Hardware Contention-Aware Real-Time Scheduling on Multi-Core Platforms in Safety-Critical Systems

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Ankit Agrawal

D 386

Eingereicht am:	30.11.2018
Tag der mündlichen Prüfung:	26.04.2019
Dekan des Fachbereichs:	Prof. Dr.-Ing. Ralph Urbansky
Promotionskomission	
Vorsitzender:	Prof. Dr.-Ing. Wolfgang Kunz
Berichterstattende:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
	Prof. Dr.Ing. Sebastian Altmeyer

"Faith and Patience"
- Shirdi Sai Baba, my Guru

Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

(Kaiserslautern, Datum) (Ankit Agrawal)

Abstract

While the computing industry has shifted from single-core to multi-core processors for performance gain, safety-critical systems (SCSs) still require solutions that enable their transition while guaranteeing safety, requiring no source-code modifications and substantially reducing re-development and re-certification costs, especially for legacy applications that are typically substantial. This dissertation considers the problem of worst-case execution time (WCET) analysis under contentions when deadline-constrained tasks in independent partitioned task set execute on a homogeneous multi-core processor with dynamic time-triggered shared memory bandwidth partitioning in SCSs.

Memory bandwidth in multi-core processors is shared across cores and is a significant cause of performance bottleneck and temporal variability of multiple-orders in task's execution times due to contentions in memory sub-system. Further, the circular dependency is not only between WCET and CPU scheduling of others cores, but also between WCET and memory bandwidth assignments over time to cores. Thus, there is need of solutions that allow tailoring memory bandwidth assignments to workloads over time and computing safe WCET. It is pragmatically infeasible to obtain WCET estimates from static WCET analysis tools for multi-core processors due to the sheer computational complexity involved.

We use synchronized periodic memory servers on all cores that regulate each core's maximum memory bandwidth based on allocated bandwidth over time. First, we present a workload schedulability test for known even-memory-bandwidth-assignment-to-active-cores over time, where the number of active cores represents the cores with non-zero memory bandwidth assignment. Its computational complexity is similar to merge-sort. Second, we demonstrate using a real avionics certified safety-critical application how our method's use can preserve an existing application's single-core CPU schedule under contentions on a multi-core processor. It enables incremental certification using composability and requires no-source code modification.

Next, we provide a general framework to perform WCET analysis under dynamic memory bandwidth partitioning when changes in memory bandwidth to cores assignment are time-triggered and known. It provides a stall maximization algorithm that has a complexity similar to a concave optimization problem and efficiently implements the WCET analysis. Last, we demonstrate dynamic memory assignments and WCET analysis using our method significantly improves schedulability compared to the state-of-the-art using an Integrated Modular Avionics scenario.

Acknowledgements

First, I am deeply grateful to Prof. Gerhard Fohler — my PhD advisor and mentor for this research opportunity, his invaluable guidance and supervision and the many ensuing discussions that enabled this work. I am also thankful to him for the many opportunities to interact with researchers from all over the world. It helped to improve my understanding of safety-critical systems design and real-time scheduling. Further, it is his unwavering support and motivation during the past five years, even during my personal crises, that enabled me to overcome the many hurdles during the PhD, especially when giving up was easy and going further was difficult. I am also greatly thankful to Prof. Sebastian Altmeyer for accepting to be on my PhD Defense Committee and to Prof. Wolfgang Kunz for accepting to chair the Committee.

During my PhD, I had the opportunity to collaborate and write research papers with researchers from both the industry and the academia. I would like to extend special thanks to the co-authors of my publications: Michael Paulitsch, Sascha Uhrig, Jan Nowotsch, Johannes Freitag, Renato Mancuso, and Rodolfo Pellizzoni. I am grateful to Bernd Koppenhöfer and Max Gapp from Cassidian for the avionics case study. Many thanks to Björn Brandenburg for his feedback on an earlier draft of my publication. I also thank Claire Pagetti, Daniel Gracia Pérez, Rob Davis and Marcus Völp for the enriching discussions. Thanks to also the anonymous reviewers of my publications that helped to see the boundaries of my work and fine the rough edges.

Special appreciation goes to Steffi whose help and support was invaluable to wade through the administrative steps over the past five years. I am particularly grateful to Markus Müller, the king of servers, whose immeasurable help and support enabled swimming through the multitude of tools, their technical issues and also their upgrade cycles.

Many thanks go to my colleagues at department: Rodrigo Coelho, Florian Heilmann, Kristin Krüger, Alexandre Venito and Gautam Gala, Steven Dietrich and Marine Kadar. Your support and feedback especially on the earlier drafts pf this work were invaluable. I would also like to thank my ex-colleagues: Jens Theis, Ali Syed, Rajesh, and Simara Perez.

I also had the opportunity to supervise student assistants and final-year students for theses at master- and bachelor-level. This role helped to enrich another facet of my research life. Many thanks to you for the collaboration: Luc Welter, Hyamali Bello, Frank Dischner, Kristing Krïger, Clemenz Reibetanz and Oliver.

Finally, I would like to extend my special gratitude to my friends Stefan, Michel, Nan, Ahmed, Shirish, Akshay, Sabine, Hannia and Fawsy. Your friendship and support helped cheer me in some of the most difficult moments during this endeavor.

Ankit Agrawal
Kaiserslautern, 30.11.2018

The work presented in this thesis has been partially supported by the European research projects *Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments* (EMC^2) within the EU ARTEMIS Joint Undertaking under grant agreement number 621429.

Publications

I have authored or co-authored the following publications:

Peer-Reviewed Conference Papers

- Ankit Agrawal, Renato Mancuso, Renato Pellizzoni, and Gerhard Fohler. *Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems.* Proceedings of 39$^{\text{th}}$ IEEE Real-Time Systems Symposium (RTSS 2018), Nashville, USA, December 2018.

- Ankit Agrawal and Gerhard Fohler. *DRAM-related Challenges in Task Scheduling with Timing Predictability on COTS Multi-cores for Safety-critical Systems.* Proceedings of 3$^{\text{rd}}$ International Symposium on Memory Systems (MEMSYS 2017). Washington D.C., USA, October 2017.

- Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. *Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study.* Proceedings of 29$^{\text{th}}$ Euromicro Conference on Real-Time Systems (ECRTS 2017), Dubrovnik, Croatia, June, 2017.

Newsletter, Poster Abstract and Work-in-Progress Papers

- Sascha Uhrig, Johannes Freitag, Gerhard Fohler, and Ankit Agrawal. *COTS Multicores in Avionic Applications.* EMC^2 Newsletter 4, November 2016.

- Ankit Agrawal, Gerhard Fohler, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. *Poster Abstract: Slot-Level Time-Triggered Scheduling on COTS Multicore Platform with Resource Contentions.* Proceedings of 22$^{\text{nd}}$ IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016), Vienna, Austria, April 2016.

- Ankit Agrawal, Gerhard Fohler, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. *Slot-Level Time-Triggered Scheduling on COTS Multicore Platform with Resource Contentions.* Proceedings of Work-in-Progress and Demo Session of 22$^{\text{nd}}$ IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016), Vienna, Austria, April 2016.

Contents

List of Figures

List of Tables

I

Introduction

"I'd like to know what this whole show is all about before it's out."
– Piet Hein, *Grooks*

Computing systems enhance human lives. Broadly, these systems are classified as general-purpose computing systems (GPCSs) like smartphones and safety-critical systems (SCSs) like autopilot in airplanes. While failure of GPCSs discomforts humans, failure of SCSs jeopardizes human lives. SCSs must be approved by certification authorities (CAs) of being safe.

Emerging SC applications like sense-and-avoid for electric autonomous vehicles demand not only more computing performance, but also reduced vehicle weight to boost vehicle's power-to-weight ratio. While the computing industry has shifted from single-core processors to multi-core processors for performance gain, SCSs still use single-core processors and are performance constrained.

Multi-core processors implicitly share network-on-chip (NoC) and main memory hardware resources among logically-independent cores that introduces two challenges before their use in SC systems. The first challenge — unpredictable contention and deadline miss — stems from sharing of hardware resources among cores and results in unpredictable performance. When tasks that co-execute on different cores concurrently issue memory requests, the requests suffer contention that slows down co-executing tasks, and can lead a task's deadline miss and system failure, in the worst-case. The second challenge — circular dependency — is a consequence of contentions. Real-time (RT) scheduling decides tasks that co-execute with a task under analysis. The choice of co-executing tasks impacts the execution time of the task under analysis, unlike conventional RT scheduling. This results in circular dependency between scheduling and execution time.

The goal of dissertation is to enable safe scheduling of SC applications on multi-core processors, such that all tasks meet their deadlines even under contentions.

Chapter Structure: Sections I.1 and I.2 present a motivation for this dissertation. Section I.1 provides an insight into emerging safety-critical systems that require increased performance. Section I.2 briefly describes a multi-core architecture and the challenges it introduces for real-time scheduling for multi-core processor use in safety-critical systems. Section I.3 describes the two problems considered in this work. Section

I.4 lists the contributions made by this work. Section I.5 outlines the chapter-wise structure of the dissertation.

I.1 Emerging Safety-Critical Systems

Emerging safety-critical systems demand performance gain. For example, industrial domains like aerospace, automotive, are pursuing electric-powered autonomous vehicles for future mobility. Specifically, Airbus is developing helicopter-style ultra-light autonomous and electric passenger aircraft to ease traffic for urban landscape. E.g., CityAirbus [Air16], and the Vahana aircraft [Lya16].

These emerging aircraft have tight weight constraint — to boost their power-to-weight ratio—, tight area constraint — to increase passenger space, and tight avionics system power constraint — to eliminate need for active cooling that increases weight and area. These aircraft require a sense-and-avoid application for autonomous flying, unavailable today, in addition to the most avionics applications used in today's aircraft.

While autonomous sense-and-avoid applications are under development, a precursor application with sense-display-and-warn (SDAW) function is available today. It is called helicopter terrain awareness and warning system (HTAWS). It shows a helicopter pilot the surrounding topographical layout (including large buildings, power lines) with "flyable" areas together with warnings when the helicopter approaches rough terrain, e.g., when vision is degraded. Such a system also needs to be integrated into future autonomous aircraft to allow the aircraft to perform autonomous path planning and in-flight re-planning.

HTAWS application is currently implemented on a dedicated single-core processor and comprises a mix of memory-intensive and central processing unit (CPU)-intensive workload. It is not feasible for ultra-light autonomous aircraft due to its size, weight and power (SWaP) constraints. Thus, it warrants new solutions that *(i)* increase performance, *(ii)* reduce SWaP consumption (SWaP), *(iii)* require no legacy source-core modifications, and *(iv)* enable incremental certification and composability.

I.2 Multi-cores and Hard Real-Time Scheduling

Since 2005, while computing industry shifted to multi-core processors from their single-core counterparts for more performance, SCS still mainly rely on single-core processors and are yet to shift. Unlike single-core processors, multi-core processors are significantly more challenging to analyze due to the extensive sharing of hardware resources among logically independent execution flows. The primary source of performance unpredictability, in this class of processors, is the memory hierarchy. The memory hierarchy in multi-core platforms is comprised of a number of components that are concurrently accessed by multiple cores. These include: multi-level CPU caches, shared memory controllers and dynamic random access memory (DRAM) banks, and shared input/output (I/O) devices. The interplay of requests originated by multiple cores has a direct impact on the timing of subsequent memory accesses.

When co-executing tasks concurrently issue memory requests, the requests suffer contention. When a task's memory request suffers contention, it slows down the task. The resulting temporal variability is in the range of multiple orders of magnitude, meaning that inaccurate performance modeling and analysis can lead to overly pessimistic worst-case execution time (WCET) estimates. Further, when contention is left unmitigated, it can result in task's deadline miss, that can lead to system failure, in the worst-case. The CAST-32a joint position paper [CAS14] from aerospace certification authorities, further highlights these issues.

SCS require hard real-time scheduling for efficient use of computing resources and to ensure tasks meet deadlines to prevent system failures. HRT scheduling decides tasks that co-execute at a time in a multi-core processor. This introduces a new circular dependency challenge between the execution time of a task and CPU scheduling. When CPU scheduling chooses co-executing tasks that are memory-intensive — issue large number of memory requests —, it results in larger slowdown of a task under analysis, compared to when CPU scheduling chooses co-executing tasks that are non memory-intensive.

A significant body of work exists for hard real-time (HRT) scheduling on conventional multiprocessor system. However, these works require extensions before use in multi-core processors for conventional multiprocessor system models are contention-agnostic and assume a fixed WCET of a task that is independent of co-executing tasks. An even slimmer body of works has provided general results to reason on scheduling of CPU and memory contention. The majority of works in this area assume static assignment of memory resources to CPUs.

Approaches based on static memory bandwidth to cores have been shown to be useful and gained adoption. In addition to controlling the amount of memory bandwidth available to cores, they allow the use of existing HRT scheduling algorithms with minor modifications, as the effect of steady memory bandwidth to cores can be seen as a slower processor. However, the SDAW application comprises memory-intensive and non-memory intensive workloads. Using steady memory bandwidth may lead to either deadline miss of memory-intensive workload or provide only little remaining bandwidth to the other cores.

I.3 Problem Statements

Future SCSs in avionics domain not only require emerging applications like sense-and-avoid but also many existing certified safety-critical avionics applications like flight management system. These existing certified applications mainly execute on single-core processors. There is an urgent industry need for solutions that enable transition to multi-core processors from single-core processors while preserving an existing certified application's CPU schedule in a composable way, i.e., make no assumptions on the CPU schedule of workloads on other cores. Such solutions can result in significant certification cost and time savings. This problem is difficult as, first, the CPU schedule of migrating application's tasks must be guaranteed under contentions while ensuring composability. Second, the transition step requires no modification of a certified appli-

cation's source code to reduce re-development costs, that are generally significant for legacy applications.

While independent safety-critical (SC) applications comprising one or more workloads partitioned across single-core processors benefited from WCET analysis of a workload being independent of co-executing workloads, multi-core processors break this property. Unlike conventional hard real-time multiprocessor scheduling in SCSs, a workload's WCET on multi-core processors depends on co-executing workloads as there is circular dependency between execution time of a workload and its co-executing workloads on other cores. When concurrently executing workloads issue memory requests, their memory requests suffer contention, as memory is shared among cores. Contentions increase a workload's execution time as contentions can stall a core. The increase in a workload's execution time due to stall depends not only its own characteristics like maximum number of memory requests issued but the time of these request with respect to the time and number of memory requests from the other cores. The circular dependency challenge makes it especially difficult to perform WCET analysis for multi-core processors.

Moreover, SC applications typically exhibit disparate memory bandwidth demand over time at two levels. Workloads of an application generally differ in their memory bandwidth demand on the workload-level. On the intra-workload level, a workload's demand changes over time. E.g, workloads generally demand more memory bandwidth at start time due to cold caches, in the first few iterations of a loop, and just before completion due to write-backs to memory. As memory bandwidth is a significant performance bottleneck in multi-core processor, solutions that over-provision memory bandwidth for a workload can make an otherwise feasible taskset unschedulable. On the other hand, under-provisioning can result in a workload's deadline miss. There is need for solutions that enable reserving resources tailored to a workload's demand as it can significantly improve system schedulability. Dynamic memory bandwidth partitioning over time can ease matching memory bandwidth assignment to a workload's requirements over time, but changing memory bandwidth assigned during workload's execution can impact its WCET. While supplying lower than required memory bandwidth slowdowns a workload, supplying more than required may not speedup a workload. Thus, there is need of solutions that allow tailoring memory bandwidth assignments to workloads over time and guarantee the computed WCET is safe.

Moreover, SCSs manufacturers demand solutions that preserve certified single-core application schedules to reduce re-development and re-certification costs for especially legacy applications as they are generally substantial. Further, these manufacturers demand solutions that require no source-code modifications and support incremental certification and composability as certification costs and development costs are significant for SCSs. Composability eases incremental certification by simplifying proving that combining two certified systems will not negatively impact either of the two certified systems. No source-code modifications on migration significantly saves re-development and re-certification costs especially for legacy applications.

I.4 Contributions

In this dissertation, we make the following contributions:

First, we introduce a new scheduling problem for commercial-off-the-shelf (COTS) multi-core processors when some workloads in a partitioned workload set are memory-intensive, as presented in the problem statement. This problem originates from a real certified avionics application—HTAWS— that we show in this work. Workloads in this application drastically differ in their average memory bandwidth demand. Memory bandwidth in multi-core processors is shared across cores and is a significant performance bottleneck. The application's measured resource requirements show that it is not schedulable on a multi-core platform using existing static memory bandwidth partitioning approaches. While reserving resources tailored to a workload's demand can significantly improve workload schedulability, there is a need for solutions that allow reserving memory bandwidth tailored to a workload and compute WCET of a workload under dynamic memory bandwidth assignment to cores.

Second, we present a workload schedulability test for known even-memory-bandwidth-assignment-to-active-cores over time, where the number of active cores represents the cores with non-zero memory bandwidth assignment. The test considers not only workload's characteristics — CPU time demand in isolation and worst case number of memory requests— but also worst-case amount of contentions a workload can suffer from other cores based on the memory bandwidth assignments over time. An offline scheduler requires it to test if a set of slots under consideration with given memory budget assignments meet a workload's worst-case demand of its core-local execution time and worst-case number of memory requests. If the test is successful, it gives guarantees that the given set of slots with their memory budget assignment will meet workload's worst-case requirements at runtime. An offline scheduler is likely to use the test for a workload multiple times to ensure schedulability as well as tightness of interval of slots reserved for a workload, i.e., to prevent under- and over-provisioning of resources. The test constructs an analytical worst-case memory request pattern that capture the maximum amount of stall memory requests can suffer under given memory budget assignments. The test is generic with respect to offline schedulers. Further, it requires sorting of memory budget assignments that dominates the test's computational complexity, but it has low computational complexity in comparison to the static WCET analysis tools. While such tools can in combination with measurements find a worst-case memory request pattern, such an approach is pragmatically infeasible due to the computational complexity involved.

Next, we provide a general framework to perform WCET analysis under dynamic memory bandwidth partitioning when changes in memory bandwidth to cores assignment are time-triggered and known. The method uses memory stall curves to represent the cumulative maximum interference-induced stall for a given number of memory requests performed by a core under analysis. It constructs core-specific memory-stall curves for each memory budget assignment to cores. It uses a fixed-point iterative algorithm to compute the worst-case span of the workload on a core under analysis. We first present the algorithm for the WCET analysis under static budget. The algorithm

improves the WCET analysis for static case. Later, we present a fixed-point iterative algorithm to compute the WCET of a workload on a core, when the memory bandwidth assigned to core changes over time. The method guarantees that the algorithm always completes in a fixed number of iterations and that the computed WCET covers the worst-case memory request pattern a task may exhibit at runtime. We then present a stall maximization algorithm that efficiently implements the fixed-point iterative algorithm for span analysis. It has a complexity similar to a concave optimization problem. Further, the framework can be used to analyze memory schedulers when bandwidth-to-cores allocation over time is known.

Finally, we demonstrate use of our proposed methods for SCS using two Integrated Modular Avionics (IMA) systems case studies. The first case study focuses on preserving an existing certified application's single-core CPU schedule under contentions on a multi-core processor. This is an important industrial requirement to aid transition of existing SCSs from single-core processors to multi-core processors and substantially reduces re-development and re-certification costs especially for legacy applications. Our method uses a resource server mechanism and the schedulability test for dynamic even budget assignment. It formulates ILP constraints for a two-core system and guarantees the certified HTAWS single-core CPU schedule on a multi-core processor. It highlights benefits of our method, no application source-code modifications, composability and incremental certification as our method removes circular dependency between WCET analysis and CPU schedule of other cores.

The second case study considers an IMA scenario. It performs an empirical evaluation of dynamic memory bandwidth assignment policy against two static bandwidth assignment policies — static even and static uneven—, under a fixed workload priority order on each core. The comparison metric is schedulability ratio, i.e., the ratio of schedulable workload sets to generated workload sets for each bandwidth assignment policy. The evaluation shows that our proposed method under dynamic memory bandwidth allocation dominates static partitioning when partitioned workload sets contain disparate memory-intensive workloads, similar to the HTAWS application.

I.5 Dissertation Structure

The dissertation is structured as follows:

Chapter II

This chapter provides a basic understanding of *(i)* multi-core architectures, especially memory sub-system that causes temporal variability in time taken to perform memory request, *(ii)* real-time scheduling and *(iii)* safety-critical systems, especially avionics, relevant for this work.

Chapter III

This chapter describes the system model, the related assumptions and formulates the problem considered in this work. The system model comprises multi-core platform model, workload model, time model, and scheduling model. It also present insights on how to restrict a real multi-core platform to the multi-core platform model considered.

Chapter IV

This chapter presents a resource server based runtime mechanism and a workload schedulability test for dynamic even memory bandwidth assignment to cores over time. It solves the circular dependency between computing a task's WCET and CPU scheduling on other cores. The evaluation shows the combined technique enables preserving CPU schedule of the HTAWS application under contentions on a multi-core.

Chapter V

This chapter presents an analysis to compute a WCET of a workload when memory-bandwidth-to-core allocates is time-triggered. It also presents the evaluation that shows the improvement in performance achieved by our proposed solutions against existing static memory bandwidth partitioning solutions in IMA safety-critical systems.

Chapter VI

This chapter presents the state-of-the-art on multi-core scheduling that includes contention-aware approaches and contention-free approaches. It also presents the works that act as enablers for scheduling on multi-core processors by providing analyses and mechanisms to determine memory request time in isolation and under contentions.

Chapter VII

This work discusses the presented solution, related system model and assumptions, and evaluation in context of *(i)* certification, *(ii)* use in real multi-core platforms, and *(i)* performance.

Chapter VIII

This work summarizes the contributions made in this work and concludes the dissertation.

II

Background

"Architecture is an interface between technology and applications"

– Unknown

This chapter provides an overview of multi-core processor architecture, real-time scheduling and safety-critical systems. Section II.1 presents a multi-core platform architecture, a main memory architecture and the various causes of temporal variability in time taken to perform a memory request. Section II.2 provides an overview of real-time scheduling. Section II.3 describes the characteristics of safety-critical systems (SCSs).

II.1 Multi-core Platforms

In the quest for more processor performance, the consumer computing domain began transitioning to multi-core architectures from single-core architectures in the 2000s. The year 2005 was a milestone as Intel announced the release of their first multi-core platform. Single-core processors suffered from the limited heat dissipation problem that required reducing core frequency to limit power dissipation. Improvements in transistor scaling and chip fabrication led to doubling of the transistor count every 18 to 24 months since the 1960s (as predicted by Moore) that packed ever more transistors in the same area. The improvements allowed increasing core frequency as well to increase performance but dissipated more heat for the same surface area, eventually resulting in the transition to multi-cores. Section II.1.1 presents a multi-core platform architecture. Section II.1.2 describes the main memory architecture and also the various causes of temporal variability in time taken to perform a memory request.

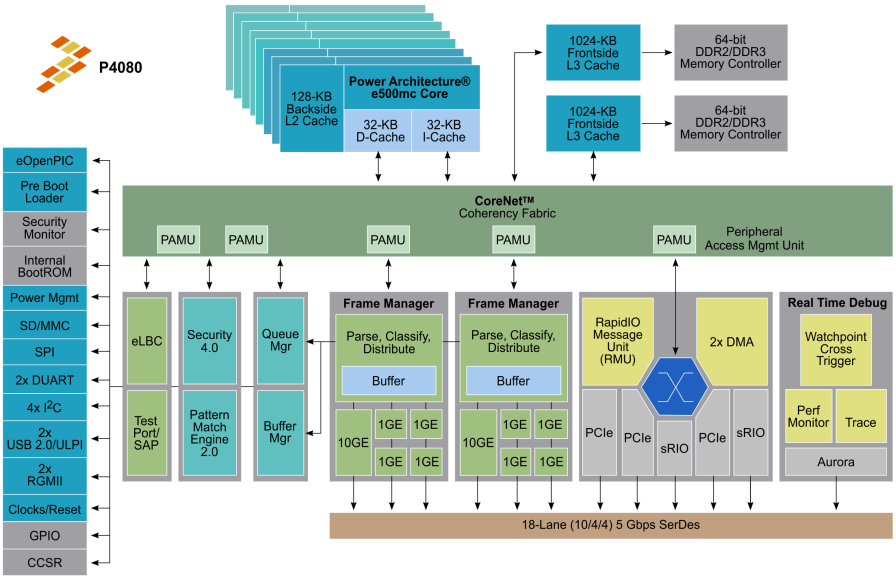

Figure II.1: *Platform Architecture of the 8-core COTS NXP P4080 platform P4080 [Fre13a]*

II.1.1 Platform Architecture

Multi-core platforms feature multiple cores that share the memory hierarchy. These platforms typically use a memory hierarchy that is a combination of fast but low storage capacity memories — caches — and slow but large storage capacity main memory, e.g., dynamic random access memory (DRAM). A DRAM-like main memory is a sub-system that comprises a memory controller and a memory device. While the memory controller is on a processor chip, the memory device is off-chip. The caches and the DRAM are volatile memories. Thus, a multi-core platform has a secondary memory like solid-state drives (SSDs), hard disks, etc.. Secondary memory is non-volatile and much slower than main memory.

Multi-core platforms require interconnects to allow moving data to and from the cores and along the memory hierarchy. There are core-level interconnects that connect a core to a cache and then platform-level interconnect that connect DRAM sub-system, caches, secondary memory and many other peripherals like input/output (I/O) controllers, e.g., ethernet controller, etc.. Figure II.1 shows the platform architecture of a real commercial-off-the-shelf (COTS) multi-core platform comprises 8 cores.

The focus of this work is on cores and the memory hierarchy until the main memory. Thus, the relevant components in the Figure II.1 for this work are the corenet coherency fabric — an interconnect — , power architecture e500mc cores, frontside caches and the memory controllers. The top part of the figure depicts these components. Corenet coherency fabric connects the shared L3 caches and each cores private L1 caches, thereby establishing a path between each core and main memory. Note that, the figure shows only on-chip components and does on show the memory device that is off-chip.

NXP QorIQ P4080 is a high performance multicore platform introduced in 2008 [Fre13a]. It combines eight 32-bit e500mc cores based on Power architecture. It can deliver a per-

formance of 12 GFLOPS (1.5 GFLOPS/core-pre silicon; single-precision floating point) operating at maximum frequency of 1.5 GHz, at a power budget of around 30 W.

It is suited for applications that are highly compute intensive or I/O intensive or both. It is based on a 45 nm technology and targets general purpose embedded computing systems domains like networking, telecom, industrial, aerospace and defense markets.

P4080 is based on a shared-memory with concurrent network multi-core architecture, having eight e500mc cores. The e500mc is a 32-bit core based on Power Architecture. The eight cores, the memory and other peripherals are connected through the CoreNet coherency fabric, providing a concurrent communication network for the processor as shown in Figure II.1 [Fre14]. It also provides cache coherency across all levels in the system.

e500mc Core: The e500mc is a 32-bit core based on Power Architecture [Fre13a] and has 7 pipeline stages. It has 2 levels of private cache (L1 and L2). All cores share the L3 cache. It has 3 integer units, 2 of which are simple units and the third one is a complex unit-providing integer multiply and divide function. It provides 3 instruction levels: User, Supervisor and the Hypervisor. It provides 36-bit of physical addressing resulting in maximum address range of 64 GB. The maximum frequency the core can run on is 1.5 GHz. **Private L1 Cache:** Each core has a separate private L1 data and instruction cache. Each is 32 KB in size and 8-way associative. And they run at the same speed as the core. A single cache line corresponds to 64 bytes. L1 Data cache line replacement selection occurs at the reload time. L1 Instruction cache line replacement is performed when a L1 Instruction cache miss occurs. At the time of cache line replacement selection, if any cache line in the set is invalid, the cache line with the lowest numbered way is chosen. If there is no invalid cache line, the pseudo-least recently used (PLRU) replacement algorithm is used.

