SEMINAR REPORT
Seminar: Embedded Systems and Robotics in Summer term 2025

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Interference Channels and Their Mitigation in Modern Systems on Chip

Lev Sonnenfeld
Rheinland-Pfälzische Technische Universität Kaiserslautern, Department of Computer Science

Note: *This report is a compilation of publications related to some topic as a result of a student seminar. It does not claim to introduce original work and all sources should be properly cited.*

The increasing integration of multi-core processors in modern systems on chip (SoCs) has enabled significant performance gains while meeting stringent size, weight, power, and cost (SWaP-C) constraints. This is accompanied by a trend of consolidating systems into fewer multi-task components that were originally implemented as many distributed, specialised components. However, these trends also introduce timing interference between tasks that are executing concurrently. This is particularly problematic in real-time and safety-critical embedded systems such as avionics, automotive safety systems, and autonomous systems.

Such interference occurs when multiple processing units compete for shared hardware resources, including caches, main memory, and system interconnects. This report surveys the main interference channels in multi-core SoCs and analyses their impact on predictability and performance. For each channel, it discusses in detail a representative software-based mitigation strategy, such as cache partitioning through dynamic page colouring, memory channel partitioning, and memory bandwidth regulation.

1 Introduction

An increase in computational demand, coupled with the need to optimise for constraints of size, weight, power and cost (SWaP-C), has prompted a change in the design philosophy in embedded systems.

Traditionally, single-core systems on chip (SoCs) were used in such systems. However, since it is not possible to improve the performance of single-core CPUs efficiently by increasing the clock frequency, additional CPU cores have been introduced instead. This enables performance to be increased by harnessing the performance gains achieved through parallelism. Additionally, rather than implementing individual functionalities on different physically distributed components, systems are now being adapted to implement multiple functionalities on unified hardware components.

However, while these changes address their respective requirements, they also necessitate multiple tasks accessing the same hardware resource. Since processing units run in parallel and independently, they may need to access the same resource simultaneously. Additionally, even if they are not accessed simultaneously, one task's operation might change the state of the hardware. This could necessitate reverting the hardware for the other task once it requires the resource again.

In both cases, some execution units must wait for the resource to become available again. This means that processes that were previously independent in their execution can now affect each other, leading to unpredictable increases in execution times. This undesired phenomenon is called interference.

While interference can reduce the overall performance, in systems that additionally have time constraints for response times, it can lead to additional complications. These systems are known as real-time systems and are common in embedded and reactive environments. Typical examples include avionics, autonomous driving and automotive safety systems [6, p. 2]. These systems are usually safety-critical and must therefore be highly reliable and fail-safe. To ensure this, those systems are extensively verified, tested and certified [9]. This requires high predictability of each subsystem, since testing and verifying the whole system is practically impossible.

This predictability requirement is directly affected by interference. Here, since all systems cannot be verified at once, a maximally pessimistic execution time must be assumed in each component. This means the hardware cannot be utilised to its full potential. For example, in some applications, only one core of a multi-core SoC is used to avoid interference altogether [8]. While deactivating all but one core has removed the interference, the hardware potential is still wasted. Applying this solution, in turn, also increases the number of components, working against the SWaP-C constraints. Therefore, interference and its impact on system performance must be addressed differently.

In this report, typical interference channels causing timing interference in multi-core SoCs are derived from the structure of typical SoCs and analysed. Then, techniques for minimising the impact of interference on these channels are investigated and explained.

The following report is structured as follows. First, Section 2 introduces the background knowledge necessary to understand the following sections. Specifically, it introduces real-time systems as well as the architecture of multi-core SoCs. Using this background, Section 3 derives common interference channels for the described real-time systems. Then, in Section 4, mitigation methods for the different interference channels are presented and analysed. Following this, a summary of related work and aspects not covered by this report is given in Section 5. Finally, the report is then concluded by a summary of the key points that were discussed in the report in Section 6.

