzadeh, A. Kazem, *et al.*, "A method for dynamic software updating in real-time systems," in *Computer and Information Science. IEEE/ACIS International Conference on*, IEEE, 2009.

[10] 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.

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

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

[13] J. Rosa, J. Craveiro, and J. Rufino, "Exploiting AIR composability towards spacecraft onboard software update," *Actas do INForum-Simpósio de Informática*, 2010.

[14] 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.

[15] R. Stefan, A. Molnos, A. Ambrose, and K. Goossens, "A TDM NoC supporting QoS, multicast, and fast connection set-up," in *Proc. of the Conference on Design, Automation and Test in Europe*, 2012.

[16] B. Akesson, A. Hansson, and K. Goossens, "Composable resource sharing based on latency-rate servers," in *Digital System Design. 12th Euromicro Conference on*, IEEE, 2009.

[17] E. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, 1987.

[18] K. Gilles, "The semantics of a simple language for parallel programming," in *In Information Processing74: Proc. of the IFIP Congress*, vol. 74, 1974.

[19] A. Nelson, A. Nejad, *et al.*, "CoMik: A predictable and cycle-accurately composable real-time microkernel," in *Proc. of the Conference on Design, Automation and Test in Europe*, March 2014.

[20] A. Hansson, M. Wiggers, *et al.*, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *Computers & Digital Techniques, IET*, vol. 3, 2009.

[21] 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.

[22] 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.