Private L2 Cache: Each core also has a write-back, private backside L2 cache having size 128 KB and is 8-way associative. It can be configured as a unified instruction and data cache or only as a data/instruction cache. When used as a unified cache, it contains instructions that are fetched from the memory subsystem or data resulting from L1 cache cast outs. Also, the associativity can be partitioned between data and instruction i.e. say 5-way data and 3-way instruction when used as a unified cache. The cache line replacement algorithm can be configured to one of the following available choices: PLRU, streaming PLRU, or streaming PLRU with aging.

The next section describes a typical main-memory architecture of an NXP P4080-like platforms.

II.1.2 Main Memory Architecture

In this section, we provide an understanding of the main memory architecture based on the third-generation of the dual-data rate (DDR) DRAM memory. Our aim is to show the architecture features that cause temporal variability in memory request time. The sub-system comprises a memory controller and one or more memory devices. A memory device stores data.

Memory Device

We now describe the architecture of a memory device. A memory device is logically organized into ranks and banks. A rank comprises multiple banks. A bank is a two-dimensional memory storage organized in rows and columns. Each row has a fixed size. A bank has a row buffer, that acts like a cache. A read or write to a row is only possible when the contents of a row are in the row buffer. The size of the row buffer equals the size of a row.

Basic Commands: There are three basic commands to access contents of a bank: PRE, ACT and CAS. A PRE command closes a row, i.e., copies the contents of the row buffer back to its respective row. An ACT command open a row, i.e., copies the contents of a request row to row buffer. A memory controller must issue a PRE command before issuing an ACT command, if some other row is open. A CAS command allows read/write to an open row.

Memory refresh: The DRAM technology stores each data in bits — 0 or 1 — in a capacitor by storing electric charge in two levels corresponding to the bit stored. Capacitors are prone to discharge over time, that if not mitigated, will result in data loss. So, this technology requires a critical maintenance operation called refresh that occurs periodically. A DRAM is unavailable to service memory request when it performs refresh. A refresh operation on a row requires three steps: closing the current open row, opening a row that must be refreshed and closing the opened row. Closing a row recharges a row's capacitors.

DDR DRAM Memory: We now present typical architectural information for DDR3 DRAMs. A rank comprises 8 banks. A bank contains 8192 rows each of 4 KB. A row buffer is of 4 KB in size. The periodic refresh rate is 64 ms. A DRAM controller refreshes a row in a bank every $\frac{64}{8192} = 8\mu s$. A refresh operation requires 24 ns to refresh one row in a bank (and at the same time in all banks). It requires issuing a PRE command and an ACT command to each bank. So, every 8 microseconds, one row is refreshed in all banks in a serial order. Thus, the time available for useful work, i.e., to service memory requests is 8 μs minus 24 ns. Note that, some DRAM sub-systems allow configuring the refresh policy to increase memory performance by either refreshing a row early when a bank is idle or by postponing refresh to a row by a few refresh periods, but requires in-depth knowledge of the DRAM module being used and differs between manufacturers.

Memory Controller

A memory controller comprises two parts: front-end and back-end. The back-end of a memory controller arbitrates among the memory requests from cores. The front-end of a memory controller breaks each memory request into PRE, ACT and CAS commands to be issued to a memory device for the selected memory request. It is also generally responsible for refreshes if a DRAM device does not support self-refresh. A memory device typically has a private data and address bus, called a channel, to communicate with a memory controller. Each channel has separate channel controller responsible for converting memory requests to PRE, ACT and CAS commands.

The front-end controller must respect different timing constraints as specified in the JEDEC standard. For example, tFAW timing constraint only allows four ACT commands in any time window of 20 cycles. This constraint is to limit the maximum current drawn by a DRAM device in a 20 cycle window.

Causes of Temporal Variability

DRAM-like memories generally employ locality principles and multi-level scheduling policies to improve average-case memory performance. These memories in COTS domain consider both temporal and spatial locality as opening a row to read (say) 64 bytes of data opens a 4KB row, and a request for 4 bytes of data is answered with (say) 64 bytes of data as data stored nearby is likely to be accessed. A memory request that targets an open row completes much faster compared to that targets a closed row as CAS command to an open row does not require PRE and ACT commands, thereby increasing temporal variability. The multi-level scheduling policies aim at maximizing the bandwidth of the memory sub-system but increase maximum delay experience by a memory request. For example, a memory controller may employ first-ready first-come-first-serve (FR-FCFS) policy for the back-end of the memory controller and round robin at the front-end of the memory controller, in the worst-case. FR-FCFS selects a memory request that is ready for scheduling (whose timing constraints have been met) even if this request arrived later than a waiting request, also known as request re-ordering. FCFS policy is used for tie-breaking. The request re-ordering while good for average-case performance, further increase the temporal variability of a memory request. Moreover, as memory refreshes and timing constraints like tFAW are dynamically handled, such behavior further increase the temporal variability of a memory request.

II.2 Real-Time Scheduling

Real-time (RT) scheduling has been increasingly used for scheduling applications in safety-critical systems. Its purpose is to not only extract performance from hardware for applications but also meet their timing constraints. Timing constraints are common in safety-critical systems as these systems demand applications not only compute a correct result but also deliver it timely to ensure safety. E.g., when a pilot moves a joystick to the left, the plane must react and steer to left within a few milliseconds. If the plane steers to left after a few minutes, it jeopardizes passenger safety. RT scheduling has three branches —- hard, soft and adaptive. While soft RT scheduling suits non-safety-criticality applications, adaptive RT scheduling must react to environment changes. Hard RT scheduling give timing guarantees and is suited for safety-critical systems.

Application split in tasks An application is too coarse a unit for software development, execution and scheduling. Safety-critical system designers split an application in sub-units. The name of sub-units differ among domains. In order to be generic, we refer to a sub-unit as a task in this section. On a software-level, a task is a program that takes an input set, executes a series of statements based on the input set and produces an output set. The instructions are of three types: assignment, decision, and loop. A task

requires CPU, takes input values and produces output values. A task exhibits variable execution times. When a program executes on a CPU, the input values to a program affect the series of statements executed. Between any two execution runs, the series of statements executed can differ. Each decision represents a different execution path. Further, each loop bound represents a different execution path. Execution time is the performance demand of a task on the CPU.

Worst-case: In this past, for single-core processors and multiprocessors, the WCET analysis was performed for each task in isolation. Industrial tools are available that provide the values. E.g., measurement-based like RapiTime and static WCET analysis tools like aiT. While a significant body of research on hard real-time scheduling literature — both for single-core processors and multiprocessors — is available, it cannot be used for predicting execution time of a task in multi-cores. It is memory and NoC contention-agnostic. It assumes the execution time of a task is fixed and independent of other tasks.

II.3 Safety-Critical Systems in Avionics

Safety-critical systems are systems that on failure can result in a catastrophe jeopardizing human lives. These systems require performance guarantees under worst-case. So, such systems employ hard real-time scheduling for SCSs. Based on the likelihood and the impact of failures, these system are classified into different criticality-levels. The aerospace domain considers five safety-criticality-levels, design-assurance level (DAL)-A, the highest criticality, to DAL-E. The DO-178B/C [RTC12] standard associates DAL-A level to catastrophic consequence on failure, DAL-B to hazardous, DAL-C to major, DAL-E to minor and DAL-E to no effect on failure. For example, the flight control system is classified as DAL-A, the in-flight announcement as DAL-C and the in-flight entertainment system as DAL-E. Automotive domain and industrial domain (ISO26262) have similar criticality-levels. The criticality-levels affect the system design and development process and thereby costs.

Certification: SCS require certification to ensure safety. Certification is a process when a SCS manufacturer demonstrates to a certification authority (CA) predictable behavior of the SCS under all foreseeable scenarios including identified potential risks like guaranteeing safe execution of logically independent applications under contentions on a multi-core platform with shared memory sub-system. Certification in avionics domain ensures airworthiness of an aircraft. CAs like Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA) ensure an aircraft design and development meets stringent airworthiness requirements to ensure safety in civilian airspaces in North America and Europe, respectively. Criticality-levels affect the documentation requirements, the allowed qualified tools, etc. It is well-known that the higher the criticality, the costlier is the software development process, as the requirements become more stringent and assumptions more pessimistic that impact the safety margin that must be added to the estimated WCET. Further, criticality-level additionally requires tools like for WCET analysis qualified for the same level.

Fail-safe and Fail-operational: SCS are also classified as fail-safe and fail-operational. A fail-safe system can reach a safe-state immediately on failure, e.g., automotive domain

and railways, as in worst-case, these system can halt on ground. A fail-operational system cannot reach a safe-state immediately and must continue operation before reaching a safe-state. Example of such systems are aircrafts. A flying aircraft on engine-failure must continue operation to land safely, ensured by each large plane having at least two engines with each capable of flying the plane alone. Triple modular redundancy (TMR) is widely used in avionics as they must ensure fail-operational behavior for especially for DAL-A systems.

Properties: SCSs in the avionics domain generally employ time-triggered (TT) release of activities to ensure determinism that eases certification as it requires proving the system is safe for just one-case. Further, these systems require incremental certification [WRPR08], i.e, use an existing certified system in a new system and only certify the additions. Many SCS employ product families to reduce design, development and certification costs by distributing costs across similar products. Composability eases incremental certification as it eases proving that combining two certified systems will not negatively impact either of the two certified systems, saving re-certification costs. In addition, these system have a long life-span, typically 30 years for aircrafts. A SCS manufacturer typically has contractual obligations to provide spare-parts and maintenance for the life-span. As the life-span of current computer technology is much less, SCS manufacturers prefer solutions that do not require source-code modifications when migrating to a new computer platform. This is especially the case legacy applications as redevelopment and re-certification costs are generally substantial [GJR+15].

Shift to COTS Components: COTS platforms are known to significantly reduce development and maintenance costs over the lifetime of a SCS [BF06]. The standardized hardware use eases integration of safety-critical applications from multiple vendors [AC97]. This is evident by the shift from federated architecture to integrated modular avionics (IMA) architectures since the 1990s in the aerospace domain [Iti07]. Boeing B777 [Ada03] airplane was the first to use IMA architecture that improved integration of applications from multiple vendors onto standardized hardware platforms. Similar shift has partly occurred in automotive domain and is starting to occur in railway domain.

Applications and Scheduling: SCS in avionics consider an application comprising partitions. These system consider TT scheduling on inter-partition level, also known as ARINC 653 scheduling [ARI03]. It ensures strict temporal and spatial isolation on a single-core processor between any two partitions that provides determinism and eases certification. The spatial isolation is with respect to the ,e.g., memory regions to prevent a partition writing to another partition's memory region(s). The second scheduling level is the intra-partition level that typically uses a fixed-priority event-triggered (ET) scheduling policy like rate-monotonic. In this work, out focus is on inter-partition level scheduling. To minimize risks, safety-critical (SC) domains tend to rely on multiple vendors to not only increase competition and reduce cost. Moreover, use of TMR for DAL-A criticality applications requires three-different implementations of the same application for replication. This ensures bugs in one application implementation do not affect all replicas. Further, SCSs typically load the essential data and source-code of SC applications to main memory before entering safety-critical mode, as a standard

practice. E.g., avionic systems load the essential data and applications to main memory required for take-off when at a gate. In avionics domain, the aircraft manufacturers like Airbus, Boeing, etc. are responsible for certification of their aircrafts.

We now describe the traditional design and development process followed when designing a new SCS system in avionics domain for single-core processors. A SCS designer collaborates with multiple vendors for developing the system. For a new application, as an application's source-code is yet to be developed, a designer is likely to rely on empirical estimates and past experience to determine the number of partitions required and the worst-case execution time (WCET) of each partition. The designer then communicates to each partition vendor the WCET they must ensure is met. Later, the designer fine tunes each partition's WCET estimates when a vendor delivers a partition's source code. Note that, for DAL-A partition, the DO-178B/C [RTC12]standard requires use of a DAL-A qualified WCET analysis tool like aiT and compiler. Moreover, it is required to certify the hardware platform, the real-time operating system and the scheduler to the level of the highest criticality partition allocated to a platform.

III

System Model and Problem Formulation

This chapter presents i) the system model, assumptions and the related terms and notations and ii) formulates the problem based on the system considered in this work. The system model comprises multi-core platform model, time and scheduling model and workload model. Sections III.1, III.2 and III.3 respectively present a part of the system model and related assumptions. Section III.4 uses the system model to formulate the problem considered and ends the chapter. Next, we present each of the three models in detail.

III.1 Multi-Core Platform Model

We assume a multi-core platform comprises three types of hardware resources: i) n *processing resources*, where each resource is referred as a core or central processing unit (CPU), ii) a *memory resource* shared implicitly among cores and iii) $n + 1$ monitoring resources that monitor the use of the first two resource types. Next, we present the model of each resource type.

III.1.1 Core Model

We assume the platform comprises n homogeneous cores, i.e., all cores have identical core architecture and clock frequency f^c. The c in superscript stands for core. We use index i to refer to a generic core in the set of cores $\mathcal{N}$ i.e., $i \in \mathcal{N} = \{1, \ldots, n\}$.

Caches: We restrict the core cache-hierarchy to only private caches. We assume a core performs in-order execution. We assume a last-level private (LLP) cache miss results in a request to the shared memory resource and stalls the issuing core until the request is completed. Further, we assume LLP cache hits do not generate memory traffic. We do not restrict the number of private cache-levels and make no assumption on the cache replacement policy. Note that, static timing analysis tool may restrict the number of private-cache levels as each cache level increases the computational complexity.

Use in a Real Platform: Achieving this behavior in a real commercial-off-the-shelf (COTS) multi-core platform requires *(i)* either disabling shared caches or configuring them as static random access memories (SRAMs) and *(ii)* disabling pre-fetchers and speculative execution units like branch predictors, to prevent modification of cache states and introduction of unpredictable memory traffic.

III.1.2 Memory Resource Model

We consider a memory resource is implicitly shared among n cores. A core requires access to the memory resource to read and write data. A core's memory request is either a read request or a write request. We assume a memory request stalls a core from the time a LLP cache miss occurs until the time the memory resource completes the request. We assume a write request completes when data is written to memory. We assume a read request completes when the issuing core's LLP cache receives the requested data from the memory.

Memory Request Size: We assume a maximum size (in bytes) of memory request is the worst case. However, we do not require all memory requests to be of a constant size. In NXP P4080 [Fre13b], the memory request size is 64 bytes and it equals the size of a cache-line.

III.1.3 Memory Contention Timing Model

In this work, we consider two memory contention models. The first model assumes a pure round-robin (RR) arbitration policy among cores for memory requests at memory level and considers a memory request requires a constant time L_{max} for completion in worst-case under no contentions and that maximum contention time from a single memory request is also L_{max}. As many high-performance COTS platforms that do not conform to pure RR arbitration policy among cores for memory requests, we consider a second model. The second model assumes a non-decreasing maximum time taken to complete a memory request as the number of cores j increase in the system from 1 to n. This model captures the complex multi-level scheduling schemes in real multi-core processors at interconnect level and memory controller level.

Chapter V uses the constant memory time model. Chapter IV uses the non-decreasing memory request time model.

Constant Memory Request Time Model

This model focuses only on cores and shared memory with a pure RR arbitration policy among cores for memory requests [SCM+16, MPC+15, MPTC17]. This model assumes time to perform a single memory request (both read type and write type) without stall is bounded in the interval: $[L_{min}, L_{max}]$. When using this model, we always use L_{max} as the time taken by each memory request for completion. Further, we assume L_{max} also represents the maximum contention from a single memory request.

Use in a Real Platform: This model requires a multi-core platform that uses pure RR arbitration policy at memory level and allows private bank mapping to cores [YMWP14, KdNA+14, PY16]. Private bank mapping prevents request re-ordering of different cores within dynamic random access memory (DRAM) banks.

Non-decreasing Memory Request Time Model

This model originates from industrial research [NP12] and abstracts timing interferences in high-performance multi-core platforms that do not conform to pure RR arbitration policy among cores for memory requests. Such platforms use complex multi-level arbitration policies at interconnect-level and memory controller-level, e.g., first-ready first-come-first-serve (FR-FCFS) policy to improve average-case performance.

We consider δ_j represents the maximum latency to perform a memory request under contentions when memory requests of j cores contend for the shared memory resource, with $j \in \{1, \ldots, n\}$. We assume δ_j also includes the maximum time a memory request requires to arrive at the memory resource from a core's LLP cache, e.g. through interconnect. Note that, δ_1 represents the time taken to perform a request when only a single core issues memory requests, i.e., a no-contention scenario. Set $\Delta = \{\delta_1 \ldots \delta_n\}$ represents the set of memory latencies of a memory request when the number of contending cores j varies from 1 to n. Further, we assume δ_j is non-decreasing for values of $1 \leq j \leq n$ as follows:

$$\frac{\delta_1}{1} \leq \frac{\delta_2}{2} \leq \ldots \leq \frac{\delta_j}{j} \leq \ldots \leq \frac{\delta_n}{n} \tag{III.1}$$

The Inequality III.1 allows to safely bound the time taken by a memory request accounted with, e.g., δ_4 latency in a 4-core system while at runtime the request may experience a lower latency δ_2 due to no contention from 2 out of the 4 cores. Further, we consider δ_j is the maximum time between the two memory request types — read and write.

Use in a Real Platform: The *non-decreasing memory request time model* is compatible with the 8-core NXP P4080 and the 2-core NXP P5020 platforms. Although the chip manufacturer has not publicly shared the arbitration policy used, it is evident that the policy is definitely not RR from the memory arbitration behavior described in the documentation [Fre14]. While the set of Δ values can be obtained from the hardware

architecture model using white-box approaches like static-analysis, such approaches require knowledge of design time parameters like length of memory buffers, arbitration policy etc. that chip manufacturers are quite reluctant to share publicly. As NXP did not provide the hardware architecture model for any of the two platforms — P4080 and P5020, we rely on an alternate measurement-based method from Nowotsch and Paulitsch [NP12] that uses a black-box approach to obtain the set of memory latency Δ values. As this method has been used for scheduling in safety-critical context in existing works [NPB+14, NP13], we consider it is "good enough" for research purposes in this work. Table III.2 and III.1 show the values of the set of memory latencies Δ as observed on P4080 and P5020 platforms by Airbus Innovations (previously EADS). Note that, it is well-known that volatile memories like DRAM periodically perform a critical operation, e.g., refreshes to prevent loss of data. During this time the memory can not service memory requests. In our model, we do not consider the impact of refresh operations, although [KdNA+14] mentions around 2% increase in memory request time for DRAMs.

Table III.1: *NXP P4080 platform: Maximum observed memory latencies δ_j (ns) for different number of contending cores j*

No. contending cores j	Mem. Lat. δ_j (in ns)
1	34.17
2	136.67
3	204.17
4	385.83
5	430.83
6	614.17
7	653.33
8	839.17

Table III.2: *Maximum observed memory latencies δ_j (in ns) for different number of contending cores j on COTS multi-cores P5020 and P4080*

COTS Multi-core	Mem. Lat. δ_1 (ns)	Mem. Lat. δ_2 (ns)
P5020	24.17	49.17
P4080	34.17	136.67

III.1.4 Monitoring Resources Model

We assume the platform has two types of hardware monitoring resources: a *timer* and *n memory monitors*. The *timer* tracks the platform time since the platform began execution. We assume it periodically notifies all cores of current platform time, e.g., via a timer interrupt. We assume a one-to-one mapping between *memory monitors* and

cores. A *memory monitor* of core i counts core i memory requests to the shared memory resource. It notifies core i when the counter reaches a preset value, e.g., via an interrupt. We allow memory monitors to have different preset values that a scheduler can modify at runtime.

Use in Real Platform: Our monitoring resources model is compatible with many COTS multi-core platforms like NXP P4080 [Fre13b] and [NXP16]. These platforms contain a platform-level hardware timer that tracks time. Further, these platforms also contain memory monitor for each core in the form of performance monitor. A performance monitor requires configuration to function as a memory monitor.

III.2 Time Model and CPU Scheduling Model

Time Model: We consider a time-triggered (TT) system. In a TT system, activities, e.g., release of a workload, pre-empting a workload, are initiated by the progression of time [Kop11]. A common way of operation in TT systems is to assume a minimum temporal granularity of operation at runtime called slots [Kop11]. We divide the timeline into time granules of fixed duration called slots, where Υ represents the duration of a slot s. In a safety-critical domain like avionics, the system designer determines a suitable time duration. We consider the unit of slot duration is CPU clock cycles *cycles*. We assume all cores in a multi-core have a common notion of time and that the slots are synchronized across all cores.

CPU/Core-level Scheduling Model: We consider a partitioned system, where each core has its own CPU scheduler. We restrict to CPU schedulers that adhere to our time model. Such a scheduler only executes at the start of each slot s and selects a workload to execute on its respective core. Our CPU scheduling model is compatible with TT systems and allows a CPU scheduler to select a workload to execute either only from a schedule table created at pre-runtime, or from a combination of runtime CPU scheduling and a schedule table. Note that, however, we do not make any assumptions on CPU scheduling policy used.

III.3 Workload Model

We consider a set of independent safety-critical workloads $\mathcal{W}$ is statically partitioned among n cores. We denote core n workload set as $\mathcal{W}n$. In a real system, a workload corresponds to either a single task or a group of tasks, like an avionic partition. We abstract away the details of each task and only consider the "load", or "workload" in terms of CPU time and the number of memory requests that a workload must complete until a given deadline.

III.3.1 Characterization

A workload ω_u is characterized by the tuple $\langle rl_u, dl_u, E_u, \mu_u \rangle$, where

- rl_u is the earliest start time,

- dl_u is the absolute deadline,
- E_u is the core-local execution time (in CPU clock cycles) excluding the time taken by memory requests,
- μ_u is the maximum number of memory requests to the memory.

We assume each workload is safety-critical, i.e., must complete until its deadline. We assume st_u and ft_u denote the start time and finish time of a workload ω_u. We assume rl_u, dl_u, st_u and ft_u are integer multiples of the Υ. This is standard assumption in TT systems. Further, the workload characterization parameter μ_u is agnostic of memory request type and represents the cumulative maximum of number of read and write memory requests of a workload ω_u. Note that, our workload model captures workload-level memory intensity with parameters E_u and μ_u. It is independent of the runtime memory request pattern of a workload, when and how many memory requests a workload issues at runtime.

III.3.2 Span

Conventional real-time multiprocessor scheduling assumes worst-case execution time (WCET) to be independent of other workloads. It assumes that WCET is a property of a workload and the processor considered and that it captures the worst-case execution time demand of a workload. WCET is contention-agnostic as it does not capture the additional stall from contentions that increase a workload's execution time demand.

However, based on our multi-core model, a workload's requirements of core-local execution time and number of memory requests, and additional CPU stall due to the contentions suffered by the issued memory requests affects the maximum number of slots a workload requires for completion. This is referred to as span, is contention-aware and is defined below.

Definition 1 (Span)**.** *The span of a workload ω_u is the maximum number of slots required to complete core-local execution time E_u and μ_u number of memory requests. We denote span by C_u.*

Nowotsch and Paulitsch [NPB+14, NP13] presented a methodology to acquire worst-case values of the parameters E_u and μ_u of a workload ω_u using a combination of static timing analysis tool like aiT and measurements. Note that, since the obtained E_u and μ_u values may relate to different execution runs of a workload ω_u, it can inflate span of workload ω_u.

III.4 Problem Formulation

A multi-core processor implicitly shares a single memory resource, thereby memory bandwidth among cores. When workloads executing on cores concurrently issue memory requests to shared memory, their requests suffer contention. Consider a generic workload ω_u allocated to core i When a workload ω_u memory request suffers contention,

it increases the stall suffered by the core i, and thereby stall suffered by ω_u, that increases its completion time. Workloads differ not only in their requirements of core-local execution time and maximum number of memory requests, but also on the interleaving between the two requirements. We refer to this interleaving as memory request pattern. The increase in workload ω_u stall depends not only depends on its own memory request pattern, but also on the memory request patterns of the co-executing workloads on the remaining cores. Thus, there is *circular dependency* between completion time of a workload ω_u and co-executed workloads, even though we assume an independent partitioned workload set. The *circular dependency challenge* makes it especially difficult to perform span analysis for multi-core processors.

Conventional multiprocessor scheduling lacks WCET time analysis that considers the circular dependency challenge as it is contention-agnostic. It assumes WCET of a workload is fixed and independent of all other workloads in an independent workload set. While static memory bandwidth partitioning approaches are known to ease span analysis, as static bandwidth partitioning among cores overcomes the circular dependency challenge and ensures composability. However, static partitioning is known to be suboptimal in general case. When workloads assigned to a core substantially differ in their ratio of memory demand to CPU demand, e.g., sense-display-and-warn (SDAW) application, static approaches are prone to waste shared memory bandwidth, constraining schedulability.

Another approach for span analysis is synchronized scheduling of co-executing workloads. While this approach eases span analysis by requiring start-time synchronization of all co-executing workloads, this requirement constrains scheduling. Further, it is not composable as a minor increase in a workload's memory bandwidth requirement, not only increases the completion time of all co-executing workloads, but also affects the start time of upcoming set of co-executing workloads.

This work considers the problem of performing a span analysis for a partitioned independent workload set on multi-core processors where homogeneous cores implicitly share memory bandwidth, while ensuring composability such that the completion time of a workload is independent of co-executing workloads while improving schedulability compared to the static memory bandwidth partitioning approaches.

IV

Resource Servers and Scheduling

In this chapter, we propose resource servers to extend resource control and reservation to multiple resources — CPUs and memory — and overcome the circular dependency between WCET and core-level scheduling of the co-executing cores. Section IV.1 provides an background of partitioning memory bandwidth across cores and its associated impact on a workload. Section IV.2 presents the resource servers. In Section IV.3, we present a workload schedulability test when memory bandwidth is equally partitioned among a subset of cores where the subset of cores can differ across slots. Section IV.4 applies the proposed method of resource servers with schedulability test to a real avionics use and highlights the our method's benefit compared to static partitioning approaches.

IV.1 Background

In multi-core systems the available memory bandwidth can be arbitrarily distributed among cores. Take a 2-core system for instance, as depicted in Figure IV.1. Workload on the two cores can be either CPU-intensive (blue), or memory-intensive (red). For simplicity, the figure assumes that CPU-intensive workload is unaffected by changes in memory bandwidth (BW) assignment. Conversely, memory-intensive workload is roughly linearly affected by it. An even assignment as depicted in Figure IV.1(a) would provide 50% of the available memory bandwidth to each core. Even partitioning is not flexible: mostly memory intensive workload is deployed on core Core A, while mostly CPU-intensive workload is scheduled on Core B. As such, workload is penalized on Core A while memory bandwidth is wasted on Core B. Under this setup, the memory-intensive workload on Core A and B take 5 and 3 time units to complete, respectively. The overall utilization is 95%.