2 Background

This section provides an overview of the fundamentals required for the analysis of interference channels. The first part provides insight into real-time systems. Then Subsection 2.2 presents the typical components of a multi-core SoC and how they interact.

2.1 Real-Time Systems

Real-time systems are time-critical systems that have to respond within a certain deadline when receiving a signal. Generally, these systems can be categorised as either hard or soft real-time [3]. Missing a deadline in a hard real-time system often leads to system failure, whereas in a soft real-time system, the system only degrades with time and is usually able to recover. This

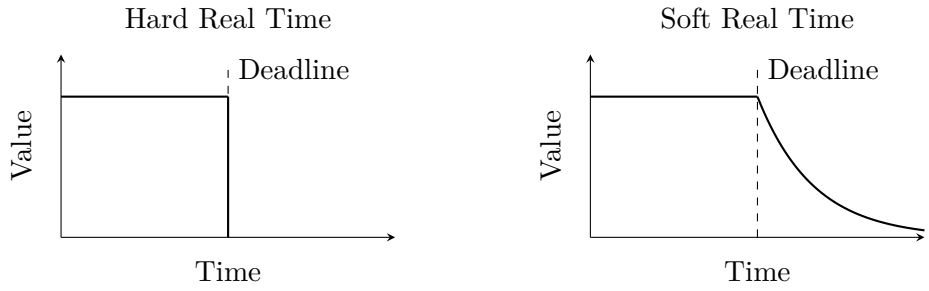


Figure 1: A comparison between soft and hard real-time systems. In hard real-time systems, the value of the response is zero or negative after the deadline. By contrast, soft real-time systems (right) still produce a response after the deadline, albeit with reduced value.

is illustrated in Figure 1.

In the hard real-time case, after missing the deadline, the response value is zero or less, whereas in the soft real-time case, the response still has some value. The typical application domains, therefore, differ between the two categories: hard real-time systems are found in safety-critical applications such as automotive braking systems and avionics. Soft real-time systems, on the other hand, are found in multimedia and telecommunications systems, where missing deadlines is occasionally tolerable. Unless stated otherwise, a real-time system will always be assumed to be hard real-time in the following.

To ensure that a real-time system can meet all its deadlines, a schedulability analysis must be conducted. Although this report does not focus on schedulability analysis, a brief overview is provided in Section 5.2. Such an analysis requires knowledge of either the worst-case execution time (WCET) or the worst-case response time (WCRT) [5, 10]. While these two metrics are closely related, the WCET describes the time required to execute the task. Conversely, WCRT is the time required for the system to respond to a request. Both metrics assume worst-case scheduling conditions and represent an upper bound. Once again, it is clear that both metrics can be affected by hardware as well as interference.

Finally, since missing deadlines is considered catastrophic, predictability is valued more highly than peak performance in real-time systems. Modern hardware features such as caches, pipelines and shared memory interconnects complicate WCET and WCRT analysis because they make execution times highly variable. For this reason, the design of real-time systems typically balances performance improvements against the need for analyzable, dependable behaviour.

2.2 Multi-Core Systems on Chip

To analyse and identify possible interference channels in multi-core SoCs, it is essential to understand the system components and how they interact. A high-level system overview of a typical SoC is given in Figure 2. Here, all components are interconnected by a central bus. Each processing core is connected to the interconnect through its private cache hierarchy. The memory is also connected through a last-level cache (LLC) to the system. Peripheral components, such as timers, general-purpose I/O (GPIO) controllers, direct memory access (DMA) engines, and

