Caching in a Mixed-Criticality 5G Radio Base Station

Emad Jacob Maroun[1,*], Luca Pezzarossa[1] and Martin Schoeberl[1]

[1]Technical University of Denmark, Department of Applied Mathematics and Computer Science

Abstract

Telecommunication is a critical driver of economic and social development. 5G technologies are state-of-the-art in telecommunication, setting strong and open-ended requirements for implementing systems. Current systems for implementing baseband technologies in 5G depend on hardware separation to ensure high- and low-criticality tasks do not interfere in such a way as to violate guarantees. To increase performance and lower costs, this paper sets the research direction into future mixed-criticality systems that can handle both the high- and low-criticality tasks of the baseband unit. We analyze the 5G requirements and the common systems that currently implement them. We propose using T-CREST as the research platform with a specific architecture targeting mixed-criticality workloads. We present two cache proposals to reduce the interference of low-criticality tasks on high-criticality tasks but ensure high cache utilization and efficiency. The first cache proposal uses timeouts to automatically free cache lines reserved for high-criticality tasks. The second proposal uses contention tracking to limit how much low-criticality tasks may influence high-criticality tasks. Lastly, we propose a third cache architecture to unify the method and stack caches unique to T-CREST into a single level-2 cache.

Keywords

5g, t-crest, real-time systems, low latency, caches, radio baseband

1. Introduction

Socio-technical evolution is dependent on mobile communications as a critical driver to allow for economic and social development [1]. As such, the evolution of communication technologies is essential in enabling societal development. 5G is state-of-the-art in mobile communication technologies, promising unprecedented speeds, ultra-low latency, and massive connectivity capabilities. With its lofty promises, implementing 5G communication networks is a significant industrial challenge. Continued investment in 5G technologies is needed to reach beyond the minimal promises of the technology. Improvements in technical implementations will ensure better service characteristics for customers and users at lower costs.

One critical aspect of telecommunications technology is the radio base station (RBS), which provides wireless transmission to and from mobile devices. The 5G functionality is implemented in these RBSs. Continued improvement of the RBS is critical to staying at the forefront of the industry. As such, research on how to best implement RBS for optimizing performance and cost ensures long-term competitiveness in the industry.

The requirements of 5G introduce a hierarchy of prioritized tasks that the RBS has to complete. The RBS, therefore, becomes a mixed-criticality system [2], where minimum guarantees are upheld to ensure critical tasks are completed correctly and in a timely fashion. On the other hand, non-critical tasks need to be performed as fast as possible; however, they only need to provide good quality of service (QoS) on average, so they may be de-prioritized to ensure that critical tasks meet their deadlines. To ensure non-critical tasks do not interfere with the critical ones, hardware systems are divided into several layers with differing responsibilities correlating to the open systems interconnection (OSI) model [3]. This hardware division makes it easier to control interference but decreases resource utilization, which hurts performance and price. Therefore, we are interested in investigating future system designs incorporating mixed-criticality system research to merge the currently divided systems into a single platform that can handle the varying criticality of tasks. While the current heavy use of shared scratchpads and the phased execution model [4] give high predictability to systems managing the OSI layer 1, it is wasteful and difficult to unify with the use of shared caches in the systems managing the OSI layer 2. Therefore, innovative techniques are needed to facilitate the unification of the layer 1 and layer 2 systems into a unified hardware system.

This paper addresses the challenge of sharing a level 2 (L2) cache between different tasks and executing on different cores while still delivering low-latency execution of critical tasks. We propose to use the T-CREST platform [5] to explore different solutions of the challenges around memory management for mixed-criticality systems by presenting three distinct caching architectures for future exploration. All solutions are centered around regulating access to different cache lines for high- and low-criticality jobs. More specifically, we propose two shared caches that use timeouts and contention tracking to limit the interference of low-criticality tasks on high-criticality ones, as well as an L2 cache that unifies the split caches unique to T-CREST since they exhibit unique access characteristics that can be sped up predictably.

The contributions of this paper are: (1) A description of common 5G RBS technologies and implementations, (2) a discussion of the challenges future systems face in the pursuit of lower cost, higher efficiency, and improved performance, and (3) three proposals for caching architectures that we intend to explore to address the challenges described.

The rest of this paper is structured into four sections. The following section will provide some background on how current systems implement 5G and their challenges. Section 3 introduces the T-CREST platform and how it can be used as a basis for research into a mixed-criticality system. Section 4 discusses the three cache architecture proposals. Section 5 presents related work and Section 6 concludes the paper.

2. 5G Radio Baseband

During the initial phases of the 5G specification, three usage scenarios were identified as being critical for the future of

3rd workshop on Resource AWareness of Systems and Society (RAW 2024), July 2–5, 2024, Maribor, Slovenia

*Corresponding author.

ejama@dtu.dk (E. J. Maroun); lpez@dtu.dk (L. Pezzarossa); masca@dtu.dk (M. Schoeberl)

0000-0002-3675-3376 (E. J. Maroun); 0000-0002-0863-2526 (L. Pezzarossa); 0000-0003-2366-382X (M. Schoeberl)