If Core A is known to run memory-intensive tasks while Core B mostly processes CPU-intensive workload, it is beneficial to perform an uneven assignment – e.g., 80% and 20% of the available bandwidth assigned to Core A and B, respectively. This is depicted in Figure IV.1(b). In this case, bandwidth can be distributed to better meet the CPU/memory needs of workload on the various cores. The memory-intensive workload on Core A can benefit from this assignment, now completing in 2 time units. However, the (shorter) memory-intensive workload on Core B is negatively affected, completing in 4 time units. Overall utilization decreases to 85% in our example.

In both uneven and even partitioning, memory bandwidth allocation is static, i.e., it does not change over time. The workload on each core, however, can undergo variations in terms of memory requirements. This is often the case as more/less memory-intensive tasks (or partitions) are scheduled on each core. As such, it is natural to consider a scheme where bandwidth-to-cores assignment is varied over time. In this case, we talk of dynamic bandwidth partitioning, i.e., memory scheduling. Figure IV.1 depicts one such example. Here, when only one core is executing memory-intensive workload, it is given 80% of the available bandwidth; when both are executing the same type of workload, the bandwidth is evenly distributed. Under the new scheme, the system operates at 80% utilization. In general, dynamic bandwidth assignment can yield significant performance improvement, because it is possible to produce an assignment scheme that follows the memory requirements of scheduled workload over time.

IV.2 Resource Servers

In this section, we extend resource control and reservation to multiple resources — CPUs and memory — in parallel via resource servers. A slot represents a fixed temporal granularity of resource control. It can be seen as a temporal granularity of resource reservation, i.e., reserving chunks of CPU time for a workload. Slot can be also be used as a temporal granularity to partition shared memory bandwidth among cores over time. Section IV.2.1 presents the system model, the memory contention model and the server

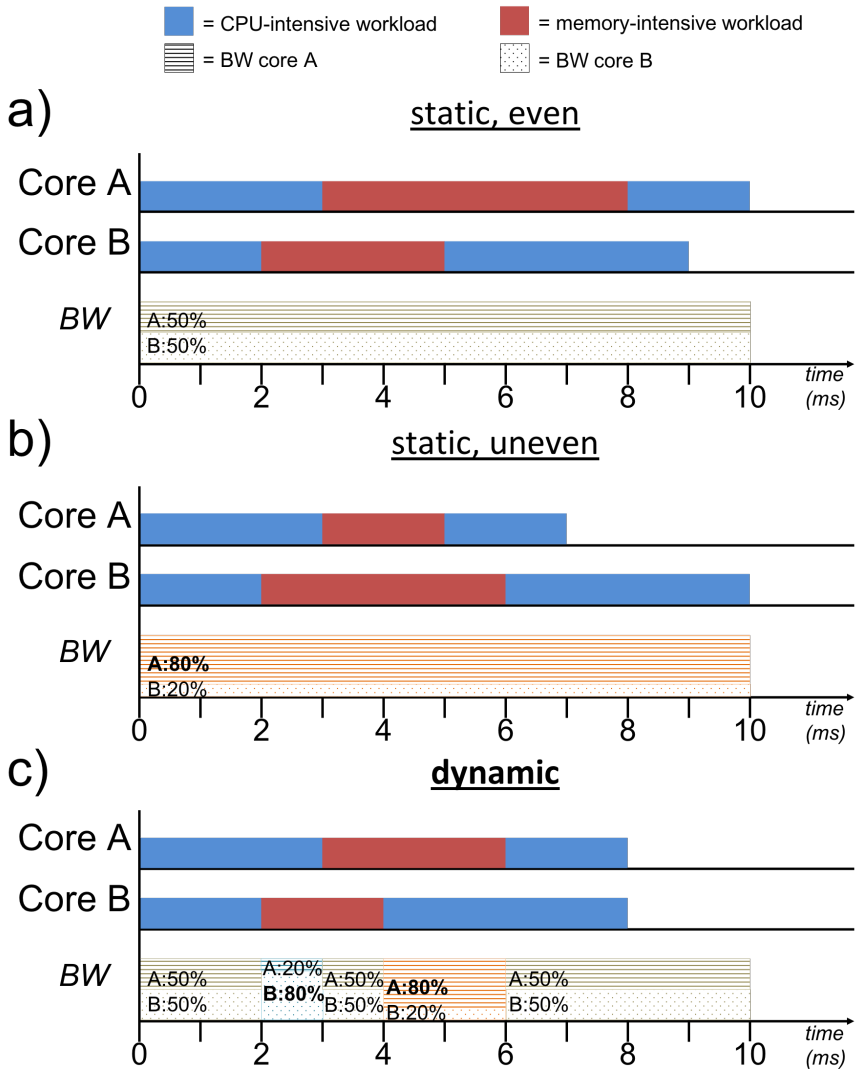

Figure IV.1: *Example of static and even (a), static and uneven (b), and dynamic (c) memory bandwidth management on a 2-core (A and B) system.*

model. Sections IV.2.2 and IV.2.3 present the CPU servers and memory servers that in a pair jointly bound contention on each core. Section IV.2.2 describes the interaction between the servers. Section IV.2.5 describes valid budget assignments for each instance of CPU server and memory server. Section IV.2.6 highlights how the resource servers overcome circular dependency between WCET and core-level scheduling on co-executing cores. Section IV.2.7 illustrates the functioning of resource servers at runtime.

IV.2.1 System Model and Server Model

System Model: This chapter considers the the system model presented in Chapter III. Further, we consider the *non-decreasing memory time model* as presented in Chapter III.1.3. The system model comprises the multi-core platform model, the time model and CPU scheduling model and the workload model, presents in Sections III.1, III.2 and III.3, respectively.

Server Model: We propose two types of resource servers — CPU server σ^c and memory server σ^m. We consider n pairs of a CPU server σ^c_i and a memory server σ^c_i, one pair for each core $i \in \mathcal{N} = \{1, \ldots, n\}$. We assume all servers have the same period that equals the slot duration Υ. We assume all servers are released synchronously at the begin of each slot s on their respective core i. We describe the functioning of a pair of resource servers on a generic core i, as identical rules govern all resource servers pair. For simplicity, we refer to a generic workload on core i as ω_i.

IV.2.2 CPU Server

A CPU server $\sigma^c_{i,s}$ is released at the start of each slot s on each core i. Its period T^c_i equals the slot duration Υ. A CPU server's $\sigma^c_{i,s}$ budget is replenished at the start a slot s. Budget $q^c_{i,s}$ represents a CPU server's $\sigma^c_{i,s}$ replenished amount at the start of slot s and is denoted in CPU clock cycles. We allow CPU server budget to change across slots. A CPU server $\sigma^c_{i,s}$ regulates the maximum amount of CPU time given to a workload ω_i in a slot s based the assigned CPU budget $q^c_{i,s}$. A CPU server's budget $q^c_{i,s}$ decreases with the progression of time until it is zero. During runtime, an executing workload ω_i consumes the CPU server's budget $q^c_{i,s}$ in slot s for its core-local execution time, the time taken for memory requests and the stall time due to contentions from the memory requests of the other cores in the system. $\bar{q}^c_{i,s}(t)$ denotes a CPU server's remaining budget in slot s on core i at time t, with $t \in [s \times \Upsilon, (s+1)\Upsilon)$. We represent a CPU budget assignment of n cores in slot s as $\mathcal{Q}^c_s = \{q^c_{1,s}, \ldots, q^c_{i,s}, \ldots, q^c_{n,s}\}$, where $q^c_{i,s}$ is the CPU budget-to-core i assignment in slot s. For simplicity, we now refer to a CPU server budget as *CPU budget*.

IV.2.3 Memory Server

The key idea of a memory server is to periodically regulate the number of memory requests that a core i is allowed to perform during a slot s, based on the works [YYP+13, YYP+16]. A memory server $\sigma^m_{i,s}$ is released at the start of each slot s on each core i. Its period T^m_i equals the slot duration Υ. A memory server's $\sigma^m_{i,s}$ budget is replenished

at the start a slot s. Budget $q^m_{i,s}$ represents the replenished amount at begin of slot s on core i and is denoted in number of memory requests. We allow memory budgets to change across slots. A memory server $\sigma^m_{i,s}$ regulates the maximum number of memory requests a workload ω_i can issue in a slot s based the assigned memory budget $q^m_{i,s}$. A memory server's budget $q^m_{i,s}$ decreases by one unit when an executing workload issues a memory request (read/write) until it is zero. During runtime, an executing workload ω_i consumes the memory server's budget $q^m_{i,s}$ in slot s for its memory requests. $\bar{q}^m_{i,s}(t)$ denotes a memory server's remaining budget in slot s on core i at time t with $t \in [s \times \Upsilon, (s+1) \times \Upsilon)$. We represent a memory budget assignment of n cores in slot s as $\mathcal{Q}^m_s = \{q^m_{1,s}, \ldots, q^m_{i,s}, \ldots, q^m_{n,s}\}$, where $q^m_{i,s}$ is the memory budget-to-core i assignment in slot s. For simplicity, we now refer to a memory server budget as *memory budget*.

IV.2.4 Behavior at Runtime

During runtime, each core-level scheduler sets the budgets of its CPU server and memory server at the start of each slot s. The server budgets need not be same across slots. For time-triggered systems, these slot-level budgets are read from a core's schedule table found in the offline phase. For time-triggered (TT) systems with online flexibility [IF09] a core-level scheduler can also compute and then assign these budgets at the begin of each slot s. However, such behavior requires ensuring a recomputed memory server's budget on a core i either has no adverse effect on the timing of co-executing workloads on the other cores, or that the recomputed budget has been verified and agreed upon by the other cores. Both the scenarios are non-trivial to handle at runtime and are out of scope of this work.

A CPU budget decreases with the progression of time in a slot. A memory budget decreases by 1 unit on each memory request issued by an executing workload in a slot. A workload ω_i continues execution in a slot s at time t with $t \in [s \times \Upsilon, (s+1) \times \Upsilon)$, if and only if both the servers have positive budgets, i.e.,

$$\bar{q}^m_{i,s}(t) \geq 0 \wedge \bar{q}^m_{i,s}(t) \geq 0 \tag{IV.1}$$

When in a slot s at time t on core i, if either of the two servers exhausts its budget, then an executing workload ω_i is stalled until the next slot and the core is idle. If any of the two servers has a positive remaining budget, it is discarded at time t. Jointly, the two servers on each core guarantee that the budgets provided for each slot in the offline schedule table, hold at runtime. This enables contention-aware dynamic memory bandwidth isolation between cores.

IV.2.5 Valid Server Budgets

Our proposed resource servers work for an arbitrary number of dynamic memory bandwidth levels and unequal distribution among cores. We now describe on what constitutes a valid memory budget assignment and a valid CPU budget assignment.

Consider a memory budget assignment in slot s as $\mathcal{Q}^m_s = \{q^m_{1,s}, \ldots, q^m_{y,s}, \ldots, q^m_{n,s}\}$, sorted in increasing number of memory budgets. Also, consider a corresponding CPU

budget assignment $\mathcal{Q}_s^c = \{q_{1,s}^c, \ldots, q_{y,s}^c, \ldots, q_{n,s}^c\}$ in slot s. Let p represent the number of cores with non-zero memory budget in $\mathcal{Q}_s^m$ and that at least one core has non-zero memory budget $p \neq 0$ in slot s.

$$q_{y,s}^c \leq T_y^c = \Upsilon \tag{IV.2}$$

$$q_{y,s}^m = 0, \text{ if } q_{y,s}^c = 0 \tag{IV.3}$$

$$q_{y,s}^c = 0, \text{ if } q_{y,s}^m = 0 \tag{IV.4}$$

$$q_{y,s}^c \geq \sum_{z=n-p+1}^{y} q_z^m - q_{z-1}^m * \delta_{n-z}, \text{ where } q_0^m = 0 \tag{IV.5}$$

The memory and CPU budget assignments are valid in a slot s if the conditions in Equations IV.2–IV.5 hold. Inequality IV.2 ensures an assigned CPU server budget $q_{y,s}^c$ is never greater than its period T_y^c that equals the slot duration Υ. Equations IV.3 ensure an assigned memory server budget $q_{y,s}^m$ always has 0 budget when the corresponding CPU server budget $q_{y,s}^c$ assigned equals 0, and vice-versa in Equation IV.4. The Inequality IV.5 ensures that a memory server's corresponding CPU has enough CPU budget to account for maximum time required to perform $q_{y,s}^m$ under contentions on a generic core y in slot s. It also shows the relationship between the two servers and the memory latencies The L.H.S of the Inequality IV.5 is based on the interference-sensitive WCET computation [NPB+14, NP13]. When all memory budget values in a slot s are greater than 0, i.e., $p = n$, the L.H.S of the Inequality IV.5 considers that only a maximum of $q_{1,s}^m$ memory requests from each core contending core will suffer memory latency δ_n, $q_{2,s}^m - q_{1,s}^m$ will suffer latency δ_{n-1} and so on, in the worst case.

IV.2.6 Resolving Circular Dependency

Resource servers remove the circular dependency between span of a workload and core-level scheduling of other cores. The memory servers enable partitioning of shared memory among cores, thereby bounding contentions that can occur in a slot. Static memory bandwidth partitioning mechanisms assign a constant amount of memory bandwidth to each core before runtime. This limits the worst-case number of contentions any workload on a core can experience at any time, which allows computation of workload span separately from scheduling. But, such approaches limit tailoring the memory bandwidth partitioning to a workload.

In contrast, under dynamic memory bandwidth isolation, the resource servers provide an offline scheduler the freedom to tailor partitioning of memory bandwidth dynamically across cores to match memory bandwidth demands of workloads on per slot-basis, in comparison to static memory bandwidth partitioning approaches. The resource servers aid offline scheduler to overcome the circular dependency between core-level CPU

scheduling and span of a workload. Under dynamic memory bandwidth, resolving circular dependency for span computation of a workload on a core requires knowledge of memory budget assignment in each slot, that are computable by an offline scheduler. This raises the question of how to optimally assign memory budgets with respect to maximizing workload schedulability, that is outside the scope of this work.

$$\forall \delta_j \in \Delta \ , \ \hat{q}^{m,j} = \left\lfloor \frac{\hat{q}^{c,1}}{\delta_j} \right\rfloor \tag{IV.6}$$

While the resource servers allow offline scheduler to consider arbitrary number of budget assignments, for the rest of this chapter we limit ourselves to dynamic even memory budget assignments levels. The next Chapter V considers arbitrary number of memory budget assignments levels.

We illustrate the dynamic even memory budget assignments using an example. Consider a Υ of 1ms and a CPU budget $\hat{q}^{c,1}$ equals to Υ. Then, Table IV.1 shows the memory budgets $\hat{q}^{m,j}$ considering the memory latencies from Table III.2. Note that, the memory latency δ_0 equals ∞ that represents the case when a core is not active in a slot and we assume that all inactive cores in a slot have memory budget and CPU budget equal to 0.

Table IV.1: *Per core memory server budgets $\hat{q}^{m,j}$ with equal distribution of requests corresponding to memory latencies from Table III.2 for CPU server budget $\hat{q}^{c,1} = \Upsilon = 1ms$*

COTS Multicore	Mem. bud. $\hat{q}^{m,0}$	Mem. bud. $\hat{q}^{m,1}$	Mem. bud. $\hat{q}^{m,2}$
P5020	0	41379	20338
P4080	0	29268	7317

IV.2.7 Example

Figure IV.2 shows an illustrative example of our proposed method for $n = 2$ cores using memory latencies for P4080 platform (row 3 in Table III.2). The y-axis represents a time line divided into slots with duration $\Upsilon = 1ms$. Each core i with $i \in \{1, 2\}$ has two servers — a CPU server σ_i^c and a memory server σ_i^m, as shown on the left side in the figure. All servers have the same period that equals the slot duration $\Upsilon = 1ms$. All servers are released synchronously at the start of each slot s as shown by the vertical arrows that mark release and deadline of each server instance. For each CPU server — σ_1^c and σ_2^c— the dotted horizontal line marked with Υ depicts the maximum allowed CPU budget that equals the slot length Υ.

The example considers three different dynamic bandwidth levels for each memory server — σ_1^m and σ_2^m that represent three unique memory budget values that can be assigned to a memory server, as described in Section IV.2.6, with valid budget distributions shown in row 3 of Table IV.1. The symbols $\hat{q}^{m,0}, \hat{q}^{m,1}, \hat{q}^{m,2}$ denote the three unique budget values, where $\hat{q}^{m,0} = 0$ represents the memory budget value when no core is

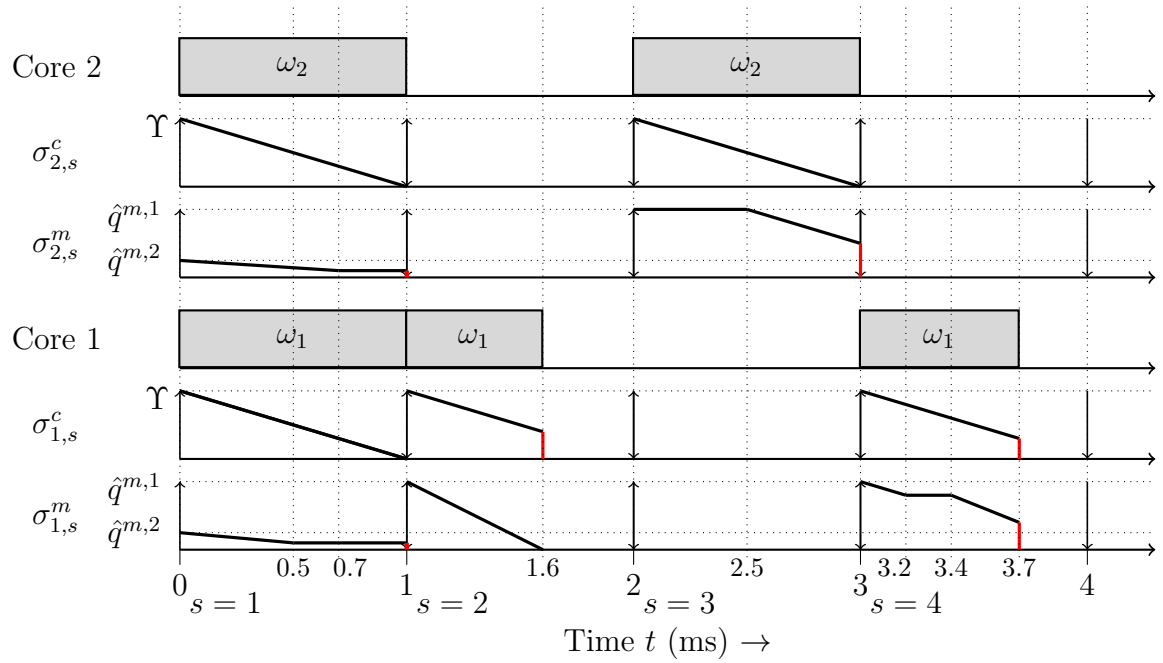

Figure IV.2: *Illustrative example of our proposed method with three dynamic bandwidth levels*

active in a slot, $\hat{q}^{m,1} = 29268$ represents the budget value when only one core is active in a slot and $\hat{q}^{m,2} = 7317$ represents the budget value when two cores are active in a slot. In the figure, the labeled dotted horizontal lines show the two budget levels $\hat{q}^{m,1}$ and $\hat{q}^{m,2}$ for each memory server. The red line represents a server discarding its unused budget. During runtime, at the start of each slot s, each core-level scheduler sets the corresponding server budgets for each server on its respective core, based on the offline schedule table (shown via budgets and workload assignment in Figure IV.2).

We now describe the interaction of resource servers on core 1 with workload $wl1$ and core-level scheduler on core 1. In slot $s = 1$ at $t = 0$, both the cores are active and each core-level scheduler sets the corresponding memory budget to $q^m_{i,1} = 7317$ and CPU budget to $q^c_{i,1} = 1ms$. Core 1 scheduler dispatches workload ω_1 on core 1 and core 2 scheduler dispatches ω_2 on core 2.At the start, the workload on each core issue lots of requests, possibly due to cold caches and data loading, as evident by each core's remaining memory budget decreasing in the time interval $[0, 0.5]ms$. At $t = 0.5ms$, ω_1 stops issuing memory budgets but continues execution just until t=1ms. At t=1ms, core 1's memory server drops the remaining unused budget as shown in the figure by thick red line.

At the start of slot $s = 2$ (at time $t = 1$ms), only core 1 is active in the system. The assigned memory budget $q^m_{1,2}$ equals 29268 and CPU budget $q^c_{1,2}$ equals 1ms. In the time interval $[1, 1.6)$ms, the workload ω_1 issues memory requests as indicated by the corresponding decrease in remaining memory budget. At $t = 1.6$ms, the remaining memory server budget equals 0 that causes the CPU server to discard its budget as shown thick the vertical line and preempt the workload ω_1. Core 1 is in idle state until t=2ms. We refer to such behavior of resource servers as *memory regulation*.

In slot $s = 3$, core 1 servers are assigned a 0 budget, i.e., the core is idle and is not allowed to execute any workload. In slot $s = 4$, only core 1 is again active in the system. The assigned memory budget $q^m_{1,4}$ equals 29268 and CPU budget $q^c_{1,4}$ equals 1ms. In the time interval $[3, 3.2)$ms, ω_1 performs memory requests as indicated by the decrease in memory budget. Later, ω_1 issues no memory requests in the time interval $[3.4, 3.2)$ms. Finally, in the time interval $[3.4, 3.7)$ms ω_1 performs a lot of memory requests, possibly writes to memory as it is approaching completion. At $t = 3.7$, workload ω_1 completes execution and both the servers discard the unused budget and core 1 is idle in the time interval $[3.7, 4)ms$.

IV.3 Scheduling under Dynamic Even Memory Bandwidth

Our goal is to provide a test for an offline scheduler to check if a workload ω_i is schedulable in the considered slots that are allowed differ in their memory budget assignment. We limit the memory budget assignment to dynamic even case, i.e., memory budget assignment is always even, i.e, equally divided among a subset of cores, where the subset of cores is allowed to differ across slots. Equation IV.6 provides valid memory budget assignments for the even case. Section IV.3.1 illustrates the problem of checking for workload schedulability under dynamic memory budgets that arises from memory request pattern. Section IV.3.2 presents analytical worst-case memory request pattern for dynamic even memory budgets. Section IV.3.3 uses the analytical worst-case memory request pattern and presents how an offline scheduler tests for workload schedulability before assigning slots to a workload under known slot-level even memory budget assignment.

IV.3.1 Memory Request Pattern

Static memory bandwidth isolations mechanisms assign same memory budget in each server instance on a core. Such mechanisms are agnostic of the number of active cores in each slot and need to consider the worst-case memory latency corresponding to a maximum number of active cores. Therefore, the use of a static memory budget per core considering worst-case memory latency ensures that if a workload performs a memory request in any of its assigned slots, a workload's execution time does not increase. However, under dynamic memory bandwidth, a workload may receive different memory budgets in each slot. Further, since the latencies of memory requests may differ between different memory budgets, the interaction between memory budgets and a workload's memory request pattern impacts a workload's execution time. We use an example to highlight this interaction.

Consider a workload ω_1: $\langle rl_1 = 0, dl_3 = 3, E_1 = 1, \mu_1 = 60\rangle$.

An offline scheduler identifies three slots for ω_1 on core 1. Consider two illustrative runtime memory request patterns of ω_1 considering the budget assignment as shown in Table IV.2 (columns 2,3 and 4):

Table IV.2: *Example slots to depict the interaction between memory request patterns and dynamic memory server budgets*

Slot $s =$	$s = 1$	$s = 2$	$s = 3$	$s = 1'$	$s = 2'$	$s = 3'$
Mem. ser. bud. $q^m_{1,s}$	45	100	15	100	45	15
CPU ser. bud. $q^c_{1,s}$	1	1	1	1	1	1

1. If this workload uses slot $s = 1$ for its core-local execution time E_1, and slot $s = 2$ for its $\mu_1 = 60$ memory requests, then slots $\{1, 2\}$ are sufficient to meet its E_1 and μ_1 requirements.

2. On the other hand, if ω_1 uses slot $s = 1$ for 45 memory requests, slot $s = 2$ for its E_1, and slot $s = 3$ for the remaining 15 memory requests, then ω_1 requires slots $\{1, 2, 3\}$ to meet its requirements.

If the memory budgets of the three slots were different as shown in columns 5,6 and 7 in Table IV.2, then using the previously considered memory request pattern 2, ω_1 may use slot $s = 1'$ for 45 memory requests and the remaining 0.55 time in the slot for a part of its E_1, slot $s = 2'$ for its remaining 0.45 time units of E_1 with the remaining slot for its 15 memory requests. Thus, it just requires 2 slots. Similarly, using previously considered memory request pattern 1, the workload requires three slots.

Our proposed method is independent of the runtime memory request patterns of a workload as we consider the worst-case memory request pattern of a workload for assigned memory budgets in each slot.

IV.3.2 Analytical Worst-case Pattern

The previous example illustrates the need to construct an analytical worst-case memory request pattern for each workload to ensure the slots with dynamic memory budgets under consideration will meet the workload's requirements. Static WCET analysis tools in combination with measurements can help to find a worst-case memory request pattern. However, pragmatically such an approach is infeasible due to the computational complexity involved. Further such methods need to explore all valid inputs. To overcome these issues, our method uses a worst-case memory request pattern that only requires information about the workload model and the memory budget assigned in each slot.

The worst-case memory request pattern for a workload ω_i assigned to execute on i in slots s $\forall s \in [a, b]$ manifests when a workload ω_i uses $\frac{E_i}{\hat{q}^{c,1}}$ slots $\in [a, b]$ with the largest assigned memory budgets for its core-local execution time E_i requirement, and the remaining slots for its μ_i memory request requirement. This insight on worst-case memory request pattern helps to overcome this issue and not needing an exploration of the search space, but just requires checking for a single worst-case memory request pattern if the slots under consideration will meet the workload's requirements. However, the single worst-case memory request pattern is specific to the combination of the memory budget assignments of the slots under consideration and a workload's E_i and μ_i requirements.

Figure IV.3 shows this general case with exemplary dynamic memory request server budgets for slots a, $b-1$ and b. Note, that the worst-case memory request pattern may not manifest at runtime. Nevertheless, considering it allows an offline scheduler to provide execution time guarantees, irrespective of the runtime memory request patterns. In the next section, we present how an offline scheduler tests for schedulability before assigning slots to a workload under dynamic memory budget assignment.

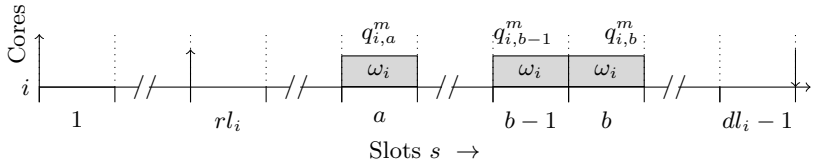

Figure IV.3: *General case when an offline scheduler considers slots $\forall s \in [a,b]$ on some core i for workload ω_i with possibly different memory budgets.*

IV.3.3 Offline Scheduling and Span Computation

This section describes a workload schedulability test for offline schedulers to check for schedulability before assigning slots to a workload under known slot-level even memory budget assignment considering an analytical worst-case memory request pattern.

