ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

COMCOS: AN ENHANCED CACHE PARTITIONING TECHNIQUE FOR INTEGRATED MODULAR AVIONICS

M.Sc. THESIS

Yakup HÜNER

Department of Defence Technologies

Defence Technologies Programme

JUNE 2023

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

COMCOS: AN ENHANCED CACHE PARTITIONING TECHNIQUE FOR INTEGRATED MODULAR AVIONICS

M.Sc. THESIS

Yakup HÜNER
(514201037)

Department of Defence Technologies

Defence Technologies Programme

Thesis Advisor: Asst. Prof. Ramazan YENİÇERİ

JUNE 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

COMCOS: ENTEGRE MODÜLER AVİYONİKLER İÇİN GELİŞMİŞ BİR ÖNBELLEK BÖLÜMLEME TEKNİĞİ

YÜKSEK LİSANS TEZİ

Yakup HÜNER
(514201037)

Savunma Teknolojileri Anabilim Dalı

Savunma Teknolojileri Programı

Tez Danışmanı: Asst. Prof. Ramazan YENİÇERİ

HAZİRAN 2023

Yakup HÜNER, a M.Sc. student of ITU Graduate School student ID 514201037 successfully defended the thesis entitled "COMCOS: AN ENHANCED CACHE PARTITIONING TECHNIQUE FOR INTEGRATED MODULAR AVIONICS", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Asst. Prof. Ramazan YENİÇERİ** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Sıddıka Berna ÖRS YALÇIN** Istanbul Technical University	.....
	Dr. Etem DENİZ TÜBİTAK BİLGEM	.....

Date of Submission : **23 May 2023**
Date of Defense : **14 June 2023**

To my spouse,

FOREWORD

As someone who values innovation, I wanted my professional efforts to be recognized in the academic field. With this purpose in mind, I conducted this thesis work to develop a new perspective on the topic of "Cache Partitioning." I hope that my work will be well-received both academically and in the business world.

First of all, I would like to express my heartfelt gratitude to my advisor Assoc. Prof. Dr. Ramazan YENİÇERİ for his guidance and support throughout my academic journey. His mentorship and inspiring vision have greatly contributed to my academic development and achievements.

I would also like to extend my sincere thanks to TÜBİTAK BİLGEM for providing me with the resources that facilitated my research and academic endeavors.

Furthermore, I would like to thank my dear spouse and family. Their unwavering support, patience, and understanding have been a constant source of strength and motivation during the challenges of my studies. Their love and encouragement have played a significant role in my success, and I am truly grateful to them.

Once again, I extend my heartfelt thanks to everyone who has played a part in shaping my academic journey and making my research efforts possible.

June 2023

Yakup HÜNER
(Computer Engineer)

TABLE OF CONTENTS

ABBREVIATIONS

EASA : European Union Aviation Safety Agency
FAA : Federal Aviation Administration
I/O : Input/Output
IMA : Integrated Modular Avionics
API : Application Programming Interface
ARINC : Aeronautical Radio, Incorporated
CAST : Certification Authorities Software Team
AMC : Acceptable Means of Compliance
ComCoS : Combined Color Stacks
OS : Operating System
SRAM : Static Random-Access Memory
DRAM : Dynamic Random-Access Memory
APEX : Application Executive
DAL : Design Assurance Level
RTCA : Radio Technical Commission for Aeronautics
AFDX : Avionics Full-Duplex Switched Ethernet
PMC : Performance Monitor Counter
WCET : Worst-Case Execution Time
QoS : Quality Of Service
RTOS : Real-Time Operating System
CAST : Certification Authorities Software Team
AMC : Acceptable Means of Compliance

SYMBOLS

N **:** Number of Cache Pools

LIST OF TABLES

Page

LIST OF FIGURES

Page

COMCOS: AN ENHANCED CACHE PARTITIONING TECHNIQUE FOR INTEGRATED MODULAR AVIONICS

SUMMARY

Integrated Modular Avionics (IMA) has been widely used in safety-critical aviation applications due to its reusability, portability, modularity, and cost-effective re-certification. IMA-based systems effectively manage numerous applications with varying levels of criticality by utilizing shared hardware and middleware supported by hardware-independent application programming interfaces (APIs), such as the ARINC 653 (Aeronautical Radio Incorporated) interfaces. It enables integrating multiple applications with varying criticality levels through robust time and memory partitioning. The standardized and well-defined software architecture of ARINC 653 facilitates the development of safety-critical systems by ensuring a clear separation of software functions within their designated partitions. This separation minimizes the risk of interference or failure, enhancing the overall reliability of the systems.

To ensure the reliability of aviation systems in all operational conditions, they are subject to strict safety standards and regulations, such as the DO-178C software guidance. Compliance with DO-178C requires detailed documentation of the software development process, including planning, design, coding, testing, and verification, to ensure that the software meets the required level of safety and reliability. Analyzing the system's worst-case execution Time (WCET), limitations, and utilization is recommended in the DO-178C software verification process. WCET measures a software task's maximum completion time under worst-case conditions.

In IMA-based systems, applications are typically segmented into distinct memory and time partitions to ensure they have the necessary resources to execute consistently and reliably. However, shared resources such as cache, busses, and memory can still impact the WCET of the applications on multi-core platforms, even partitioned. These shared resources can lead to contention and may cause delays in the execution of applications, affecting the system's determinism and overall performance. To address this issue, CAST-32A (Certification Authorities Software Team) and AMC 20-193 (Acceptable Means of Compliance) guidance has been published for certifying multi-core avionics software.

These shared resource contentions can be mitigated through Cache Partitioning, which involves allocating different applications to separate cache regions. There are two main approaches to cache partitioning: hardware-based and software-based. Software-based cache partitioning is more suitable for IMA-based systems due to its hardware independence, ability to provide finer granularity, and the capability to configure it based on specific application requirements.

This study conducted numerous experiments to demonstrate the improvements achieved by cache partitioning in mitigating interferences between applications. The

first example demonstrates that the execution time of the matrix multiplication application increased by up to 101,44% due to the trashing effect caused by another application. Implementing cache partitioning can reduce this effect by up to 0,6%. In another experiment, a matrix multiplication operation experienced a significant 2483% increase in execution time due to bus interference, while cache partitioning reduced this increase by 30%.

The use of a shared cache can cause both determinism and performance degradation not only in multi-core systems but also in single-core systems. Although cache flushing during application switching eliminates determinism problems in single-core systems, this situation leads to performance degradation. The performance degradation caused by cache flushing, resulting in a loss of 47,4%, was reduced to as low as 0,314% through the implementation of cache partitioning.

In safety-critical avionics systems, physical memory for application program areas is fully allocated during application loading. Memory management algorithms for real-time systems are divided into static memory allocation and dynamic application allocation. The CAST-32A discourages dynamic memory allocation and recommends that applications allocate memory at system startup. In static memory allocation, all cache regions are preloaded during system startup to lists, and applications acquire memory from these preloaded lists during system initialization. However, this approach can significantly increase the system's boot time. While the CAST-32A guidelines do not recommend dynamic memory allocation, the AMC 20-193 directives allow for dynamic allocation under specific deterministic conditions and limits. In large-scale IMA systems, there may be a need for dynamic loading of applications due to requirements such as load balancing or transferring an application from a faulty card to other cards. However, it is crucial to ensure that the system's other applications are unaffected from dynamic loading.

Hundreds of applications work in a complex structure in IMA-based aviation systems. Each application has unique cache characteristics and requires a different cache size. Applications can be assigned to multiple cache regions. However, in existing cache allocation methods described in the literature, cache requests are provided page by page, even when applications reside in contiguous cache regions. To overcome this limitation, a new ComCoS (Combined Color Stacks) technique has been developed to allocate multiple cache pools in a single request. Instead of keeping physically adjacent cache regions in separate stacks, they are placed together in a shared stack. That enables the allocation of multiple cache regions in a single service request. The ComCoS reduces system startup time for systems that prefer static allocation and offers faster and more deterministic cache allocation for systems favoring dynamic application allocation.

Experimental results on an ARINC 653 compatible RTOS demonstrate that the ComCoS technique provides an average performance improvement of 52% and reduces the standard deviation in memory distribution by 2,91 times. Additionally, the technique achieves a 3,48 times improvement in WCET and a 6,23 times reduction in standard deviation for the memory allocation service.

COMCOS: ENTEGRE MODÜLER AVİYONİKLER İÇİN GELİŞMİŞ BİR ÖNBELLEK BÖLÜMLEME TEKNİĞİ

ÖZET

Havacılık endüstrisi geleneksel olarak sıkı güvenlik ve güvenilirlik standartlarına uyan karmaşık aviyonik sistemlere dayanmaktadır. Geçmişte, aviyonik sistemler her bir bileşenin ayrı bir değiştirilebilir hat birimi olarak çalıştığı Federatif Mimari yaklaşımını takip etmiştir. Değiştirilebilir hat birimleri kendi işlemcileri, giriş/çıkış (G/Ç) cihazları ve haberleşme arayüzlerine sahiptirler ve haberleşme kanallarını kullanarak iletişim kurmaktadırlar. Ancak, havacılık teknolojisinin ilerlemesi ve artan karmaşıklık düzeyi nedeniyle Federatif Mimari yaklaşımını kullanarak uçak tasarımlarını yönetmek, giderek zorlaşmıştır. Entegre Modüler Aviyonik (IMA: Integrated Modular Avionics) mimarisi, bu zorlukların üstesinden gelmek için alternatif olarak önerilmiştir.

IMA mimarisi, daha az merkezi işlem birimi kullanarak birden fazla donanım modülünü yönetir ve bunu donanım bağımsız uygulama programlama arayüzleri (API: Application Program Interface) aracılığıyla başarır. Bu yaklaşım, alt sistemlerin işlem gücü, veri depolama ve haberleşme gibi aviyonik kaynakların paylaşılmasını sağlar. Böylece kaynak kullanımında, sistem boyutunda ve ağırlığında azalma sağlanır. Boeing, 787 Dreamliner uçağının tasarımının IMA olarak gerçekleştirilmesi ile uçak aviyoniğinde 2000 libre azalma yaşandığını söylemektedir.

IMA mimarisi tekrar kullanılabilirlik, taşınabilirlik, modülerlik ve maliyet etkin sertifikasyon özellikleri sayesinde aviyonik dünyada popüler hale gelmiştir. Bu mimari yaklaşım, sistem tasarımı ve bakımında esneklik sağlarken, yeni uygulamaların daha kolay entegrasyonunu ve aviyonik kaynaklarının daha verimli kullanımını mümkün kılar. Bu sayede daha etkin bir şekilde kaynak kullanımı sağlanır ve sertifikasyon maliyetleri düşer.

IMA mimarili sistemler genellikle INTEGRITY ve TÜBİTAK GIS gibi gerçek zamanlı işletim sistemlerini kullanırlar. Aviyonik dünyada kullanılan gerçek zamanlı işletim sistemler, ARINC 653 (Aeronautical Radio Incorporated) gibi donanım bağımsız arayüzler kullanarak uygulamalar ile iletişim kurmaktadırlar. ARINC 653 katı zaman ve hafıza paylaşımı yaklaşımı sayesinde farklı kritiklik seviyelerindeki uygulamaların bir arada, birbirini etkilemeden çalışabilmesini sağlar. ARINC 653'ün standartlaştırılmış ve iyi tanımlı yazılım mimarisi, yazılım uygulamaların belirlenen bölmeler içinde katı bir şekilde ayrılmasını sağlayarak güvenlik açısından kritik sistemlerin geliştirilmesini kolaylaştırır. Bu ayrım, müdahale veya arıza riskini en aza indirir ve sistemlerin genel güvenilirliğini artırır. Ayrıca, ARINC 653 sertifikasyon maliyetlerini azaltmaya yardımcı olur. Sertifikasyon sürecini daha hızlı ve hatalara daha az duyarlı hale getirir.

Havacılık sistemleri, tüm koşullar altında güvenilirliğini sağlamak için sıkı güvenlik standartlarına tabi tutulurlar. Avrupa Birliği Havacılık Emniyeti Ajansı (EASA: European Union Aviation Safety Agency) ve Federal Havacılık İdaresi (FAA: Federal Aviation Administration), havacılık endüstrisinde en önemli düzenleyici kurumlardan ikisidir. Temel görevleri, uçak sertifikasyonu ve işletmesi için düzenlemeler ve standartlar oluşturarak ve bunları uygulayarak hava seyahatinin güvenliğini sağlamaktır. Bu süreçler, DO-178C rehber dokümanı gibi çeşitli düzenlemelere ve standartlara uyum gerektirir. DO-178C emniyet kritik sistem yazılımlarının planlama, tasarım, kodlama, test ve doğrulama gibi tüm yaşam döngüsünü kapsamaktadır. Uygulamaları ölümcül, tehlikeli, önemli, az önemli ve etkisiz olarak Tasarım Güvence Seviyelerine (DAL: Design Assurance Level) ayırmaktadır ve her biri için sağlaması gereken hedef listesi tanımlamaktadır. DO-178C yazılım doğrulama süreci, sistemin en kötü işlem süresinin (WCET: Worst-Case Execution Time), sınırlarının ve kaynak kullanımının analiz edilmesini tavsiye etmektedir. Sistemdeki kaynak paylaşımı en kötü işlem süresine göre yapılmaktadır.

IMA mimarili sistemlerde uygulamalar genellikle katı zaman ve hafıza bölmelerine ayrılsalar da özellikle çok çekirdekli işlemcilerde ortak kullanılan önbellek, veriyolu ve hafıza birimleri ile birbirlerinin WCET'ini etkilemektedirler. Bu paylaşılan kaynaklar arasındaki çekişme, sistem determinizmini ve performansını etkileyebilir. Çok çekirdekli aviyonik yazılımların sertifikasyonu için CAST-32A ve AMC 20-193 rehber dokümanları yayınlanmıştır.

Ortak kaynak paylaşım problemi önbellek bölümleme ile giderilebilir. Bu yöntemde uygulamalar farklı önbellek bölgelerine yerleştirilerek, birbirlerinin önbellekdeki verilerini tahliye etmezler. Önbellek bölümleme için donanım tabanlı ve yazılım tabanlı olmak üzere iki ana yaklaşım bulunmaktadır. Donanım tabanlı bölümlendirme, donanıma bağımlılığı ve düşük bölümleme çözünürlüğü nedeniyle IMA tabanlı sistemler için uygun değildir. Uygulamalar sahip oldukları önbellek miktarlarına göre farklı davranış göstermektedirler. Bu yüzden her uygulamanın talep edeceği önbellek miktarı farklı olacaktır. Ancak uygulamaların derleme anında ihtiyacı olan önbellek miktarını belirlenmesi zordur. Uygulamalara verilen önbellek miktarının dinamik olarak belirleneceği pek çok çalışma yapılmıştır. Ancak, emniyet kritik sistemler için, uygulamaya tahsis edilen önbellek bölgelerinin sistem yürütülmesi sırasında dinamik olarak yer değiştirmesi istenmeyen bir dinamik davranıştır. Bu nedenle, uygulamaya tahsis edilecek önbellek bölgelerinin yapılandırılması statik olarak belirlenmelidir.