© 2024 Copyright © 2024 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

CEUR-WS.org/Vol-3867/paper5.pdf

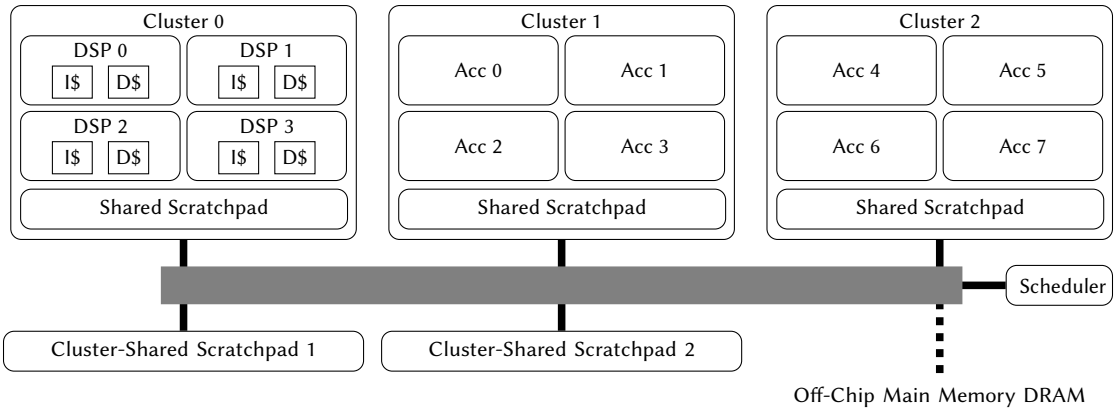

Figure 1: Hypothetical baseband unit architecture.

mobile communications [1]:

Enhanced Mobile Broadband (eMBB): Focuses on providing significantly higher data rates and capacity compared to previous telecommunication generations, enabling applications such as high-definition video streaming, virtual reality, and augmented reality. This scenario covers the day-to-day activities of private users and data-heavy but less critical industrial applications.

Ultra-Reliable and Low Latency Communications (URLLC): Emphasizes ultra-reliable and low-latency communication, critical for applications that demand real-time responsiveness and mission-critical reliability, including autonomous vehicles, remote surgery, and industrial automation.

Massive Machine Type Communications (mMTC): Targets the connectivity of a massive number of devices using minimal energy, enabling the Internet of Things (IoT) to scale to unprecedented levels, facilitating applications such as smart cities, industrial IoT, and environmental monitoring.

These scenarios resulted in a requirement specification that includes the following criteria [6]:

- Peak Data Rate: 20 Gbit/s download, 10 Gbit/s upload. This is only in ideal conditions.
- Transmission Latency: 4 ms for eMBB, 1 ms for URLLC. This is the latency added by the 5G network to the overall communication latency between endpoints.
- Device Mobility: up to 500 km/h for rural eMBB, less for more dense areas.
- Density: up to 1.000.000 devices per square kilometer in the mMTC scenario.

Note how each requirement applies in specific scenarios and is not necessary in others. For example, the peak data rate is unnecessary for scenarios covered by URLLC or mMTC. Meanwhile, the extreme latency requirement of 1 ms only applies to URLLC.

An RBS must manage these diverse requirements and, therefore, becomes a mixed-criticality system. For example, tasks within the URLLC scenario must be prioritized over eMMB tasks to uphold the URLLC latency requirements. Not only do we have a range of priorities, but these priorities may also change as usage changes. Adapting to ongoing changes in network usage is, therefore, a critical aspect of implementing 5G.

2.1. System Architecture

Typical RBS systems are divided into three hardware units:

1. The Remote Radio Unit (RRU). It is immediately connected to the antennas and handles the initial input stream from the antennas. The antenna streams are initially processed in this unit and grouped into user streams (e.g., 8 antenna streams are compressed to one group) to be sent to the next unit.
2. The Baseband Unit (BBU). It takes the input streams from the RRU and further processes them. The RRU and BBU units together constitute the physical layer of the OSI model (layer 1), handling the physical aspects of transmitting and receiving wireless 5G signals [7].
3. The Layer 2 unit handles the data link layer of the OSI model (layer 2). This includes Medium Access Control (MAC) and Radio Link Control (RLC) tasks.

The varying characteristics of the workloads of the different units result in different hardware designs. While both the BBU and layer 2 must handle high- and low-criticality tasks, they do so in different ways. This research aims to explore a merged system to handle the BBU and layer 2 tasks in one hardware system. The new system is to be centered around the design of a BBU but explore technologies that allow layer 2 tasks to run efficiently.

2.2. Baseband Unit

The BBU system handles physical layer tasks centered around signal processing of incoming and outgoing transmissions. Its design ensures maximum predictability at the expense of resource utilization efficiency. Figure 1 provides an overview of the system. It is not meant to be representative of any specific system but to give an idea of the components often present and their interactions.

2.2.1. Hardware

We focus on systems centered around a clustered and heterogeneous design. Each cluster contains a set of processors or accelerators (for illustration, we show four in Figure 1). First, the general computing capability is provided by digital signal processor (DSP) cores with high predictability [8]. Each DSP has a private instruction and data cache and shares a

single scratchpad memory with the other processors in the cluster.

The other clusters contain acceleration cores for specific and common workloads. The accelerators in each cluster also share a scratchpad. The exact architecture of the accelerators is out of the scope of this paper.

The clusters may also share scratchpads, two are shown as an example. These split scratchpads handle different data with specific access characteristics. For example, some configuration data might be mostly read and changed rarely, while user-specific data may be updated continuously.

Lastly, a hardware scheduler can be present to orchestrate task execution on the relevant cores and movement of data. We have omitted to describe any other application-specific devices or connections to peripherals.

2.3. Data Processing

Data processing starts once every millisecond. While the RRU is processing the antenna streams, the BBU starts with a set of configuration tasks that prepare for the delivery of data from the RRU. These configuration tasks must run on the DSP cores to, e.g., configure the accelerators before they start executing. This could result in configuration data initially going to one of the cluster-shared scratchpads, from where it is moved to the cluster scratchpads as needed. This data starts in the shared scratchpad of the core running the job and is off-loaded to the cluster-shared scratchpad when the configuration job is done. In parallel with the configuration tasks, the data from the RRU is being loaded into the cluster-shared scratchpads. When that is ready, proper processing tasks can begin executing on DSPs or accelerators as needed.

We consider only strict data access characteristics of the tasks. All shared data is read-only. User-specific data is segmented into the relevant tasks and updated only by the task currently being worked on. At no point are two tasks working on the same user data. These strict data access characteristics mean that synchronization and coherence are not issues we will consider.