When scheduling offline, consider a general case as shown in Figure IV.3 with a workload ω_i: $\langle rl_i, dl_i, E_i, \mu_i \rangle$, assigned to a core i. Recall that C_i represents a workload's span. An offline scheduler considers the slots $s \in [a,b]$ to assign to ω_i and needs test if these slots will meet E_i and μ_i requirements of a workload ω_i. For simplicity, we consider all server budget assignments on core i in the range of slots $[a,b]$ have non-zero values and that workload's deadline $dl_i > b$, where b denotes the last slot in the range of slots under considerations. Note that, when the slots with their budget assignments under consideration for a workload change, an offline scheduler must retest for schedulability. An offline scheduler will typically require many such test steps $s \in [a,b]$ to ensure slots reserved for a workload do neither over-provision nor under-provision resources.

We now describe hows to test for the schedulability for the dynamic even memory budget assignment. For each slot s, each memory request server's budget $q^m_{i,s}$ relates to a valid memory budget distribution described in Section IV.2.6. The offline scheduler must test if the slots s $\forall s \in [a,b]$ on core i are sufficient to meet the E_i and μ_i requirements of a ω_i workload. Then the offline scheduler determines the minimum span C_i^{min} needed to meet the core-local execution time and memory requirements of ω_i and is given by the Equation (IV.7). If the number of slots under consideration $b-a+1$ is less than C_i^{min}, workload ω_i is unschedulable under slots being considered.

$$C_i^{min} = \left\lceil \frac{E_i}{\hat{q}^{c,1}} + \frac{\mu_i}{\hat{q}^{m,1}} \right\rceil \tag{IV.7}$$

Otherwise, if $b-a+1 \geq C_i^{min}$, the offline scheduler uses the worst-case memory request pattern described in Section IV.3.2. The offline scheduler sorts and renumbers

the slots s $\forall s \in [a,b]$ in descending order according to their memory budget $q^m_{i,s}$. Let $[a'b']$ represent the sorted and renumbered indices of slots $[a,b]$. Then the scheduler determines the number of slots required by workload for its core-level execution time E_i using Equation (IV.8).

$$\kappa = \frac{E_i}{\hat{q}^{c,1}} \tag{IV.8}$$

If κ is not an integer, the offline scheduler computes the amount of memory requests possible in the remaining part of slot $a' + \lceil\kappa\rceil$, i.e.,

$$\rho = \lceil(\lceil\kappa\rceil - \kappa) * q^m_i a' + \lceil\kappa\rceil\rceil$$

Then the offline scheduler sums the memory budgets of the remaining slots from $[a' + \lceil\kappa\rceil, b']$, i.e.,

$$\forall s' \in [a' + \lceil\kappa\rceil, b'] \ , \ \psi = \sum q^m_{i,s'}$$

If the inequality holds $\mu_i \leq \rho + \psi$, then the slots $\forall s \in [a,b]$ will meet the workload's requirements. Note that, the interval of slots $[a,b]$ is a tight assignment if $\rho+\psi-\mu_i \leq \mathcal{Q}^m_n$ and workload's span C_i equals $b-a+1$. Note that, the given slots $[a,b]$ definitely over-provision resources, if $\rho + \psi - \mu_i \geq \mathcal{Q}^m_1$. In such a case, the offline scheduler can tighten the interval of slots and must retest to obtain schedulability guarantees as well as tightness of interval of slots reserved for a workload. Moreover, the sorting step has a computational complexity of $\mathcal{O}\big((b-a+1)\cdot\log(b-a+1)\big)$ that dominates the workload schedulability test's computational complexity, as all actives cores in a slot s cores are assigned equal budgets at the begin of a slot s. Thus, using the valid dynamic even memory budget assignments and the worst-case memory request pattern allows core-level scheduling and testing for workload schedulability in a single step without circular dependency.

IV.4 Avionics Case Study: Scheduling Certified SDAW Application

This chapter applies our proposed method with resource servers and dynamic even memory bandwidth assignment to an industrial avionics use-case certified single-core application — helicopter terrain awareness and warning system (HTAWS)— from Airbus. Section IV.4.1 presents HTAWS application's single-core data. As our focus is on multi-core systems, Section IV.4.2 provides application characteristics observed on a multi-core platform. Section IV.4.3 describes a preparatory step required before using our proposed method for scheduling this application on a 2-core multi-core platform. Section IV.4.4 describes a proof-of-concept schedule table generation using a constraint solver using our proposed method.

For future mobility, Airbus is pursuing autonomous electric aircraft targeting urban landscape to ease traffic. In addition to the avionics applications used in current aircraft, DAL-A sense-and-avoid applications for autonomous flying are needed, which still need to be developed. Nevertheless, as a starting point, HTAWS application represents a non-autonomous version of a sense-and-avoid application. This application is an optional feature of Airbus' helicopters.

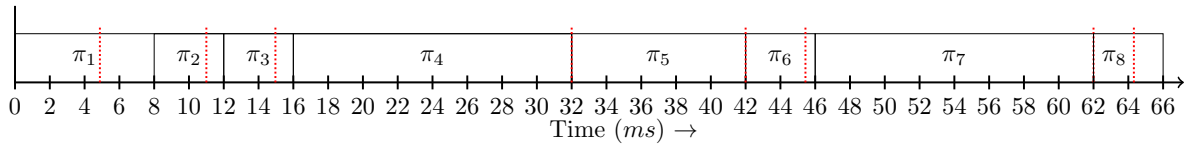

Figure IV.4: *DAL-C certified ARINC 653 single-core schedule of the HTAWS application from Airbus. Dotted red lines indicate the maximum observed execution time of each of the partition, without contention, as shown in column 3 of Table IV.4.*

HTAWS enables safer flying by assisting the pilot especially in degraded visual environments like flying at night, poor visibility conditions due to fog, rough terrain, and low-altitudes, useful for search and rescue mission by air ambulances and coastguards. Furthermore, it contains a map of power transmission lines and other obstacles which are hard to detect even in good weather conditions. This application is a DAL-C certified safety-critical avionics application executing on a single-core processor running VxWorks 653 RTOS from Wind River [Riv15]. The HTAWS application's and the VxWorks RTOS's source-code is closed-source.

IV.4.1 Single-core Data

HTAWS includes eight partitions with a major time frame H (MAF) of 66ms. In this chapter, we consider each partition represents a workload and use π_i to represent a partition. Figure IV.4 shows its ARINC 653 partition-level schedule. At runtime, the inter-partition scheduler schedules the partition based on a static schedule. Table IV.3 shows the partition-level timing constraints. I/O operations are not part of the study here.

Table IV.3: *HTAWS application: Partition-level timing data*

Partition π_i	abs. release time rl_i (ms)	abs. deadline dl_i (ms)	Duration (ms)
π_1	0	8	8
π_2	8	12	4
π_3	12	16	4
π_4	16	32	16
π_5	32	42	10
π_6	42	46	4
π_7	46	62	16
π_8	62	66	4

IV.4.2 Measured Data on COTS Multi-core

Cassidian, part of the Airbus group, performed measurements on the real HTAWS application in a test setup using a dual-core COTS P5020 platform running VxWorks with only 1 active core. A core is considered active if, in the time interval under consideration, it is allowed to issue memory accesses. Table IV.4 shows the maximum execution time and the maximum number of memory accesses for each partition observed in 1000 runs. Partitions π_4, π_5, and π_7, are memory-intensive.

Table IV.4: *HTAWS application: Measurements on the dual-core COTS P5020 platform with only 1 active core*

Partition	Max. obs. num. memory accesses	Max. obs. ET (ms)
π_1	6618	4.88
π_2	2764	3.12
π_3	7381	2.97
π_4	477886	16.00
π_5	262962	10.00
π_6	4275	3.44
π_7	477886	16.00
π_8	7020	2.32

Airbus Innovations, using its proprietary OS for research, provided the maximum observed memory latencies for a different number of active cores. Table III.2 lists these values for two COTS multi-core platforms: the dual-core P5020 platform and the eight-core P4080 platform. Our proposed resource servers were integrated in the proprietary OS for research by Airbus that runs on the P4080 platform. For each CPU server, a multicore programmable interrupt controller (MPIC) timer was used. It is a hardware timer with auto-reload feature. It ensures slot-level synchronization among all cores and provides a unique interrupt instance [Fre13a] to each core on each interrupt. For the memory server, a core-level hardware performance counter was used that was configured to count the requests to the on-chip network [Fre13a] from each core. Observed overheads due to the CPU server and memory server equal 10μs, that is 1% of the considered slot duration of 1ms [Now14]. The platform clock frequency was configured to 600 MHz and each core's clock frequency to 1200 MHz. The hardware timers used: MPIC timer, and Timebase timer, are configured to 37.5 MHz. The Timebase timer was used to generate timestamps. L1 and L2 caches were enabled, while L3 caches were disabled. The setup used two memory controllers.

Rationale for Dynamic Memory Bandwidth Isolation: As some partitions in our reference HTAWS application require upto 73% of memory bandwidth (column 4 in Table IV.5), it results in at least 1 active core under static memory bandwidth isolation mechanisms, defeating the purpose of using COTS multi-core for the HTAWS application. Adding more cores under static memory bandwidth isolation will not solve

the problem as it will not increase the memory bandwidth. However, except partitions $\{\pi_4, \pi_5, \pi_7\}$, the memory bandwidth demand of the remaining partitions is under 16%. Using dynamic memory bandwidth isolation, allows using additional cores when the remaining partitions are active.

IV.4.3 Preparation of the HTAWS Application Data

The numbers shown in Table IV.4 are the sole input to the study presented here as actual application internals are confidential. They give maximum observed execution times and memory requests for each partition but do not mention the distribution of read/write requests as well as how many requests end in L3 cache. Our partition model needs the core-local execution times, which are not given. Thus, the observed data had to be "reverse engineered" for this study, making safe assumptions about the number of memory requests and core-local execution time of each partition. We use the memory latencies shown in Table III.2 to obtain core-local execution times shown in column 2 of Table IV.5. Partition π_5 requires on average, 73% of memory bandwidth using δ_1 memory latency. Column 3 shows the minimum constant memory bandwidth that needs to be reserved using existing interference-sensitive WCET computation [NPB+14, NP13], to meet each partition's E_i and μ_i requirements with only 1 active core. The memory latencies are the largest observed latencies for the different number of active cores under different combinations of read and write memory requests. The execution time computation further assumes that the core stalls on each memory request. As a consequence, the processed data is pessimistic, calling for more detailed information available from the original application. The processed data is, however, safe for the experiments in this study.

Table IV.5: *"Reverse-engineered" core-local execution time and minimum constant memory bandwidth that needs to be using reserved using existing interference-sensitive WCET computation [NPB+14, NP13] for the HTAWS application*

Partition	Computed core-local ET E_i (ms)	Min. mem. b/w (in %) reqd. throughout partition considering δ_1 latency
π_1	4.72	4.88
π_2	3.05	7.03
π_3	2.79	14.74
π_4	4.45	100.00
π_5	3.64	100.00
π_6	3.34	15.66
π_7	4.45	100.00
π_8	2.15	9.17

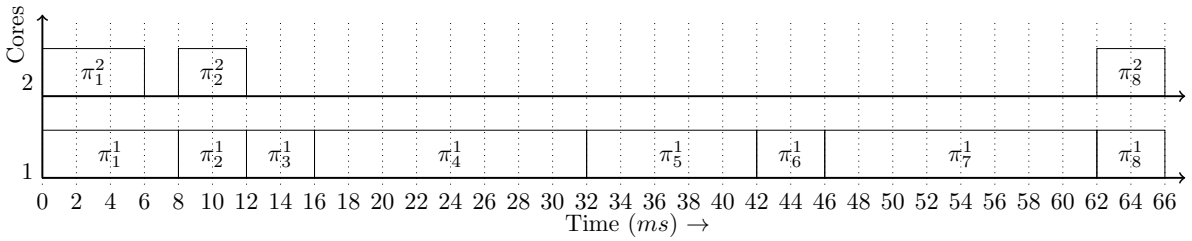

Figure IV.5: *Partition schedule generated through Gecode constraint solver using our proposed method for the HTAWS application with some replicated partitions, such that existing HTAWS schedule is preserved.*

IV.4.4 Time-Triggered Schedule

As a proof-of-concept, we modeled the partition allocation and scheduling problem for the Gecode [Gec] constraint programming (CP) solver, to find a valid schedule table for each of the two cores. The specified constraints schedule the HTAWS application on 1 core and additional partitions on the other core under dynamic memory bandwidth, while preserving the single-core HTAWS schedule. As no real avionics application with more than one active core at the same time exists today, we construct a scenario by replicating some partitions such that multiple cores are active at the same time. We obtained schedule tables using the Gecode constraint solver with a model specified using our proposed method. Figure IV.5 shows one such schedule found by the Gecode solver in which partitions π_1, π_2, and π_8 are replicated on the second core. Appendix A lists the formulation of the key constraint that checks if the slots under consideration meet a partition's π_i requirements of E_i and μ_i as described in the Section IV.3.3 using a number of bandwidth levels with equal memory budget distribution. It demonstrates the feasibility of scheduling and computing execution time offline, in the same step, under dynamic memory bandwidth, overcoming the inter-dependency challenge. At runtime, our proposed server-based runtime mechanism, described in Section IV.2, will execute the offline-computed scheduling decisions including assigning of the server budgets on each core in each slot.

V

Span Analysis under Dynamic Arbitrary Budgets

This chapter considers computing span of a workload that undergoes through slots of arbitrary memory budget assignment. Section V.1 presents the system model considered. Section V.2 presents the concept of memory stall that is basic to this chapter. Then Section V.3 uses the memory stall concept to present the stall curves. We present the span under static budget analysis in Section V.4 to ease exposition for the later presented span analysis under dynamic budget in Section V.5. Section V.6 presents the evaluation for an Integrated Modular Avionics (IMA) case study that compares a dynamic budget assignment heuristic that uses our approach to compute span to static budget assignment policies.

V.1 System Model

Further, we consider the time model, CPU scheduling model and workload model as described Chapter III. We consider the multi-core model with *constant memory request time model* as presented in Chapter III.1.3. Recall that this model considers L_{max} is the maximum time taken to perform a memory request in isolation and that it is also the maximum delay due to contention from a single memory request from an another core. In this chapter, our focus is on the computing a workload's span when the memory bandwidth allocated to a workload changes over time. So, we focus on memory server's behavior for the analysis and ignore the CPU server from the discussion. Further, we consider the maximum number of memory requests Q possible in a server period is $\frac{T^m}{L_{max}}$. For simplicity, we focus on a generic core i and refer to a generic workload on core i as ω_i. In this chapter, we assume that a workload's parameter E_i is expressed in units of L_{max}.

V.1.1 Memory Schedule

We assume that the memory schedule is known. Instead of reasoning on a per-slot basis, on sequence of memory budget assignment intervals, where each interval represents contiguous slots with same memory budget assignment across cores, as depicted in Figure IV.1(c). A a memory schedule $\mathcal{S} = \{B^1, \ldots, B^N\}$ is a time-ordered sequence of N *memory budget assignment intervals* B^j. Each B^j is of the form $B^j = (\mathcal{Q}^j, L^j)$, where $\mathcal{Q}^j = \{q_1^j, \ldots, q_n^j\}$ is the budget-to-cores assignment used in interval j, and L^j is the length in slots. For instance, the memory schedule in Figure IV.1(c) is $\mathcal{S} = \{(\mathcal{Q}, 2), (\mathcal{Q}', 1), (\mathcal{Q}, 1), (\mathcal{Q}'', 2), (\mathcal{Q}, 4)\}$.

V.1.2 Workload Span

The workload, however, may span throughout a number of different memory scheduling intervals $B^1, \ldots, B^N$. While the total span is indicated with C_i, the span of the workload over each interval B^j is indicated with C_i^j. It must hold that $\sum_{j=1}^{N} C_i^j = C_i$.

Since the intervals have fixed length and are ordered, C_i^1 must be equal to either L^1 or C_i, whichever is shorter. Then assuming $C_i > L^1$, the span C_i^2 over interval B^2 must be equal to the minimum of L^2 and $C_i - L^1$. In general, noticing that $\sum_{k=1}^{j-1} L^k$ is the cumulative length of intervals preceding B^j, the execution over interval B_j must thus be equal to:

$$C_i^j = \max\Big(0, \min\big(L^j, C_i - \sum_{k=1}^{j-1} L^k\big)\Big). \tag{V.1}$$

V.2 Memory Stall

A fundamental concept is the notion of *memory stall*. In general, a memory request originating from the core i under analysis can be "stalled" for two reasons. The first reason is that the hardware memory arbiter has prioritized one or more other cores over i for access to the memory subsystem (memory interference). The second reason is that the core under analysis has exhausted its budget and is stalled until the beginning of the next memory server instance.

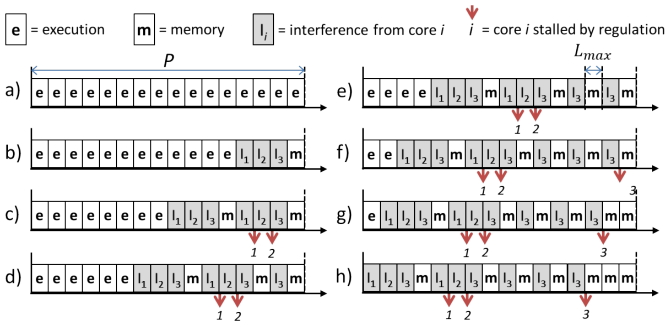

Figure V.1: *M/C configurations for core $i = 4$ under analysis in a 4-core system with $\mathcal{Q} = \{2, 2, 5, 7\}$.*

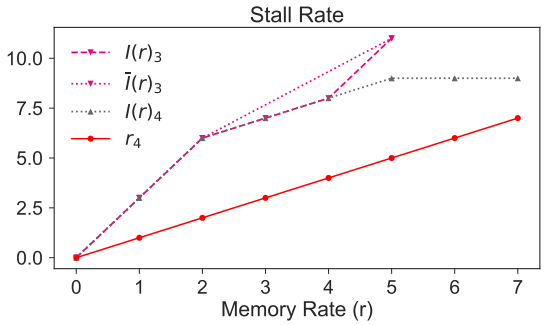

Figure V.2: *Plot of stall curves for a 4-core system with $\mathcal{Q} = \{2, 2, 5, 7\}$.*

We make no assumption on the behavior of tasks in cores other than core i. It follows that the maximum stall that can be suffered by a memory request on core i depends only on the number of memory requests performed by i in the same server period. This is exemplified in Figure V.1, where $i = 4$ and $\mathcal{Q} = \{2, 2, 5, 7\}$. The budget for Core 4 is $q_4 = 7$. It follows that there are 8 possible worst-case scenarios, denoted as (a)-(h) in the figure. In general, there are always $q_i + 1$ possible cases. In the figure, pure execution is represented with "e" and is considered in units of length L_{max}, as mentioned in Section V.1.

The pattern in Figure V.1(a) depicts the worst-case memory interference from other cores (cores 1 to 3) provided that core 4 performs zero memory accesses within the server period. A more interesting case is Figure V.1(e). Here, Core 4 performs 4 memory accesses. For the first two memory accesses, since three cores (1, 2 and 3) can cause stall, 3 units of stall are accumulated per memory access. If we depict the stall as a curve, then the "slope" of the stall introduced by the first two accesses is 3. After the first two accesses, cores 1 and 2 are temporarily stopped due to regulation – they have exhausted their respective budgets. Core 3, however, can still cause stall on requests from Core 4 under analysis. Hence, the stall slope for the 3^{rd} and 4^{th} requests is 1.

A more efficient way to visualize the possible stall scenarios is by plotting the per-period memory requests and resulting stall. The q_i+1 possible cases represent a discrete domain. A corresponding continuous curve for the memory stall can be derived by

"connecting" these discrete points. Call $r_i \in \mathbb{R}_{\geq 0}$ the number of memory requests per server period being performed. We introduce the notion of *memory rate*.

Definition 2 (Memory rate)**.** *We define as* memory rate *the number of memory requests performed per server period. Memory rates are indicated throughout the paper with the symbol* r_i.

A memory rate is often used to indicate the rate at which a total number of memory requests μ_i is performed over the span of the considered workload C_i. Hence, $r_i = \frac{\mu_i}{C_i}$. We can also indicate the number of memory requests performed during a specific interval B^j as μ_i^j. It must hold that $\sum_{j=1}^{N} \mu_i^j = \mu_i$. Analogously, we indicate the memory rate over each interval as r_i^j. By definition, we have $r_i^j = \frac{\mu_i^j}{C_i^j}$.

V.3 Memory-stall Curves

The memory-stall curve for a core i represents the cumulative maximum interference-induced stall for a given memory rate r and is denoted as $I(r)_i$. Consider same setup used for Figure V.1. The memory-stall curves for cores 3 and 4 are provided in Figure V.2. Considering Core 4, We have already discussed how the first two requests introduce stall at a "slope" of 3. This is reflected in the $I(r)_4$ curve, since the curve has slope 3 when $r \in]0, 2[$.

For clarity, let us construct the memory-stall curve for core 3. The y-axis represents the cumulative maximum stall $I(r)_3$ that can be experienced by workload on Core 3 with a memory rate r (x-axis). Workload on Core 3 can perform of 0, 1, 2, 3, 4 or 5 memory requests in a server period. The first step is to compute the maximum stall in each of these cases. If the workload does not perform any memory request ($r = 0$) in a server period, then it will experience no stall, i.e. $I(0)_3 = 0$. When $r = 1$, then it can be stalled by a maximum of 1 memory request by each of the $m - 1 = 3$ cores resulting in $I(1)_3 = 3$. Similarly, for all values of r until $r = \min_i(q_1, \ldots, q_m)$, the maximum stall rate $I(r)_3 = (m-1) \cdot r$, hence $I(2)_3 = 3 \cdot 2 = 6$. When Core 3 performs an additional memory request, i.e. $r = 3$, it can only be stalled by Core 4, since cores 1 and 2 have been regulated after their second memory access. Thus, the cumulative stall rate is $I(3)_3 = I(2)_3 + 1 \cdot 1 = 7$. Similarly, for $r = 4$, $I(4)_3 = I(3)_3 + 1 \cdot 1 = 8$. Finally, for $r = 5 = q_3$, Core 3 is regulated. Here the maximum cumulative stall is $I(5)_3 = Q - q_3 = 16 - 5 = 11$. The memory-stall curve $I(r)_3$ is obtained by connecting the discrete values of $I(k)_3, k \in \{0, \ldots, 5\}$ calculated so far.

Generalizing the example provided above, for any fixed budget $\mathcal{Q}$ we can define the stall curve $I(r)_i$ as follows:

$$I(r)_i = \begin{cases} \sum_{k \neq i} \min(r, q_k) & \text{if } r < q_i \\ Q - q_i & \text{if } r = q_i \end{cases} \tag{V.2}$$

Since the budget assignment $\mathcal{Q}^j$ changes every scheduling interval B^j, a different $I(r)_i^j$ curve needs to be considered on each interval.

If the resulting curve is concave, then the memory-stall curve is already final. This is the case for $I(r)_4$ in Figure V.2. Conversely, a refinement step is necessary to produce the final curve. Specifically, we take the upper-envelope of each of the convex segments to obtain a concave curve. The result of this step is depicted as $\bar{I}(r)_3$ in Figure V.2.

Definition 3 (Stall rate)**.** *We define as* stall rate *the amount of memory stall $\bar{I}(r)_i$ suffered per server period with a memory rate r. When considering multiple intervals, $\bar{I}(r)_i^j$ is the stall rate for core i on interval B^j.*

If the span over B^j is C_i^j and μ_i^j requests are performed in the interval, we can compute the worst-case total stall S_i^j over B^j as:

$$S_i^j := \bar{I}(r_i^j)_i \cdot C_i^j = \bar{I}\Big(\frac{\mu_i^j}{C_i^j}\Big)_i \cdot C_i^j. \tag{V.3}$$

It follows that the total stall is $S_i = \sum_{j=1}^{N} S_i^j$. If the maximum memory stall that can be suffered by the workload under analysis can be derived, then the worst-case amount of time (in multiples of L_{max}) required to complete the considered workload is $C_i \cdot Q = \beta_i + S_i$. The rest of the paper is concerned with the calculation of the maximum total stall, and hence span, over a generic memory schedule $\mathcal{S} = \{B^1, \ldots, B^N\}$.

For a fixed budget $\mathcal{Q}$, a given memory rate r_i and span C_i, Lemma 1 guarantees that computing S_i according to Equation V.3 always results in an upper-bound on the maximum possible memory stall.

Lemma 1. *$S_i = \bar{I}(\mu_i/C_i)_i \cdot C_i$ is an upper bound to the cumulative stall suffered by a workload on core i that performs μ_i memory accesses over C_i server periods, with $\mu_i \leq C_i \cdot q_i$.*

Proof. In each of the C_i server periods, a number of memory accesses between 0 and q_i could have been performed; hence, note that we cannot have $\mu_i > C_i \cdot q_i$. Let us indicate with a_k the number of periods in which k memory accesses were performed. It must hold that $\sum_{k=0}^{q_i} a_k = C_i$. We can then write:

$$\mu_i = a_0 \cdot 0 + a_1 \cdot 1 + \ldots + a_{q_i} \cdot q_i. \tag{V.4}$$

The cumulative stall suffered over C_i can be computed as:

$$a_0 \cdot I(0)_i + a_1 \cdot I(1)_i + \ldots + a_{q_i} \cdot I(q_i) = \sum_{k=0}^{q_i} I(k)_i \cdot a_k. \tag{V.5}$$

Consider now computing the stall rate as $\bar{I}(\mu_i/C_i)_i$. From Equation V.5, by $I(r) \leq \bar{I}(r)$ we have:

$$\sum_{k=0}^{q_i} I(k)_i \cdot a_k \leq \sum_{k=0}^{q_i} \bar{I}(k)_i \cdot a_k \tag{V.6}$$

Next recall that by definition of concavity for a generic function $f(x)$, it must hold that:

$$\lambda_k \in \mathbb{R} \text{ s.t. } \sum_k \lambda_k = 1 \implies f\Big(\sum_k x_k \lambda_k\Big) \geq \sum_k f(x_k)\lambda_k \tag{V.7}$$

Note that:

$$\frac{\sum_{k=0}^{q_i} a_k}{C_i} = \sum_{k=0}^{q_i} \frac{a_k}{\sum_{k=0}^{q_i} a_k} = 1. \tag{V.8}$$

Hence, we can write:

$$\frac{\sum_{k=0}^{q_i} \bar{I}(k)_i \cdot a_k}{C_i} = \sum_{k=0}^{q_i} \frac{\bar{I}(k)_i \cdot a_k}{\sum_{k=0}^{q_i} a_k} \leq \bar{I}\left(\sum_{k=0}^{q_i} \frac{k \cdot a_k}{\sum_{k=0}^{q_i} a_k} \right)_i = \bar{I}\left(\frac{\mu_i}{C_i}\right)_i. \tag{V.9}$$

This implies that $\bar{I}(\mu_i/C_i)_i \cdot C_i$ is an upper bound to the cumulative stall $\sum_{k=0}^{q_i} I(k)_i \cdot a_k$ suffered by the workload for *any* pattern of memory accesses over C_i periods, concluding the proof. □

V.4 Span under Static Memory Budget

In this section, we present a fixed-point iterative algorithm to compute the worst-case length of the workload on a core under analysis i under static memory budget $\mathcal{Q}$. This is useful to understand the basic mechanisms to compute the span over a generic single memory scheduling interval.