Bu çalışmada, önbellek bölümlemenin uygulamalar arasındaki girişimleri azaltmada sağladığı gelişmeleri göstermek için birçok deney gerçekleştirilmiştir. Gerçekleştirilen ilk deneyde, bir matris çarpma uygulamasının başka bir uygulamanın neden olduğu önbellek boşaltma etkisi nedeniyle yürütme süresinin %101,44'e kadar arttığı gözlemlenmiştir. Önbellek bölümlemenin bu etkiyi %0,6'ya kadar azaltabileceği görülmüştür. Başka bir deneyde, bir matris çarpma işlemi, veriyolunda oluşan girişim nedeniyle yürütme süresinde %2483'lük önemli bir artış yaşamış, ancak önbellek bölümleme bu artışı %30 oranına kadar azaltmaktadır. Önbellek bölümleme, veriyolu girişim problemini doğrudan çözmese de girişimin etkisini büyük ölçüde azalttığı gözlemlenmiştir. Veriyolu girişim problemi uygulamaların belirlenen zaman dilimi

içerisinde, tahsis edilen işlem kapasitesini aşması durumunda sonlandırılması ile önlenebilir.

Paylaşılan önbellek kullanımı, sadece çok çekirdekli sistemlerde değil, tek çekirdekli sistemlerde de determinizm ve performans bozulmalarına neden olabilir. Tek çekirdekli sistemlerde uygulama geçişleri sırasında önbellek boşaltılması determinizm sorunlarını ortadan kaldırsa da bu durum performans kaybına yol açar. Önbellek boşaltılması %47,4'lük bir performans kaybına sebep olurken, önbellek bölümleme sayesinde bu kayıp %0,314'e kadar azaltılmıştır.

Talep üzerine sayfa alma, sayfaların yalnızca ihtiyaç duyulduğunda ana belleğe yükleyen bir bellek yönetimi düzenidir. Bu yöntemde, uygulama sadece ihtiyaç duyduğunda ikincil depolama alanından (örneğin bir sabit disk) ana belleğe yüklenir. Bu teknik, yalnızca gerekli olan sayfaların belleğe yüklenmesine izin vererek bellek kaynaklarının daha verimli kullanılmasını sağlar. Kullanılmayan sayfalar gerektiği zamana kadar ikincil depolama aygıtında kalabilir. Talep üzerine sayfalandırma, Linux gibi geleneksel işletim sistemlerinde kullanılarak, programların fiziksel olarak mevcut olan bellekten daha fazla bellek kullanmasına olanak tanır. Ancak bu yöntem gerçek zamanlı işletim sistemleri için uygun değildir, çünkü gerekli sayfalar ikincil hafızadan geç getirilebilir veya ana hafızada hiç yer kalmamışsa getirilemeyebilir. Bu durum determinizm problemi yarattığı için, emniyet kritik aviyonik sistemlerde uygulamalar yürütülmeye başlamadan önce tamamıyla ana hafızaya yüklenmektedirler.

Emniyet kritik sistemlerde hafıza yönetimi statik ve dinamik olmak üzere iki grupta incelenebilir. CAST-32A, dinamik bellek tahsisini teşvik etmez ve uygulamaların belleği sistem başlatıldığında tahsis etmelerini önerir. Statik bellek tahsisinde, tüm önbellek bölgeleri sistemin başlatılması sırasında listelere önceden doldurulur. Uygulamalar ana belleğe yüklenirken, önceden doldurulmuş bu listelerden önbellek bölgeleri tahsis edilir. Bu yaklaşımda tüm sistem kaynakları sistem ilklenmesi sırasında tahsis edileceği için, sistem başlatma süresini önemli ölçüde artırabilir. CAST-32A yönergeleri dinamik bellek tahsisini önermezken, AMC 20-193 direktifleri belirli deterministik koşullar ve sınırlar altında dinamik bellek tahsisine izin verir. Büyük ölçekli IMA mimarili sistemlerde, yük dengelemesi yapmak veya arızalanan bir karttaki uygulamanın başka kartlara taşınması gibi ihtiyaçlardan dolayı uygulamaların dinamik olarak yüklenmeleri gerekebilir. Ancak bu dinamik yükleme sırasında sistem içerisinde halihazırda çalışmakta olan uygulamaların çalışmasını kesinlikle etkilememelidir.

Her uygulamanın benzersiz önbellek özellikleri vardır ve farklı bir önbellek boyutuna ihtiyaç duyar. Bu yüzden uygulamalar birden çok önbellek bölgesine atanabilirler. Literatürde tanımlanan mevcut önbellek tahsis yöntemlerinde, uygulamalar bitişik önbellek bölgelerinde yer alsalar bile önbellek istekleri sayfa sayfa sağlanmaktadır. Uygulamaların önbellek karakteristiğini korumak adına talep edilen farklı önbellek bölgeleri sırası ile dağıtılmaktadır. Bu tezde, tek bir istekte birden çok önbellek havuzunun tahsis edilebilmesini sağlayan, ComCoS (Combined Color Stacks, Birleşik Renk Yığınları) isimli yeni bir önbellek tahsis tekniği geliştirilmiştir. Fiziksel olarak bitişik yer alan önbellek bölgeleri ayrı ayrı yığınlarda tutulmaz; bunun yerine, birlikte paylaşılan bir yığına yerleştirilirler. Bu sayede, tek bir servis isteğinde birden fazla önbellek bölgesinin tahsisine olanak sağlanır. ComCoS, statik tahsisi tercih eden

sistemler için sistem başlatma süresini azaltırken ve dinamik uygulama tahsisini tercih eden sistemler için daha hızlı ve deterministik bir önbellek tahsisi sunar.

ARINC 653 uyumlu bir gerçek zamanlı işletim sistemi üzerinde yapılan deneysel sonuçlar, ComCoS tekniğinin ortalama performansı %52 artırdığını ve bellek dağılımındaki standart sapmayı 2,91 kat azalttığını göstermektedir. Ayrıca, ComCoS bellek tahsis hizmeti için en kötü işlem süresinde 3,48 kat iyileşme ve standart sapmada 6,23 kat azalma sağlamaktadır.

1. INTRODUCTION

Aviation systems are safety critical because any malfunction or error in these systems could lead to catastrophic consequences, including loss of life and property damage. To ensure the reliability of aviation systems in all operational conditions, they are subject to strict safety standards and regulations. European Union Aviation Safety Agency (EASA) and Federal Aviation Administration (FAA) are two of the most important regulatory bodies in the aviation industry. Their primary role is to ensure the safety and security of air travel by establishing and enforcing regulations and standards for aircraft certification and operation. EASA regulates aviation safety in Europe, while the FAA oversees aviation safety in the United States. These processes involve complying with various regulations and standards, such as the DO-178C software guidance [1]. This standard sets guidelines for the developing software used in avionics systems. The DO-178C requires detailed documentation of the software development process, including planning, design, coding, testing, and verification, to ensure that the software meets the required level of safety and reliability. Certification authorities evaluate this documentation to ensure the software meets safety requirements.

Historically, aviation systems have employed a Federated Architecture approach, wherein each system component is a separate line replaceable unit with its own processors, input/output (I/O) devices, and communications [3]. However, managing aircraft designs using this approach has become increasingly challenging with the advancement and increasing complexity of aviation technology. To address this issue, the Integrated Modular Avionics (IMA) architecture has been proposed as an alternative to Federated Avionics. The IMA approach manages multiple hardware modules using fewer central processing units, accomplished through hardware-independent application programming interfaces (API). Figure 1.1 demonstrates the comparison of Federated and IMA architectures.

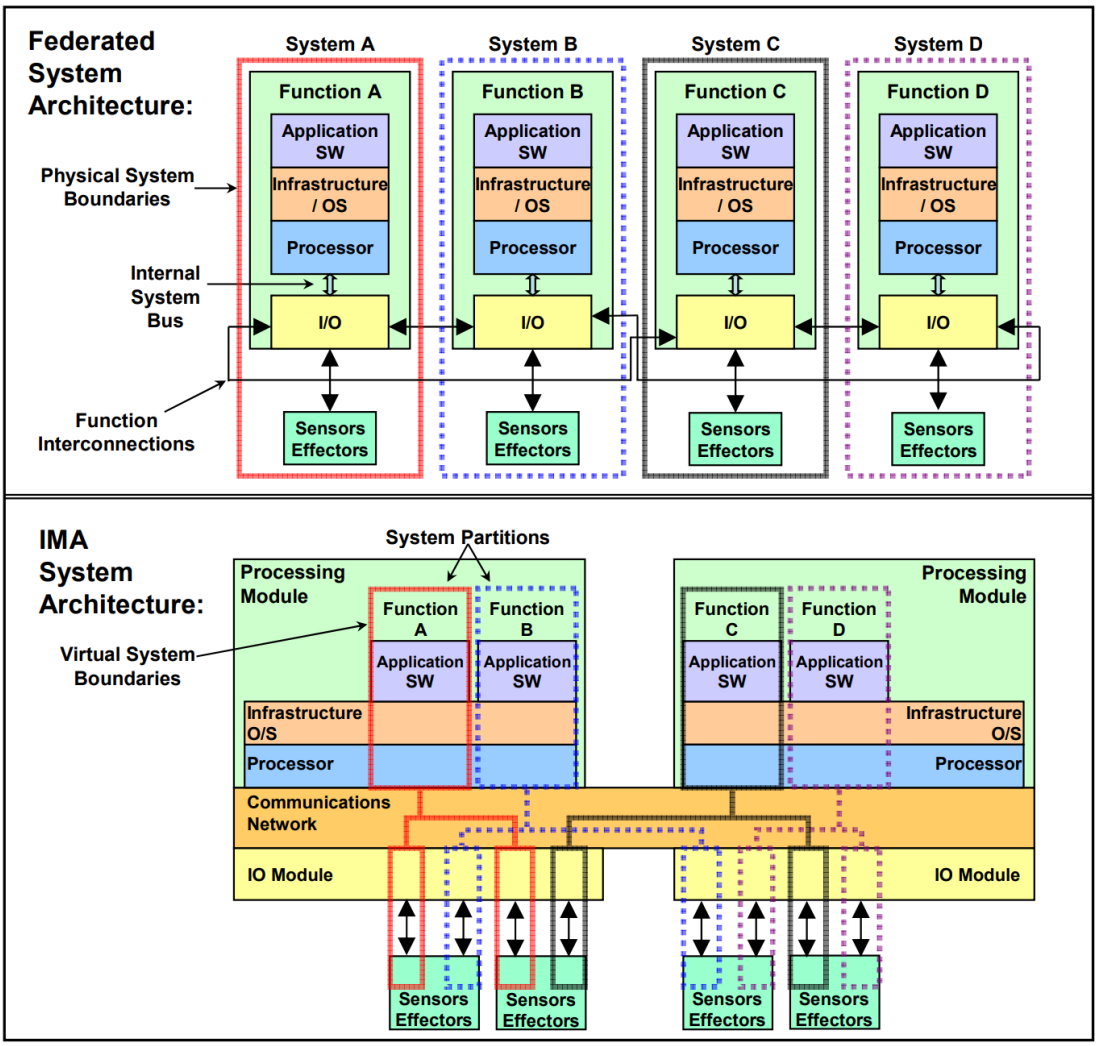


Figure 1.1 : Comparison of Federated and IMA architectures [2].

In IMA-based systems, subsystems share avionics resources such as processing power, data storage, and communication, enabling more efficient resource utilization while reducing overall system size and weight. For instance, Boeing reported a 2000-pound weight reduction in the 787 Dreamliner by implementing the IMA architecture. Similarly, Airbus has reported that IMA architecture reduced the processing units by half for the new A380 avionics suite [4]. Due to its reusability, portability, modularity, and cost-effective re-certification, IMA architecture has gained popularity in safety-critical avionics applications.

IMA-based systems typically use real-time operating systems (RTOS) such as INTEGRITY [5] and TUBITAK GIS [6], which employ generalized APIs such as

ARINC 653 (Aeronautical Radio INC) interfaces [7]. The ARINC 653 standard defines the interface between the operating system and applications running on IMA-based systems, enabling the integration of multiple applications with different criticality levels with robust time and memory partitioning. Moreover, using ARINC 653 facilitates the easy integration and substitution of new and existing applications, improving the flexibility and adaptability of the systems. ARINC 653 is a crucial component in meeting certification requirements for developing safety-critical avionics systems. Its standardized and well-defined software architecture facilitates the creation of such systems by clearly separating software functions and ensuring each operates within its own partition. This minimizes the risk of interference or failure, making it easier for developers to demonstrate compliance with safety standards and regulations. Moreover, ARINC 653's widespread adoption within the avionics industry means it provides a well-documented and validated software architecture that can help reduce the cost and time required for certification. Therefore, ARINC 653 makes the certification process faster and less prone to costly errors or delays.

With the rapid evolution of aviation technology, there is an increasing demand for processing power, which has resulted in a transition from single-core processors to multi-core processors. Multi-core support is provided by the ARINC 653 standard, with Supplement 4 published in 2015. However, shared resources such as shared caches, buses, and memory management in multi-core processors can cause multi-core interferences. To address this issue, the Certification Authorities Software Team (CAST) published a guidance paper in November 2016, CAST-32A [8], providing objectives for certifying multi-core safety critical systems. CAST-32A identifies concerns that may impact the safety, performance, and integrity of software executing on multi-core processors and proposes objectives to address these concerns. In response to CAST-32A, the FAA and EASA collaborated to develop official guidance for certifying multi-core avionics software. EASA's Acceptable Means of Compliance (AMC) 20-193 was released in early 2022 [9]. This guidance document is similar to CAST-32A and provides a set of objectives and terminology for multi-core avionics systems certification.

In IMA-based systems, applications are typically segmented into distinct memory and time partitions to ensure they have the necessary resources to execute consistently and reliably. However, shared resources such as processing power, data storage, and communication buses can still impact the worst-case execution time (WCET) of the applications on multi-core platforms, even partitioned. These shared resources can lead to contention and may cause delays in the execution of applications, affecting the system's determinism and overall performance. These shared resource contentions can be mitigated through Cache Partitioning, which involves allocating different applications to separate cache regions. The thesis proposes a novel ComCoS (Combined Color Stacks) approach for implementing cache partitioning in operating systems (OS) with better performance and determinism.

1.1 Purpose of Thesis

This thesis aims to comprehensively investigate the challenges associated with memory allocation in safety-critical IMA-based systems for cache partitioning. It identifies the limitations and drawbacks of current memory allocation approaches and proposes a novel cache partitioning algorithm as an effective solution to overcome these challenges.

The CAST-32A discourages dynamic memory allocation and recommends that applications allocate memory at system start-up. However, allocating memory for all applications at system start-up can significantly increase system boot time. In safety-critical IMA-based systems, the health monitoring mechanism triggers if an error occurs, which may require resetting the entire module depending on the error's severity. For example, catastrophic consequences may occur if systems such as engine control units or flight control systems do not operate for an extended period. Therefore, system boot time is of great importance and should be as fast as possible.

AMC 20-193 states that "justification for using dynamic allocation features within the scope of this AMC may rely on robust and proven limitations that lead to deterministic behavior [9]." However, AMC 20-193 also notes that users should avoid pure dynamic allocation and use appropriate configurations that do not affect other applications. In large-scale IMA systems, there may be a need for dynamic loading of applications due

to requirements such as load balancing or transferring an application from a faulty card to other cards. However, it is crucial to ensure that the system's other applications are unaffected by dynamic loading.

The thesis proposes a new cache partitioning algorithm called ComCoS to provide better performance and deterministic behavior. This algorithm reduces system start-up time for systems preferring static allocation and provides faster and more deterministic cache allocation for systems favoring dynamic application allocation.

1.2 Organization of Thesis

In Section 2, a detailed background and literature review were conducted on various topics, including Cache, ARINC 653, Certification Process, Shared Resource Interferences, Cache Partitioning, and Memory Management in Operating Systems. Section 3 presents our novel ComCoS cache partitioning technique. Section 4 evaluates our technique with experiments, followed by conclusions and future work in the final section.