2.3.1. Phased Execution

The use of scratchpads in the BBU reduces the variability in execution times. However, this requires methodical orchestration to ensure each job has the needed data. As such, every job is divided into three phases:

1. **Read**: Any data a task requires is moved onto its cluster's scratchpad from the cluster-shared caches.
2. **Execute**: The task's job is executed to completion without needing to access memory other than the cluster's scratchpad.
3. **Write**: All the data previously fetched for the job, which has been updated, is written back to the main memory.

This is a classic implementation of the phased execution [4, 9], also called the simple-task model [10]. The task scheduler ensures that a task's **Execute** is only scheduled on a processor when its corresponding **Read** has terminated on the same cluster. Data movement is performed using DMAs, allowing processors to execute other jobs' **Execute** phase in parallel with data movements.

A cluster's scratchpad is partitioned so that each running job has exclusive access to its memory portion. If two tasks use the same data, the **Read** of each will load that data into their respective partitions. This means data might be duplicated in the cluster scratchpads. However, such shared data is rarely written to, and synchronization is explicitly handled at the application level and, therefore, is not an issue.

2.4. Layer 2 Design

The common computing architectures for layer 2 are more traditional, with, e.g., superscalar cores and standard caching. The workload on the system requires less stringent predictability than the BBU, allowing for a more traditional design. The tasks also require higher performance, provided by the more complex design at the cost of predictability. To ensure high-criticality tasks meet their deadlines, the hardware resources can be partitioned by clusters and intentionally over-provisioned.

Layer 2, therefore, can have much wastage where high-criticality tasks are concerned. This unit's more complex design makes it challenging to ensure tasks meet their deadlines. The only possibility of ensuring the deadlines are met is to provide the tasks with such an overabundance of resources that even when low-criticality tasks interfere, the high-criticality tasks will not be adversely affected. Therefore, the inefficient use of resources in layer 2 is a supporting reason for merging the layer 2 subsystem with the BBU subsystem.

2.5. Challenges

We aim to research new methods for implementing 5G RBS technologies to achieve better performance at lower cost. Therefore, the current challenges of increased costs and lower performance must be alleviated in any future system.

Challenge 1: The primary challenge for the above-mentioned RBS systems is a divided hardware architecture. The physical division ensures that high-critical tasks can maintain their needed deadlines, which increases costs and reduces overall performance. First, the separation necessitates manufacturing two physical systems, which is costly. Second, the separation means the two systems cannot share resources, reducing the efficient use of available resources.

Challenge 2: On the BBU system specifically, there is also a challenge with efficient use of resources. While using scratchpads ensures execution-time predictability for all tasks, it also forces data duplication. If two tasks use the same data, that data is moved into both tasks' scratchpads partition. This is both a waste of scratchpad memory and memory bandwidth. This is especially prevalent with configuration data, which is often shared between many tasks and does not change often. The data loaded into the scratchpads is also loaded on a pessimistic basis. Some tasks may only need some of the data, meaning some data might be unnecessarily loaded into the scratchpads.

Challenge 3: Memory bandwidth is wasted when dependent tasks use the same data. The **Write** phase in the BBU system always runs after the **Execute** phase. A subsequent job using the same data must reload it in its **Read** phase. This is sub-optimal in cases where the subsequent task can run on the same cluster as the first task. In such a case, omitting the **Write** phase of the first task and the **Read** phase of the second task would be better.

3. The T-CREST Platform

We propose to use the T-CREST platform as a basis for research into future platforms for 5G RBS. This section describes the platform's current capabilities and how they relate to the challenges present in divided RBS systems.

3.1. T-CREST and Patmos

The Patmos processor [11] is designed to serve real-time systems. Several Patmos cores are combined with a network-on-chip, a memory arbitration tree, and a memory controller to the time-predictable multi-core platform T-CREST [5]. As such, T-CREST provides techniques that make task execution time more predictable and reduce the worst-case execution time (WCET). Around the Patmos cores, it builds a platform with time-predictable components to reduce WCET analysis complexity and increase accuracy. T-CREST uses networks-on-chips [12, 13, 14] that ensure data is moved between processing cores with a known maximum latency. For accessing shared main memory, T-CREST uses the dedicated arbitration tree-based network-on-chip [15]. Regardless of how many cores are accessing the memory, each access will be serviced within a bounded latency.

Patmos uses an in-order pipeline to ensure every instruction has a known and constant execution time. To exploit instruction-level parallelism predictably, Patmos is also a very long instruction-word (VLIW) architecture with a dual-issue pipeline. VLIW architectures are a predictable way of increasing performance without increasing complexity [16, 17]. Patmos executes instructions in *bundles* of up to two instructions. The compiler must designate instructions as part of a bundle by setting a specific bit in the first instruction. All Patmos instructions are predicated: Based on one of eight predicate registers, each instruction is either *enabled* or *disabled*. If the predicate register's value is true, the instruction is enabled, meaning it executes normally. If the value is false, the instruction is disabled and does not affect registers or memory. It effectively becomes a *noop*. However, the execution time of disabled instructions is the same as when enabled. Predicated instructions allow the compiler to minimize execution time variability or even eliminate it entirely [18].

3.2. Predictable Caching

While caching is usually associated with unpredictability and difficulties for static analysis, T-CREST deploys two predictable and easily analyzable caches. The first is a method cache [19] that replaces a traditional instruction cache in Patmos [20]. the method cache caches whole or parts of functions (sub-functions) such that instruction fetching never misses except at specific points. The compiler manages this cache by splitting the code into blocks that fit in the method cache and inserting cache-fill instructions where needed. For the Patmos ISA function call and return instructions ensure that the callee or the caller are in the method cache. To support sub-function caching Patmos has cache filling variants of branch instructions. Using a method cache limits the number of places cache misses can occur to the specific cache-filling instructions. The method cache is simpler to model for an analyzer to provide tight WCET bounds [21].