In each iteration, the algorithm recomputes the maximum stall and thereby, the workload span, based on the workload span from the previous iteration (except for the base iteration) and the corresponding memory schedule. The key intuition behind iterative recomputation is that the increase in workload span in an iteration is likely to increase the maximum stall in the consecutive iteration due to a different worst-case distribution of memory requests across *(a)* different per memory-stall curves and/or *(b)* different memory scheduling intervals.

In the rest of the paper, we will always focus on the generic core under analysis. As such, we will drop the index i from all the notation introduced so far, unless required to resolve an ambiguity. Since we will be introducing a series of iterations of the algorithm, we subscript the iteration number (e.g., (k)) in the notation introduced so far.

Iterative Algorithm: the span W over a static memory budget can be computed using Equation V.10.

$$\begin{aligned} C_{(0)} &\leftarrow \lceil \beta/Q \rceil, \\ C_{(k)} &\leftarrow \left\lceil \left(\beta + \bar{I}(\min(\mu/C_{(k-1)}), q) \cdot C_{(k-1)}\right)/Q \right\rceil, \end{aligned} \tag{V.10}$$

where the iteration continues until convergence with $C_{(k)} = C_{(k-1)}$, or until $C_{(k)} \cdot Q \cdot L_{max} > D$. In the latter case, the workload in not schedulable. Since $\bar{I}(r)$ is only defined for $r \in [0, q]$, the term $\bar{I}(\min(\mu/C_{(k-1)}), q)$ ensures that the function is never evaluated on a value outside its domain.

Theorem V.1. *The iteration in Equation V.10 terminates in a finite number of steps by either obtaining a value $C_{(k)} \cdot Q \cdot L_{max} > D$, or by converging, in which case $C_{(k)}$ is an upper bound on the span of the workload on the core under analysis.*

Proof Sketch. Notice that we omit the proof for Theorem V.1 here, as it is a corollary of the more general Theorem V.2. As such, the proof is provided in Appendix B. □

For ease of explanation, Section V.4 illustrates how to apply the algorithm in a specific instance. Subsequently, Section V.5 presents the generic algorithm.

Example of Span over Static Budget: consider the static budget $\mathcal{Q} = \{2, 2, 5, 7\}$. Let us now compute the span W of the workload with $E = 40$ and $\mu = 35$ (i.e. $\beta = 75$) executing on Core 3. For simplicity, we ignore the workload's deadline dl and focus only on its length. Since workload on Core 3 is being analyzed, we consider the stall curve $\bar{I}(r)_3$ in Figure V.2, with $Q = 16$ and $q = 5$.

The first step in the iterative Equation V.10 is $C_{(0)} = \lceil \beta/Q \rceil = \lceil 75/16 \rceil = 5$. We then have:

$$\begin{aligned} C_{(1)} &= \left\lceil (75 + \bar{I}(\min(35/5, 5) \cdot 5)/16 \right\rceil \\ &= \left\lceil (75 + \bar{I}(5) \cdot 5)/16 \right\rceil = \lceil (75 + 11 \cdot 5)/16 \rceil = 9 \end{aligned}$$

Since $C_0 \neq C_k$, an additional iteration needs to be performed. We have:

$$\begin{aligned} C_{(2)} &= \left\lceil (75 + \bar{I}(\min(35/9, 5) \cdot 9)/16 \right\rceil \\ &= \left\lceil (75 + \bar{I}(3.88) \cdot 9)/16 \right\rceil = \lceil (75 + 9 \cdot 9)/16 \rceil = 10 \end{aligned}$$

No convergence has been reached yet, so one more iteration is performed:

$$\begin{aligned} C_{(3)} &= \left\lceil (75 + \bar{I}(\min(35/10, 5) \cdot 10)/16 \right\rceil \\ &= \left\lceil (75 + \bar{I}(3.5) \cdot 10)/16 \right\rceil = \lceil (75 + 8.5 \cdot 10)/16 \rceil = 10 \end{aligned}$$

Since $C_{(3)} = C_{(2)}$, convergence has been reached and the worst-case length (in multiples of L_{max}) for the workload under analysis can be computed as $C_{(3)} \cdot Q = 160$.

V.5 Span under Dynamic Memory Budget

In this section, we extend our analysis to the case of dynamic bandwidth assignment. In this case, the workload could span across one or more memory scheduling intervals $B^1, \ldots, B^N$. Recall from Section V.1.1 that each interval $B^j = (\mathcal{Q}^j, L^j)$ is characterized by a budget-to-cores assignment $\mathcal{Q}^j = \{q_1^j, \ldots, q_m^j\}$ and a length L^j expressed in number of L_{max} units. For the core under analysis, it is always possible to compute $\bar{I}(r)^j$ using Equation V.2, i.e. the stall curve resulting from $\mathcal{Q}^j$.

Similarly to Equation V.10, we follow an iterative approach. Let once again $C_{(k)}$ be the span of the considered workload at iteration k. It is always possible to determine the span of the workload $C_{(k)}^j$ in each of the intervals B^j using Equation V.1.

Example: To better understand this setup, consider the situation depicted in Figure V.3. In this case, the memory schedule is composed of 3 intervals B^1, B^2, B^3. The intervals have length $L^1 = 5, L^2 = 3$, and $L^3 = 7$, respectively. Similarly we have three budget-to-cores assignments $\mathcal{Q}^1, \mathcal{Q}^2, \mathcal{Q}^3$ with the budgets values specified in the

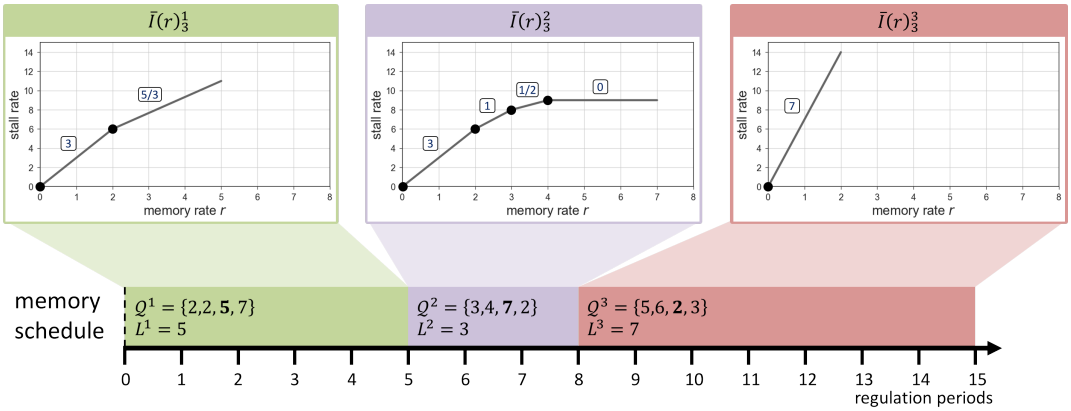

Figure V.3: *Example of memory schedule of length* 15 *regulation periods composed of 3 intervals* B^1, B^2, B^3 *of length 5, 3, and 7, respectively. Considering Core 3, the curves* $\bar{I}(r)^1_3, \bar{I}(r)^2_3, \bar{I}(r)^3_3$ *are reported above each interval. For each* $\bar{I}(r)^j_3$ *curve, segment start points are highlights with a* • *and slopes are annotated above the segments.*

figure. Assume that we want to analyze the behavior of workload on Core $i = 3$: the figure reports the $\bar{I}(r)^1_3, \bar{I}(r)^2_3, \bar{I}(r)^3_3$ curves that need to be considered in each of the intervals. Let us assume that at a certain iteration k, we have $C_{(k)} = 9$ and apply Equation V.1. We obtain $C^1_{(k)} = \max\big(0, \min(5, 9)\big) = 5, C^2_{(k)} = \max\big(0, \min(3, 9-5)\big) = 3$, and $C^3_{(k)} = \max\big(0, \min(7, 9-8)\big) = 1$. Note that it always hold $\sum_{j=1}^{N} C^j_{(k)} = C_{(k)}$.

Assume that the span $C_{(k)}$ and hence the various $C^j_{(k)}$ terms at a given iteration k are known. Then, the challenge is to determine how to distribute the total μ memory requests among the $B^1, \ldots, B^N$ intervals in a way that maximizes the overall stall. A *distribution* of memory requests simply means that we derive the quantities $\mu^j_{(k)}$ for each B^j interval. Obviously, it must hold that $\sum_{j=1}^{N} \mu^j_{(k)} = \mu$. But among all the possible, valid distributions, we are interested in the one[1] that maximizes the overall stall, i.e. $S_{(k)} = \sum_{j=1}^{N} S^j_{(k)}$, where each of the $S^j_{(k)}$ terms is defined as in Equation V.3.

To simplify exposition, we first introduce an optimization problem in Algorithm V.1 that computes the number $\mu^j_{(k)}$ of memory requests assigned to each interval B^j at the k-th iteration to maximize the overall stall $S_{(k)}$. Then, in Section V.5.2 we show how to efficiently solve the optimization problem. Note that once $\mu^j_{(k)}$ has been determined, based on Lemma 1 the stall in interval B^j can be upper-bounded as $S^j_{(k)} = \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}$. Hence, the cumulative stall at Line 8 of the algorithm is computed as $S_{(k)} = \sum_{j=1}^{N} S^j_{(k)}$. Due to regulation, in Line 12 we assign at most q^j memory requests in any server instance inside interval B^j. This is equivalent to the following constraint: the number of requests $\mu^j_{(k)}$ performed in B^j is upper bounded by $C^j_{(k)} \cdot q^j$. Finally, since we know that the workload comprises at most μ requests, it must hold $\sum_{j=1}^{N} \mu^j_{(k)} \leq \mu$ at Line 13.

[1]This distribution may not be unique.

Listing V.1: *Stall maximization over multiple intervals*

```
1  Input: B^1, ..., B^N                                   /* sequence of intervals */
2  Input: C^1_(k), ..., C^N_(k)                            /* span in each interval */
3  Input: μ                                                /* total memory requests */
4
5  Output: μ^1_(k), ..., μ^N_(k)                           /* memory requests in each interval */
6
7  Maximize:                                               /* max cumulative stall */
8      S_(k) = Σ_{j=1}^N S^j_(k) = Σ_{j=1}^N Ī(μ^j_(k)/C^j_(k))^j · C^j_(k)
9
10 Subject to:
11     μ^j_(k) ∈ ℕ                                         /* number of requests is natural */
12     μ^j_(k) ≤ C^j_(k) · q^j                             /* max q^j requests per server period */
13     Σ_{j=1}^N μ^j_(k) ≤ μ                               /* total requests constraint */
```

Based on Algorithm V.1, $C^j_{(k)}$ is then computed according to the following iteration:

$$C_{(0)} \leftarrow \lceil \beta/Q \rceil,$$
$$C_{(k)} \leftarrow \left\lceil \left(\beta + \sum_{j=1}^{N} \bar{I}(\mu^j_{(k-1)}/C^j_{(k-1)})^j \cdot C^j_{(k-1)}\right)/Q \right\rceil. \tag{V.11}$$

Note that at each iteration $k > 0$, the values $C^j_{(k-1)}$ are computed using Equation V.1 from $C_{(k-1)}$, and the values $\mu^j_{(k-1)}$ are computed using Algorithm V.1. As in Section V.4, the iteration continues until convergence or $C_{(k)} \cdot Q \cdot L_{max} > D$.

Example: Suppose we are analyzing the behavior of workload with $E = 15$ and $\mu = 25$ on Core 3 under the memory schedule depicted in Figure V.3. Assume that at a given step k we have $C_{(k)} = 6$. In this case we have $C^1_{(k)} = 5, C^2_{(k)} = 1$ and $C^3_{(k)} = 0$. Invoking Algorithm V.1 returns the memory-to-intervals distribution that maximizes the overall stall, in this case: $\mu^1_{(k)} = 22, \mu^2_{(k)} = 3, \mu^3_{(k)} = 0$. The stall per interval can be computed as $S^j_{(k)} = \bar{I}\left(\mu^j_{(k)}/C^j_{(k)}\right) \cdot C^j_{(k)}$. For this example, we have $S^1_{(k)} = 50, S^2_{(k)} = 8$ and $S^3_{(k)} = 0$. The new value of $C_{(k+1)}$ can then be computed as $C_{(k+1)} = \lceil (35 + 50 + 8)/16 \rceil = 6$. Note that in this case Equation V.11 has reached convergence.

V.5.1 Proof of Correctness

We now formally prove that Equation V.11 computes a valid upper bound for the workload length in number of server periods. We begin with some helper lemmas; Lemma 2 show that the value of $C_{(k)}$ increases monotonically, which is required for the iteration to terminate, while Lemma 3 shows that if the iteration converges, we are able to distribute all μ memory requests among the N memory scheduling intervals.

Lemma 2. *At each iteration step in Equation V.11 it holds:* $C_{(k)} \geq C_{(k-1)} > 0$.

Proof. First note that functions $\bar{I}(r)^j$ are concave and $\bar{I}(0)^j = 0$. For any such function and positive constant μ, one can prove that $\bar{I}(\mu/x)^j \cdot x$ is monotonic non-decreasing in $x > 0$ (a formal proof is reported in Lemma 6 in Appendix B). The proof then proceeds by induction over the index k.

Base Case: Since we assume $\beta > 0$, we have $C_{(0)} > 0$. Furthermore, since by definition all $C^j_{(0)}$ terms are non-negative and functions $\bar{I}(r)^j$ have non-negative ranges, $\sum_{j=1}^{N} \bar{I}(\mu^j_{(0)}/C^j_{(0)})^j \cdot C^j_{(0)}$ is non-negative. By definition of Equation V.11, this implies $C_{(1)} \geq C_{(0)} > 0$.

Inductive Step: Consider $k \geq 2$. Note that in Equation V.11, $C_{(k)}$ is computed based on the value of $C_{(k-1)}$, from which we obtain the values of $C^j_{(k-1)}$ in Equation V.1 and $\mu^j_{(k-1)}$ in Algorithm V.1; similarly, $C_{(k-1)}$ is computed based on $C_{(k-2)}$ and $C^j_{(k-2)}, \mu^j_{(k-2)}$. By induction hypothesis, we have $C_{(k-1)} \geq C_{(k-2)} > 0$; based on Equation V.1, this implies $C^j_{(k-1)} \geq C^j_{(k-2)}$ for all intervals B^j.

Now consider Line 8 of Algorithm V.1: since $\bar{I}(\mu/x)^j \cdot x$ is monotonic for $x > 0$, it must hold:

$$\sum_{j=1}^{N} \bar{I}(\mu^j_{(k-2)}/C^j_{(k-1)})^j \cdot C^j_{(k-1)} \geq \sum_{j=1}^{N} \bar{I}(\mu^j_{(k-2)}/C^j_{(k-2)})^j \cdot C^j_{(k-2)}. \tag{V.12}$$

In other words, when running Algorithm V.1 at iteration k based on the values $C^j_{(k-1)}$, there exists an assignment of variables ($\mu^j_{(k-1)} = \mu^j_{(k-2)}$, i.e., the same assignment as the previous iteration) that results in a value of the objective function that is greater than or equal to the one at iteration $k-1$. Furthermore, the assignment $\mu^j_{(k-1)} = \mu^j_{(k-2)}$ is feasible, in the sense that it satisfies the constraints at Lines 11-13 of the algorithm: note that $\mu^j_{(k-2)} \leq C^j_{(k-2)} \cdot q^j$ implies $\mu^j_{(k-1)} \leq C^j_{(k-1)} \cdot q^j$ since $C^j_{(k-1)} \geq C^j_{(k-2)}$. Hence, given that the optimization problem is maximizing the objective function, it is guaranteed to find an assignment for variables $\mu^j_{(k-1)}$ such that:

$$\sum_{j=1}^{N} \bar{I}(\mu^j_{(k-1)}/C^j_{(k-1)})^j \cdot C^j_{(k-1)} \geq \sum_{j=1}^{N} \bar{I}(\mu^j_{(k-2)}/C^j_{(k-2)})^j \cdot C^j_{(k-2)}. \tag{V.13}$$

In turn by definition of Equation V.11 this implies $C_{(k)} \geq C_{(k-1)}$, concluding the induction step. □

Lemma 3. *If the iteration in Equation V.11 converges to a value $C_{(k)}$, then there exists a feasible assignment to variables $\mu^j_{(k)}$ that maximizes the objective function at Line 8 of Algorithm V.1 and for which $\sum_{j=1}^{N} \mu^j_{(k)} = \mu$.*

Proof. Note that for a feasible assignment it cannot hold $\sum_{j=1}^{N} \mu^j_{(k)} > \mu$ due to the constraint at Line 13. Hence by contradiction, assume that for all assignments that maximize the objective function it holds: $\sum_{j=1}^{N} \mu^j_{(k)} < \mu$.

By definition, all $C^j_{(k)}$ terms are non negative. Furthermore, functions $\bar{I}(r)^j$ have non-negative ranges. Hence, increasing the value of a variable $\mu^j_{(k)}$ cannot cause the objective function to decrease. Therefore, $\sum_{j=1}^{N} \mu^j_{(k)} < \mu$ must hold even when each variable $\mu^j_{(k)}$ is assigned its maximum value, which is $\mu^j_{(k)} = C^j_{(k)} \cdot q^j$ based on the constraint at Line 12. We thus obtain: $\sum_{j=1}^{N} C^j_{(k)} \cdot q^j < \mu$. Furthermore, note that we have $\bar{I}(\mu^j_{(k)}/C^j_{(k)})^j = \bar{I}(q^j)^j = Q - q^j$.

Finally, given $E > 0$ and based on Equation V.11 at convergence, we derive:

$$\begin{aligned}
C_{(k)} &= \left\lceil \left(\beta + \sum_{j=1}^{N} \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}\right)/Q \right\rceil \\
&= \left\lceil \left(E + \mu - \sum_{j=1}^{N} q^j \cdot C^j_{(k)} + \sum_{j=1}^{N} Q \cdot C^j_{(k)}\right)/Q \right\rceil \\
&> \left\lceil \left(E + \sum_{j=1}^{N} (Q \cdot C^j_{(k)})\right)/Q \right\rceil \\
&= \lceil E/Q + C_{(k)} \rceil \\
&\geq 1 + C_{(k)}, \qquad \text{(V.14)}
\end{aligned}$$

which is a contradiction. □

Theorem V.2. *The iteration in Equation V.11 terminates in a finite number of steps by either obtaining a value $C_{(k)} \cdot Q \cdot L_{max} > D$ or converging, in which case $C_{(k)}$ is an upper bound to worst-case span of the workload on the core under analysis.*

Proof. We first show that the algorithm terminates. By Lemma 2, $C_{(k)} \geq C_{(k-1)}$. Since $C_{(k)}$ is a natural number, it follows that the algorithm must either converge or terminate with a value of $C_{(k)}$ greater than the deadline in a finite number of steps.

Hence, assume that the algorithm converges to $C_{(k)}$. Based on Lemma 3, we can find an assignment to variables $\mu^j_{(k)}$ that maximizes the stall in the objective function of Algorithm V.1 and such that $\sum_{j=1}^{N} \mu^j_{(k)} = \mu$. Hence, the assignment is valid, in the sense that the workload is able to perform its worst-case number of memory requests. Furthermore, due to Line 12, it holds $\mu^j_{(k)} \leq C^j_{(k)} \cdot q^j$ for all intervals; hence, by Lemma 1 and for each B^j, $S^j_{(k)} = \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}$ is an upper bound to the stall when performing $\mu^j_{(k)}$ memory accesses in $C^j_{(k)}$ regulation periods. Now given that Algorithm V.1

maximizes the objective function at Line 8 over all possible assignments to variables $\mu^j_{(k)}$, it follows that $\sum_{j=1}^{N} S^j_{(k)} = S_{(k)}$ is an upper bound to the cumulative stall when performing μ memory accesses over $\sum_{j=1}^{N} C^j_{(k)} = C_{(k)}$ regulation periods.

Finally, by definition, the worst-case length of the workload can be obtained (in number of units of L_{max}) as the sum of β and the stall suffered by the workload. By convergence to $C_{(k)}$, we have:

$$C_{(k)} \cdot Q = \left\lceil \frac{C_{(k)}}{Q} \right\rceil \cdot Q \geq \beta + \sum_{j=1}^{N} S^j_{(k)} = \beta + S_{(k)}, \tag{V.15}$$

and since $S_{(k)}$ is an upper bound to the stall suffered in $C_{(k)}$ server periods, this implies that $C_{(k)}$ is indeed an upper bound to the total span of the workload. □

V.5.2 Implementing the Stall Algorithm

In this section, we show how to efficiently implement Algorithm V.1. The algorithm is similar to a concave optimization problem, except that variables are integer rather than real.

By construction, each $\bar{I}(r)^j$ function is a concave piece-wise linear, and can be thought as a sequence of *segments* with decreasing slope. For each $\bar{I}(r)^j$ curve, consider the set of integer values of r corresponding to the beginning of a segment. Call each of this values a *start point*, and $\mathcal{E}^j$ the set of all the start points in $\bar{I}(r)^j$. Start points are highlighted with a solid dot (•) in Figure V.3. Considering Core 3, for the example in the figure we have: $\mathcal{E}^1 = \{0, 2\}, \mathcal{E}^2 = \{0, 2, 3, 4\}$, and $\mathcal{E}^3 = \{0\}$.

Using this formulation, we introduce two helper functions defined on $\mathcal{E}^j$ for $\bar{I}(r)^j$. First, the $\text{next}^j(r)$ function returns the next start point strictly greater r:

$$\text{next}^j(r) := \min_p \{p \in \mathcal{E}^j \mid p > r\}. \tag{V.16}$$

Second, the function $\text{slope}^j(r)$ simply returns the slope of the segment at r. If r is a start point, the function returns the slope of the starting segment. Formally:

$$\text{slope}^j(r) := \Big(\bar{I}\big(\text{next}^j(r)\big)^j - \bar{I}(r)^j\Big) / \big(\text{next}^j(r) - r\big) \tag{V.17}$$

All the slopes are annotated in Figure V.3 right above the corresponding segment.

Algorithm V.2 first initializes all variables $\mu^j_{(k)}$ to zero. Then, the algorithm iterates over Lines 9-19 until either (1) $\sum_{j=1}^{N} \mu^j_{(k)} = \mu$ holds, meaning that all μ memory requests have been distributed among the N memory scheduling interval; or (2) $\mu^j_{(k)} = C^j_{(k)} \cdot q^j$ for all intervals, meaning that we cannot assign any more memory requests due to the regulation constraints. When the condition $\mu^j_{(k)} = C^j_{(k)} \cdot q^j$ holds for some interval B^j, we say that B^j is *saturated*. The set of all the unsaturated intervals is computed at Line 11, and their respective memory rates r^j given the current assignment $\mu^j_{(k)}$ is computed at Line 13.

Listing V.2: *Stall maximization over multiple intervals*

```
1  Input: B^1, ..., B^N                                   /* sequence of intervals */
2  Input: C^1_(k), ..., C^N_(k)                           /* span in each interval */
3  Input: μ                                               /* total memory request */
4
5  Output: μ^1_(k), ..., μ^N_(k)                          /* memory requests in each interval */
6
7  ∀j : μ^j_(k) ← 0
8
9  do:
10                                                        /* consider only unsaturated intervals */
11   B ← {j | μ_j(k) < C^j_(k) · q^j}
12                                                        /* compute current memory rate r on each unsaturated interval */
13   ∀j ∈ B : r^j ← μ^j_(k)/C^j_(k)
14                                                        /* find curve p where r yields maximum stall slope */
15   p ← argmax_{j∈B}{slope^j(r^j)}
16                                                        /* assign as many as possible requests to this interval */
17   μ^p_(k) ← min(μ − Σ_{j≠p} μ^j_(k), next^p(r^p) · C^p_(k))
18                                                        /* stop if all μ assigned, or all intervals are saturated */
19 until (Σ_{j=1}^N μ^j_(k) = μ or ∀j : μ^j_(k) = C^j_(k) · q^j)
```

For each iteration, at Line 15 the algorithm selects the interval B^j with the highest slope for the currently assigned memory rate r^j among all the unsaturated intervals – in case two intervals have the same slope, the tie can be broken arbitrarily. Finally, at Line 17 the value of memory requests $\mu^p_{(k)}$ assigned to the selected interval B^p is modified to the minimum of two expressions: (1) $\mu - \sum_{j\neq p} \mu^j_{(k)}$, that is, all remaining requests. In this case, after the assignment, it holds that $\sum_{j=1}^{N} \mu^j_{(k)} = \mu$ and the algorithm terminates immediately. (2) $\text{next}^p(r^p) \cdot C^p_{(k)}$, that is, $\mu^p_{(k)}$ is incremented so that $r^p = \mu^p_{(k)}/C^p_{(k)}$ becomes equal to the next segment start point.

Note that the segment start points, μ and $C^p_{(k)}$ are all natural numbers; hence, assuming that the values of variables $\mu^j_{(k)}$ were integer before the assignment at Line 15, the new value assigned to $\mu^p_{(k)}$ is also integer. Furthermore, the new assignment cannot violate the constraints $\sum_{j=1}^{N} \mu^j_{(k)} \leq \mu$ or $\mu^j_{(k)} \leq C^j_{(k)} \cdot q^j$, since we use the minimum of the two expressions. Hence, this shows that the assignment to variables $\mu^j_{(k)}$ operated by Algorithm V.2 is feasible according to the constraints at Lines 11-13 of Algorithm V.1. Furthermore, note that Algorithm V.2 is guaranteed to terminate after the assignment at Line 17 selects the first expression, or after all intervals have been saturated. The number of segment start points for each function $\bar{I}(r)^j$ is $\mathcal{O}(m)$; hence, the number of iteration of the algorithm is $\mathcal{O}(N \cdot m)$.