2. BACKGROUND AND LITERATURE REVIEW

This section provides background information on the working principle of caches, the ARINC 653 standard, and the certification of IMA-based systems. Then, the challenges arising from the shared cache, buses, and memory in multicore platforms are examined, along with the contributions of cache partitioning to address these issues, supported by examples from the literature. Afterward, different cache partitioning approaches are evaluated from the perspective of IMA-based systems. Finally, existing memory management algorithms developed for cache partitioning in the literature are examined, and their limitations are identified.

2.1 Cache

To improve the performance of computer systems, the cache is generally used, which is based on static random-access memory (SRAM). However, due to the high unit cost of SRAM, the amount of cache memory in these systems is limited compared to the main memory, which is typically dynamic random-access memory (DRAM). The cache structure is an intermediate layer between the main memory and the processor, reducing data access time and energy consumption. It is not feasible to optimize cost, power consumption, volume, and speed using a single type of memory component. As a solution, the hierarchical memory model has been adopted as in Figure 2.1 [10].

The processor checks for a corresponding entry in the first-level cache (L1D/ L1I). If the desired location is in the cache, a *cache hit* occurs; otherwise, a *cache miss* occurs, and the processor requests the data from the next memory hierarchy level. The *hit ratio* measures cache performance and can be defined as the number of cache hits over the total number of memory accesses. The hit ratio can be improved by increasing the cache block size, associativity, temporal and spatial locality, or cache partitioning.

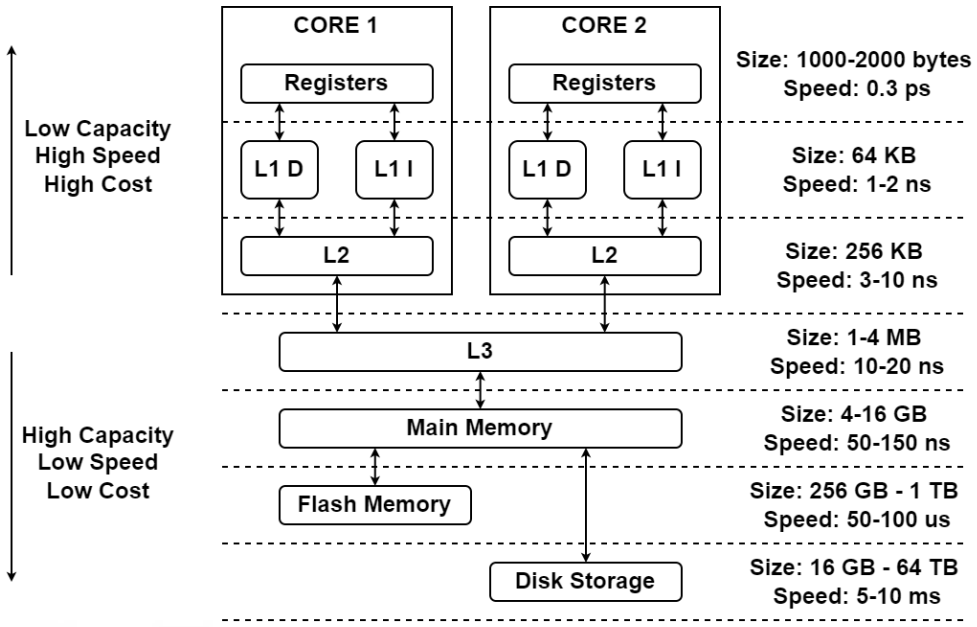


Figure 2.1 : Schematic of the memory hierarchy.

The cache mechanism follows general programming practices. After accessing an address, the near address will probably be accessed soon because instructions and program segments follow the sequential memory order. So data exchange between cache and main memory is performed with blocks because of *spatial locality*. After accessing an address, high probably it will be reaccessed soon. Frequently used addresses are not removed from the cache according to the *temporal locality*.

Memory mapping between the cache and main memory can be performed using different methods, as seen in Figure 2.2. The *direct mapping* method allows each memory block to be placed in a dedicated cache location. However, the main problem with this technique is that even if there are several unused lines in the cache, two memory blocks mapped to the same cache region cannot exist simultaneously. In contrast, with the *fully associative mapping* method, each memory block can be placed in any of the cache memory lines. However, this method can only be used for tiny cache memories due to the high hardware cost of searching the entire cache.

In *set associative mapping*, the cache is divided into multiple *sets*, and each set has a specified number of *ways*. It is determined in which set the memory block will be placed, but it is not certain which way is selected for placement. Memory blocks can map to any way in the specified cache set and ways of the sets are accessed in parallel.

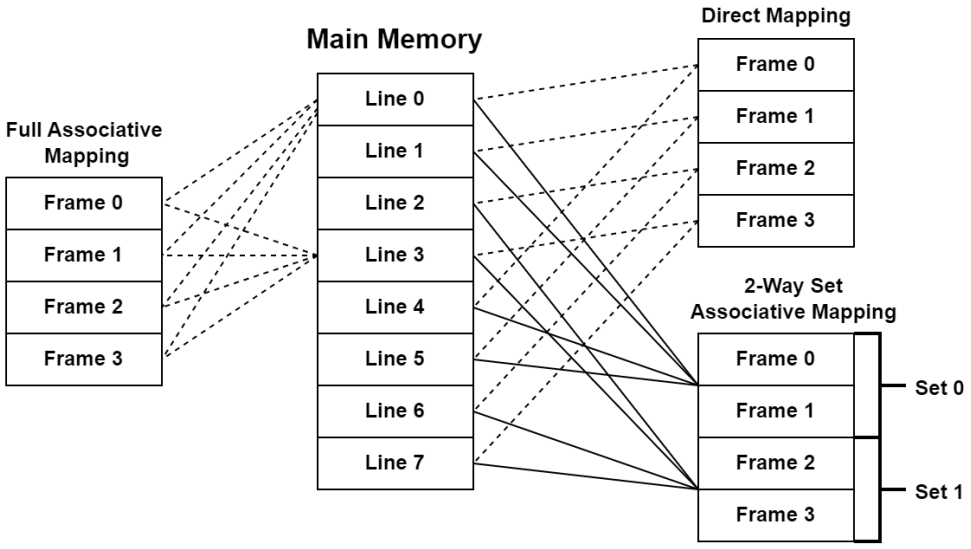


Figure 2.2 : Cache mapping methodologies.

The set associative mapping technique can be considered a combination of direct and fully associative mapping techniques.

2.2 ARINC 653

The primary responsibility of operating systems is to facilitate the sharing of system hardware resources, such as memory, processor time, network connections, etc., among the tasks running on it. However, resource sharing for safety-critical systems must be more restricted. The ARINC 653 specifications provide an interface for operating systems to protect applications of varying criticality levels from one another. The primary goal of these specifications is to define the Application Executive (APEX) interfaces between the operating system and applications as follows:

- Partition Management
- Process Management
- Time Management
- Interpartition Communication
- Intrapartition Communication
- Health Monitoring

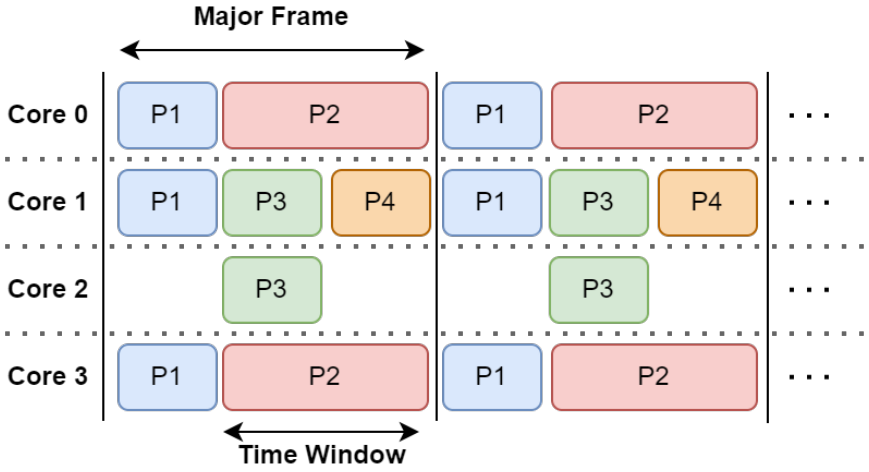


Figure 2.3 : ARINC 653 time configuration.

The ARINC653 standard does not offer memory management services; instead, it expects applications to determine and define their memory requirements during system configuration. Additionally, another responsibility of ARINC653 is to provide a configuration file for system configuration. The system then allocates each application's required physical memory space before it starts running. This means that each application is responsible for managing its own memory within the allocated space. Partitions (applications) can define one or more processes to perform their desired functions. The ARINC 653 standard provides robust memory partitioning, ensuring that partition processes share the same physical space but cannot access the physical space of other partitions.

In ARINC 653, robust time partitioning is a mechanism to guarantee the independent and interference-free operation of different applications in an IMA-based system. This mechanism divides the processor's time into fixed intervals called *time windows* and assigns specific time windows to each partition. Applications execute within their assigned time window. The length of the time window, determined by the size of the time partition, is specified in the configuration file. The time window ensures that an application completes its execution within the assigned time and prevents it from exceeding its allocated time. Furthermore, ARINC 653 defines the major frame as a periodic time interval. During the major frame, time windows execute in a predetermined sequence. Figure 2.3 shows an ARINC 653 time configuration example.

Table 2.1 : Design Assurance Levels (DAL) [11].

DAL	Failure Condition	Resulting Conditions	Objectives
A	Catastrophic	Failure may result in deaths and loss of the aircraft.	71
B	Hazardous	Failure creates a major negative impact on safety or performance or reduces the aircraft crew's ability to operate the aircraft. This can result in serious or fatal injuries.	69
C	Major	Failure causes significant reduction of the safety margin or significant increase in the aircraft crew workload. Passenger discomfort or minor injuries can result.	62
D	Minor	Failure slightly reduces the margin of safety or causes slight increase in aircraft crew workload. Results can include passenger inconvenience or changes to a routine flight plan.	26
E	No effect	Failure causes no impact or effect on safety, crew workload, or operation of the aircraft.	0

2.3 Certification Process

The certification of IMA-based systems is critical to ensuring the safety and reliability of aircraft. These complex structures integrate various critical applications with different levels of criticality. Criticality levels are defined by the Design Assurance Level (DAL) in the DO-178C guidance and given in Table 2.1. RTCA (Radio Technical Commission for Aeronautics) published the DO-178 document in response to the need to define software processes for compliance with flight safety requirements due to the increasing use of software in aircraft systems and equipment in the early 1980s. The document has been revised three times in response to feedback and experience.

Certifying IMA-based systems under DO-178C involves an aviation safety association or certification body that evaluates the system design to ensure its conformity with DO-178C and verifies its reliability. This process follows the design and development of the system and includes various tests and analyses to confirm that the design meets certain criteria. The certification process covers the system's entire life cycle, from its initial design to its production and testing to its use in operation. The life cycle of processes can be divided into three main categories: the software planning process, the

software development processes (requirements, design, coding, and integration), and the integral processes (verification, configuration management, quality assurance, and certification liaison) [1].

Analyzing the system's WCET, limitations, and utilization is recommended as part of the DO-178C software verification process for safety-critical systems in aviation. The WCET measures the maximum time a software task or process may take to complete under worst-case conditions. It is a crucial parameter for allocating sufficient processing resources and ensuring the system can meet all required deadlines. Another critical parameter for measuring system utilization in safety-critical systems is the standard deviation of task or service execution times. High standard deviations can result in significant losses in system utilization since resources are allocated based on worst-case scenarios. Therefore, it is necessary to evaluate the system design comprehensively, identify factors that may influence task execution time, and implement strategies to mitigate or eliminate them whenever possible to achieve determinism in ARINC 653 and IMA-based systems.

While it is more cost-effective for system integrators to re-certify completed applications upon integration, other applications in the system must not impact the resource usage and analysis of individual applications. This is particularly challenging because even when applications are separated by time and memory partitioning with the IMA and ARINC 653 standard, they can still affect each other due to using shared resources on multi-core platforms. This creates a significant challenge for the system's reusability, safety, and determinism, especially when considering WCET analysis of applications and the distribution of system resources.

EASA and FAA have created guidelines for developing multi-core systems for aerospace projects, including AMC 20-193 and CAST-32A. These documents guide compliance with airworthiness regulations and recommend best practices to consider when dealing with multi-core processors. These two guidelines share many similarities except for a few topics, such as the dynamic allocation of software execution, simultaneous multithreading, IMA, etc. [12]. The alteration made to dynamic allocation is crucial for this thesis. While the CAST-32A document does not

recommend dynamic memory allocation, AMC 20-193 states that dynamic allocation can be performed under specific deterministic conditions and limits.

2.4 Interferences

Interferences in multi-core systems can occur either on-chip or off-chip. Shared off-chip resources, such as I/O devices, network devices, and other peripherals, are the principal source of interference in multi-core systems. Concurrent access to these resources by multiple cores can create interferences, particularly when the device has limited bandwidth. These interferences can lead to contention for resources and delay the execution of applications. These problems can be avoided with partitioned device designs, such as Avionics Full-Duplex Switched Ethernet (AFDX) [13], which utilizes a partitioned network architecture. However, partitioning device design is not the subject of this thesis.

Another source of interference is shared on-chip resources, such as shared caches, buses, and memory management units. Simultaneous access to these resources by multiple cores can create interference channels that impact the WCET of applications.

2.4.1 Cache eviction

In multi-core systems, cache eviction occurs when an application running on one core removes the data of another application running on a different core from the cache. In this case, the execution time of one application is directly affected by the other application, creating a determinism problem. Single-core systems can eliminate this problem by flushing the cache during application switches. However, in multi-core systems, cache partitioning can prevent this issue. The experiment in Section 4.2 shows the cache eviction determinism problem and the cache partitioning improvement.

2.4.2 Bus interferences

On multi-core systems, applications requesting simultaneous memory access from different cores can cause bus interference due to using a shared bus, even if the applications are located in different cache regions. Applications that perform memory-intensive tasks such as logging and mathematical calculations can saturate the

system's bus bandwidth, causing delays for critical applications with higher priority running concurrently. These applications are constrained to a designated processing capacity during specific time intervals to address this issue and terminate if they exceed that capacity. The system tracks the number of memory accesses made by these applications using Performance Monitor Counters (PMC) and throws an exception if they exceed a certain threshold. Safety monitoring terminates these applications, guaranteeing critical priority applications' proper functioning. Many studies have been conducted to determine and manage the memory bus bandwidth [14,15].

Although cache partitioning does not directly solve this situation, it can reduce the impact on memory bus bandwidth by preventing applications from evicting each other's cache regions. The experiment in Section 4.3 illustrates the problem of bus interference and demonstrates the improvement achieved through cache partitioning.

2.4.3 DRAM bank eviction

Processor cores access the main memory (DRAM) through memory banks, organized into ranks and channels to enable parallel access. The memory banks consist of rows and columns, and the memory controller identifies the specific location within the banks based on the memory address. Accessing the same row allows direct access from the row buffer, but accessing a different row evicts the previous data. The memory controller queues access requests due to the speed disparity between the processor cores and the main memory.