The second unique cache of the T-CREST is the stack cache [22]. It caches function-local data, often accessed predictably, and can be loaded at function entry and exit points. Accessing this data is also done without experiencing cache misses. The compiler also manages the stack cache, setting it up and tearing it down at function entry and exits and using stack-targeting load and store instruction variants. An analyzer can assume any stack-targeting instruction will hit in the stack cache. Therefore, the cache size must only be modeled to account for the stack setup and tear-down time [23]. Data accesses that are not function-local may still go through the conventional data cache or circumvent all caching to target the main memory directly.

These two cache architectures are supported by the Platin WCET-analyzer [24]. Platin models instruction execution and tracks which blocks of code are likely to be in the method cache at a given point. It accounts for this at control-flow point to know whether a method-cache miss is likely and how many bytes would have to be loaded. For the stack cache, it models the program stack's size at any point and tracks stack-cache-control instructions added by the compiler. At points where the stack must grow, Platin knows whether the cache has free space or needs to spill some of the program stack to main memory.

3.3. Missing Capabilities

The T-CREST platform is missing some features and capabilities compared to the BBU system. We will enumerate these missing capabilities and highlight how we might either simulate them using existing capabilities or discuss how to implement them into the platform as part of the research project.

3.3.1. Acceleration and Clustering

The specific processing requirements of an RBS means dedicated accelerators can be used for maximum efficiency. The T-CREST platform does not include anything resembling these accelerators. Likewise, the T-CREST platform does not use any clustering, whose benefit is mainly driven by a multi-layered intermediate memory, which we will discuss in the next section.

As this research mainly focuses on the efficient use of resources, notably memory, we will not investigate or implement any hardware acceleration. Instead, we will use the Patmos cores as substitutes for specific accelerators. We will implement clustering into the T-CREST platform so that each cluster can be designated to be allowed to execute specific tasks. This will allow us to treat one cluster as a substitute for a BBU DSP cluster and others for different types of acceleration clusters.

3.3.2. Hierarchical Memory

The Patmos cores of T-CREST are each paired with private caches, as described earlier. However, no further hierarchy of intermediate memory exists. In contrast, the BBU system contains three levels of intermediate storage: First, each DSP (or accelerator) has its caches. Second, each cluster has a shared scratchpad. lastly, cluster-shared scratchpads are present for a last level of storing various types of data.

A multi-layered memory hierarchy is necessary for the experiments to be representative, especially given the unique data access characteristics. Therefore, we will build a second layer of intermediate memory, which is shared between the Patmos cores of each cluster. We will omit a last memory layer, as any methods of managing the second layers

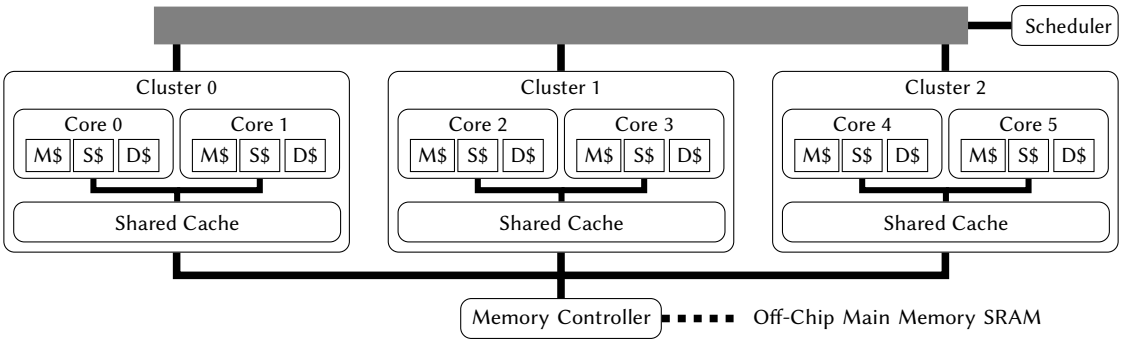

Figure 2: Proposed T-CREST system for researching novel cache architectures.

we develop can be transferred to the rest of the layers of a real-world system.

3.3.3. Hardware-Assisted Scheduling

The BBU systems often use hardware to accelerate scheduling. T-CREST does not implement any hardware that can assist with scheduling. While using a hardware scheduler in the BBU system ensures that the extreme amount of tasks gets scheduled in a reasonable time, the smaller scale of this project's prototypes can likely handled by software-managed scheduling.

Therefore, the initial proposed system will not have any scheduling hardware, but dedicated Patmos cores will replace it to handle the scheduling. Software-defined scheduling can be a flexible way to test our scheduling strategies as the system matures. Moving to a hardware scheduler should be easily doable at later stages of research, where the scheduling has been studied and techniques chosen. Patmos already supports adding custom devices and accelerators[25]. A hardware scheduler is a device that interacts with the rest of the clusters, memories, and processors and issues commands in the same manner a Patmos core would.

3.4. Proposed System Architecture

Figure 2 shows a diagram of our proposed system. It comprises three clusters, each with a set of Patmos cores with private split caches (**M**ethod, **S**tack, and **D**ata) and a shared cluster cache. The cores use the T-CREST memory tree to access the shared cache, providing us with predictable and low-latency access. The clusters use the T-CREST memory tree to connect to the memory controller, which manages access to the off-chip, main memory. A shared bus (in gray above the clusters) facilitates cross-cluster and cross-core communication. This allows a Patmos core or a hardware device scheduler to issue scheduling commands to the whole system.

This system architecture will allow research on efficiently managing the cluster caches. The different clusters can simulate the DSP or accelerator clusters on the BBU system, while the cluster-shared scratchpads of that system do not introduce new challenges. Therefore, limiting ourselves to the two levels of cache (private and cluster caches) will allow for fruitful experimentation during the research.