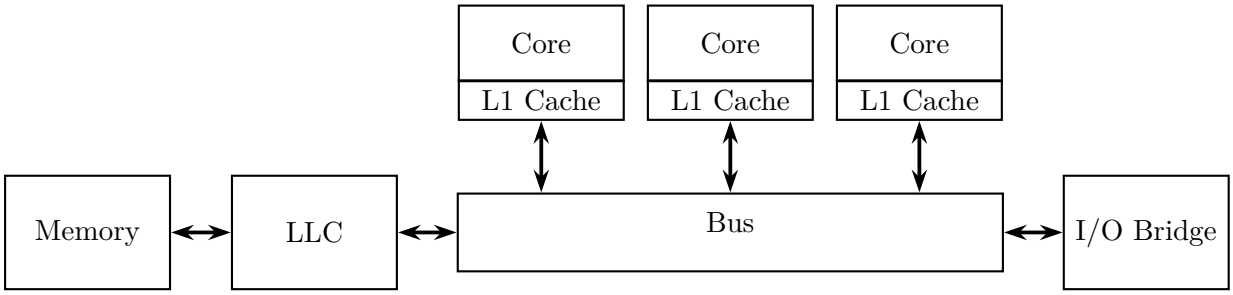


Figure 2: Overview of a typical Multi-Core SoC architecture. All cores are connected via a central bus. This bus is then connected to memory and external components. This overview is derived from the overview given in [3].

external interfaces (e.g. PCIe, Ethernet, UART), are connected via an I/O bridge to the same internal bus. While instruction and data caches are often physically separated in modern designs, this overview treats them generically as data caches for simplicity. In the following, each component is described in detail.

2.2.1 Processing Cores

Processing cores are the primary computational units of the SoC. In a multi-core SoC, several identical or heterogeneous cores are integrated in the design, enabling parallel execution of multiple software threads or processes. In the following, they are handled like homogeneous cores for simplicity. Each core typically consists of an instruction fetch unit, decode logic, execution units, and register files, often supported by hardware features such as branch predictors and vector processing extensions. Additionally, Cores are typically equipped with Hardware Performance Counters (HPCs) used for measuring different performance characteristics [11,12]. Cores typically execute instructions in a pipeline and access data through their local cache hierarchy.

2.2.2 Caches

Caches are small but fast memory units that are positioned between the cores and the main memory. They are used to overcome the performance gap between processor speeds and memory latency. This is achieved by saving recently used data blocks in the cache, assuming they will be reused. A cache always loads and unloads whole cache lines containing multiple addresses, not just the single requested address. If a requested memory address is currently stored in a line in the cache, a cache hit is triggered, and the result is returned. Otherwise, a cache miss is triggered, which requires loading the data from a larger memory.

Since caches are limited in size, an existing cache line must be replaced when a new one is loaded. First, a placement policy decides where the cache line can be placed. There are three policies: direct mapping, which maps a fixed set of addresses to a specific cache line; fully associative mapping, which allows any memory block to be stored in any cache line; and set associative mapping, which is a compromise between the other two. In this case, a fixed set of