Row buffer eviction causes contention between applications. Which bank will be accessed in the DRAM chip is determined according to the physical address. Like cache partitioning, bank partitioning can be accomplished by allocating appropriate memory zones to different partitions. If the cache and bank selection bits match, cache partitioning and bank partitioning can be implemented jointly. However, some modern processors can use bit randomization techniques to select the memory banks, and they do not document address mappings to banks [16]. Unlike a generalizable method such as cache address mapping, the design of bank address mapping can vary depending on the individual designs of the processors. There have been many studies about combining cache and bank partitioning by revealing the relationship between cache and

bank mapping bits [16,17]. On the other hand, memory controllers reorder the queue to reduce row buffer eviction and interleave the memory request to the banks. Cache and bank partitioning can be evaluated together based on the compatibility of cache and bank mapping bits. However, hardware manufacturers do not have a generalized approach.

2.5 Cache Partitioning

Cache partitioning in multi-core systems is the technique of dividing the shared cache among applications to reduce contention and improve performance. Basically, cache partitioning methods are implemented as hardware or software. Sparsh Mittal conducted a literature survey on cache partitioning from various perspectives [18]. Literature works are evaluated from the perspective of IMA-based systems.

2.5.1 Hardware based cache partitioning

Way partitioning is a hardware-based solution. The cache control registers provided by the hardware are set, and processor cores are assigned to different cache ways. For example, L2PIRn, L2PARn, and L2PWRn registers of PowerPC e6500 core are responsible for way partitioning [19]. There have been many studies based on way partitioning [20]–[22]. Not only way partitioning but also hardware-provided bits can be used for deterministic memory, such as research by Farshchi et al. [23].

Hardware cache partitioning methods are straightforward and do not require additional software design for the operating system. However, not all processor families support this feature, and hardware dependency in IMA-based systems is undesirable. In addition, many different partitions run on the same processor cores over time, each having different memory needs. However, way partitioning forces partitions to have similar cache behavior. Because modifying the hardware configuration during run-time is not recommended for safety-critical projects, as it poses potential risks to the system's safety and integrity. Furthermore, the system can only support a limited number of ways, and the cache associativity is proportional to the cache way count. Processors commonly used in the market have 4-8 cores and 8-16 way set associative shared caches, such as Xeon E5606, PowerPC e6500, and ARM Cortex-A77. In way

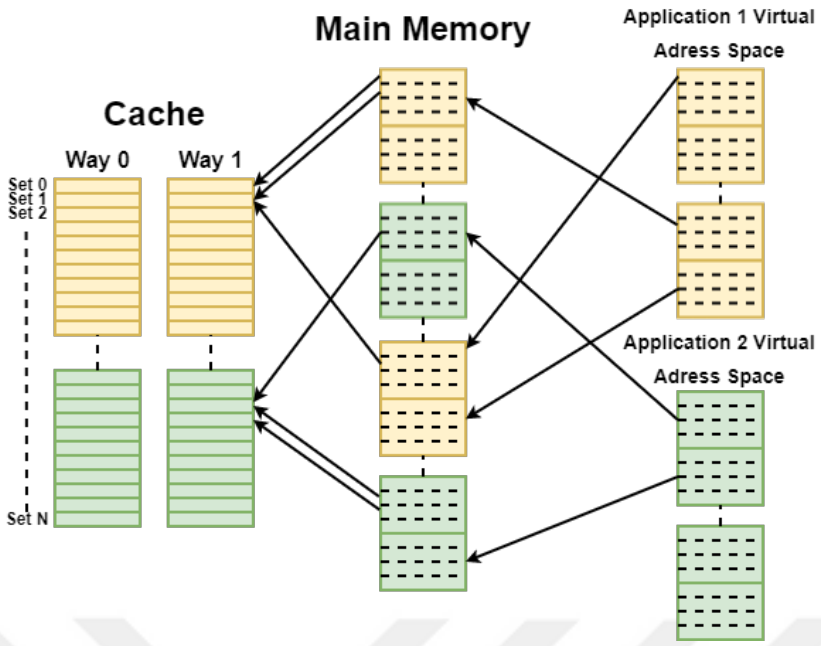


Figure 2.4 : Schematic of the page coloring.

partitioning, the partitioning granularity is limited by the number of cores, and the cache partition size assigned to each core is restricted by the number of ways.

The use of hardware-based cache partitioning methods in large-scale IMA-based systems is not recommended due to hardware dependency, decreased cache associativity, and the potential risks of modifying hardware configuration during system run-time.

2.5.2 Software based cache partitioning

Set partitioning, or page/cache coloring, is a software-based approach that determines the placement of physical memory regions in cache sets. The applications can access a specific cache region only when the appropriate physical memory regions are assigned to them. These cache regions are considered as different colors. Applications may request more contiguous physical memory than their cache region size, which can be solved by memory virtualization. The working principle of cache coloring is illustrated in Figure 2.4

There have been several studies on set partitioning, which does not require hardware support and does not decrease associativity [24]–[26]. The entire mechanism uses software but requires changes to the operating system's memory management. Since

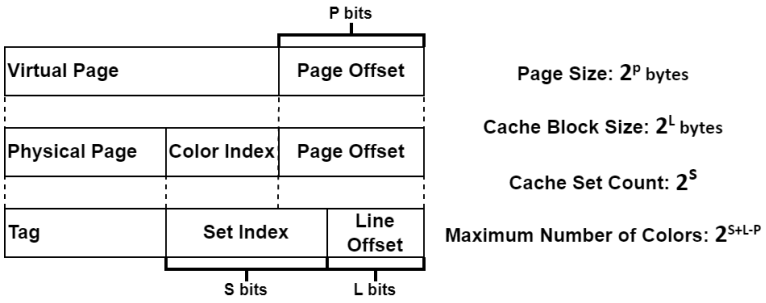


Figure 2.5 : Address translation of the virtual to physical.

the whole set partitioning mechanism is implemented in software, it is more suitable for IMA-based systems in terms of hardware independence and common hardware usage. With this method, each application can create its configuration independent of the hardware.

The granularity of set partitioning depends on the virtual page size, the number of sets, and the size of cache blocks, as seen in Figure 2.5. An example is the PowerPC e6500 processor, which has a shared L2 cache consisting of 4 banks, 512 sets, 16 ways, and 64-byte block sizes shared among four cores. If virtual memory is divided into fixed 4KB pages (minimum page size for the processor), the cache can be divided into 32 cache partitions. Compared to way partitioning, set partitioning has a much better granularity.

In order to achieve deterministic WCET, IMA-based systems rely on a well-partitioned cache. This is especially crucial in complex systems like aircraft, where numerous applications of varying criticality levels coexist and necessitate different cache sizes. A high level of cache granularity is essential to accommodate these diverse requirements.

2.5.3 Dynamic and static cache partitioning

Determining the cache need of system applications at compile time is challenging. Therefore, many Quality Of Services (QoS) have been defined for running applications with optimum cache partitioning [27,28]. However, optimal cache partitioning is an NP-Hard problem. Several statistical [29,30] and artificial intelligence-based [31,32] studies have been carried out. When the application requests a new memory, the location to be given is changed according to the QoS. The cache size and region allocated for the partition can be updated. However, since IMA-based systems are

generally used in safety-critical applications, dynamic cache partitioning cannot be accepted in run-time because it will change the system's determinism. Therefore, static cache partitioning is preferred in IMA-based systems.

2.6 Cache Partitioning in Single Core Systems

Time partitioning is crucial for ensuring determinism in IMA-based systems. In the ARINC 653 standard, applications run in their sequentially repeated time windows. In single-core systems, after an application completes its window, it switches to the other application, and the new application can evict the cache data of the old application. According to the CAST-20 document [33], cache flushing is recommended during partition switching to eliminate determinism issues. However, this can cause a significant performance loss. On the other hand, if cache flushing is not performed, the thrashing rate varies depending on the time window size. If the time window is small, the application cache becomes contaminated before it uses the data it puts in the cache. The experiment in Section 4.4 demonstrates that cache flushing results in performance degradation, while cache partitioning resolves the determinism problem without compromising performance.

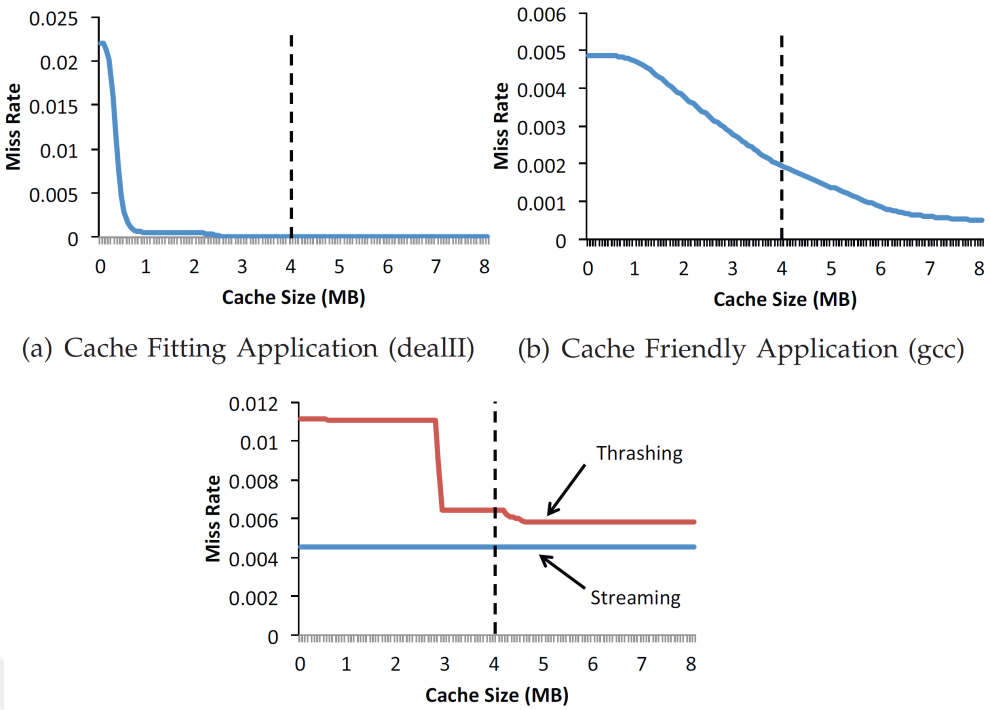
2.7 Application Cache Characteristics

Applications may exhibit different behavior depending on the size of the cache allocated to them. Applications can be categorized into three groups based on their cache characteristics: fitting, friendly, and streaming/thrashing, as in Figure 2.6.

Hundreds of applications work in a complex structure in IMA-based aviation systems. Each application has unique cache characteristics and requires a different cache size. In IMA-based systems, the operating system must provide a memory management structure that allocates memory in a way suitable for the cache characteristics of each application.

2.8 Memory Management Implementations

Demand paging is a memory management scheme in which pages are loaded into main memory only when needed. In demand paging, a page is loaded from secondary



(a) Cache Fitting Application (dealII) (b) Cache Friendly Application (gcc)

(c) Cache Thrashing / Streaming (lbm, milc)

Figure 2.6 : Application cache characteristics [20].

storage, such as a hard drive, into main memory, such as DRAM, once a running program requests it. This technique allows for more efficient use of memory resources since only the necessary pages are loaded into memory. Unused pages can remain on the secondary storage device until needed. Demand paging is widely used in traditional operating systems, such as Linux, to enable programs to use more memory than is physically available. However, demand-based paging is generally incompatible with real-time systems as it can introduce unpredictable delays due to the time it takes to fetch the necessary pages from secondary storage. Ensuring a maximum response time for critical tasks is necessary in real-time systems. Demand-based paging can compromise these guarantees if it leads to unexpected page faults. In safety-critical avionics systems, physical memory for the program areas of applications is fully allocated during application loading. Page coloring memory management algorithms for real-time systems, dividing them into two groups. The first group involves allocating memory for applications at system startup following the directives of CAST-32A. The second group is based on dynamic application allocation according to the directives of AMC20-193.

2.8.1 Static memory allocation

In static memory allocation, all physical memory for applications is allocated during system startup according to the CAST-32A guidance. In the work of Suzuki et al. [16], linked lists are created for each color, and application page requests are satisfied from these lists. Applications may request multiple pages based on their cache requirements. To maintain cache characteristics, the colors assigned to the application are sequentially given in a round-robin order. This approach ensures a deterministic physical cache layout for the contiguous user space of the application, preventing variations in WCET and determinism issues. Dugo et al. [17] mentioned that they store data in an array instead of using linked lists due to the high cost of linked list node linking and lack of trust in linked lists. This optimization resulted in accelerated system boot time. The initialization time is a critical criterion for these techniques as it involves acquiring all physical memory during system initialization. In both studies, applications make single page requests at a time, but the assigned colors may be contiguous. In the proposed ComCoS algorithm in this thesis, multiple colors can be allocated simultaneously with a single request, significantly speeding up application allocation and reducing the system boot time.

2.8.2 Dynamic memory allocation

Applications can dynamically load within the robust and proven limitations specified in the AMC20-193 directives. In large-scale IMA-based systems, dozens of companies design applications integrated by system integrators. To balance the load on the processing units within the system, applications with low criticality levels can be migrated to another processor. Alternatively, if a card fails, the other cards may need to assume that responsibility dynamically. Therefore, applicants may require dynamic application loading. However, this allocation should not impact other applications on the system and should allocate resources with fast and deterministic behavior within the robust and proven limitations outlined in the AMC20-193 directives.

Tam et al. [24] modified the Linux page frame to support color awareness by dividing the single free list into multiple lists, each associated with a specific page color.

However, this design has issues with both performance and determinism because the operating system memory allocator needs to invoke repeatedly until obtaining a page of the desired color.

COLORIS [26] proposed keeping free pages of the same color together in linked lists to address these problems. Applications can request multiple colors, and these allocations are performed round-robin. If a linked list does not have the desired color, the application requests another color assigned to it. However, this may alter the cache characteristics and behavior of the application. The operating system memory allocator is consulted when all requested cache regions are exhausted in lists. If the allocated memory is not the desired color, it is added to the list of the relevant color. Although this approach performs better than the previous design, it is still uncertain when the desired cache color will be reached.

Part et al. [34] proposes allocating all pages from the operating system memory manager when the desired colored page is not found in the color lists. These pages are split into all color lists simultaneously. In this method, there is a difference between the average and worst-case execution times. Because in the best case, only one color is retrieved from the list, while in the worst case, all colors are allocated from the main memory, and each page is added to the appropriate color lists. This causes the service not to exhibit deterministic behavior. Our proposed method differs from previous approaches because we do not store each color as a separate node. Instead, adjacent colors are stored together as a single node, allowing for the allocation of multiple pages with a single service request. This results in faster and more deterministic dynamic allocation.

Using the memory reservation method is appropriate to ensure sufficient memory allocation for dynamically loaded applications. This method reserves a specific memory area from which dynamic memory regions are to be allocated. The reserved memory area is restricted and separated from existing applications within the system. Applications already in the system cannot consume this reserved memory region.

There is no need to fill the color lists for the dynamic allocation at system boot time as this would unnecessarily extend system initialization time. The ComCoS algorithm

allows multiple page nodes to be filled dynamically to lists. This allows dynamic application allocation to occur faster and within a limited time compared to other applications in the literature.

3. DESIGN AND IMPLEMENTATIONS

In this study, two cache allocation algorithms were implemented. The first algorithm, referred to as the reference implementation, was used for comparison purposes with the ComCoS design. To avoid the cost of node linking, separate stacks were maintained for each color, following the approach described by Dugo et al. [17]. The newly proposed ComCoS design's second implementation utilized new stacks to handle color combinations.

Two different versions were defined for each implementation. In the first version, applications were statically initialized, and the total memory required by the applications was allocated from the main memory. The stacks were filled with this allocated memory during start-up, and the applications were initialized using these stacks. In the second version, dynamic allocation was used to assign memory to the applications at run-time. This version utilized memory reserved for dynamic allocation to feed the stacks dynamically.