4. Cache Proposals

To start addressing the challenge of merging layer 1 and layer 2 systems, we focus on the challenge of using a shared cache in each cluster. As described earlier, the BBU architecture sacrifices the efficient use of resources to ensure low variability in execution times. We aim to maximize resource usage in the proposed system while maintaining low variability. We propose exploring three caching solutions that address the challenges of predictable caching: (1) a criticality timeout cache, (2) a contention tracking cache, and (3) a unified method/stack cache.

4.1. Criticality Timeout Cache

In cases where strict predictability is unnecessary but flexibility and utilization efficiency are essential, we propose a cache using a partitioning approach based on cache line timeouts. For that cache, we need an n-way set associative cache configuration. We can configure the cache at the granularity of cache ways. Each cache way can be assigned either a criticality or a task/core ID (we will use criticality moving forward).

In this proposal, each cache way can be assigned either to high or low criticality. Cache lines can be used by high- or low-criticality tasks. However, naturally high-criticality tasks are *preferred*. A low-criticality task cannot evict a high-criticality cache line. Therefore, to avoid starvation of low-criticality tasks, at least one way must not be assigned for the high-criticality tasks.

When an access of the high criticality arrives, a cache line in one high-criticality way is tagged as being occupied by that criticality, and an associated timeout begins. As long as the timeout is not reached, accesses of low-criticality tasks cannot evict the cache line. If there is no access to the line before the timeout is reached, the line criticality is downgraded, allowing low-criticality jobs to evict the line. The cache can either be configured right before each job starts executing, or the criticalities can be configured ahead of time to match the tasks that will run on the cluster. With timeouts, there is no need to explicitly release any data, as the timeout mechanism will do so automatically. Configuring the cache is done by setting the criticality of a cache way. When a way is configured with a criticality, all its cache lines will prefer accesses from that criticality, as described above.

A significant drawback of this approach is its unpredictability. Because timeouts might cause a cache line to be evicted even when it might be used in the future, it can be

difficult for a WCET analysis tool to track which cache lines have reached the deadline and which have not. The effect of the timeouts on WCET bounds can be challenging to estimate and would require dedicated analysis. However, it can also be omitted, as this cache architecture is better suited for measurement-based WCET estimation. With detailed testing and measurements, getting a sufficiently safe WCET bound should be feasible.

This cache architecture is designed for high utilization and low scheduling complexity. Because it reserves each cache line, only the necessary subset of a cache way is reserved at a given time. Cache lines that either timed out or were not used by the job are free to be used by low-criticality tasks, increasing the utilization of the cache. In this proposal, we also do not pre-load data into the cache. This means only data that is used will be loaded. Therefore, we avoid both bandwidth wastage and cache space wastage when loading data that is not used. When a job stops executing, its associated cache lines will eventually time out and release their contents automatically. The scheduler, therefore, does not need to manage the phased execution of jobs, reducing the pressure on the scheduler.

4.2. Contention Tracking Cache

In this proposal, a combination of contention tracking in the cache and contention-aware task scheduling will allow for maximal cache utilization through dynamic partitioning, with high predictability through cache contention tracking and mitigation.

In a multicore system without shared caches, the execution time of a job is affected by the cache behavior without that behavior being affected by other jobs. Through cache analysis, we can bound the execution time attributable to the cache. This is done by estimating the number of cache misses that will occur. When the cache is shared, this analysis is no longer possible, as the interference of other jobs will cause additional cache misses in a manner that cannot be estimated. In this proposal, we want to let the task scheduler limit the contention that a job is allowed to experience such that it is guaranteed to meet its deadline.

We give two example types of contention: (1) A job J_1 experiences a contention event if a cache line C_1 it populated with data D_1 is evicted by an access by another job J_2. This is because J_1 will experience a cache miss on the next access to D_1 that it would not have experienced if J_2 had not interfered. (2) J_1 also experiences a contention event if a cache miss occurs when accessing D_1 results in the eviction of a cache line that J_1 also populated in the same cache set (with data D_2). This event is a contention with any other job with at least one populated cache line in the same set. Without the other jobs, J_1 would have populated an empty cache line instead of evicting one of its other populated lines. The evicted line will cause a cache miss in the future when J_1 needs to access D_2 again.

We only consider contention between different jobs. Self-contention also happens in private caches and is, therefore, already managed in the cache analysis for the private cache.

We limit the maximum allowed contention as defined above to ensure that a job meets its deadline without interference from other jobs. The scheduler will configure the cache with a maximum allowed contention. The cache controller will track contention by checking and counting the above contention events for each job. When a job reaches its contention limit, any cache access that would cause a contention event will be blocked or mitigated. For example, say J_1 is high criticality, and J_2 is not. As long as J_1 has not reached its contention limit, the cache treats accesses from both jobs equally. When the limit is reached, contention events are mitigated between J_1 and J_2. In the case of the first event type, accesses from J_2 that would cause an eviction of J_1's cache lines would be rejected by the cache. The access must then be rerouted directly to the main memory, which the system must have support for. In the second event type, if the default replacement policy would have J_1 evict its own cache line in the set, it would instead evict a cache line from J_2.

Setting the contention limit is the responsibility of the job scheduler. Through traditional static WCET analysis with the assumption of private caches, jobs get their WCET bound. Any excess time between the bound and the task deadline is therefore open to contention. Before the scheduler starts a job, it sets the contention limit, ensuring the WCET of the job, with contention, still meets the deadline. The contention limit can be static, and it can be calculated as part of schedulability analysis. It can also be dynamic, so the scheduler changes it for the runtime condition. If the task was started early, the contention is increased to match the slack time available. If the task was started late, the contention is reduced or set to zero to ensure that the deadline is still met.