Finally, we show that once the algorithm terminates, the assignment to variables $\mu^j_{(k)}$ maximizes the cumulative stall $S_{(k)} = \sum_{j=1}^{N} S^j_{(k)} = \sum_{j=1}^{N} \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}$, that is, the

objective function in Algorithm V.1. This follows from the way intervals are selected at Line 15. By contradiction, assume that there exists a different feasible assignment, call it $\{\bar{\mu}^1_{(k)}, \ldots, \bar{\mu}^N_{(k)}\}$, such that $\sum_{j=1}^{N} \bar{I}(\bar{\mu}^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)} > \sum_{j=1}^{N} \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}$; then can obtain $\{\bar{\mu}^1_{(k)}, \ldots, \bar{\mu}^N_{(k)}\}$ by iteratively modifying $\{\mu^1_{(k)}, \ldots, \mu^N_{(k)}\}$, subtracting some number of memory requests, say ζ, from one variable $\mu^j_{(k)}$ and adding them to another variable $\mu^p_{(k)}$. Now define:

$$\text{slope}^p = \frac{\bar{I}\big((\mu^p_{(k)} + \zeta)/C^p_{(k)}\big)^p - \bar{I}(\mu^p_{(k)}/C^p_{(k)})^p}{\zeta/C^p_{(k)}},$$
$$\text{slope}^j = \frac{\bar{I}(\mu^j_{(k)}/C^j_{(k)})^j - \bar{I}\big((\mu^j_{(k)} - \zeta)/C^j_{(k)}\big)^j}{\zeta/C^j_{(k)}}, \tag{V.18}$$

as the resulting slopes for functions $\bar{I}(r)^p_i$ and $\bar{I}(r)^j_i$. Note that the modification to the variables will increase the cumulative stall by $\text{slope}^p \cdot \zeta$ and reduce it by $\text{slope}^j \cdot \zeta$. But because Line 15 always selects the function with the highest slope, it must be $\text{slope}^p \leq \text{slope}^j$; hence, the cumulative stall cannot increase, a contradiction. In summary, we have shown the following lemma:

Lemma 4. *Algorithm V.2 terminates in a finite number of steps. Furthermore, the resulting assignment to variables* $\{\mu^1_{(k)}, \ldots, \mu^N_{(k)}\}$ *determines a value of* $\sum_{j=1}^{N} \bar{I}(\mu^j_{(k)}/C^j_{(k)})^j \cdot C^j_{(k)}$ *equal to the value of the objective function computed by Algorithm V.1.*

Computational Complexity: note that each iteration of the algorithm can be easily optimized to execute in $\mathcal{O}(1)$. The sum of variables $\mu^j_{(k)}$ at Lines 17, 19 can be executed in constant time by keeping the sum in a variable and updating it each time $\mu^p_{(k)}$ is modified at Line 17. The selection at Line 15 can be performed in constant time by creating a table of segments ordered by slope. Since ordering the segments then dominates the complexity of the algorithm, this results in a $\mathcal{O}\big(N \cdot n \cdot \log(N \cdot n)\big)$ time for Algorithm V.2. Note that, the analysis assumes a know memory schedule.

V.6 IMA Case Study: Schedulability Ratios

Integrated Modular Avionics (IMA) systems use time-triggered scheduling of partitions, also known as ARINC 653 scheduling [ARI03], where each partition is assigned, at compile time, a fixed start time and span in a major cycle i.e., a hyperperiod (H). These partition-level scheduling decisions are stored at compile time resulting in a static CPU schedule, which is repeated every major cycle.

Our analysis (Section V.5) works with known memory assignment across cores and known workload parameters. IMA systems are a natural fit, representing a real-world scenario. We consider a set of IMA partitions with a fixed major cycle and assignment of partitions to cores. For simplicity, we assume the order of execution of the partitions is known, and we assume that each partition executes once in the major cycle, and that the major cycle is synchronized among cores. Our goal is to use our analysis from Section

V.5 and perform an empirical evaluation comparing the ratio of schedulable tasksets to generated tasksets, under dynamic memory budget assignment policy against the static budget assignment policies, under a fixed partition execution order on each core.

In the next Subsections, we describe the setup used to compare the budget assignment policies and the two sets of experiments, one, that varies the number of cores and two, that varies the number of memory intensive partitions in a system.

V.6.1 Setup

IMA Partition Set Generation: For each experiment run, we consider n cores and a set of $4 \times n$ IMA partitions, with a fixed major cycle, i.e., hyperperiod (H) of $128ms$. The earliest start time of each partition is set to $t = 0$ and the deadline to the hyperperiod i.e. $128ms$. From the perspective of the analysis (Section V.5), each partition is a workload.

We characterize the varying memory demand between partitions as exhibited by avionic applications IV.4, using a parameter — *memory intensity (MI)* — , that represents the ratio of pure memory demand to the sum of pure processing demand and pure memory demand of a partition under single-core case i.e., no contentions. We then use a bi-modal distribution for MI, where each partition either has a HIGH MI mode or a LOW MI mode. The use of two modes is first, consistent with the memory intensity behavior exhibited by partitions in a real avionic application IV.4, and second, some partitions perform I/O activity that is memory- intensive. All HIGH MI mode partitions are randomly assigned an MI value in the range of $[0.5, 0.99]$, whereas for LOW MI mode partitions, the MI value range is $[0.001, 0.1]$. We use a parameter *memory intensity ratio* MIr to vary the number of partitions in the HIGH MI mode to that in the LOW MI mode in the system.

Each partition is then randomly assigned a core, such that each core ends up with 4 partitions. The setup then generates per partition single-core utilization using UUniFast algorithm [BB05] such that U is the cumulative single-core utilization of each core. The parameter U allows varying the cumulative single-core utilization of partitions assigned to a core. Next, the setup generates E and μ values for each partition based on its single-core utilization and memory intensity (MI) value, assuming no stall. The E and μ values of each partition respectively represent an aggregated E demand and an aggregated μ demand of all tasks assigned to it, in line with existing work like [NPB+14].

System-wide Parameters: We use realistic system-wide parameters: $L_{max} = 2.4 \times 10^{-6} ms$, $T^m = \Upsilon = 1ms$, resulting in $Q = 41666$ as described in [MPC+15].

Budget Assignment Policies: We consider two static budget assignment policies: *Static and even (SE)* that assigns to each core a constant and identical budget of $1/n$ times the total budget, e.g., for a 4-core system $\mathcal{Q}$ ={10416, 10416, 10416, 10416}, and *static and uneven (SU)* that assigns to each core a constant budget based on the weight of each core, e.g., $\mathcal{Q} = \{416, 20416, 5416, 15416\}$. For the SU policy, we use a heuristic to generate the weight of each core and thereby, a budget assignment, based on the input partition set.

The key idea behind the heuristic is to assign cores with higher memory demand

a higher memory budget. The heuristic computes weight of each core based on the ratio of the remaining cumulative μ to that of the sum of remaining cumulative μ and cumulative E on a core. Then, the memory bandwidth Q is partitioned among cores based on the computed weights resulting in a budget assignment. This is similar to the term memory intensity, albeit on a core-level.

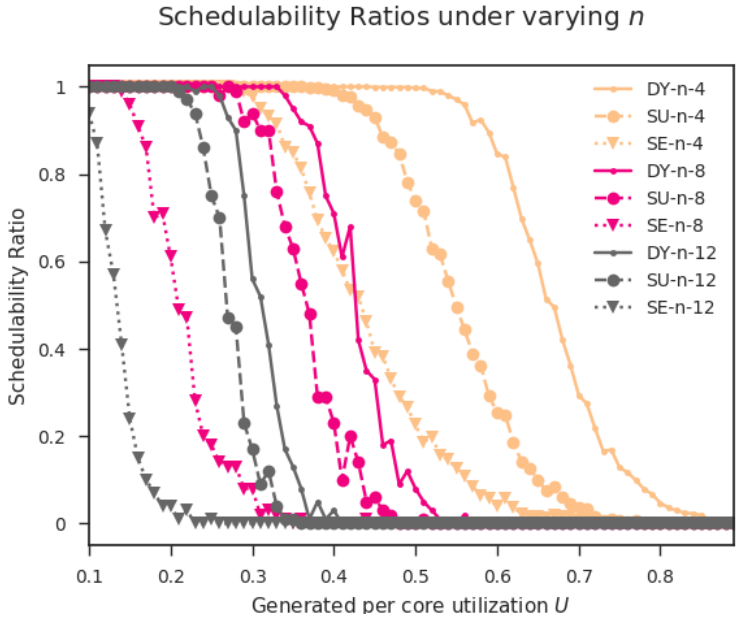

Figure V.4: *Schedulability ratios under varying n from* 4 *to* 12 *cores in steps of* 4. *Memory intensity ratio MIr of* 0.25. *Utilization per core U is varied from* 0.1 *to* 0.90 *in steps of* 0.1. *The MI value range considered is* $[0.5, 0.99]$.

We compare the static policies (SE and SU) against a *dynamic policy (DY)*, which assigns dynamic memory budgets to cores, using the heuristic. As compared to the static SU policy that uses the heuristic at time $t = 0$ only, DY policy recomputes the weight of every core each time a partition finishes execution, resulting in a dynamic budget assignment.

V.6.2 Effect of Varying Number of Cores

Figure V.4 compares the schedulability ratios for each of the three budget assignment policies — DY, SU and SE —- under varying the number of cores n from 4 to 12 in steps of 4, for a fixed memory intensity ratio MIr of 0.25. In Figure V.4, for each value of U, we generated 1000 partition sets for $n = 4$ case, and 100 partition sets for each of $n = 8$ and $n = 12$ cases. On the x-axis, we vary the cumulative per core utilization U from 0.1 to 0.9 in steps of 0.01.

First, we observe that as the number of cores n increases, the schedulability ratio decreases for the plots shift towards the left, for each of the three budget assignment policies. This is because, with increasing the number of cores, the total memory supply

remains constant, albeit the total memory demand increases as the number of HIGH MI mode partitions increase in the system. Second, for each value of n, the dynamic policy DY dominates the static policies SU and SE.

V.6.3 Sensitivity to Memory Intensity Ratio

Now, we vary the memory intensity ratio MIr from 0.15 to 0.50 that impacts the number of HIGH MI partitions in the system, and consequently, the number of LOW MI partitions in the system. We set the number of cores n to 8.

Figure V.5 shows the schedulability ratios for each of the three budget assignment policies — DY, SU and SE —- under varying MIr. On the x-axis, we vary the cumulative per core utilization U from 0.1 to 0.9 in steps of 0.01. In Figure V.5, we generated 100 partition sets for every combination of U and MIr.

As the MIr ratio increases, the cumulative memory load from all cores on the memory increases, in general. Consequently, we observe that schedulability ratio plots shift towards the left on increasing the MIr ratios. Further, for each memory intensity ratio MIr, the dynamic budget assignment policy DY dominates static policies SU and SE.

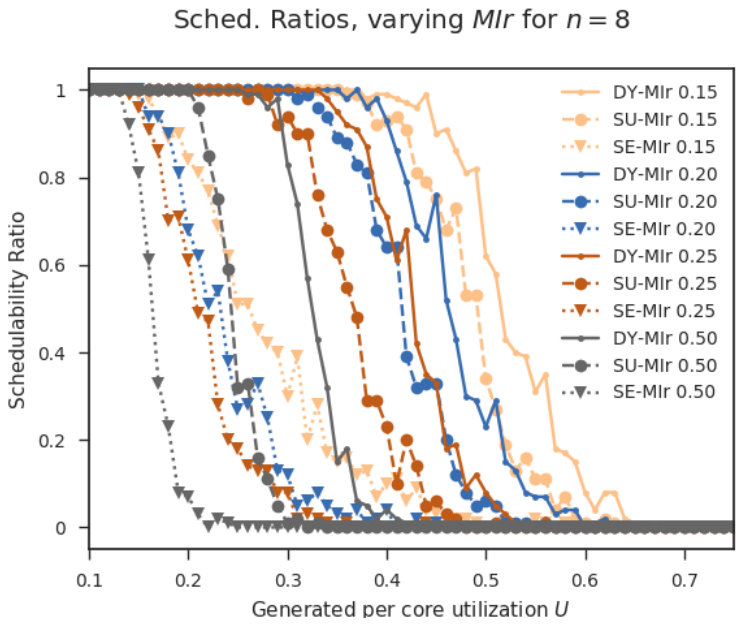

Figure V.5: *Schedulability ratios under varying memory intensity ratio MIr from* 0.15 *to* 0.50. *The number of cores n is set to* 8. *Utilization per core U is varied from* 0.1 *to* 0.90 *in steps of* 0.1. *The MI value range considered is* $[0.5, 0.99]$.

V.6.4 Sensitivity to Range of Memory Intensity Value

We now increase the lower limit of the memory intensity (MI) value from 0.5 to 0.7. The higher limit is fixed to 0.99. A partition's MI value determines the distribution

of its single-core utilization between pure execution time demand E and pure memory demand μ values, assuming no stall. So, increasing the lower limit of MI value, is likely to result in increase of cumulative memory demand of HIGH Mode partitions, thereby increasing the cumulative memory demand of the system and reducing schedulability. This is observed in Figures V.6 and V.7 as the schedulability ratios shift slightly to the left i.e. increase of lower limit of MI value is likely to increase the cumulative memory demand of HIGH Mode partitions and of the system, that is likely to reduce schedulability.

For each data point 1000 partition sets were generated. The x-axis represents the cumulative per core utilization U that was varied from 0.1 to 0.9 in steps of 0.01.

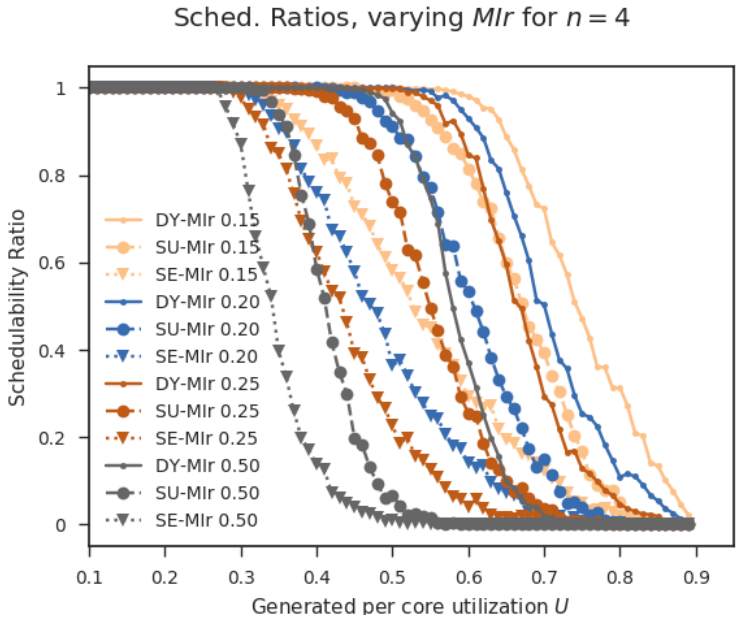

Figure V.6: *Schedulability ratios under memory intensity (MI) value* $[0.5, 0.99]$ *for* $n = 4$ *cores. The* MIr *ratio varies from* 0.15 *to* 0.50. *Utilization per core* U *is varied from* 0.1 *to* 0.90 *in steps of* 0.1. *The* MI *value range considered is* $[0.5, 0.99]$.

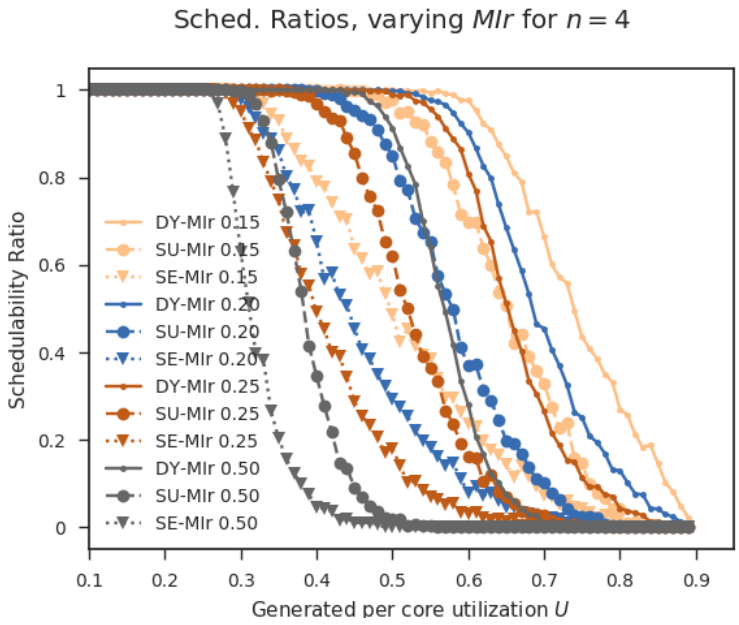

Figure V.7: *Schedulability ratios under memory intensity (MI) value* $[0.7, 0.99]$ *for* $n = 4$ *cores. The* MIr *ratio varies from* 0.15 *to* 0.50*. Utilization per core* U *is varied from* 0.1 *to* 0.90 *in steps of* 0.1*. The* MI *value range considered is* $[0.7, 0.99]$*.*

VI

Related Work

> *"A scientific accomplishment is always the cumulative result of many people working in the same field or related fields. Our present concept and knowledge cannot exist without past experiences, may not originate without present stimulations, . . . "*
>
> – Tsung-Dao Lee, *in* speech at the 1957 Nobel Banquet

Recent literature on the design of real-time systems on multi-core platforms considers main memory as a significant source of unpredictability, and an important interfering channel to mitigate. Predictable memory controllers have been proposed in [BLL+11, AGR07, VY15]. Further, a framework has been proposed to evaluate predictable performance of different multi-core architecture configurations in the design phase based using real traces of tasks [DAI+18].

[KdNA+14] and [YPV15] consider white-box approach assuming FR-FCFS DRAM scheduling to compute worst-case DRAM latency. [KdNA+14] considers private and shared DRAM banks between cores and 1 outstanding request per core. [YPV15] improves the analysis of [KdNA+14] by considering 12 outstanding request per core (6 reads and writes each) with high/low watermark scheme and private banks for each core. It requires knowledge of read buffer size; write buffer size, high watermark value and low watermark value and minimum writes per batch. DRAM subsystem with single-channel DRAM controller and a DDR memory module and FR-FCFS with watermark - private banks for each core. non-preemptive task execution with known reads and writes. mostly simulation based as do not know exactly the memory scheduling algorithm use. It shows that in the worst-case the analytically computed execution time of a task can be three times larger compared to the measured values when co-executing with 3 memory-intensive tasks. These works rely on the knowledge of DRAM scheduling policy and related parameter values like buffer size, writes per batch, high/low watermark values etc. It limits their applicability as silicon vendors are generally reluctant to share such information publicly to protect from the competition providing optimized designs. Further, these approaches do not consider the exact timing impact of DRAM refreshes, though [KdNA+14] mentions around 2% increase in DRAM interference delay.

[NP12, YYP+13, IMB+14] consider black-box approach to obtain maximum DRAM latency from measurements so as to overcome the key limitation of white-box approach - unavailability of COTS DRAM controller model. The main idea is to measure the

time taken by a pre-configured number of DRAM requests from a measurement core under maximum inter-core interference from the other cores. Averaging the largest of the observed values with the number of DRAM requests issued by the measurement core gives the maximum observed DRAM latency. [NP12] measures the combined NoC and DRAM latency for NXP P4080 multi-core for different combinations of DRAM requests: read-read, write-write, read-write-write-read. It shows that the combined latency of a DRAM request increases from 34.16 ns to 839.17 ns when the number of cores is increased from 1 to 8 [NP13]. [YYP+13] considers the Intel Core2Quad Q8400 multi-core and measures a guaranteed DRAM bandwidth of only 1.2 GB/s, which is then converted to latency. These approaches try their best to create worst-case inter-core interference scenario for different number of active cores, but are not guaranteed to do so. The lack of "baseline" comparison due to unavailability of COTS multi-core hardware model is their key shortcoming. Further, these approaches implicitly assume a fair DRAM arbitration and scheduling scheme, that ensures no starvation in all shared resource arbitration mechanism with round-robin and FIFO per core scheduling

Yet another body of work has investigated the idea of strictly serializing access of cores to main memory. By clustering and serializing access to shared memory, interference is avoided by design. Schranzhofer et al. [SPC+11], Pellizzoni et al. [PBB+11], Boniol et al. [BCNP12] propose deterministic execution models to control the access to shared resources. The basic concept is to divide program execution into multiple phases and restrict their capabilities. Schranzhofer et al. [SPC+11] propose to divide each task into superblocks where each superblock consists of three phases: acquisition phase, execution phase and replication phase. A task is allowed to access the shared hardware resource only in acquisition and replication phases. Further, no two co-executing tasks on different cores can have overlapping acquisition and replication phases. Pellizzoni et al. [PBB+11] propose the PRedictable Execution Model (PREM) for single-core COTS processors, introducing co-scheduling for shared resources. It splits each task into a sequence of non-preemptable intervals: predictable intervals, and compatible intervals. Predictable intervals are used to pre-load all data and instructions into local caches, while system calls and interrupt preemptions are prohibited. System calls and interrupt preemptions are, however, allowed in compatible intervals. Each predictable interval is further divided into two phases: execution phase and memory phase. Traffic from peripheral devices is only permitted during the execution phase of a predictable interval, resulting in an architecture with very few contentions for accesses to the shared resources. Boniol et al. [BCNP12] present a sliced execution model. It splits each task into sub-tasks and each sub-task further into execution slices and communication slices. The access to a shared resource is only allowed in a communication slice. The work in [YPB+11] clusters memory operations in tasks via cache pre-fetching using compiler-level transformations, defining memory- and execution- phases. Then, a central scheduler only allows at most one memory-phase to be active at any point in time. A similar scheme was adopted in [TMW+16, WP13, SP17, DFG+14] using DMAs instead of CPU-initiated pre-fetches and scratchpad memories. Such approaches require modification of the source code of legacy applications [GJR+15], to support transition from single-core processors to COTS multi-core processors, a non-trivial step.

Compared to clustering and serializing access approach, monitoring and regulation has the advantage of being entirely implementable at OS-level and requiring no application source-code modification. Bellosa [Bel97b, Bel97a] introduced the idea to leverage built-in processor counters to acquire additional task runtime information. Yun et al. [YYP+12] proposed controlling memory accesses from all but one cores to limit contentions experienced by hard real-time tasks executing on one core to ensure that they meet their deadlines. Yun et al. extended the work in [YYP+13] by introducing a memory throttling mechanism - MemGuard, that regulates memory accesses using a memory server on each core. It assumes that memory bandwidth is statically partitioned between cores before runtime for safety-critical systems. Behnam et al. [BINS12] propose a method to isolate the behaviour of different cores. They apply a hierarchy of servers to all cores using a server-based approach which assigns a certain limit on the amount of cache misses.

With respect to static and even budget assignment, a first analysis was derived in [MPC+15]. It also reduces the worst case number of DRAM requests of a task by locking frequently accessed data in the last-level shared cache or shared SRAM. It requires analyzing each task before runtime and identifying pages that when locked in caches/SRAM will reduce the memory interference delay.

In [YYW+16], an analysis for static and uneven bandwidth partitioning was performed assuming only knowledge of the memory budget for the core under analysis, and assuming arbitrary assignment to the other cores. More recently, the work in [MPTC17] demonstrated that by leveraging *exact* knowledge of each core's budget it is possible to drastically reduce the pessimism of the analysis.

A few works [YYP+13, YYP+16] proposed unused budget reclamation. However, no offline guarantees can be provided on the dynamic portion of the assigned budget. The work in [FLY14] considers budget reclamation and derives WCET guarantees assuming full knowledge of the workload on all cores. In a more recent work, Nowotsch et al. [NP13, NPB+14] consider avionics temporal partitions with pre-defined budget assignment. The authors consider a frame that consists of co-executing partitions and introduce a WCET analysis that computes, in offline phase, the execution time of each partition considering contentions from the worst-case number of memory accesses from each of co-executing partitions in a frame.

What sets this work apart? Our focus on dynamic memory bandwidth isolation and WCET guarantees for safety-critical systems differentiates this work from the aforementioned works. Another differentiator is our focus on no-source code modifications requirement and support for incremental certification of an existing single-core safety-critical (SC) application by preserving its single-core CPU schedule on migration.

VII

Discussion

While the computing industry has shifted from single-core to multi-core processors for performance gain, safety-critical systems (SCSs) still require solutions that enable their transition while guaranteeing safety, requiring no source-code modifications and substantially reducing re-development and re-certification costs. This dissertation considered two related problems. The first problem is how to enable migration of an existing SC application to multi-core platforms without source-code modifications while preserving its ARNC 653 CPU schedule, SCS require this migration step to prevent re-development and re-certification costs that are generally substantial for legacy applications in aerospace domain. The second problem considered was of span analysis under contentions when deadline-constrained workloads in independent partitioned workloads set execute on a homogeneous multi-core processor with dynamic time-triggered shared memory bandwidth partitioning in SCSs.

The problems originate as multi-core processors implicitly share network-on-chip (NoC) and main memory hardware resources among logically-independent cores. DRAM-like main memories generally employ locality principles and multi-level scheduling policies to improve average-case memory performance that has large temporal variability resulting in poor predictable performance. These memories in commercial-off-the-shelf (COTS) domain consider both temporal and spatial locality as opening a row to read (say) 64 bytes of data opens a 4KB row, and a request for 4 bytes of data is answered with (say) 64 bytes of data that stored nearby is likely to be accessed. A memory request that targets an open row complets much faster compared to that targets a closed row, thereby increasing temporal variability. The multi-level scheduling policies aim at maximizing the bandwidth of the memory sub-system that increase maximum delay experience by a memory request. For example, a memory controller may employ first-ready first-come-first-serve (FR-FCFS) policy for the back-end of the memory controller and round robin at the front-end of the memory controller. FR-FCFS selects a memory request that is ready for scheduling (whose timing constraints have been met) even if this request arrived later than a waiting request, also known as request re-ordering. FCFS policy is used for tie-breaking. The request re-ordering while good for average-case performance, further increase the temporal variability of a memory request. Further, the timing impact of memory refreshes and timing constraints like tFAW that are dynamically handled, further increase the temporal variability of memory request time.

Multi-core Model

Our system model in Chapter III focuses on COTS multi-cores because of two reasons. First, use of COTS platforms is known to reduce development and maintenance costs in safety-critical domains compared to custom platforms like field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs) etc. E.g., COTS platforms ease integration of safety-critical applications from multiple vendors, thereby reducing development and maintenance costs. This is evident by the shift from federated architecture to integrated modular avionics (IMA) architectures since the 1990s in the aerospace domain. Boeing B777 airplane was the first to use IMA architecture that improved integration of applications from multiple vendors onto standardized hardware platforms. Similar shift has partly occurred in automotive domain and is starting to occur in railway domain. Second, of the high-performance COTS platforms — multi-cores and many-cores—, multi-core platforms have a large support ecosystem of debugging tools, compilers, real-time operating systems, that is essential for use in safety-critical systems. Most chip producers migrated to multi-cores and many offer backwards-compatible instruction-set architecture (ISA) to their single-core platforms that eases migration of legacy applications to multi-cores. While some chip producers offer many-cores, we think many-cores lack maturity, partly due to limited adoption.

We considered a multi-core model based on real COTS multi-core platforms. Although COTS high-performance platforms feature out-of-order execution, our proposed core model assumes an in-order execution that considers each memory request stalls the issuing core. While this assumption is pessimistic, published works [NPB+14, NP13] from avionics domain have made similar assumptions, possibly due to stringent certification requirements. For the memory request contention model, we considered two models: a constant memory request time and delay model, and a non-decreasing memory request time model. While the first model caters to multi-core processors that employ pure round-robin (RR) arbitration policy among cores for memory requests [MPC+15], the second model allows using multi-core platform with complex multi-level memory scheduling policies provided memory latencies are obtainable as shown by [NP12].

White box approaches to estimate memory latency provide formal proof, which is especially useful in certification, as compared to black-box approach's reliance on measurements. However, it requires silicon vendors to provide *(a)* DRAM controller model with scheduling policy used and values of related parameters like size of read and write buffers, writes per batch, etc., and *(b)* NoC model with size of buffers etc., used in a COTS multi-core platform. For the considered platforms in this work, silicon vendors have not publicly shared the models and the arbitration policy used. Silicon vendors are generally quite reluctant to share COTS DRAM controller model due to fear of the competition providing optimized designs. Therefore, in this work, we rely on an alternate measurement-based method from Nowotsch and Paulitsch [NP12] that uses a black-box approach to obtain the set of memory latency Δ values. A common challenge in using black-approaches is the lack of "baseline" for comparison. However, our proposed model and the proposed analyses will work with memory request time value generated from white-box approaches.