3.1 Reference Implementation (Separate Color Stacks)

How to calculate the maximum number of colors that can be created for the relevant platform is given in Figure 2.5. In this thesis, the symbol of N is used to represent the number of defined colors. N different stacks are created by adjusting the *STACK_CAPACITY* configuration parameter based on the specific requirements of the IMA-based systems in the reference implementation.

In the static memory allocation version, the memory region reserved by the system configuration for colored page allocation is pushed onto the color stacks one by one during system start-up. This reserved area will be used to fulfill the colored page requests of all applications. Algorithm A.1 defines the *Cache_Alloc* service, which enables memory allocation in specific cache colors. The service takes a stack index as a parameter, representing a different cache color ranging from *1* to *N*. In the first

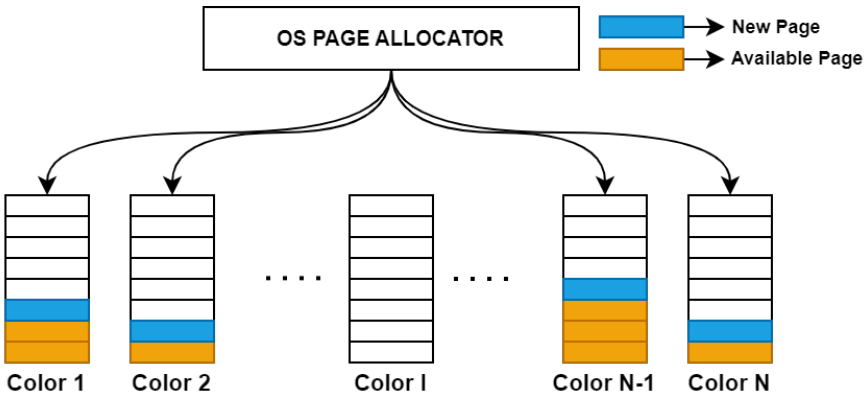


Figure 3.1 : Dynamic memory allocation from separate color stacks.

step, the algorithm attempts to pop a memory region from the corresponding stack. The static allocation version returns an error if there is no available memory region of the relevant color in the stack. This is because all the stacks are filled during start-up according to the configuration, and there is no dynamic memory allocation. Otherwise, the desired colored memory region return from the service.

In the dynamic memory allocation version, the algorithm may request new dynamic memory from the OS page allocator if the requested stack is exhausted. However, following the AMC 20-193 directives, the dynamic memory areas must be pre-reserved and dedicated exclusively for dynamic application loading. This measure ensures determinism and imposes limitations on the allocation of dynamic memory. When there is no desired page remaining in the stack, to guarantee that the allocated memory will contain the desired color, *N* pages of memory are obtained from the main memory using the OS page allocator service, following the approach by Part et al. [34]. The page in the appropriate color is returned from the service within the *N* pages, and the remaining *N-1* pages are pushed onto their respective stacks using the *Push_Pages* service in Algorithm A.1. Figure 3.1 demonstrates the memory allocation process the OS page allocator performs when requesting an *I* indexed cache region in the dynamic cache allocation version.

The application can be assigned more than one color, but the *Cache_Alloc* service can only allocate a single colored page at a time. In such cases, the assigned colors are distributed to the application using the round-robin method to maintain a consistent behavior. When the stacks reach their capacity, the system's behavior may vary. Two

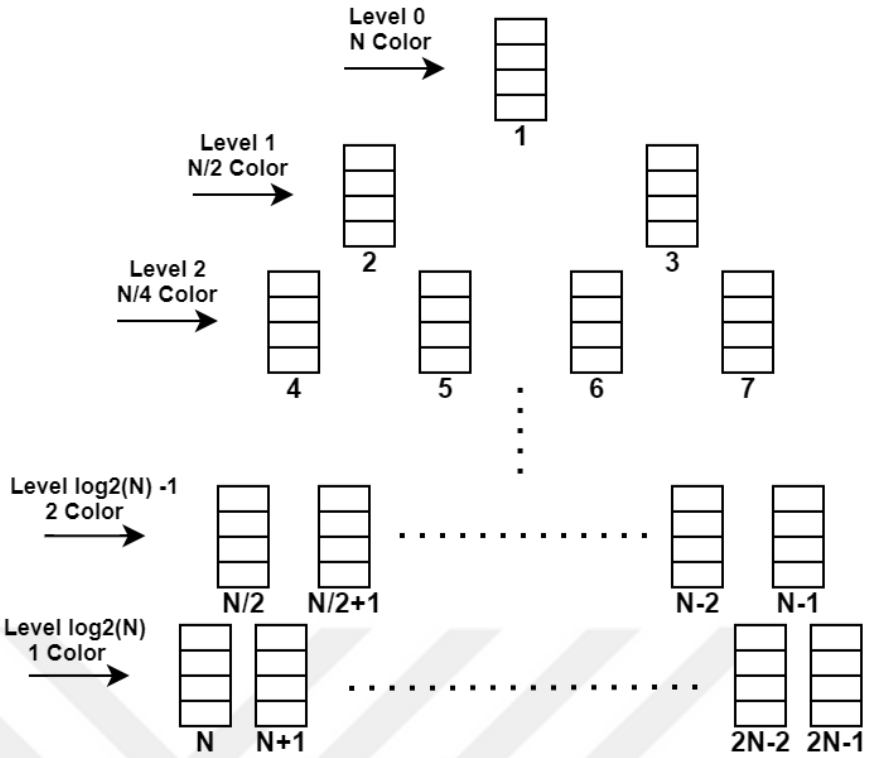


Figure 3.2 : Binary tree structure of ComCoS technique.

options can be considered in this situation: increasing the stack size or returning non-fitting pages directly to the OS page manager. However, in time-critical IMA systems, it is undesirable to have free or deletion services. Therefore, it is necessary to reconsider the *STACK_CAPACITY* parameter.

3.2 Combined Color Stacks

Assigning different amounts of cache pools is essential for each partition due to their distinct cache characteristics. However, the existing cache allocation services used in previous methods had performance and service determinism limitations because they could only return one page at a time. To overcome this limitation, a new ComCoS technique has been developed to allocate multiple cache pools in a single request.

ComCoS defines new stacks for adjacent free colors instead of having a separate stack for each color. Specifically, a perfect binary tree structure is constructed with *2N-1* stacks, each covering a range of adjacent free-colored pages. The root stack covers all *N* different colors, with its left child covering adjacent free colored pages from *1* to *N/2*, and its right child covering adjacent free colored pages from *N/2+1* to *N*. The

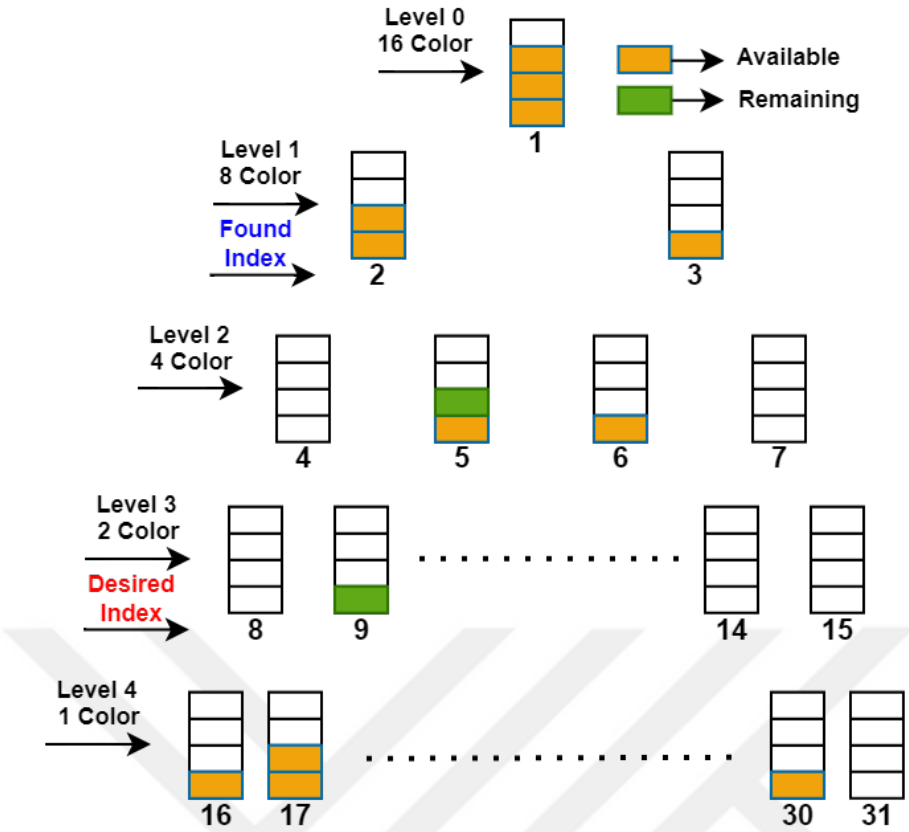


Figure 3.3 : Example memory allocation from parent index for ComCoS.

number of pages kept together in the stack is halved at each level of the tree. The leaf nodes of the binary tree are separate color stacks. Figure 3.2 illustrates the constructed binary tree structure. All stacks are created statically with the *STACK_CAPACITY* configuration parameter.

In the static memory allocation version, the memory region designated by the system configuration for colored page allocation is pushed into the root stack in groups of N colored pages. ComCoS provides the *Cache_Alloc* service, defined in APPENDIX B, which allows for memory allocation in multiple colors. This service takes a stack index as a parameter in the range of 1 and $2N-1$, and it is determined which stack index corresponds to which color(s). For instance, index 2 represents $N/2$ colors between index 1 and $N/2$, and the starting address of the continuous $N/2$ colored page is returned from the service.

Memory fragmentation becomes a significant concern in the ComCoS technique, as it allows allocating multiple colors in a single request. This means that if the desired color belongs to a smaller color set, utilizing the larger color set can potentially lead

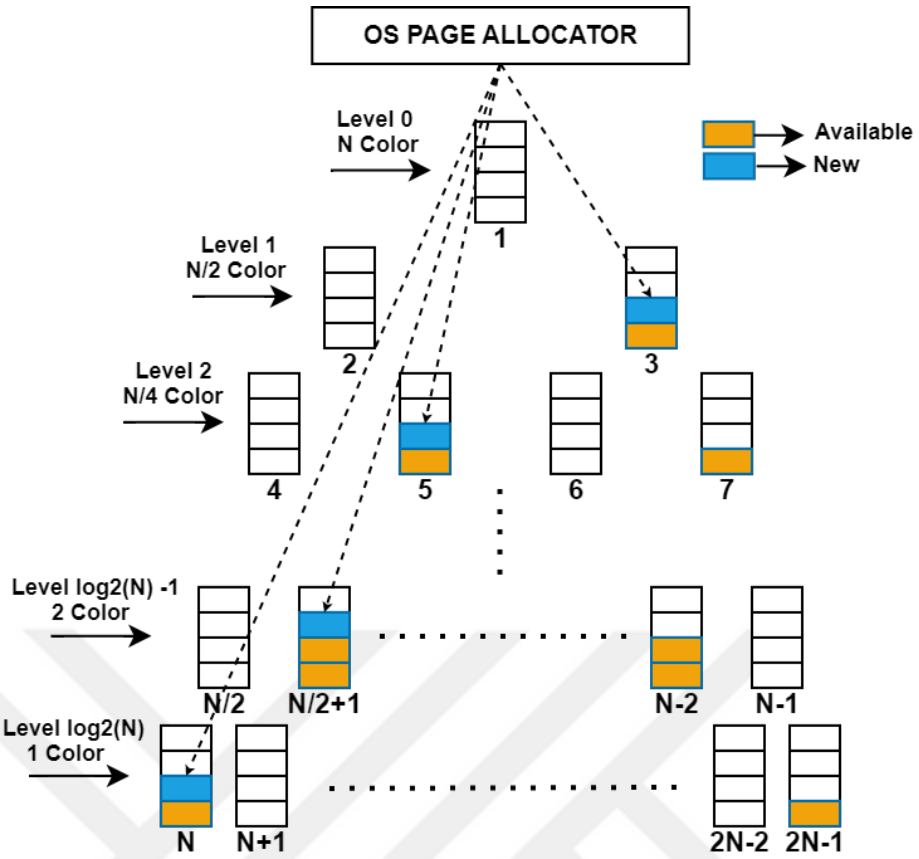


Figure 3.4 : Dynamic memory allocation for ComCoS.

to applications that require this memory region being unable to obtain it later on. Therefore, the algorithm first utilizes the *ALLOC_FROM_STACKS* service to search for the desired index in the stack indexed by the relevant index and all of its parents. The service returns the memory region if the desired index is found directly. If it is found in the parent indexes, the remaining pages are pushed to the appropriate stacks using the *Push_Pages* service. Finally, the desired page block address is updated according to the index, as the stacks always hold the starting address of the N page memory block taken from the main memory and returned.

Figure 3.3 presents an example scenario with *16* different colors. In this scenario, the *Cache_Alloc* service is called to request a cache region with index *8*. Firstly, the algorithm attempts to pop from the cache stack indexed by *8*, but it is unsuccessful. Then, it sequentially requests cache from the parent indexes *4* and *2*. It finds an available cache block in the stack indexed by *2*. While returning the two-page memory

block corresponding to index *8*, the remaining two-page are pushed to index *9*, and the four-page are pushed to index *5*.

In dynamic allocation, if the desired cache index is not found in any parent stack, the OS page allocator takes N pages. It then pushes the pages other than the desired stack index to the appropriate stacks using the *Push_Pages* service. For example, in Figure 3.4, when a free page with index $N+1$ is requested but not found in any parent stack, the algorithm takes N pages from the main memory. It returns the page with index $N+1$ and pushes the remaining $N-1$ pages to the stacks of its sibling and the siblings of its parent.

3.3 Cache Partitioning Configuration

Numerous software applications from different suppliers in IMA-based systems must work together safely. To ensure the coexistence of these applications, a configuration process is required that partitions system resources like memory, processors, and I/O devices among the various applications. For static and dynamic cache allocations, separate memory regions should be reserved through in that configuration. In the configuration of the memory to be taken statically, the whole reserved memory region should be distributed among stacks according to their colors, and applications should allocate the relevant memory areas from the stacks. In the configuration of the memory to be taken dynamically, the reserved memory is not entirely distributed to color stacks at system initialization; instead, it is filled into stacks in response to dynamic application loading requests. Additionally, the depth of these stacks must be configured and passed to the cache management algorithms as *STACK_CAPACITY*.

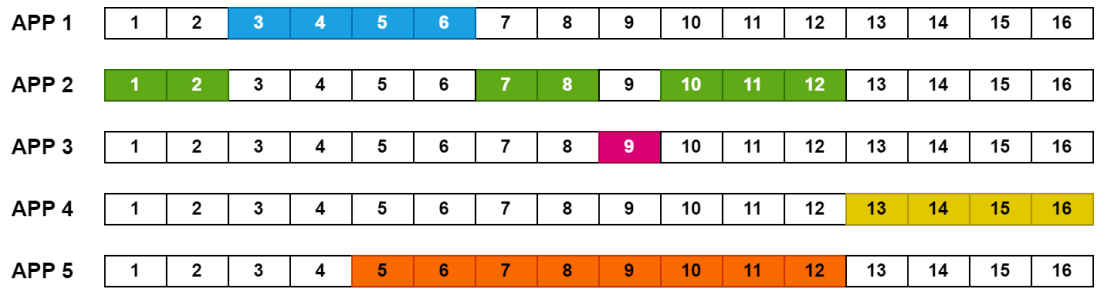


Figure 3.5 : Applications color distributions.