This proposal's major strength is that it disconnects the analysis of tasks with differing criticalities. Because of the contention limit, high-criticality tasks will never be adversely affected by low-criticality tasks. Therefore, we just need to ensure that all high-criticality tasks meet their deadlines with other methods.[1] It also does not statically partition or lock the cache. At worst, when a contention limit is reached, the cache will be dynamically partitioned automatically simply by prioritizing the jobs that have reached the limit. This maximizes cache utilization. It also allows for maximizing the performance of low-criticality tasks as long as it does not adversely affect any high-criticality tasks.

This proposal does increase the complexity of the cache controller, which needs to track contention events and mitigate them for jobs that have reached their contention limit. Each cache line needs to be associated with a job (or core), each job needs a contention counter, and logic needs to ensure the correct mitigation at contention limits. The proposal also increases scheduler complexity. This complexity can be initially lowered by simply having statically determined contention limits. However, further work should explore dynamically determined limits, which would increase the workload on the scheduler.

4.3. Unified Method/Stack Cache

The Patmos processor on T-CREST uses the special method and stack caches. While these caches have been researched for their impact on predictability, and the Platin analyzer has analysis implementations for them, additional work is needed to integrate them into a shared L2 cache. Therefore, we propose investigating a shared L2 cache that integrates the features of both the method cache and the stack cache. It is meant to complement either a traditional L2 data cache or scratchpad, with extended research avenues for a fully integrated L2 cache that supports the method-, stack-, and

[1] For example, we could use partitioning between high-criticality tasks only.

	Priority Timeout	Contention Tracking	Unified Method/ Stack
All Data	✓	✓	✗
Shared	✓	✓	✗
Mixed-Criticality	✓	✓	✗
Analyzable	✗	✓*	✓
Needs Scheduling	✗*	✓*	✗
Guaranteed	✗	✓*	✓

Table 1
Comparison between features of the three cache proposals.

data caches. This proposal can also complement either of the previous proposals.

The method and stack caches have particular access patterns to their data. The method cache accesses a block of code at a time, pre-loading a complete block at once. It also uses a first-in, first-out (FIFO) replacement policy to account for functions earlier in the call stack being less likely to be called again soon. On the other hand, the stack cache is not backed by main memory unless some data is spilled when the cache is full. This allows the L2 cache to store the spilled stack data first without sending it to the main memory. Access to this stored data would have the same characteristics as access to the stack cache. Additionally, when space is tight in the L2 cache, the replacement policy is the same as the stack cache: spill the data furthest up the stack.

An open question is how to partition the cache between the method and stack data. Since both have a replacement policy that depends on reaching the space limit, a policy is needed for deciding how much of the cache should be meant for the methods, and how much should be used for the stack. We should also investigate if this division can be dynamically configured such that if the stack is not expected to use much space, then most of the L2 cache should be saved for the methods and vice versa. A different approach could be to say that the stack gets priority up to a point. When the stack needs to store more data, methods are evicted to make room up to a point (e.g., half the L2 cache size). Any space not used by the stack cache can store methods. This can also be done in reverse, where the method data gets priority.

An open question that would need answering following the above initial research, would be how to implement a unified method/stack cache that is also shared between cores. Since each core has a distinct stack, and is also likely to use different functions, we need to explore ways for a single cache to effectively manage multiple stacks and call trees.

4.4. Discussion

The three caching proposals—Criticality Timeout Cache, Contention Tracking Cache, and Unified Method/Stack Cache—each address the challenge of predictable caching in different ways. Table 1 compared the various features of our proposals. The first big difference is between the Unified Method/Stack Cache and the two other caches. The Priority Timeout and Contention tracking caches both support all program data, whereas the Unified Method/Stack Cache only supports instruction data (methods) and stack data. Even more specifically, the traditional stack does not support all stack data, only that which does not need an address, as the stack cache is not backed by main memory. Any data whose address is taken in the program cannot be put in the stack cache, going instead to the *shadow stack*, which is backed by main memory. Another big difference between the Unified Method/Stack Cache and the others is that the proposal does not share the cache between multiple cores, which also means it does not alleviate any challenges for mixed-criticality systems.

Analyzability is different between all the cache proposals. The Priority Timeout cache does not support analyzability very well, as it is difficult for analyzers to track when cache lines have timed out. The contention cache is analyzable, but only in the sense that it simplifies mixed-criticality analysis by disallowing interference between tasks of different criticalities. For tasks with the same criticality, the cache does not provide any assistance but does not complicate the analysis. The Unified Method/Stack Cache is the most analyzable. Analyzers can reuse the analysis done for the separate method and stack caches and likely reuse it for the unified one with different configurations and minor customization.

The proposals also differ in how much support is needed from the job scheduler at runtime. The Priority Timeout Cache can be implemented without scheduler support if the way-based partitioning is configured ahead of time. If the partitioning is done dynamically, it would be the scheduler's responsibility. The Contention Tracking Cache needs support from the scheduler to ensure the amount of allowed contention is within the correct limit. The scheduler needs to account for when a high-criticality job is started so that an appropriate contention limit is chosen. A static approach can also be used where the contention limit is chosen ahead of time. However, that does not provide much benefit compared to traditional partitioning. The Unified Method/stack Cache needs no scheduling support at all. The only thing that might be configurable would be how much of the cache is prioritized for methods or the stack. However, this could better be done by the program itself, e.g., through compiler management of the cache.

Lastly, each cache has different guarantees on its behavior. The Priority Timeout Cache provides priority guarantees for only a specific time. If that is not managed such that it does not run out, programs cannot be guaranteed that a specific amount of the cache is reserved for them. While giving no guarantees on partitioning, the Contention Tracking cache guarantees how much contention could affect a job. However, this is only contention from lower criticality job contention, and so does not make any guarantees about the contention from similar-criticality tasks. However, the Unified Method/Stack Cache is predictable and guarantees similar behavior to the split caches.