Resource Servers

In Chapter IV.2, we proposed two types of resource servers — CPU server and memory server, with period equal to slot duration. The resource servers extend resource control and reservation to multiple resources — CPUs and memory — and remove the circular dependency between a workload's span and core-level scheduling of the co-executing cores in a partitioned system. Each pair of a CPU server and a memory server executes on each core and is released synchronously at the start of each slot. A CPU server instance obtains a CPU budget at each release in CPU time units. A memory server instance obtains a memory budget at each release in memory request units. The resource server mechanism is generic i.e., allow reserving budgets online or offline, provide the budget assignment is valid.

Memory servers provide another knob to vary memory resource assignment to a workload, in addition to conventional CPU scheduling. We consider each server's period equals slot duration, that makes the resource servers well-suited for time-triggered (TT) systems. Our proposed resource monitors are commonly supported by many COTS multi-core platforms like [Fre13b, NXP16]. Thus, their implementation requires no additional hardware on such platforms, supporting COTS hardware use, as preferred by SC domains, especially avionics. Further, a CPU and memory server pair has a low-overhead of 10μs when slot duration is fixed to $1ms$, as observed on the NXP P4080 multi-core platform [Now14]. For use in avionic SCSs, it requires integration and certification at the inter-partition-level scheduler.

Certification authorities (CAs) demand significant precaution before approving COTS multi-core use in avionics. E.g., current certification guidelines (CAST-32a [Tea16]) on COTS multi-core platform use in avionics require using a safety-net processor to monitor a multi-core platform. A safety-net processor is a single-core processor tasked to monitor applications executing on a multi-core processors and take mitigating actions if a multi-core processor shows unexpected behavior. Thus, avionics companies have developed safety-net processors. Some of the solutions using this approach only rely on measurements and safety-net processors to ensure a SC applications meets its deadlines. Further, such solutions are limited to when only one core executes safety-critical workloads. We think such solutions are only transitional and due to lack of experience with multi-core use especially in avionics domain for SCS applications. As resource servers have low runtime overhead and ensure all cores to execute critical workloads, we think the resource servers are well-suited for domains like avionics.

Workload Schedulability and Span Analyses under Dynamic Budgets

In Chapter IV.3, we proposed a workload schedulability test for known even-memory-bandwidth-assignment-to-active-cores. An offline scheduler requires it to test if a set of slots under consideration with given memory budget assignments meet a workload's worst-case demand of its core-local execution time and worst-case number of memory

requests. If the test is successful, it gives guarantees that the given set of slots with their memory budget assignment will meet workload's worst-case requirements at runtime. The memory budgets sorting step dominates the tests' computational complexity. An offline scheduler is likely to use the test for a workload multiple times to ensure schedulability as well as tightness of interval of slots reserved for a workload, i.e., to prevent under- and over-provisioning of resources. The test is generic with respect to offline schedulers.

Chapter V.4 presented the span analysis for a workload that undergoes through slots of arbitrary memory budget assignments, extending the workload schedulability test from Chapter IV.3. It considers the maximum time taken to perform a memory request time in isolation is fixed and it also denotes the maximum delay due to contention from a single memory request from an another core. The analysis constructs core-specific memory-stall curves for each memory budget assignment to cores that represent the cumulative maximum interference-induced stall for a given number of memory requests performed by a core under analysis. If a resulting memory stall curve is concave, then it is final. If the curve has convex segments, then a refinement step recomputes the curves by taking the upper envelope of each convex segment to obtain a concave curve. The analysis uses memory stall curves to illustrate span analysis under static budget that eases exposition for the later presented span analysis under dynamic budgets. We presented a fixed-point iterative algorithm to compute the worst-case span of the workload on a core under analysis under static memory budget, that is an improvement compared to [MPTC17]. Later, we then presented a fixed-point iterative span analysis under dynamic arbitrary budget assignments in Chapter V.5. The key intuition behind iterative re-computation is that the increase in workload span in an iteration is likely to increase the maximum stall in a consecutive iteration due to a different worst-case distribution of memory requests across *(a)* different per memory-stall curves and/or *(b)* different memory scheduling intervals. We then presented a stall maximization algorithm that efficiently implements the fixed-point iterative algorithm for span analysis. It has a complexity similar to a concave optimization problem.

Benefits, Trade-offs and Extension Possibilities

The presented schedulability test and the two analyses construct analytical worst-case memory request patterns for a workload under analysis and given memory budget assignments. While such a worst-case pattern is not unique with respect to the issue of requests in time, it is an invariant considering workload's each memory request suffers maximum contention based on the budget assignments. As a sorting step dominates the computational complexity of our methods, they provide lower computational complexity compared to using static WCET analysis tools that are pragmatically infeasible for WCET analysis on multi-core platforms when workloads suffer contention. Each of the method gives guarantees that the given set of slots with their memory budget assignment will meet workload's worst-case requirements at runtime.

Our presented schedulability test and the two analyses assume a known memory schedule, workload-to-CPU assignment and workload priority order on a CPU. Thus,

they aid in response time analysis with respect to memory resource and can be used with existing CPU schedulers that schedule on slot granularity. Moreover, they can be used to analyze memory bandwidth schedulers (or memory bandwidth assignment schemes) that assign dynamic memory-bandwidth-to-cores over time.

As our proposed methods construct analytical worst-case memory request pattern, they can significantly ease SCSs certification and SCSs system design and development in the avionics domain especially in the early phases. While workload's real memory request pattern-aware approaches must prove all patterns worse than the considered pattern will not occur under contentions, use of our approach only requires a proof for a single case, that can ease certification. Further, in the early phases of SCSs system design and development in the avionics domain an application's source-code is unavailable and thus there is a lack of knowledge of an application's memory request pattern. Traditionally, a system designer determined the number of partitions required and the worst-case execution time (WCET) of each partition. The designer then estimated the numbe rof single-core processors required by generating ARINC 653 schedules. For scheduling on multi-core under contentions, a system designer requires new tools to estimate a partition span under contentions. Under such a scenario, our method can help to estimate span of a partition under dynamic memory budget assignments as our methods require no knowledge of a workload's request pattern and the CPU schedule of other cores. using our method for system-level partition scheduling will require integration in offline scheduling tools.

Our methods trade tightness of span to have a low computational complexity solution by distributing memory requests to budget assignments to maximize stall. For contention-aware approaches that work with workload-pattern-awareness can result in tight span values as a real workload pattern use can reduce pessimism and tighten span of a workload. Using our method at the the granularity of a memory request is unlikely to be practically feasible. However, as a future extension, one way to reduce pessimism while using our method is to consider computing span at sub-workload level, where a workload is abstractly split into sub-units called sub-workloads. Thus, our method will treat each sub-workload as a workload in the regular model. Note that, our method then requires for each sub-workload the maximum core-local execution time demand and worst-case number of memory requests. The sum of spans of all sub-workloads of a workload will give a workload's span. A natural-way to abstractly split a workload is along a workload's memory intensive phases, that are typically at the start — when caches are cold —, towards the completion — as write-backs occur to memory — and during the begin of loops. Note that, obtaining a sub-workload's maximum core-local execution time demand and worst-case number of memory request, will require making some assumptions on start cache-state. Note that, the span analysis must be performed in sub-workload occurrence order at runtime. This extension can be especially useful to match memory budget assignments to a workload's memory bandwidth demand.

Our presented method is composable and supports incremental certification as it does not require knowledge of CPU schedule of other cores. Moreover, it does not require any source-code modifications enabling low-cost migration of legacy applications in comparison to execution model based approaches [SPC+11, BCNP12, PBB+11], that

require significant source-code modifications.

VIII

Conclusion

"Live long and prosper"

– Vulcan Salute — Star Trek Universe

In this dissertation, we focused on multi-core processors to meet performance demand of emerging safety-critical applications like sense-and-avoid for autonomous flying. Memory bandwidth in multi-core processors is shared across cores and is a significant cause of performance bottleneck and temporal variability of multiple-orders in task's execution times due to contentions in memory sub-system. Further, the circular dependency between not only WCET and CPU scheduling of others cores but also WCET and memory bandwidth assignments over time to cores. Safety-critical systems (SCSs) require solutions that enable their transition to multi-core processors while guaranteeing safety, requiring no source-code modifications and substantially reducing re-development and re-certification costs. This dissertation considered the problem of worst-case execution time (WCET) analysis under contentions when deadline-constrained workloads in an independent partitioned workload set execute on a homogeneous multi-core processor with dynamic time-triggered shared memory bandwidth partitioning in SCSs.

VIII.1 Summary of Contributions

We presented a new scheduling problem in the introduction for commercial-off-the-shelf (COTS) multi-core processors when some workloads in a partitioned workload set are memory-intensive. This problem originates from a real certified avionics application— helicopter terrain awareness and warning system (HTAWS)— that we shows in this work. Workloads in this application drastically differ in their average memory bandwidth demand. The application's measured resource requirements show that it is not schedulable on a multi-core platform using existing static memory bandwidth partitioning approaches. While reserving resources tailored to a workload's demand can significantly improve workload schedulability, there is a need for solutions that allow reserving memory bandwidth tailored to a workload and compute WCET of a workload under dynamic memory bandwidth assignment to cores.

In Chapter II, we considered a multi-core model based on real COTS multi-core platforms. The model is compatible with both black-box and white-box approaches to obtain number for time taken to perform a memory request both, in isolation and under contention. We presented practical considerations for with respect to core, memory sub-system and monitoring resources configuration.

In Chapter IV, we proposed two types of resource servers — CPU server and memory server, with period equal to slot duration. The resource servers extend resource control and reservation to multiple resources — CPUs and memory — and overcome the circular dependency between WCET and core-level scheduling of the co-executing cores. Each pair of a CPU server and a memory server executes on each core and is released synchronously at the start of each slot. A CPU server instance obtains a CPU budget at each release in CPU time units. A memory server instance obtains a memory budget at each release in memory request units. Memory servers provided another knob to vary memory resource assignment to a workload, in addition to conventional CPU scheduling. We considered each server's period equals slot duration, that makes the resource servers well-suited for time-triggered (TT) systems.

Next, we solved the problem of computing a workload's span under even-memory-budget-assignment-to-active-cores changes over time and is known. We described how to construct analytical worst-case memory request pattern. This step has the benefit of significantly less computational complexity compared to using static WCET analysis tools to find such a pattern. Next, we used the analytical worst-case memory request pattern and developed a workload schedulability test useful for an offline scheduler to check if a workload is schedulable under known and changing even-memory-bandwidth-to-core assignment over time.

Further, we applied our proposed method with resource servers and schedulability test for dynamic even budget assignment to an industrial avionics use-case certified single-core application — HTAWS— from Airbus. As some partitions in the HTAWS application require upto 73% of memory bandwidth, static memory bandwidth isolation over-provisions memory bandwidth to core that matches the maximum demand of HTAWS' workloads. The application's measured resource requirements and memory latency numbers for P5020 platform showed that it is not schedulable on a multi-core

platform using existing static memory bandwidth partitioning approaches. Adding more cores under static memory bandwidth isolation will not solve the problem as it will not increase the memory bandwidth. Using our proposed dynamic memory bandwidth isolation, allowed to match memory bandwidth of the HTAWS application on per-workload basis and use additional cores when the non-memory-intensive partitions are active. It preserved HTAWS application's CPU schedule under contentions on a multi-core processor. This is an important industrial requirement to aid transition of existing SCSs from single-core processors to multi-core processors and substantially reduce re-development and re-certification costs especially for legacy applications.

In Chapter V, we presented the span analysis for a workload that undergoes through slots of arbitrary memory budget assignments. It considers the maximum time taken to perform a memory request time in isolation is fixed and it also denotes the maximum delay due to contention from a single memory request from an another core. The analysis constructs core-specific memory-stall curves for each memory budget assignment to cores that represent the cumulative maximum interference-induced stall for a given number of memory requests performed by a core under analysis. If a resulting memory stall curve is concave, then it is final. If the curve has convex segments, then a refinement step recomputes the curves by taking the upper envelope of each convex segment to obtain a concave curve. We used memory stall curves to illustrate the span analysis under static budget and presented a fixed-point iterative algorithm to compute the worst-case span of the workload. In each iteration, the algorithm recomputes the maximum stall and thereby, the workload span, based on the workload span from the previous iteration (except for the base iteration) and the corresponding memory schedule. The algorithm improves the calculation computing span under static memory bandwidth partitioning.

Then, we presented a fixed-point iterative span analysis under dynamic arbitrary budget assignments in Chapter V.5. The key intuition behind iterative re-computation is that the increase in workload span in an iteration is likely to increase the maximum stall in a consecutive iteration due to a different worst-case distribution of memory requests across *(a)* different per memory-stall curves and/or *(b)* different memory scheduling intervals. We then presented a stall maximization algorithm that efficiently implements the fixed-point iterative algorithm for span analysis. It has a complexity similar to a concave optimization problem.

The chapter concluded with an evaluation for an Integrated Modular Avionics (IMA) scenario. It compared dynamic (DY) memory bandwidth assignment policy against two static bandwidth assignment policies — static even (SE) and static uneven (SU) —, under a fixed workload priority order on each core. The comparison metric chosen was schedulability ratios, i.e., the ratio of schedulable workload sets to generated workload sets for each bandwidth assignment policy. The evaluation demonstrated our proposed method under dynamic memory bandwidth allocation dominates static partitioning when partitioned workload sets contain disparate memory-intensive workloads, similar to the HTAWS.

VIII.2 Future Directions

We presented two future extensions of our proposed methods. First, our methods can be particularly useful in the early phases of safety-critical (SC) systems design and development in the avionics domain when an application's source-code/memory request pattern is unavailable. Under such a scenario, our method can help to estimate span of a partition under dynamic memory budget assignments as they require no knowledge of a workload's request pattern and the CPU schedule of other cores. Using our method for system-level partition scheduling requires integration in offline scheduling tools.

Second, as a future extension, our methods can tighten a workload's span analysis by computing span at sub-workload level, where a workload is abstractly split into sub-units called sub-workloads. Thus, our method will treat each sub-workload as a workload in the regular model. Note that, our method then requires for each sub-workload the maximum core-local execution time demand and worst-case number of memory request. The sum of spans of all sub-workloads of a workload will give a workload's span. This extension can be especially useful to match memory budget assignments to a workload's memory bandwidth demand.

Appendix A

Core-local Execution Time and Memory Request Constraints

Constraint (A.1), specified in our constraint-solver model, is a key constraint that ensures that the number of slots allocated to a partition meet its requirements of core-local execution time and the number of memory accesses, under fixed number of dynamic memory bandwidth levels with equal budget distribution between the active cores in each slot.

$slot1(i)$ represents the number of slots with memory budget $\hat{q}^{m,1} = 41379$ considering only 1 active core assigned to a partition π_i. $slot2(i)$ represents the number of slots with memory budget $\hat{q}^{m,2} = 20338$ considering 2 active core assigned to a partition π_i.

$$
\begin{aligned}
&\forall i \in \mathcal{W}\ , \\
&\left(slot1(i) == 0\ \wedge\ slot2(i) == \left\lceil \frac{E_i}{\hat{q}^{c,1}} + \frac{\mu_i}{\hat{q}^{m,2}} \right\rceil\right) \\
\vee &\left(slot1(i) == \left\lceil \frac{E_i}{\hat{q}^{c,1}} + \frac{\mu_i}{\hat{q}^{m,1}} \right\rceil\ \wedge\ slot2(i) == 0\right)
\end{aligned}
$$

$$
\begin{aligned}
&\quad (slot1(i) > 0\ \wedge\ slot2(i) > 0) \\
&\quad \wedge \left(slot1(i) \geq E_i\ \wedge\ part(i) == slot1(i) * \hat{q}^{m,1} - E_i * \hat{q}^{m,1}\right) \\
\vee &\quad \wedge \left(slot2(i) * \hat{q}^{m,2} + part(i) - \mu_i < \hat{q}^{m,2}\right) \\
&\quad \wedge \left(slot2(i) * \hat{q}^{m,2} + part(i) - \mu_i \geq 0\right)
\end{aligned}
\tag{A.1}
$$

$$
\begin{aligned}
&\quad slot1(i) < E_i \\
&\quad \wedge \left(part(i) == -slot1(i) * \hat{q}^{m,1} + E_i * \hat{q}^{m,1}\right) \\
\vee &\quad \wedge \left(slot2(i) + \frac{part(i)}{\hat{q}^{m,1}}\right) * \hat{q}^{m,2} - \mu_i < \hat{q}^{m,2} \\
&\quad \wedge \left(slot2(i) + \frac{part(i)}{\hat{q}^{m,1}}\right) * \hat{q}^{m,2} - \mu_i \geq 0
\end{aligned}
$$

Appendix B

Proof of Theorem V.1

Lemma 5. *Consider a function* $f : \mathbb{R}_{\geq 0} \to \mathbb{R}$ *such that* f *is concave and* $f(0) = 0$. *Then* $\forall a, b \in \mathbb{R}_{\geq 0}$ *with* $b > a$*:*

$$f(a) - a \cdot \frac{f(b) - f(a)}{b - a} \geq 0. \tag{B.1}$$

Proof. By definition of concave function, $\forall x, y \in \mathbb{R}_{\geq 0}$ and $\alpha \in [0, 1]$ it holds:

$$f\big((1 - \alpha) \cdot x + \alpha \cdot y\big) \geq (1 - \alpha) \cdot f(x) + \alpha \cdot f(y). \tag{B.2}$$

Substituting $x = 0, y = b$ and $\alpha = \frac{a}{b}$ in Equation B.2 and given $f(0) = 0$ we have:

$$f(a) \geq \left(1 - \frac{a}{b}\right) \cdot f(0) + \frac{a}{b} \cdot f(b) = \frac{a}{b} \cdot f(b). \tag{B.3}$$

Finally, using Equation B.3 we obtain:

$$\begin{aligned} & f(a) - a \cdot \frac{f(b) - f(a)}{b - a} = \\ & \frac{f(a) \cdot b - f(a) \cdot a - f(b) \cdot a + f(a) \cdot a}{b - a} = \\ & \frac{f(a) \cdot b - f(b) \cdot a}{b - a} \geq \\ & \frac{\frac{a}{b} \cdot f(b) \cdot b - f(b) \cdot a}{b - a} = 0, \end{aligned} \tag{B.4}$$

which yields the hypothesis. □

Lemma 6. *Consider a function* $g : \mathbb{R}_{\geq 0} \to \mathbb{R}$ *such that* g *is concave and* $g(0) = 0$. *Then* $\forall \mu \in \mathbb{R}_{\geq 0}, \forall x \in \mathbb{R}_{>0}$*:* $x \cdot g(\mu/x)$ *is monotonic non decreasing in* x.

Proof. We have to show that $\forall x_1, x_2 \in \mathbb{R}_{>0}$ with $x_2 > x_1$:

$$x_2 \cdot g(\mu/x_2) \geq x_1 \cdot g(\mu/x_1). \tag{B.5}$$

Let us define $K = \frac{g(\mu/x_1) - g(\mu/x_2)}{1/x_1 - 1/x_2}$. We first show that:

$$g(\mu/x_2) - K/x_2 \geq 0. \tag{B.6}$$

Define $f(y) = g(\mu \cdot y), a = 1/x_2, b = 1/x_1$. Note that it holds $b > a > 0, f(0) = g(\mu \cdot 0) = 0$, and since g is concave, f is also concave. Then we obtain by substitution: $g(\mu/x_2) - K/x_2 = f(a) - a \cdot \frac{f(b)-f(a)}{b-a}$, which by Lemma 5 is greater than or equal to 0.

Finally, using Equation B.6 we obtain:

$$
\begin{aligned}
& x_2 \cdot g(\mu/x_2) = x_2 \cdot \big(g(\mu/x_2) - K/x_2\big) + K \geq \\
& x_1 \cdot \big(g(\mu/x_2) - K/x_2\big) + K = \\
& x_1 \cdot \big(g(\mu/x_2) - K/x_2 + K/x_1\big) = \\
& x_1 \cdot \big(g(\mu/x_2) + K \cdot (1/x_1 - 1/x_2)\big) = \\
& x1 \cdot \big(g(\mu/x_2) + g(\mu/x_1) - g(\mu/x_2)\big) = x_1 \cdot g(\mu/x_1), \qquad \text{(B.7)}
\end{aligned}
$$

completing the proof. □

Bibliography

[AC97] B.D. Aleksa and J.P. Carter. Boeing 777 airplane information management system operational experience. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 1, pages 3.1–21–7 vol.1, Oct 1997.

[Ada03] Charlotte Adams. Boeing: Integrated avionics takes another step forward, June 2003.

[AGR07] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, Sept 2007.

[Air16] Airbus. Future of urban mobility: My kind of flyover, December 2016.

[ARI03] ARINC. ARINC 653: Avionics application software standard interface, 2003.

[BB05] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.

[BCNP12] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012*, volume 7179 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[Bel97a] Frank Bellosa. Memory access - the third dimension of scheduling. Technical report, University of Erlangen, 1997.

[Bel97b] Frank Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical report, University of Erlangen, 1997.

[BF06] R. Black and M. Fletcher. Open systems architecture - both boon and bane. In *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*, pages 1–7, Oct 2006.

[BINS12] Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *5th Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2012.

[BLL+11] D. Bui, E.A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)*, pages 274 – 279, June 2011.

[CAS14] *CAST-32 Multi-core Processors*. Certification Authorities Software Team, May 2014.

[DAI+18] Robert I. Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, 54(3):607–661, Jul 2018.

[DFG+14] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, February 2014.

[FLY14] J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling DRAM contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 151–159, June 2014.

[Fre13a] Freescale Semiconductor. *e500mc Core Reference Manual Rev. 3*, 2013.

[Fre13b] Inc. Freescale Semiconductor. P4080: QorIQ P4080/P4040/P4081 Communications Processors with Data Path, 2013.

[Fre14] Freescale Semiconductor. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual Rev. 2*, 2014.

[Gec] Generic constraint development environment.

[GJR+15] S. Girbal, X. Jean, J. Le Rhun, D. G. Perez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core cots. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1–8D4–15, Sept 2015.

[IF09] Damir Isovic and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems*, 43(3):296–325, 2009.

[IMB+14] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjodin. The multi-resource server for predictable execution on multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 1–12, April 2014.

[Iti07] Jean-Bernard Itier. A380 integrated modular avionics: The history, objectives and challenges of the deployment of ima on a380, Nov. 2007.

[KdNA+14] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April 2014.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer-Verlag, 2nd edition, 2011.

[Lya16] Rodin Lyasoff. Welcome to vahana, Sept. 2016.

[MPC+15] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183, July 2015.

[MPTC17] Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET derivation under single core equivalence with explicit memory budget assignment. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 3:1–3:23, 2017.

[Now14] Jan Nowotsch. *Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors*. PhD thesis, University of Augsburg, 2014.

[NP12] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012.

[NP13] Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 151–160, New York, NY, USA, 2013. ACM.

[NPB+14] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 109–118, July 2014.

[NXP16] NXP. P5020: Qoriq® p5020 and p5010 64-bit dual- and single-core communications processors, Sept. 2016.

[PBB+11] R. Pellizzoni, E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 269–279, April 2011.

[PY16] R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[Riv15] Wind River. Wind river vxworks 653 platform, 2015.

[RTC12] RTCA. DO-178C/ED-12C - software considerations in airborne systems and equipment certification, 2012.

[SCM+16] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016.

[SP17] Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[SPC+11] A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 213–222, April 2011.

[Tea16] Certification Authorities Software Team. *Position Paper CAST-32A Multicore Processors*. US Federal Aviation Administration, Nov. 2016.

[TMW+16] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE 22th*, April 2016.

[VY15] P. K. Valsan and H. Yun. MEDUSA: A predictable and high-performance DRAM controller for multicore based embedded systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 86–93, Aug 2015.

[WP13] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

[WRPR08] Alex Wilson, Wind River, Thierry Preyssler, and Wind River. Incremental certification and integrated modular avionics. *Proc. of 27th IEEE/AIAA Digital Avionics Systems Conf.*, 2008.

[YMWP14] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms.

In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[YPB+11] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. In *Real-Time Systems*. Springer, 2011.

[YPV15] H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195, July 2015.

[YYP+12] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, July 2012.

[YYP+13] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64, April 2013.

[YYP+16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.

[YYW+16] Gang Yao, Heechul Yun, Zheng Pei Wu, R. Pellizzoni, M. Caccamo, and Lui Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *Computers, IEEE Transactions on*, 65(2):601–614, Feb 2016.

List of Acronymns

ASIC	Application-Specific Integrated Circuit
CA	Certification Authority
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
DAL	Design-Assurance Level
DDR	Dual-Data Rate
DRAM	Dynamic Random Access Memory
EASA	European Aviation Safety Agency
ET	Event-Triggered
FAA	Federal Aviation Administration
FPGA	Field-Programmable Gate Array
FR-FCFS	First-Ready First-Come-First-Serve
GPCS	General-Purpose Computing System
HRT	Hard Real-Time
HTAWS	Helicopter Terrain Awareness And Warning System
I/O	Input/output
ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
LLP	Last-Level Private
RR	Round-Robin

RT	Real-Time
SC	Safety-Critical
SCS	Safety-Critical System
SDAW	Sense-Display-And-Warn
SRAM	Static Random Access Memory
SSD	Solid-State Drive
SWaP	Size, Weight and Power
TMR	Triple Modular Redundancy
TT	Time-Triggered
WCET	Worst-Case Execution Time

Summary
Hardware Contention-Aware Real-Time Scheduling on Multi-Core Platforms in Safety-Critical Systems

Computing systems enhance human lives. Broadly, these systems are classified as general-purpose computing systems (GPCSs) like smartphones and safety-critical systems (SCSs) systems like autopilot in airplanes. While failure of a GPCS causes discomfort, failure of a SCS jeopardizes human lives. These SCSs systems are common in transportation domains like aerospace, automotive, railways, etc. In aerospace and railway domains, such system must obtain certification of being safe to to certification authorities (CAs), before commercial use. Certification is a process when a SCS manufacturer demonstrates to a CA predictable behavior of the SCS under all foreseeable scenarios including identified potential risks like guaranteeing safe execution of logically independent applications under contentions on a multi-core platform with shared memory sub-system.

The computing industry significantly benefited from such a shift to multi-core processors for performance gain since 2005. While the computing industry has shifted from single-core processors to multi-core processors for performance gain, SCSs still use single-core processors and are performance constrained.

Chapter I–Introduction

The chapter provides an overview of the emerging SCSs, multi-core processors and hard real-time scheduling. It also presents the problem statements considered in this work and the contributions of this work. The chapter concludes with a dissertation structure.