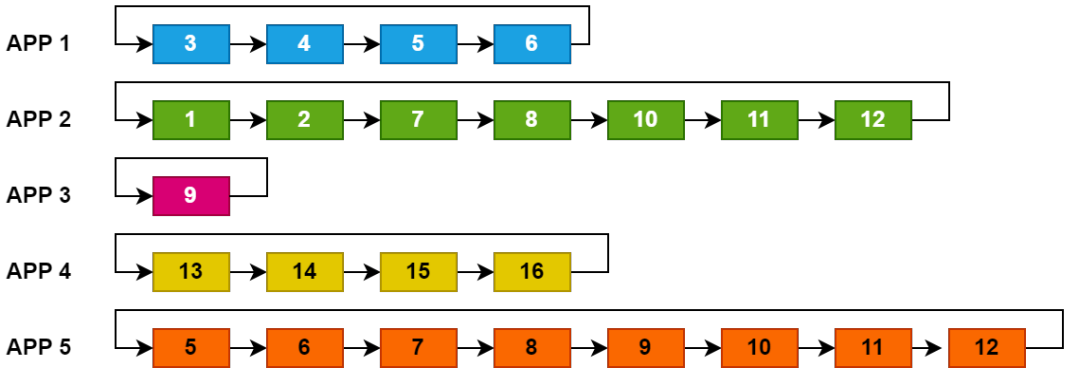


Figure 3.6 : Linked color lists in the Reference implementation.

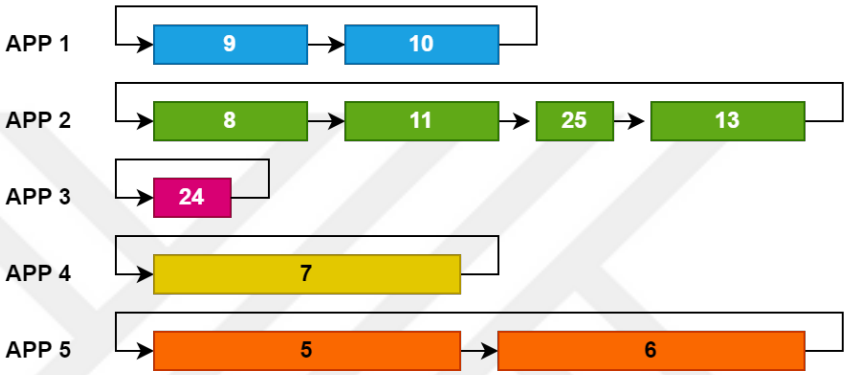


Figure 3.7 : Linked color lists in the ComCoS technique.

As discussed in Section 2.7, each application has a different cache characteristic. Therefore, assigning different amounts of cache for each application should be possible. Applications can create combinations of any desired number from *N* different colors. As mentioned, it is determined in which time window each application will run. Applications not running in the same time window can be assigned the same cache pool. For example, Figure 3.5 shows the example configuration of the *16* color distribution for five different applications.

Colors are assigned to the applications in a sequential round-robin order to maintain their behavior. The configuration file is parsed to create circular linked lists for the applications. However, there is a difference in the generated linked list between the reference and ComCoS implementations. In the reference implementation, each cache region resides in a separate stack, and as a result, each assigned cache region becomes a node in the list. Figure 3.6 illustrates the cyclic linked list for the given configuration shown in Figure 3.5 for the reference implementation.

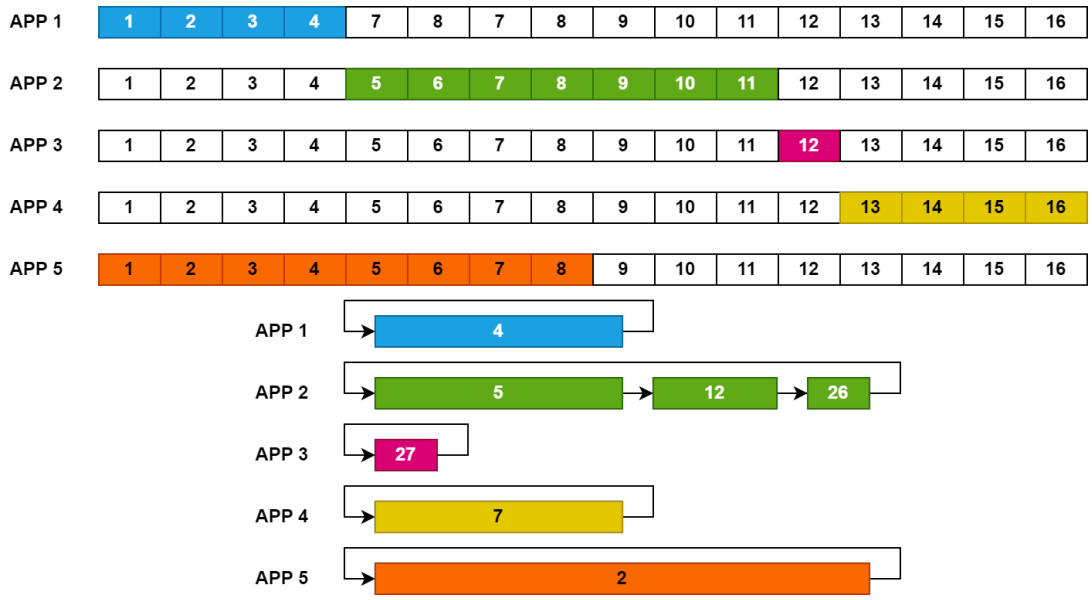


Figure 3.8 : Optimized linked color lists in the ComCoS technique.

In the ComCoS technique, adjacent colors can be placed in a single stack, retrieving multiple colors with a single call. Figure 3.7 illustrates the circular linked list created for the ComCoS algorithm using the cache configuration provided in Figure 3.5. The stack indices are based on the ordering shown in Figure 3.4.

Creating the cache configuration from adjacent colors will facilitate faster initialization of the applications. For example, in Figure 3.5, Application 5 shares the same cache region with Applications 1, 2, and 3. As a result, it will not be executed concurrently with these applications for determinism. Configuring the system, as shown in Figure 3.8, is more optimized for the ComCoS algorithm since it allows for more adjacent colors can be allocated in a single call.

3.4 Comparison of Algorithms

In the static allocation version, the system fills all stacks at system start-up. In the reference design, each color places in a separate stack, while in the ComCoS design, N adjacent pages are placed in a single stack, reducing the number of stash pushes by N times during stack filling. Figure 3.5 demonstrates that each application can be assigned to multiple cache regions. If adjacent cache regions are configured, they can be obtained from the stack in a single operation. However, if none of the

assigned colors for an application are adjacent, additional pop and push operations are performed, directly impacting initialization time. Each color request assigned to an application is fulfilled individually in the reference implementation. Through the accelerated filling of stacks by N times and the capability for applications to receive multiple colors in a single iteration, ComCoS reduces the system start-up time for systems that favor static allocation. Section 4.5 compares the initialization times for different configurations in both the reference and ComCoS implementations with examples.

In dynamic allocation, when an application requests a cache region from reference implementation, the ideal situation is that it can obtain a cache region by performing a single pop operation from the corresponding stack. However, in the worst-case scenario, the algorithm takes *N* pages from the main memory and pushes *N-1* of them to other stacks. This creates a determinism problem between the best and worst cases of the service. In the worst-case scenario of the ComCoS design, the algorithm explicitly requests a single color. If the color is not found in parent stacks, it retrieves memory from the reserved memory area. In this case, it performs $log2(N)$ pop operations and $log2(N)$ push operations, resulting in a total of $2log2(N)$ stack operations. Due to its ability to receive multiple colors in a single request and the reduced memory allocation time difference between best and worst-case scenarios, ComCoS provides faster and more deterministic cache allocation for systems that prioritize dynamic application allocation. Detailed analysis is given in Section 4.6 for dynamic cache allocation.

4. EXPERIMENTAL RESULTS

In this section, the first three experiments demonstrate the determinism issue arising from shared resource utilization and the improvements achieved through cache partitioning. In the final two experiments, the reference implementation and the ComCoS cache allocation algorithm were compared and evaluated in terms of performance and determinism for systems that prefer static and dynamic allocation.

4.1 Environmental Setup

The NXP T2080RDB board has been selected as the target hardware due to its specialized features, including four high-performance PowerPC e6500 processor cores, rich I/O and networking capabilities. The aerospace and defense industry commonly uses that target. Additionally, it contains 2 MB of shared 16-way L2 cache, with a maximum of 32-page colors.

TUBITAK GIS-653 is a Real-Time Operating System (RTOS) [6] developed explicitly for hard real-time safety-critical avionics applications. It is fully compatible with ARINC 653 Part-1 Supplement-5. In addition, it includes many possibilities, such as file system, analyzer agent, hard-soft debugging, advanced scheduling, etc. The experiments are performed on the TUBITAK GIS-653 operating system, implementing both the ComCoS and reference designs. In these implementations, the T2080RDB shared L2 cache is divided into 32 colors, and the physical spaces of the applications are allocated from these colors based on the configuration file.

4.2 Experiment 1: Cache Eviction

Purpose of the Experiment: When one application removes another application's data from the shared cache, it can disrupt the determinism of the evicted applications. This experiment aims to illustrate the cache eviction problem across different cache configurations and demonstrate the improvements achieved through cache partitioning.

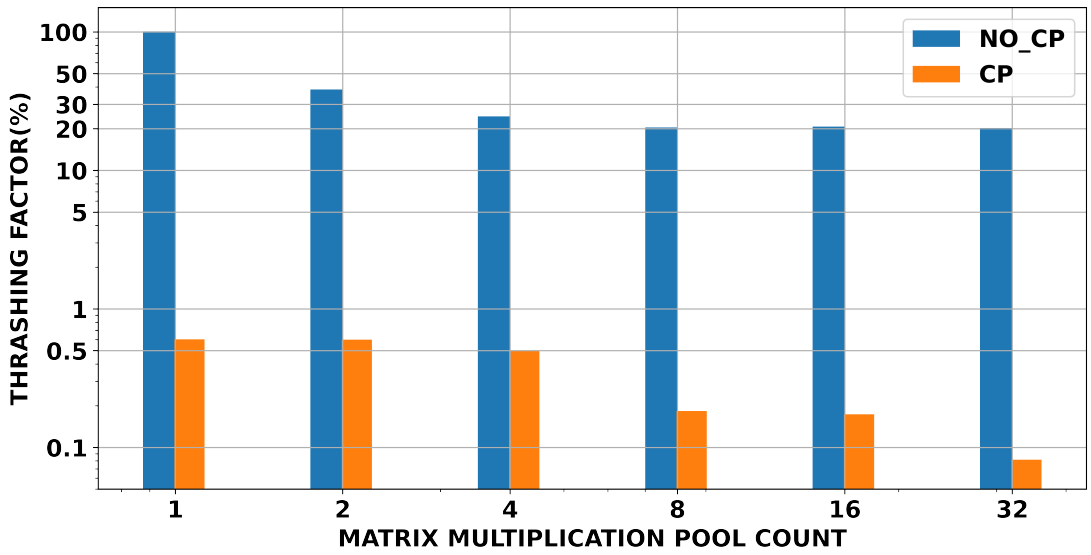


Figure 4.1 : Cache eviction problem.

Methodology: To illustrate the shared cache eviction problem, a 200*200 matrix multiplication scenario is conducted with two defined partitions. The first partition aimed to execute the matrix multiplication on the first core. The second partition aimed to deliberately pollute the cache by making sequential accesses to an array on the second core. In order to assess the impact of a thrasher application on the execution time of matrix multiplication, several experiments are conducted and plotted the results in Figure 4.1. The x-axis represents the number of cache colors assigned to the matrix multiplication application, while the y-axis represents the level of thrashing effect observed when the thrasher application is activated. Two sets of experiments were performed: one where the applications shared the same cache pools (NO_CP) and another where the applications used different cache pools (CP). These experiments aimed to analyze and compare the effects of cache sharing on execution time and thrashing behavior.

Inferences:

- The thrasher application causes a minimum thrashing effect of *20%* when it is located in the same cache region as the application. This situation creates problems in terms of both performance and determinism.

- Operating systems typically strive to avoid fragmentation when allocating physical space for user applications. However, there may be situations where the OS needs to allocate space from a fragmented portion of physical memory, which could result in two applications sharing a small cache pool. When two applications are located in the same cache region, and the thrasher application is active, the thrashing effect can be as high as *101.44%*. This can pose a significant problem for determinism and is considered an edge case for this application.

- Observations have indicated that when smaller caches are shared between applications, there is an increase in the thrashing factor. On the other hand, assigning more than eight cache pools to the matrix multiplication application does not significantly impact the thrashing effects. This observation indicates that the matrix multiplication application demonstrates cache-fitting behavior.

- When applications are located in different cache regions, the thrashing effect remains relatively low, averaging around *0.2%*. In the worst-case scenario of cache partitioning, even where applications are situated in different cache regions, the measured thrashing factor reaches *0.6%*. This phenomenon arises from the utilization of shared memory and buses between cores.

- When *32* cache colors are assigned to the matrix multiplication application and cache partitioning is enabled, the thrasher application operates in a non-cacheable manner. However, a minor thrashing effect of *0.08%* remains due to shared memory and busses.

4.3 Experiment 2: Bus Interferences

Purpose of the Experiment: Simultaneous memory access from different cores can lead to bus interference in multi-core systems. While cache partitioning cannot directly resolve the bus interference problem, it can mitigate cache eviction by ensuring that applications are placed in separate cache regions. That reduces the frequency of accessing the main memory by the applications. This experiment aims to show the bus interference problem using various configurations and highlight the enhancements achieved through cache partitioning.

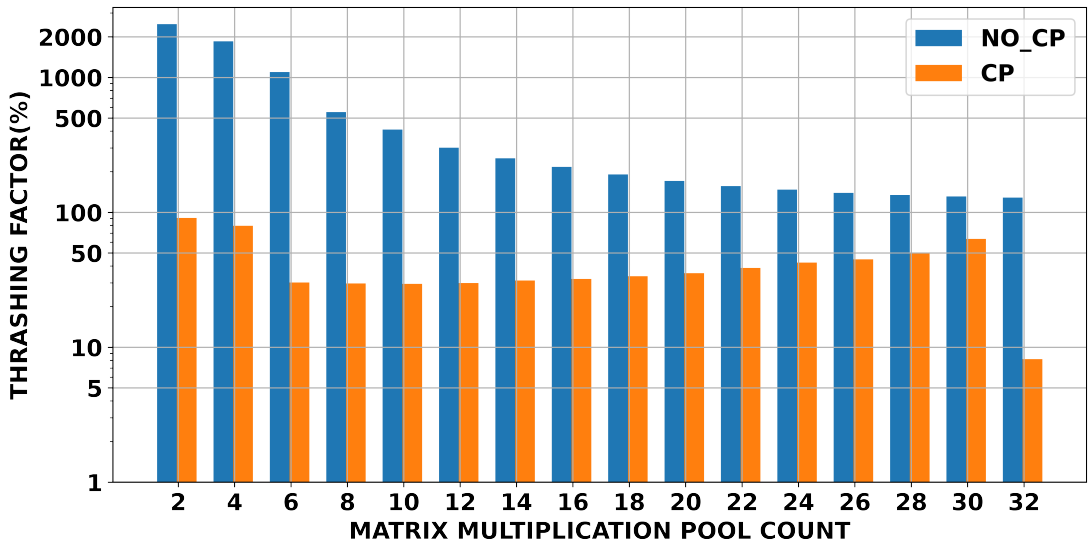


Figure 4.2 : Bus interferences problem.