5. Related Work

Shared caches are a significant challenge for predictability due to their inherent nature of allowing multiple cores to access the same cache [26]. This can lead to contention and unpredictable performance. However, several solutions have been proposed to address this issue, including cache partitioning and locking [27].

Partitioning is a technique that divides the shared cache into several partitions, each dedicated to a specific core [28]. This approach can significantly improve predictability by reducing contention [29]. Way-based partitioning involves dividing the cache ways among different cores. Each core is

assigned a specific number of ways in the cache, ensuring exclusive access to those ways. This method can effectively isolate the cache activities of different cores, improving predictability. On the other hand, index-based partitioning involves dividing the cache sets among different cores. Each core is assigned specific sets in the cache, ensuring exclusive access. This method is more flexible than way-based partitioning because the number of sets is usually large, allowing for finer-grained partitioning. However, a given set maps to specific address ranges. Therefore, this method requires more detailed memory management. Page coloring is often used to partition the cache [30]. The address space is divided into colors associated with the cache sets. Assigning colors to tasks/cores provides the partitioning, assuming an assignment that provides the correct memory for each task/core is found. The cache hardware can also support index-based partitioning for various benefits [31, 32]. However, some form of software management will always be needed.

Cache locking is another technique used to improve predictability in shared caches [33]. With locking, specific cache lines can be locked to prevent them from being evicted, ensuring they are always available for the necessary cores. This can significantly reduce cache misses and improve predictability. Locking can be costly. Lock management involves tracking the locked cache lines, increasing hardware complexity. Adding locking to a cache can reduce its capacity or speed depending on how fine-grained the locking is. Locking also reduces cache utilization, as any unused locked content cannot be evicted to free up the cache lines for needed data.

T-CREST has enabled much research within various aspects of real-time systems [5]. Because all of T-CREST's components are predictable, it is possible to implement constant execution-time code based on the single-path paradigm [34, 18]. Single-path has an inherently high overhead, necessitating optimizations to reduce executed code [35], make best use of Patmos' dual-issue pipeline [36, 17], and use custom register allocation techniques [37]. The combination of T-CREST and single-path code has been shown to be competitive with off-the-shelf ARM processors for a real-time application [38]. Research is also ongoing to port the Lingua Franca coordination language to T-CREST to enable the creation of complete real-time systems within one framework [39, 40].

6. Conclusion

The increasing importance of 5G technologies necessitates continuous research and development into the hardware systems implementing the technology. The diverse requirement specifications of this new technology necessitate a system with varying degrees of strictness and performance. Existing systems were designed with the minimal 5G guarantees in mind, ensuring the hard requirements, e.g., low latency, were met before softer requirements like throughput. This focus resulted in a divided physical system to achieve the goals.

To increase future systems' performance while maintaining the older system's guarantees, this paper sets the research direction into a mixed-criticality 5G RBS with merged BBU and layer 2 systems. The system should be able to execute high-criticality tasks, like those required by the URLLC 5G scenario, and low-criticality, QoS tasks, like those for the eMMB, in one SoC. By analyzing the 5G requirement specifications and the common system architecture, we propose using the T-CREST platform as the research platform for future mixed-criticality systems. We propose a specific system architecture that best leverages the existing system architecture's strength and increases its performance through shared caches. We propose three specific research directions within shared L2 caches for clustered systems. The various proposals have distinct strengths and weaknesses that will be further explored in future work.

Acknowledgment

This work is partially supported by the CERCIRAS (Connecting Education and Research Communities for an Innovative Resource Aware Society) COST Action no. CA19135 funded by COST (European Cooperation in Science and Technology).

References

[1] International Telecommunication Union - Radiocommunication Sector, IMT Vision - Framework and overall objectives of the future development of IMT for 2020 and beyond, Technical Report M.2083-0, International Telecommunication Union, 2015.

[2] A. Burns, R. I. Davis, Mixed criticality systems-a review:(february 2022) (2022).

[3] ISO/IEC 7498-1:1994(E), Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, Technical Report 7498-1:1994, International Organization for Standardization, 1996.

[4] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for cots-based embedded systems, in: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, 2011, pp. 269–279.

[5] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, A. Tocchi, T-CREST: Time-predictable multi-core architecture for embedded systems, Journal of Systems Architecture 61 (2015) 449–471. doi:10.1016/j.sysarc.2015.04.002.

[6] International Telecommunication Union - Radiocommunication Sector, Minimum requirements related to technical performance for IMT-2020 radio interface(s), Technical Report M.2410-0, International Telecommunication Union, 2017.

[7] Z. Kong, J. Gong, C.-Z. Xu, K. Wang, J. Rao, ebase: A baseband unit cluster testbed to improve energy-efficiency for cloud radio access network, in: 2013 IEEE International Conference on Communications (ICC), IEEE, 2013, pp. 4222–4227.

[8] E. Tell, A. Nilsson, D. Liu, A programmable dsp core for baseband processing, in: The 3rd International IEEE-NEWCAS Conference, 2005., IEEE, 2005, pp. 403–406.

[9] J. Arora, C. Maia, S. A. Rashid, G. Nelissen, E. Tovar, Schedulability analysis for 3-phase tasks with partitioned fixed-priority scheduling, Journal of Systems Architecture 131 (2022) 102706.

[10] H. Kopetz, Real-Time Systems, Kluwer Academic, Boston, MA, USA, 1997.

[11] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, D. Prokesch, Patmos: A time-predictable microprocessor, Real-Time Systems 54(2) (2018) 389–423. doi:10.1007/s11241-018-9300-4.

[12] M. Schoeberl, F. Brandner, J. Sparsø, E. Kasapaki, A statically scheduled time-division-multiplexed network-on-chip for real-time systems, in: Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS), IEEE, Lyngby, Denmark, 2012, pp. 152–160. doi:10.1109/NOCS.2012.25.