We now summarize the problem statements. Multi-core processors implicitly share network-on-chip (NoC) and main memory hardware resources among logically-independent cores. As memory bandwidth in multi-core processors is shared across cores, it is a significant cause of performance bottleneck and temporal variability of multiple-orders in task's execution times due to contentions in memory sub-system. Further, the circular dependency is not only between worst-case execution time (WCET) and central pro-

cessing unit (CPU) scheduling of others cores, but also between WCET and memory bandwidth assignments over time to cores. Thus, there is need of solutions that allow tailoring memory bandwidth assignments to workloads over time and computing safe WCET. It is pragmatically infeasible to obtain WCET estimates from static WCET analysis tools for multi-core processors due to the sheer computational complexity involved. SCSs require solutions that enable their transition to multi-core processors while guaranteeing safety, requiring no source-code modifications and substantially reducing re-development and re-certification costs. This dissertation considers the problem of WCET analysis under contentions when deadline-constrained workloads in an independent partitioned workload set execute on a homogeneous multi-core processor with dynamic time-triggered shared memory bandwidth partitioning in SCSs.

Chapter II–Background

This chapter provides an overview of multi-core platform architecture, real-time scheduling and safety-critical systems. It presents a real multi-core platform architecture — the NXP P480. It provides an understanding of the main memory architecture based on the third-generation of the dual-data rate (DDR) dynamic random access memory (DRAM) memory.

Real-time scheduling has been increasingly used for scheduling applications in safety-critical systems as such systems demand applications not only compute a correct result but also deliver it timely to ensure safety. The largest execution time of a task is called worst-case execution time. The process to compute the WCET of a task is known as WCET analysis. In this past, for single-core processors and multiprocessors, the WCET analysis was performed for each task in isolation.

Safety-critical systems are systems that on failure can result in a catastrophe jeopardizing human lives. These systems require performance guarantees under worst-case. So, such systems employ hard real-time scheduling for SCSs. Based on the likelihood and the impact of failures, these system are classified into different criticality-levels —- DAL-A (catastrophic) to DAL-E (no effect). SCS require certification to ensure safety. SCS in avionics consider an application comprising partitions. These system consider time-triggered (TT) scheduling on inter-partition level, also known as ARINC 653 scheduling. It ensures strict temporal and spatial isolation on a single-core processor between any two partitions that provides determinism and eases certification.

Chapter III–System Model and Problem Formulation

We assume a multi-core platform comprises three types of hardware resources: i) *processing resources*, where each resource is referred as a core or CPU, ii) a *memory resource* shared implicitly among cores and iii) monitoring resources that monitor the use of the first two resource types. Next, we present the model of each resource type.

We consider a multi-core model comprising cores, shared memory and monitoring resources. The core model assumes an in-order execution that considers each memory request stalls the issuing core. For the memory request contention model, two models are presented: a constant memory request time and delay model, and a non-decreasing

memory request time model. While the first model caters to multi-core processors that employ pure round-robin (RR) arbitration policy among cores for memory requests, the second model allows using multi-core platform with complex multi-level memory scheduling policies provided memory latencies.

The time model considers a timeline divided into time granules of fixed duration called slots, that are synchronized across cores. The CPu scheduler model assumes a partitioned system, where each core has its own CPU scheduler. The workload model comprises a workload set partitioned across cores. Each workload has an earliest start time, a deadline, a core-local execution time that excludes the time taken by memory requests and the maximum number of memory requests to the memory. The span of a workload must be computed and is the maximum number of slots required to complete core-local execution time and number of memory requests.

The problem formulated is to perform span analysis for a partitioned independent workload set on multi-core processors where homogeneous cores implicitly share memory bandwidth, while ensuring composability such that the completion time of a workload is independent of co-executing workloads while improving schedulability compared to the static memory bandwidth partitioning approaches.

Chapter IV–Resource Servers and Scheduling

In this chapter, first we propose two types of resource servers — CPU server and memory server, with period equal to slot duration. The resource servers to extend resource control and reservation to multiple resources — CPUs and memory — and overcome the circular dependency between WCET and core-level scheduling of the co-executing cores. Each pair of a CPU server and a memory server executes on each core and is released synchronously at the start of each slot. A CPU server instance obtains a CPU budget at each release in CPU time units. A memory server instance obtains a memory budget at each release in memory request units.

Second, we solve the problem of computing a workload's span under even-memory-budget-assignment-to-active-cores changes over time and is known. We described how to construct analytical worst-case memory request pattern. This step helps in solving the problem and has the benefit of significantly less computational complexity compared to using static WCET analysis tools to find such a pattern. Later, we used the analytical worst-case memory request pattern and developed a workload schedulability test useful for an offline scheduler to check if a workload is schedulable under known and changing even-memory-bandwidth-to-core assignment over time. The sorting step dominates the workload schedulability test's computational complexity.

Third, we applied our proposed method with resource servers and dynamic even memory bandwidth assignment to an industrial avionics use-case certified single-core application — helicopter terrain awareness and warning system (HTAWS)— from Airbus. Our proposed method preserved the application's schedule under dynamic even memory budget assignment.

Chapter V–Span Analysis under Dynamic Arbitrary Budgets

This chapter contains two analyses to compute workload WCET in worst-case. The first analysis computes WCET under static memory bandwidth partitioning across cores. The second analysis computes WCET under dynamic memory bandwidth partitioning across cores. Memory stall considers the maximum time taken to perform a memory request time in isolation is fixed and it also denotes the maximum delay due to contention from a single memory request from an another core. We use memory stall curves to illustrate span analysis that eases exposition for the later presented span analysis under dynamic budgets. The key intuition behind iterative re-computation is that the increase in workload span in an iteration is likely to increase the maximum stall in a consecutive iteration due to a different worst-case distribution of memory requests across *(a)* different per memory-stall curves and/or *(b)* different memory scheduling intervals. We then presented a stall maximization algorithm that efficiently implements the fixed-point iterative algorithm for span analysis. It has a complexity similar to a concave optimization problem.

The chapter concludes with an evaluation for an Integrated Modular Avionics (IMA) scenario. It compared dynamic (DY) memory bandwidth assignment policy against two static bandwidth assignment policies — static even (SE) and static uneven (SU) —, under a fixed workload priority order on each core. The comparison metric chosen was schedulability ratios, i.e., the ratio of schedulable workload sets to generated workload sets for each bandwidth assignment policy. The evaluation demonstrates our proposed method under dynamic memory bandwidth allocation dominates static partitioning when partitioned workload sets contain disparate memory-intensive workloads, similar to the HTAWS.

Chapter VI–Related Work

This chapter presents the state-of-the-art. It presents the various white-box and black-box approaches to obtain memory request time. It compares the serializing approaches that divide execution into processing and memory phases. It the presents the related work for span analysis under static memory partitioning. The chapter concludes with differentiating this work from the state-of-the-art.

Chapter VII–Discussion

This chapter presents the discussion on the multi-core processor model, the proposed resource servers, the workload schedulability and span analyses and concludes with the benefits, trade-offs and extension possibilities of our work.

Chapter VIII–Conclusion

The chapter summarizes the main contributions of this dissertation and concludes with future directions. We presented methods — resource servers, workload schedulability test and two span WCET analyses — that enable safe scheduling of safety-critical workloads under static and dynamic memory bandwidth partitioning. We showed that

static memory bandwidth partitioning wastes system performance when CPU-intensive and memory-intensive workloads are assigned to cores. We demonstrated that dynamic memory bandwidth partitioning dominates static partitioning and significantly improves performance in multi-core processors. We summarized two future directions. Our methods can be particularly useful in the early phases of safety-critical (SC) systems design and development in the avionics domain but require integration in offline scheduling tools. Further, our methods can tighten a workload's span analysis when computing span at sub-workload level, where a workload is abstractly split into sub-units called sub-workloads.

Appendices

Appendix A presents the integer linear programming (ILP) constraints for core-local execution time and memory requests. The HTAWS case study uses these constraints to generate CPU schedule and memory schedule under dynamic even memory budget assignment for a dual-core multi-core. Appendix B presents a proof of Theorem V.1 that ensures the iteration in Equation V.10 converges.

Zusammenfassung Hardware-Zugriffskonfliktbewusste Echtzeitaufgabenplanung auf Mehrkernplattformen in safety-kritischen Systemen

Rechnensysteme steigern die Lebensqualität meschliches Leben. Allgemein werden diese Systeme als Universalrechnensysteme (z.B. Smartphones) und Safetykritischesysteme (z.B. Autopiloten in Flugzeugen) klassifiziert. Während der Ausfall von Universalrechnensysteme Unbehagen verursacht, gefährdet der Ausfall von Safetykritischesysteme Menschenleben. Diese Safetykritischesysteme sind im Transportbereich sg. der Luft- und Raumfahrt, die Automobilindustrie, die Eisenbahn usw. üblich. In der Luft- und Raumfahrt und im Bahnbereich muss Zertifizierungsstellen gegenüber nachgewiesen werden, dass ein solches Safetykritischesysteme vor der kommerziellen Verwendung sicher ist. Die Zertifizierung ist ein Prozess, bei dem ein Safetykritischesysteme-Hersteller eine Zertifizierungsstelle vorhersagbares Verhalten der Safetykritischesysteme unter allen vorhersehbaren Szenarien demonstriert, einschließlich identifizierter potenzieller Risiken, wie z.B. die sichere Ausführung logisch unabhängiger Anwendungen unter Zugriffskonflikte auf einer Mehrkern-Plattform mit gemeinsame Hauptspeicherbandbreite.

Die Computerindustrie profitierte seit 2005 deutlich von einer solchen Umstellung auf Mehrkernprozessoren zur Leistungssteigerung. Während sich die Computerindustrie von Single-Core-Prozessoren zu Mehrkernprozessoren verlagert hat, um die Leistung zu steigern, verwenden Safetykritischesysteme immer noch Einkernprozessoren und sind leistungsschwach.

Kapitel I–Einleitung

Das Kapitel gibt einen Überblick über die aufkommenden Safetykritischesysteme, Mehrkernprozessoren und die harte Echtzeitaufgabenplanung. Es werden auch die in dieser Arbeit berücksichtigten Problemstellungen und die Beiträge dieser Arbeit vorgestellt. Das Kapitel schließt mit einer Dissertationsstruktur.

Wir fassen nun die Problemstellungen zusammen. Mehrkernprozessoren teilen sich implizit Netzwerk-on-Chip (NoC) und Hauptspeicher-Hardware-Ressourcen zwischen logisch unabhängigen Kernen. Da die Speicherbandbreite in Mehrkernprozessoren kernübergreifend genutzt wird, ist sie eine wesentliche Ursache für Leistungsengpässe und zeitliche Variabilität mehrerer Ordnungen in den Ausführungszeiten der Aufgabe aufgrund von Konflikten im Speicher-Subsystem. Darüber hinaus besteht die zirkuläre Abhängigkeit nicht nur zwischen Ausführungszeit und der Planung anderer Kerne, sondern auch zwischen Ausführungszeit- und Speicherbandbreitenzuweisungen im Laufe der Zeit an Kerne. Daher besteht Bedarf an Lösungen, die es ermöglichen, die Speicherbandbreitenzuweisungen im Laufe der Zeit an die Workloads anzupassen und einen sicheren Ausführungszeit zu berechnen. Es ist pragmatisch nicht möglich, Ausführungszeit-Schätzungen aus statischen Ausführungszeit-Analysewerkzeugen für Mehrkernprozessoren zu erhalten, da dies mit einer schieren Komplexität verbunden ist. Safetykritischesysteme benötigen Lösungen, die den Übergang zu Mehrkernprozessoren ermöglichen und gleichzeitig die Sicherheit gewährleisten, keine Änderungen am Quellcode erfordern und die Kosten für Neuentwicklung und Rezertifizierung erheblich senken. Diese Dissertation befasst sich mit dem Problem der Ausführungszeit Analyse unter Streitigkeiten, wenn termingebundene Workloads in einem unabhängigen partitionierten Workload-Set auf einem homogenen Mehrkern-Prozessor mit dynamischer zeitgetriggerter Speicherbandbreitenaufteilung in Safetykritischesysteme ausgeführt werden.

Kapitel II–Hintergrund

Dieses Kapitel gibt einen Überblick über die Mehrkernplattformarchitektur, Echtzeit-Planung und sicherheitskritische Systeme. Es präsentiert eine echte Mehrkernplattformarchitektur — den NXP P480. Es liefert ein Verständnis der Hauptspeicherarchitektur, die auf der dritten Generation des dual-data rate (DDR) basiert. dynamic random access memory (DRAM) Erinnerung.

Die Echtzeitplanung wird zunehmend für die Planung von Anwendungen in sicherheitskritischen Systemen eingesetzt, da solche Systeme nicht nur ein korrektes Ergebnis berechnen, sondern es auch rechtzeitig liefern, um die Sicherheit zu gewährleisten. Die größte Ausführungszeit einer Aufgabe wird als Worst-Case-Ausführungszeit bezeichnet. Der Prozess zur Berechnung der Ausführungszeit einer Aufgabe wird als Ausführungszeit-Analyse bezeichnet. In der Vergangenheit wurde bei Single-Core-Prozessoren und Multiprozessoren die Ausführungszeit-Analyse für jede Aufgabe einzeln durchgeführt.

Sicherheitskritische Systeme sind Systeme, die bei einem Ausfall zu einer Katastrophe führen können, die Menschenleben gefährdet. Diese Systeme erfordern Leistungsgarantien im Worst-Case. Solche Systeme verwenden also eine harte Echtzeitplanung für Safetykritischesysteme. Basierend auf der Wahrscheinlichkeit und den Auswirkungen von Ausfällen werden diese Systeme in verschiedene Kritikalitätsstufen eingeteilt —- DAL-A (katastrophal) bis DAL-E (ohne Wirkung). Safetykritischesystem benötigen eine Zertifizierung, um die Sicherheit zu gewährleisten. Safetykritischesystem in der Avionik betrachten eine Anwendung mit Partitionen. Dieses System betrachtet die time-triggered (TT) Planung auf Interpartitionsebene, auch bekannt als ARINC 653 Planung.

Es gewährleistet eine strikte zeitliche und räumliche Trennung auf einem Single-Core-Prozessor zwischen zwei beliebigen Partitionen, die Determinismus bietet und die Zertifizierung erleichtert.

Kapitel III–System Modell und Problemformulierung

Wir gehen davon aus, dass eine Mehrkern-Plattform drei Arten von Hardware-Ressourcen umfasst: i) *processing resources*, wobei jede Ressource als Kern oder central processing unit (CPU) bezeichnet wird, ii) eine *memory resource*, die implizit zwischen den Kernen geteilt wird, und iii) Monitoring-Ressourcen, die die Nutzung der ersten beiden Ressourcentypen überwachen. Als nächstes stellen wir das Modell jedes Ressourcentyps vor.

Wir betrachten ein Mehrkern-Modell, das Kerne, gemeinsamen Speicher und Überwachungsressourcen umfasst. Das Kernmodell geht von einer sequenziellen Ausführung aus, bei der jede Speicheranforderung den ausgebenden Kern blockiert. Für das Konkurrenzmodell für Speicheranforderungen werden zwei Modelle vorgestellt: ein Modell für konstante Speicheranforderungen und Verzögerungen und ein Modell für nicht abnehmende Speicheranforderungen. Während das erste Modell Mehrkernprozessoren unterstützt, die reine round-robin (RR) Arbitrierungsrichtlinien zwischen den Kernen für Speicheranforderungen verwenden, ermöglicht das zweite Modell die Verwendung einer Mehrkernplattform mit komplexen mehrstufigen Speicherplanungsrichtlinien, die Speicherlatenzen bereitstellen.

Das Zeitmodell berücksichtigt eine Zeitachse, die in Zeitgranulate fester Dauer unterteilt ist, die als Slots bezeichnet werden und kernübergreifend synchronisiert sind. Das CPu-Scheduler-Modell geht von einem partitionierten System aus, bei dem jeder Kern seinen eigenen CPU-Scheduler hat. Das Workload-Modell umfasst einen Workload-Set, der auf mehrere Kerne verteilt ist. Jeder Workload hat eine früheste Startzeit, eine Deadline, eine Kern-Lokal-Ausführungszeit, die die Zeit, die von Speicheranforderungen benötigt wird, und die maximale Anzahl von Speicheranforderungen an den Speicher ausschließt. Die Spanne eines Workloads muss berechnet werden und ist die maximale Anzahl von Slots, die benötigt werden, um die Kern-Lokal-Ausführungszeit und die Anzahl der Speicheranforderungen zu vervollständigen.

Das formulierte Problem besteht darin, eine Spannenanalyse für eine partitionierte unabhängige Arbeitslast durchzuführen, die auf Mehrkernprozessoren eingestellt ist, bei denen homogene Kerne implizit die Speicherbandbreite gemeinsam nutzen, während die Kompostierbarkeit so gewährleistet ist, dass die Fertigstellungszeit einer Arbeitslast unabhängig von der gleichzeitigen Ausführung von Arbeitslasten ist, während die Planbarkeit im Vergleich zu den Ansätzen zur statischen Speicherbandbreitenverteilung verbessert wird.

Kapitel IV–Ressourcenserver und Scheduling

In diesem Kapitel schlagen wir zunächst zwei Arten von Ressourcen-Servern vor — CPU-Server und Speicherserver, mit einem Zeitraum, der der Slot-Dauer entspricht. Die Ressourcenserver zur Erweiterung der Ressourcensteuerung und -reservierung auf mehrere

Ressourcen — CPUs und Speicher — und zur Überwindung der zirkulären Abhängigkeit zwischen Ausführungszeit und der Planung der mitausführenden Kerne auf Kernebene. Jedes Paar eines CPU-Servers und eines Speicherservers führt auf jedem Kern aus und wird synchron zu Beginn jedes Steckplatzes freigegeben. Eine CPU-Server-Instanz erhält bei jedem Release ein CPU-Budget in CPU-Zeiteinheiten. Eine Memory-Server-Instanz erhält bei jeder Freigabe in Speicheranforderungseinheiten ein Speicherbudget.

Zweitens lösen wir das Problem der Berechnung der Spannweite einer Workloads unter gleichmäßigen Änderungen der Speicher-Budget-Zuordnung zu aktiven Kernen im Laufe der Zeit und sind bekannt. Wir haben beschrieben, wie man analytische Worst-Case-Speicheranforderungsmuster konstruiert. Dieser Schritt hilft bei der Lösung des Problems und hat den Vorteil einer deutlich geringeren Rechenkomplexität im Vergleich zur Verwendung statischer Ausführungszeit-Analysewerkzeuge zur Ermittlung eines solchen Musters. Später verwendeten wir das analytische Worst-Case-Speicheranforderungsmuster und entwickelten einen Workload-Schedulability-Test, der für einen Offline-Scheduler nützlich war, um zu überprüfen, ob ein Workload unter bekannten und sich ändernden Even-Memory-Bandbreite zu Core Zuweisungen im Laufe der Zeit planbar ist. Der Sortierschritt dominiert die Rechenkomplexität des Workload Schedulability Tests.

Drittens haben wir unsere vorgeschlagene Methode mit Ressourcenservern und dynamischer Bandbreitenzuweisung sogar auf eine industrielle Avionik-Anwendung angewendet - eine zertifizierte Single-Core-Anwendung — helicopter terrain awareness and warning system (HTAWS)— von Airbus. Unsere vorgeschlagene Methode bewahrte den Zeitplan der Anwendung unter dynamischer, gleichmäßiger Speicherzuweisung.

Kapitel V–Zeitspannenanalyse unter Dynamischesbudget

Dieses Kapitel enthält zwei Analysen zur Berechnung der Arbeitslast Ausführungszeit im Worst-Case. Die erste Analyse berechnet Ausführungszeit unter statischer Speicherbandbreitenaufteilung über die Kerne hinweg. Die zweite Analyse berechnet Ausführungszeit unter dynamischer Speicherbandbreitenaufteilung über die Kerne hinweg. Der Speicherstand berücksichtigt die maximale Zeit, die benötigt wird, um eine Speicheranforderungszeit isoliert auszuführen, ist festgelegt und bezeichnet auch die maximale Verzögerung aufgrund von Konflikten mit einer einzelnen Speicheranforderung von einem anderen Kern. Wir verwenden Memory-Stall-Kurven, um die Spannenanalyse zu veranschaulichen, die die Darstellung für die später vorgestellte Spannenanalyse unter dynamischen Budgets erleichtert. Die wichtigste Intuition hinter der iterativen Neuberechnung ist, dass die Erhöhung der Arbeitsbelastungsspanne in einer Iteration ist es wahrscheinlich, dass der maximale Strömungsabriss in einer aufeinander folgenden Zeitspanne erhöht wird. Iteration aufgrund einer unterschiedlichen Worst-Case-Verteilung von Speicheranforderungen über die gesamte Bandbreite hinweg. *(a)* unterschiedliche Kurven pro Speicherplatz und/oder *(b)* unterschiedliche Speicherplanungsintervalle. Wir präsentierten dann einen Stall-Maximierungsalgorithmus, der den Festkomma-Iterationsalgorithmus für die Spannenanalyse effizient implementiert. Es hat eine Komplexität, die einem konkaven Optimierungsproblem ähnlich ist.

Das Kapitel schließt mit einer Bewertung für ein Integrated Modular Avionics (IMA) Szenario. Es verglich die dynamische (DY) Speicherbandbreitenzuweisungsrichtlinie mit

zwei statischen Bandbreitenzuweisungsrichtlinien — statisch gerade (SE) und statisch ungleich (SU) —, unter einer festen Reihenfolge der Arbeitslastprioritäten auf jedem Kern. Die gewählte Vergleichsmetrik war die Planbarkeitsquote, d.h. das Verhältnis von planbaren Workload-Sets zu generierten Workload-Sets für jede Bandbreitenzuordnungsrichtlinie. Die Auswertung zeigt unser vorgeschlagenes Verfahren unter dynamischer Speicherbandbreitenzuweisung. dominiert die statische Partitionierung, wenn partitionierte Workload-Sets unterschiedliche speicherintensive Workloads enthalten, ähnlich wie die HTAWS.

Kapitel VI–Themenbezogene Fachliteratur

In diesem Kapitel wird der Stand der Technik vorgestellt. Es werden die verschiedenen White-Box- und Black-Box-Ansätze vorgestellt, um die Speicheranforderungszeit zu erhalten. Es vergleicht die Serialisierungsansätze, die die Ausführung in Verarbeitungs- und Speicherphasen unterteilen. Es werden die entsprechenden Arbeiten zur Spannenanalyse bei der statischen Speicherpartitionierung vorgestellt. Das Kapitel schließt mit der Unterscheidung dieser Arbeit vom Stand der Technik.

Kapitel VII–Diskussion

Dieses Kapitel stellt die Diskussion über das Mehrkern-Prozessormodell, die vorgeschlagenen Ressourcenserver, die Workload-Planbarkeits- und Spannweitenanalysen vor und schließt mit den Vorteilen, Kompromissen und Erweiterungsmöglichkeiten unserer Arbeit.

Kapitel VIII–Fazit

Das Kapitel fasst die wichtigsten Beiträge dieser Dissertation zusammen und schließt mit zukünftigen Richtungen. Wir präsentierten Methoden — Ressourcenserver, Workload Schedulability Test und zwei Span-Ausführungszeit-Analysen —, die eine sichere Planung sicherheitskritischer Workloads unter statischer und dynamischer Speicherbandbreitenpartitionierung ermöglichen. Wir haben gezeigt, dass die Partitionierung der statischen Speicherbandbreite die Systemleistung verschlechtert, wenn CPU-intensive und speicherintensive Workloads den Cores zugewiesen werden. Wir haben gezeigt, dass die dynamische Speicherbandbreitenpartitionierung die statische Partitionierung dominiert und die Leistung in Mehrkernprozessoren deutlich verbessert. Wir haben zwei zukünftige Richtungen zusammengefasst. Unsere Methoden können besonders in den frühen Phasen des Systemdesigns und der Systementwicklung im Avionikbereich nützlich sein, erfordern aber die Integration in Offline-Terminierungswerkzeuge. Darüber hinaus können unsere Methoden die Spannenanalyse eines Workloads straffen, wenn der Span auf Sub-Workload-Ebene berechnet wird, bei der ein Workload abstrakt in Untereinheiten, sogenannte Sub-Workloads, unterteilt wird.

Anhänge

Anhang A stellt die integer linear programming (ILP) Einschränkungen für die KernLokalausführungszeit und Speicheranforderungen dar. Die Fallstudie HTAWS. verwendet diese Einschränkungen, um CPU-Zeitplan und Speicherplan unter dynamischer, gleichmäßiger Speicherbudgetzuweisung für einen Zweikernplattform zu erzeugen. Anhang B stellt einen Beweis für das Theorem V.1 dar, der sicherstellt, dass die Iteration in Gleichung V.10 konvergiert.

Ankit Agrawal

Erwin-Schrödinger Straße 12
TU Kaiserslautern
67663 Kaiserslautern, Germany
agrawal@eit.uni-kl.de

Education

2014–2019	**Ph.D.**	in **Real-time Multi-core Scheduling** at the *Chair of Real-Time Systems*, **University of Kaiserslautern**, Germany.
2011–2014	**Master of Science** (EMECS)	in **Embedded Computing Systems** from the **University of Kaiserslautern**, *Germany* (year 2) & the **University of Southampton**, U.K. (year 1). *Thesis*: Applicability of Offline Scheduling Algorithms in Distributed Hard Real-Time Systems to Multicore Architectures.
2008–2011	**Master of Business Administration**	in **Healthcare Management** from the *Manipal Institute of Management*, **Manipal University**, India.
2006–2010	**Bachelor of Engineering**	in **Information Technology** from the *Manipal Institute of Technology*, **Manipal University**, India.

Employment

2014–2019	**Research Assistant**	at the *Chair of Real-Time Systems*, **University of Kaiserslautern**, Germany.
2013–2014	**Student Assistant**	at the *Chair of Real-Time Systems*, **University of Kaiserslautern**, Germany.
2010–2010	**Intern**	at **eSATURNUS NV** in Leuven, Belgium.
2009–2009	**Intern**	at **IFU GmbH Privates Institut für Umweltanalysen** in Chemnitz, Germany.

Service

Mentor and Volunteer at the **ArbeiterKind Kaiserslautern** since March 2018 to support children from non-academic families for higher education

Volunteer at the *ECRTS Program Committee Meeting* in 2015 and 2016

Secondary Reviewer for papers in the following conferences:

IEEE Real-Time Systems Symposium (RTSS)	*2014, 2015, 2018*
ACM Symposium on Applied Computing (SAC)	*2018*
Euromicro Conference on Real-Time Systems (ECRTS)	*2014, 2016, 2017*
Design, Automation and Test in Europe Conference (DATE)	*2017*
IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)	*2016*
IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)	*2014, 2015*

Teaching Experience

Fall 2018, 2017, 2016, 2015, 2014	Module: Real-Time Systems 2	**Presented lectures**: Multiprocessor Scheduling 2; Multi-core Scheduling in Avionics
Spring 2018, 2017, 2016, 2015	Module: Real-Time Systems 1	**Presented lecture**: Multiprocessor Scheduling 1
Spring 2018, 2017, 2016, 2015	Module: Seminar Real-Time Systems	**Primary Instructor**
Fall 2018	Module: OS	**Presented exercise**: Processes & Threads

Technical Skills

Programming/Scripting Languages:	C, C++, ASM (PowerPC), Python, Bash
Hardware Debugging Tools:	Lauterbach, CodeWarrior
Kernel Development:	Linux-like OSes for Intel and PowerPC

Language Skills

English: Fluent	**German**: Very good *(Level: B2)*	**Hindi**: Fluent