Methodology: In this example, three thrasher applications are used instead of one, unlike Experiment 4.2. While the first partition aims to perform matrix multiplication on the first core, the remaining three partitions attempt to pollute the same cache by sequentially accessing an array on their respective cores. The three thrasher applications generate continuous memory requests to the main memory by interfering with each other on the same cache. The bus interference problem is depicted in Figure 4.2, using a similar graphical representation as Experiment 4.2.

Inferences:

- In the standard scenario, where *32* cache pools are assigned to all applications, both the matrix multiplication and thrasher applications utilize all cache regions. As a result, a significant thrashing effect of *128%* is observed. This occurs because the thrasher applications, by interfering with each other's cache, create cache contention. Consequently, the data of the matrix multiplication application in the cache is evicted by the thrasher applications.

- In the scenario of no-cache partitioning, all applications can reside in a small cache region due to memory fragmentation. In this case, the cache thrashing effect reaches an unacceptable level of *2483%*. This high value is because all applications are

crowded into a tiny cache region, leading to quick cache pollution and constant memory access.

- In the scenario where cache partitioning is not used(NO_CP), it is observed that increasing the number of cache pools assigned to thrasher applications reduces cache pollution.
- When cache partitioning is enabled, it has been observed that the bus interferences can cause a thrashing effect of up to *30%* in the best case and *90%* in the worst case. This occurs despite cache partitioning being active because the thrasher applications continue evicting each other's data, leading to contention on the main memory.
- To minimize cache misses in the matrix multiplication application, it is recommended to assign a minimum of six cache pools. It is also stated in Experiment 4.2 that the matrix multiplication application displays cache-fitting behavior. To mitigate memory contention caused by thrasher applications, it is crucial to minimize the cache eviction they impose on each other. This can be achieved by allocating the maximum number of cache regions to the thrasher applications. By following these guidelines, it is possible to limit the thrashing effect to *30%*.
- In the final scenario depicted in the graph, all cache regions are assigned to the matrix multiplication application while configuring the thrasher applications as non-cacheable. Even though cache is not shared between applications, it has been observed that there is an *8%* thrashing impact due to the shared memory and busses.

4.4 Experiment 3: Cache Partitioning in Single Core Systems

Purpose of the Experiment: The use of a shared cache can cause both determinism and performance degradation not only in multi-core systems but also in single-core systems. Although cache flushing during application switching eliminates determinism problems, this situation leads to performance degradation. This experiment demonstrates the performance degradation caused by the cache eviction and flushing for different time window configurations for the matrix multiplication application in a single-core system.

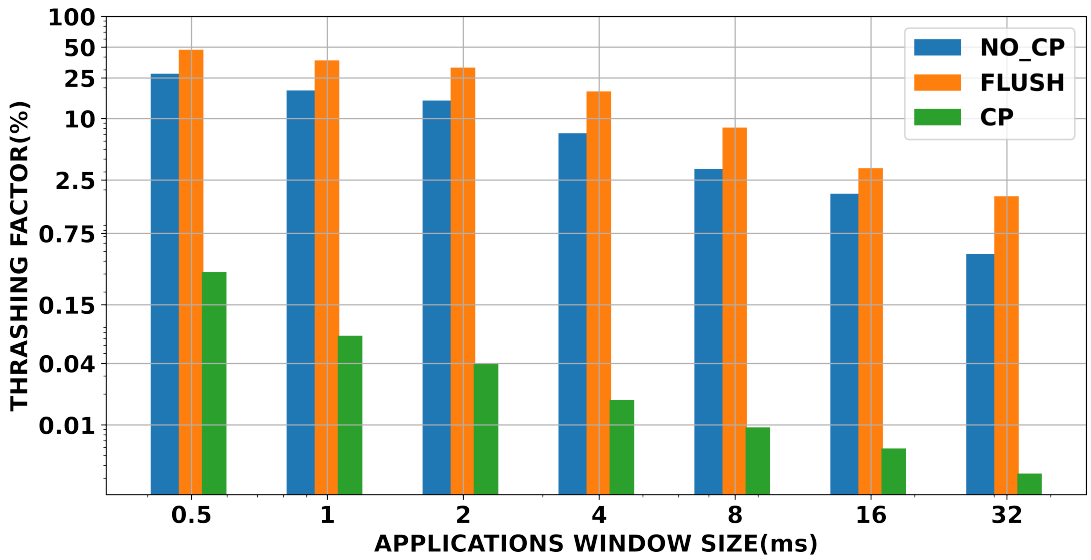


Figure 4.3 : Cache eviction problem in single core systems.

Methodology: To illustrate the shared cache eviction problem in single-core systems, two partitions were defined. While the first partition was engaged in 200*200 matrix multiplication, the second application was responsible for cache trashing. Both partitions were executed sequentially on the same core with equal time windows. In Figure 4.3, the y-axis shows how much of a thrashing effect happens on the execution time of matrix multiplication when the thrasher applications are activated, and the x-axis shows the time window interval of the applications. Two groups of experiments are available as applications using the same cache pools (NO CP) and using different cache pools(CP). These applications have been running in 16 same or different cache pools.

Inferences:

- The multiplication operation is completed within many windows, so data in the L2 cache during matrix multiplication is removed by the thrasher application when the time window ends. The pollution rate increases up to *27.85%* for *0.5 ms* time windows.

- Although the shared cache eviction problem can be resolved by cache flushing in single-core systems, this solution causes performance degradation. Despite eliminating the determinism problem, a significant performance loss of *47.4%* is

observed for the *0.5 ms* time windows. This decrease in performance is due to both cache eviction and the cache flush operation during the matrix multiplication application's time window.

- Running the applications in different cache regions eliminates the determinism problem caused by single-core cache contention. The thrashing effect has decreased up to *0.314%* thanks to cache partitioning.

4.5 Experiment 4: System Initialization Time

Purpose of the Experiment: The CAST-32A guidelines advise against dynamic memory allocation and suggest that application memory allocations should be done at system initialization. However, allocating memory for all applications at system initialization can significantly increase the system's boot time. In safety-critical IMA-based systems, the health monitoring mechanism is triggered if an error occurs. Depending on the severity of the error, the entire module may need to be reset. Especially in aviation applications, the reset time should be as short as possible to prevent catastrophic consequences. In this test, the system boot times for the reference implementation (Algorithm A.1) and ComCoS (Algorithm B.1) were compared using Monte Carlo simulation technique.

Methodology: In complex IMA-based systems, predicting the specific applications and their configurations in advance is challenging. Therefore, relying on a single configuration to analyze the impact of different algorithms on system boot times is insufficient. To address this issue, the Monte Carlo simulation technique is preferred, which allows us to simulate multiple scenarios encompassing various applications and their configurations. This approach enables us to understand better the potential outcomes and variations in system boot times.

Monte Carlo Simulation is a technique that employs statistical methods to model and analyze complex systems or processes through computational methods. The method uses random sampling and probability distributions to simulate and evaluate how the system would behave under various conditions or scenarios.

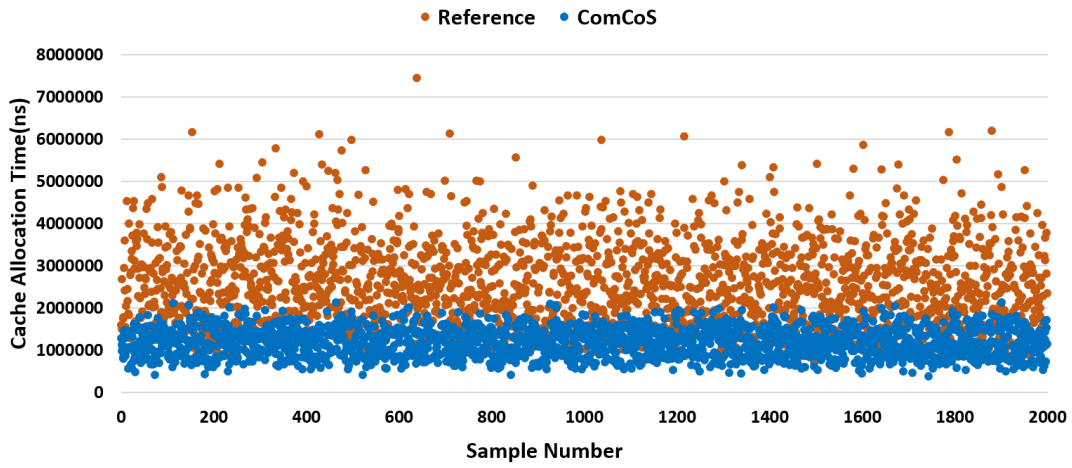


Figure 4.4 : System initialization time.

Random *2000* IMA systems are created within a Monte Carlo simulation. Each system has a random number of applications selected between *10* and *30*. Each application in these systems is created with two parameters.

The first is the stack index in the ComCoS binary tree, where the application wants to reside in the cache. This binary tree stack index is converted into a linear form for separate color allocation requests in a round-robin for reference implementation. Since the T2080rdb has a maximum of *32* colors, the stack index for ComCoS is in the range of *1-63*, while the reference implementation is between *1* and *32*.

The second is the number in the range of *100-1000* which determines how many times the cache is requested. Using uniformly generated random values within the specified ranges for these three parameters, *2000* systems are created and tested on the reference implementation and ComCoS regarding memory allocation time. Since this test examines static initialization, stacks were filled using a large memory block taken from the main memory, and the time it took to fill the stacks was added to the test times.

The distribution of system boot times for the reference and ComCoS implementations are shown in Figure 4.4. The x-axis corresponds to each example of the IMA system number, while the y-axis indicates the memory allocation time of this IMA system.

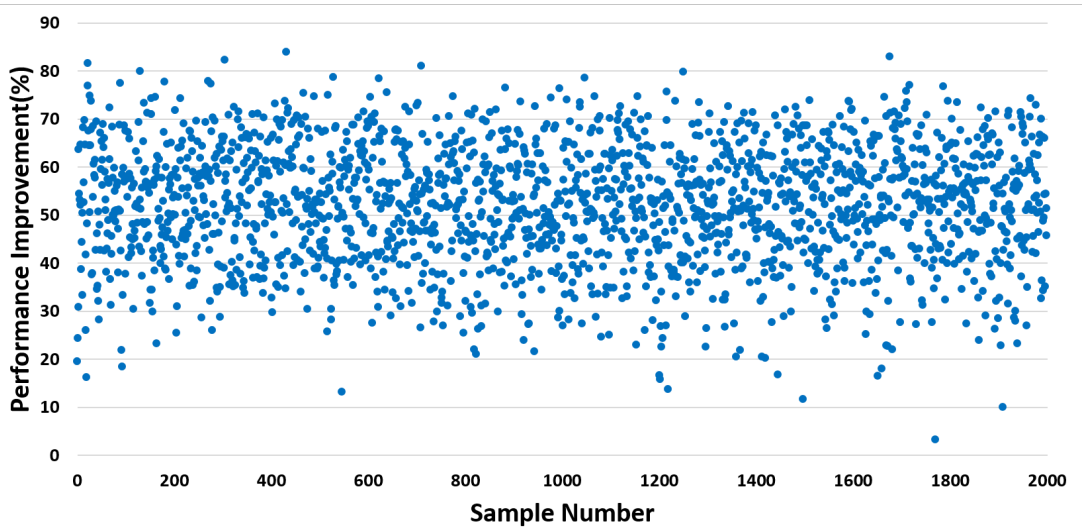


Figure 4.5 : System initialization time improvement.

Figure 4.5 shows the improvement in memory allocation times for *2000* different IMA systems. While the x-axis corresponds to each example of the IMA system number, the y-axis indicates the memory allocation improvement of the IMA system performed by the ComCoS algorithm compared to the reference algorithm.

Inferences:

- The improvement in memory allocation varies depending on the memory needs of the applications within the system, and it gets better as more combined color stacks are used. On average, a *52%* decrease in memory allocation time is observed in the system.
- The results show that the ComCoS technique outperforms the reference implementation regarding the standard deviation of boot time, with a factor of *2.91* (*337 μs* and *982 μs*, respectively). This indicates that the ComCoS algorithm has a more deterministic behavior.

4.6 Experiment 5: Dynamic Allocation Time

Purpose of the Experiment: Applications can be dynamically loaded within the robust and proven limitations outlined in the AMC20-193 directives. In large-scale IMA-based systems, load balancing or hardware failures may require application

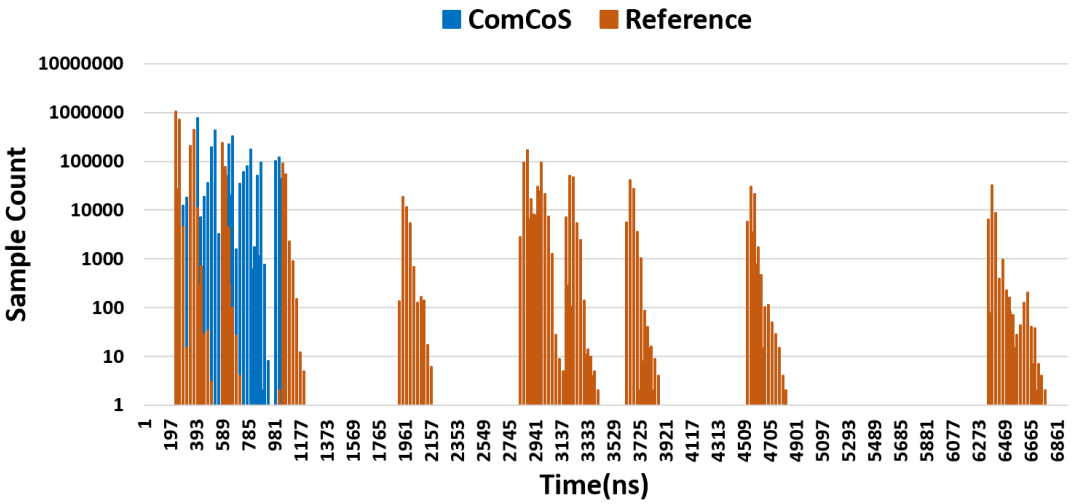


Figure 4.6 : Dynamic allocation time.

migration or dynamic loading. However, any dynamic allocation should not affect other partitions and should exhibit deterministic behavior. The time required to allocate the desired color group is an essential factor in allocating resources for an application. In this test, the allocation time of a color group for both the reference implementation (Algorithm A.1) and ComCoS (Algorithm B.1) was compared using Monte Carlo simulation in terms of WCET and standard deviation.

Methodology: A Monte Carlo simulation environment was created, which is the same as Experiment 4.5. Within this setup, *2000* different systems experienced 5019648 color group allocations in the test scenario. The distribution of the time required for each color group allocation is depicted in Figure 4.6 for both the reference and ComCoS implementations. The graph illustrates the x-axis representing the time taken for color group allocation and the y-axis indicating the number of color group allocations that occurred within each time interval.

Inferences:

- The WCET is calculated as *1947 ns* for the ComCoS implementation and *6774 ns* for the reference implementation. The standard deviation is *224 ns* and *1396 ns*, respectively. ComCoS improved the WCET by a factor of *3.48* and the standard deviation by a factor of *6.23* compared to the reference method.

- The ComCoS algorithm demonstrates much more efficient and deterministic behavior for dynamic allocation.