[13] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, J. Sparsø, Argo: A real-time network-on-chip architecture with an efficient GALS implementation, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24 (2016) 479–492. doi:10.1109/TVLSI.2015.2405614.

[14] M. Schoeberl, Exploration of network interface architectures for a real-time network-on-chip, in: Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC), IEEE, United States, 2024. doi:10.1109/ISORC61049.2024.10551364, 2024 IEEE 27th International Symposium on Real-Time Distributed Computing, ISORC ; Conference date: 22-05-2024 Through 25-05-2024.

[15] M. Schoeberl, D. V. Chong, W. Puffitsch, J. Sparsø, A time-predictable memory network-on-chip, in: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), Madrid, Spain, 2014, pp. 53–62. doi:10.4230/OASIcs.WCET.2014.53.

[16] J. Yan, W. Zhang, A time-predictable VLIW processor and its compiler support, Real-Time Syst. 38 (2008) 67–84. doi:http://dx.doi.org/10.1007/s11241-007-9030-5.

[17] E. J. Maroun, M. Schoeberl, P. Puschner, Predictable and optimized single-path code for predicated processors, Journal of Systems Architecture (2024) 103214.

[18] E. J. Maroun, M. Schoeberl, P. Puschner, Compiler-directed constant execution time on flat memory systems, in: 2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC), 2023, pp. 64–75. doi:10.1109/ISORC58943.2023.00019.

[19] M. Schoeberl, A time predictable instruction cache for a Java processor, in: On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of *LNCS*, Springer, Agia Napa, Cyprus, 2004, pp. 371–382. doi:10.1007/b102133.

[20] P. Degasperi, S. Hepp, W. Puffitsch, M. Schoeberl, A method cache for Patmos, in: Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014), IEEE, Reno, Nevada, USA, 2014, pp. 100–108. doi:10.1109/ISORC.2014.47.

[21] B. Huber, S. Hepp, M. Schoeberl, Scope-based method cache analysis, in: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), Madrid, Spain, 2014, pp. 73–82. doi:10.4230/OASIcs.WCET.2014.73.

[22] S. Abbaspour, F. Brandner, M. Schoeberl, A time-predictable stack cache, in: Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems, 2013.

[23] A. Jordan, F. Brandner, M. Schoeberl, Static analysis of worst-case stack cache behavior, in: Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013), ACM, New York, NY, USA, 2013, pp. 55–64. doi:10.1145/2516821.2516828.

[24] E. J. Maroun, E. Dengler, C. Dietrich, S. Hepp, H. Herzog, B. Huber, J. Knoop, D. Wiltsche-Prokesch, P. Puschner, P. Raffeck, et al., The platin multi-target worst-case analysis tool, in: 22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[25] C. Pircher, A. Baranyai, C. Lehr, M. Schoeberl, Accelerator interface for patmos, in: 2021 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2021.

[26] B. C. Ward, J. L. Herman, C. J. Kenna, J. H. Anderson, Making shared caches more predictable on multicore platforms, in: 2013 25th Euromicro Conference on Real-Time Systems, IEEE, 2013, pp. 157–167.

[27] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, R. Pellizzoni, A survey on cache management mechanisms for real-time embedded systems, ACM Computing Surveys (CSUR) 48 (2015) 1–36.

[28] S. Mittal, A survey of techniques for cache partitioning in multicore processors, ACM Computing Surveys (CSUR) 50 (2017) 1–39.

[29] X. Vera, B. Lisper, J. Xue, Data caches in multitasking hard real-time systems, in: RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003, IEEE, 2003, pp. 154–165.

[30] T. Lugo, S. Lozano, J. Fernández, J. Carretero, A survey of techniques for reducing interference in real-time applications on multicore platforms, IEEE Access 10 (2022) 21853–21882.

[31] A. Chousein, R. N. Mahapatra, Fully associative cache partitioning with don't care bits for real-time applications, ACM SIGBED Review 2 (2005) 35–38.

[32] M. Lee, S. Kim, Time-sensitivity-aware shared cache architecture for multi-core embedded systems, The Journal of Supercomputing 75 (2019) 6746–6776.

[33] S. Mittal, A survey of techniques for cache locking, ACM Transactions on Design Automation of Electronic Systems (TODAES) 21 (2016) 1–24.

[34] P. Puschner, A. Burns, Writing temporally predictable code, in: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), IEEE Computer Society, Washington, DC, USA, 2002, pp. 85–94. doi:10.1109/WORDS.2002.1000040.

[35] E. J. Maroun, M. Schoeberl, P. Puschner, Constant-Loop Dominators for Single-Path Code Optimization, in: P. Wägemann (Ed.), 21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023), volume 114 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 7:1–7:13. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18436. doi:10.4230/OASIcs.WCET.2023.7.

[36] E. J. Maroun, M. Schoeberl, P. Puschner, Compiling for time-predictability with dual-issue single-path code, Journal of Systems Architecture 118 (2021) 1–11.

[37] E. Maroun, M. Schoeberl, P. Puschner, Two-step register allocation for implementing single-path code, in: Proceedings of the 2024 IEEE 27th International

Symposium on Real-Time Distributed Computing (ISORC), IEEE, United States, 2024. doi:10.1109/ISORC61049.2024.10551362, 2024 IEEE 27th International Symposium on Real-Time Distributed Computing, ISORC ; Conference date: 22-05-2024 Through 25-05-2024.

[38] M. Platzer, P. Puschner, A real-time application with fully predictable task timing, in: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), IEEE, 2020, pp. 43–46.

[39] E. Khodadad, L. Pezzarossa, M. Schoeberl, Towards lingua franca on the patmos processor, in: Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC), 2024.

[40] M. Schoeberl, E. Khodadad, S. Lin, E. J. Maroun, L. Pezzarossa, E. A. Lee, Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications, in: T. Carle (Ed.), 22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024), volume 121 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 4:1–4:13. doi:10.4230/OASIcs.WCET.2024.4.