4.7 Results of the Experiments

The purpose and outputs of the conducted experiments are provided in Table 4.1, along with their respective experiment numbers.

Table 4.1 : Results of the experiments.

No	Purpose	Result
1	Cache partitioning effect on cache eviction problem	Execution time of the matrix multiplication application increased by up to 101,44% due to the trashing effect caused by another application. Implementing cache partitioning can reduce this effect by up to 0,6%.
2	Cache partitioning effect on bus interference problem	Matrix multiplication operation experienced a significant 2483% increase in execution time due to bus interference, while cache partitioning reduced this increase by 30%.
3	Cache partitioning effect on single-core systems	Cache flushing during application switching eliminates determinism problems in single-core systems but causes significant performance degradation. Implementing cache partitioning reduces this performance degradation from 47.4% to as low as 0.314%.
4	The comparison of the reference implementation and the ComCoS technique in terms of system initialization time	ComCoS technique provides an average performance improvement of 52% and reduces the standard deviation in memory distribution by 2,91 times.
5	The comparison of the reference implementation and the ComCoS technique in terms of dynamic allocation time	ComCoS technique achieves a 3,48 times improvement in WCET and a 6,23 times reduction in standard deviation for the memory allocation service.

5. CONCLUSIONS AND FUTURE WORK

As part of the certification process, detailed WCET analyses are conducted for applications and resource allocation is performed based on these analyses. The interference of one application with another within the system poses a significant determinism problem. Particularly in multi-core systems, sharing cache, bus, and memory by applications leads to determinism issues. Cache partitioning techniques can be employed to address the interferences arising from shared resource usage.

There are two main approaches to cache partitioning: hardware-based and software-based. Software-based cache partitioning is more suitable for IMA-based systems due to its hardware independence and higher granularity. Each application has different cache characteristics, making it challenging to determine the cache requirements of system applications at compile time. However, dynamically relocating application caches during system execution is an undesirable dynamic behavior for safety-critical systems. Therefore, the configuration of application caches should be set statically to avoid such dynamic behavior.

This study conducted a series of experiments to illustrate the benefits of cache partitioning in reducing interferences among applications. One example showed that the execution time of a matrix multiplication application increased by up to *101.44%* due to cache eviction interference from another application. Implementing cache partitioning reduced this interference by up to *0.6%*. In another experiment, a matrix multiplication operation experienced a significant *2483%* increase in execution time caused by bus interference, which was mitigated by *30%* through cache partitioning.

Using a shared cache can lead to decreased determinism and performance degradation, not only in multi-core systems but also in single-core systems. While cache flushing during application switching resolves determinism issues in single-core systems, it results in performance degradation. However, implementing cache partitioning

significantly reduced the performance degradation caused by cache flushing from *47.4%* to as low as *0.314%*.

Existing literature maintains each cache region in a separate list, leading to the allocation of pages individually during cache allocation and causing a degradation in performance. To overcome this limitation, a novel cache partitioning technique called ComCoS has been proposed in the thesis. This technique allows for allocating multiple cache pools in a single request, reducing the startup time for systems prioritizing static allocation according to the CAST32-A. Moreover, it provides faster and more deterministic cache allocation for systems that prefer dynamic application allocation, according to the AMC 20-193.

ComCoS technique is tested on the NXP T2080RDB platform and resulted in an average speedup of *52%* and a *2.91* times reduction in the standard deviation of memory initialization time of systems. The cache allocation mechanism also provided a *3.48* times improvement in worst-case execution time and a *6.23* times improvement in standard deviation. Although ComCoS consumes more memory than traditional implementations due to its utilization of additional N-1 stacks for adjacent colors, it provides faster and more deterministic cache allocation.

In safety-critical aviation systems, the use of hypervisors has become increasingly prevalent. The hypervisor isolates non-safe traditional OS and safety-critical RTOS. Cache partitioning is crucial in maintaining isolation and preventing interference between these two operating systems regarding cache, busses, and memory. Since real-time and traditional operating systems coexist, the ComCoS algorithm may need to perform static and dynamic allocation simultaneously. As a future work, the ComCoS cache partitioning algorithm can be implemented on a safety-critical hypervisor. Furthermore, the proposed ComCoS technique can be tested with different applications on various platforms in future works.

REFERENCES

[1] **Radio Technical Commission for Aeronautics (RTCA Inc)** (2011). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*.

[2] **Watkins, C.B.** (2006). Integrated Modular Avionics: Managing the Allocation of Shared Intersystem Resources, *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*, pp.1–12.

[3] **Watkins, C.B. and Walter, R.** (2007). Transitioning from federated avionics architectures to Integrated Modular Avionics, *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pp.2.A.1–1–2.A.1–10.

[4] **Ramsey, J.W.** (2007). *Integrated Modular Avionics: Less is More*, `https://www.aviationtoday.com/2007/02/01/integrated-modular-avionics-less-is-more/`.

[5] **Green Hills Software** (2023). *INTEGRITY Real-Time Operating System*, `https://www.ghs.com/products/rtos/integrity.html`.

[6] **TUBITAK BILGEM** (2023). *TUBITAK GIS Real-Time Operating System*, `https://bilgem.tubitak.gov.tr/en/content/rtos/gis-real-time-operating-system`.

[7] **Aeronautical Radio, Incorporated (ARINC Inc)** (2019). *ARINC 653 Specification: Avionics application software standard interface*.

[8] **Certification Authorities Software Team (CAST)** (2016). *CAST-32A, Multi-core Processors*.

[9] **European Union Aviation Safety Agency (EASA)** (2022). *AMC 20-193, Use of multi-core processors*.

[10] **Hennessy, J.L. and Patterson, D.A.** (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition.

[11] **Wind River Software** (2023). *What Is DO-178C?*, `https://www.windriver.com/solutions/learning/do-178c`.

[12] **Rapida Systems** (2023). *AMC 20-193*, `https://www.rapitasystems.com/amc-20-193#cast-32a`.

[13] **Schuster, T. and Verma, D.** (2008). Networking concepts comparison for avionics architecture, *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pp.1.D.1–1–1.D.1–11.

[14] **Liu, F.**, **Jiang, X. and Solihin, Y.** (2010). Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance, *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp.1–12.

[15] **Yun, H.**, **Yao, G.**, **Pellizzoni, R.**, **Caccamo, M. and Sha, L.** (2016). Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms, *IEEE Transactions on Computers*, *65*(2), 562–576.

[16] **Suzuki, N.**, **Kim, H.**, **Niz, D.d.**, **Andersson, B.**, **Wrage, L.**, **Klein, M. and Rajkumar, R.** (2013). Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses, *2013 IEEE 16th International Conference on Computational Science and Engineering*, pp.685–692.

[17] **Torres Aurora Dugo, A.**, **Lefoul, J.B.**, **Harnois, S.**, **Gohring de Magalhaes, F. and Nicolescu, G.** (2022). Certifiable Memory Management System for Safety Critical Partitioned System, *ERTS2022*, Toulouse, France, `https://hal.science/hal-03697093`.

[18] **Mittal, S.** (2017). A Survey of Techniques for Cache Partitioning in Multicore Processors, *ACM Comput. Surv.*, *50*(2), `https://doi.org/10.1145/3062394`.

[19] **NXP Semiconductors** (2014). *e6500 Core Reference Manual, Rev 0*.

[20] **Kaseridis, D.**, **Iqbal, M.F. and John, L.K.** (2014). Cache Friendliness-Aware Managementof Shared Last-Level Caches for HighPerformance Multi-Core Systems, *IEEE Trans. Computers*, *63*(4), 874–887, `https://doi.org/10.1109/TC.2013.18`.

[21] **Subramanian, L.**, **Seshadri, V.**, **Ghosh, A.**, **Khan, S.M. and Mutlu, O.** (2015). The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory, ***M. Prvulovic**, editor, Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, ACM, pp.62–75, `https://doi.org/10.1145/2830772.2830803`.

[22] **Kim, S.**, **Chandra, D. and Solihin, Y.** (2004). Fair cache sharing and partitioning in a chip multiprocessor architecture, *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pp.111–122.

[23] **Farshchi, F.**, **Valsan, P.K.**, **Mancuso, R. and Yun, H.** (2017). Deterministic Memory Abstraction and Supporting Cache Architecture for Real-Time Systems, *CoRR*, *abs/1707.05260*, `http://arxiv.org/abs/1707.05260`, `1707.05260`.

[24] **Tam, D.**, **Azimi, R.**, **Soares, L.B. and Stumm, M.** (2007). Managing shared L2 caches on multicore systems in software, *Proceedings of the Workshop on the Interaction Between Operating Systems and Computer Architecture*, pp.26–33.

[25] **Ward, B.C.**, **Herman, J.L.**, **Kenna, C.J. and Anderson, J.H.** (2013). Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms, *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, IEEE Computer Society, pp.157–167, https://doi.org/10.1109/ECRTS.2013.26.

[26] **Ye, Y.**, **West, R.**, **Cheng, Z. and Li, Y.** (2014). COLORIS: a dynamic cache partitioning system using page coloring, ***J.N. Amaral and J. Torrellas**, editors, International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, ACM, pp.381–392, https://doi.org/10.1145/2628071.2628104.

[27] **Qureshi, M.K. and Patt, Y.N.** (2006). Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp.423–432.

[28] **Brock, J.**, **Ye, C.**, **Ding, C.**, **Li, Y.**, **Wang, X. and Luo, Y.** (2015). Optimal Cache Partition-Sharing, *2015 44th International Conference on Parallel Processing*, pp.749–758.

[29] **Kandemir, M.**, **Yemliha, T. and Kultursay, E.** (2011). A helper thread based dynamic cache partitioning scheme for multithreaded applications, *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp.954–959.

[30] **Zhan, D.**, **Jiang, H. and Seth, S.C.** (2014). CLU: Co-Optimizing Locality and Utility in Thread-Aware Capacity Management for Shared Last Level Caches, *IEEE Transactions on Computers*, *63*(7), 1656–1667.

[31] **Bitirgen, R.**, **Ipek, E. and Martinez, J.F.** (2008). Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach, *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp.318–329.

[32] **Liu, L.**, **Li, Y.**, **Cui, Z.**, **Bao, Y.**, **Chen, M. and Wu, C.** (2014). Going vertical in memory management: Handling multiplicity by multi-policy, *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp.169–180.

[33] **Certification Authorities Software Team (CAST)** (2003). *Cast-20, addressing cache in airborne systems and equipment*.

[34] **Park, J.**, **Yeom, H.Y. and Son, Y.** (2020). Page Reusability-Based Cache Partitioning for Multi-Core Systems, *IEEE Trans. Computers*, *69*(6), 812–818, https://doi.org/10.1109/TC.2020.2968066.

APPENDICES

APPENDIX A : Pseudo Code of Reference Implementation
APPENDIX B : Pseudo Code of ComCoS Technique

APPENDIX A : Pseudo Code of Reference Implementation

Algorithm A.1 Cache allocation from Separate Color Stacks

```
/* Memory allocated from indexed stack(Color) */
function CACHE_ALLOC(stack_index)
    address ← POP(cache_stacks[stack_index-1])
    if address ≠ NULL then
        return address
    else
        if DYNAMIC_ALLOCATION_ENABLED then
            /* Memory is obtained from the reserved space for dynamic allocation. */
            address ← PAGE_ALLOC(N)
            if address ≠ NULL then
                PUSH_PAGES(address, stack_index)
                /* Address is updated according to index */
                address ← UPDATE(address, stack_index)
                return address
            else
                FAULT(MEMORY_EXHAUSTED)
            end if
        else
            /* All partitioned memory regions are pre-configured, and no dynamic
            allocation is performed. */
            FAULT(MEMORY_EXHAUSTED)
        end if
    end if
end function

/* Remaining pages are pushed to stacks */
function PUSH_PAGES(address, stack_index)
    for index ← 1 to N do
        if index ≠ stack_index then
            address ← UPDATE(address, index)
            PUSH(cache_stacks[index-1], address)
        end if
    end for
end function
```

APPENDIX B : Pseudo Code of ComCoS Technique

Algorithm B.1 ComCoS cache allocation algorithm (Part 1)

```
/* Memory allocated from indexed stack(Color) */
function CACHE_ALLOC(stack_index)
    address ← ALLOC_FROM_STACKS(stack_index)
    if address ≠ NULL then
        return address
    else
        if DYNAMIC_ALLOCATION_ENABLED then
            /* N pages allocated from from the reserved space for dynamic allocation */
            address ← PAGE_ALLOC(N)
            if address ≠ NULL then
                PUSH_PAGES(address, stack_index, ROOT_INDEX)
                /* Address is updated according to index */
                address ← UPDATE(address, stack_index)
                return address
            else
                FAULT(MEMORY_EXHAUSTED)
            end if
        else
            /* All partitioned memory regions are pre-configured, and no dynamic allocation
            is performed. */
            FAULT(MEMORY_EXHAUSTED)
        end if
    end if
end function
```

Algorithm B.2 ComCoS cache allocation algorithm (Part 2)

```
/* Try allocate cache region from stack_index or the parents of this index.*/
function ALLOC_FROM_STACKS(stack_index)
    address ←NULL
    index ←stack_index
    while index ≠ ROOT_INDEX do
        address ← POP(cache_stacks[index-1])
        if address ≠ NULL then
            PUSH_PAGES(address, stack_index, index)
            address ← UPDATE(address, stack_index)
        else
            index ← GET_PARENT_INDEX(index)
        end if
    end while
    return address
end function

/* The remaining pages from the received memory are pushed to the relevant
stacks */
function PUSH_PAGES(address, index, parent_index)
    while index ≠ parent_index do
        index ← GET_SIBLING_INDEX(index)
        PUSH(cache_stacks[index-1], address)
        index ← GET_PARENT_INDEX(index)
    end while
end function
```

CURRICULUM VITAE

Name SURNAME: Yakup HÜNER

EDUCATION:

- **B.Sc.:** 2019, Istanbul Technical University, Faculty of Electrical and Electronics Engineering, Electronics and Communication Engineering
- **B.Sc.:** 2020, Istanbul Technical University, Faculty of Computer and Informatics Electronics, Computer Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2018 1. Place in Fighting UAV Rotary Wing category in Istanbul Teknofest
- 2019 2. Place in Fighting UAV Rotary Wing category in Istanbul Teknofest
- 2019 Graduated with second-highest honors from ITU Electronics and Communication Engineering Bachelor Degree
- 2019-2020 Part-Time Embedded Developer at the ITU Aerospace Research Center.
- 2020-2023 Software Engineer at the TUBITAK BILGEM.

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Huner Y.**, Yeniçeri R. (2023). Evaluation of Cache Partitioning for Integrated Modular Avionics, *2nd International Graduate Research Symposium IGRS'23*, March 16-18, 2023 Istanbul, Turkey.
- **Huner Y.**, Yeniçeri R. (2023). ComCoS: Enhanced Cache Partitioning Technique for Integrated Modular Avionics, *26th Euromicro Conference on Digital System Design (DSD)*, Sept. 6-8, 2023 Durres, Albania.

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

- Yeniçeri R., **Hüner Y.** (2020). HW/SW Codesign and Implementation of an IMU Navigation Filter on Zynq SoC with Linux. *7th International Conference on Electrical and Electronics Engineering (ICEEE)*, Antalya, Turkey, 2020, pp. 351-354.

- **Hüner Y.**, Gayretli M.G., Yeniçeri R., (2021). HW/SW Design Space Exploration of A Complementary Filter on Zynq SoC. *8th International Conference on Electrical and Electronics Engineering (ICEEE)*, Antalya, Turkey, 2021, pp. 1-5.