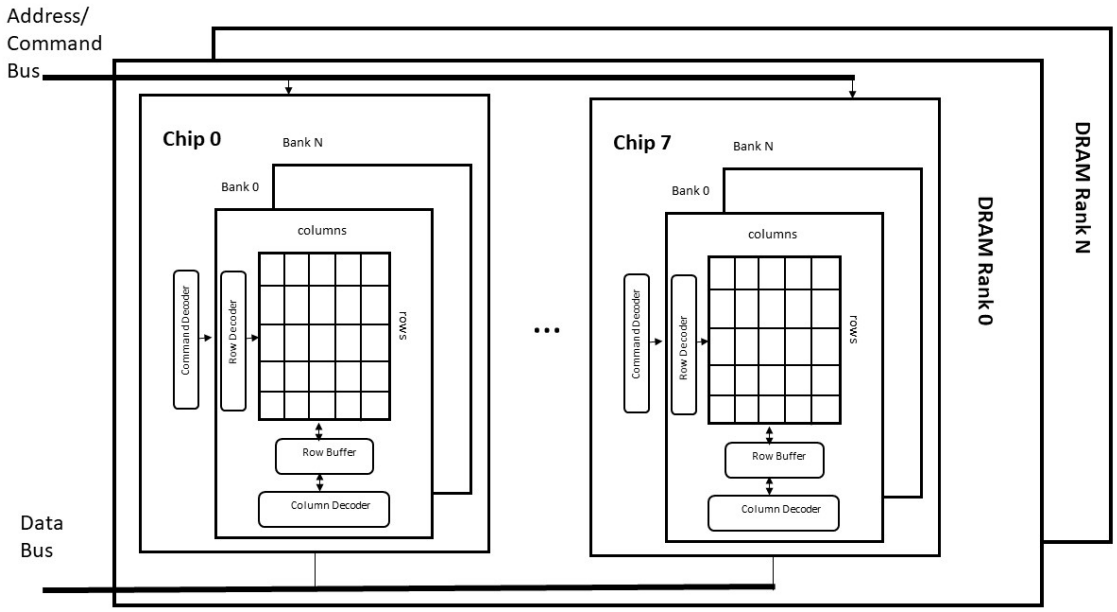


Figure 3: DRAM Device Organisation Overview from [3].

addresses is mapped to a set of cache lines, which can then be placed in any of those lines. Most modern systems use set associative mapping [11].

When the placement policy allows the memory block to be placed in multiple cache lines, the replacement policy decides which cache line is replaced. Simple replacement policies include Least Recently Used (LRU), which replaces the cache line that has not been used for the longest time, and Pseudo-LRU (PLRU), which uses a simple heuristic to approximate the behaviour of LRU. Other, more sophisticated heuristic-based policies may also be used [6, p. 120].

Caches are usually organised into cache hierarchies. The L1 caches, which are located closest to the cores, offer the lowest access latency but are the smallest in size. They are usually each dedicated to one core. The LLC, on the other hand, is shared by all cores and acts as the main point of coherence before accessing main memory. Depending on the design, some architectures employ additional intermediate caches, such as L2 caches.

Modern SoCs implement hardware cache coherence protocols (e.g., MESI, MOESI) to ensure consistency of data across private caches. These protocols rely on the interconnect to propagate coherence messages. To keep coherency after a write of a core, two different write policies can be employed: write-through and write-back. Write-through updates cache and memory simultaneously, whereas write-back updates cache and delays updates to other memory until later.

2.2.3 Memory

A variety of memory types can be used in SoCs, including programmable read-only memory (PROM), dynamic random access memory (DRAM), and flash memory. In most modern SoCs, DRAM is used for memory [9]. Therefore, in the following, DRAM is assumed to be used as the main memory. As DRAM is usually realised outside of the SoC, it is accessed by a memory controller.

The typical structure of a DRAM is visualised in Figure 3. The memory is subdivided into channels, which contain one or multiple ranks. A rank consists of multiple banks, which consist of a matrix of rows and columns. Each bank is accessed through its row buffer, which contains the most recently accessed row of that bank. If the requested address is stored inside the buffer, a row hit is triggered, and the stored data is returned. Otherwise, a row miss is triggered, and the requested row must first be loaded from the bank. At the same time, as writes are only performed on the row buffer, the buffered updates must be applied to the bank. Similar to caches, a memory always accesses a whole row, not just a single memory address [11].

In order to optimise throughput and fairness, the memory controller arbitrates access requests by applying a scheduling policy. The most common policy is First Ready First-Come First-Serve (FR-FCFS) [3, 4, 11]. This policy prioritises requests for open rows since those can be accessed the fastest.

2.2.4 System Bus

The system bus or interconnect is a central structure that is the communication backbone of the SoC. Here, a different architecture is used depending on the SoC. In earlier designs, predominantly single shared buses were used, but modern SoCs also frequently employ hierarchical buses, crossbars, or network-on-chip (NoC) fabrics to increase concurrency and scalability.

Modern bus designs typically include multiple channels, which can be accessed concurrently. The interconnect transports both data and control messages, including memory access requests, cache coherence transactions, and I/O communications. It includes arbitration logic to decide which accessor (e.g., a core, DMA engine, or peripheral) gains access to shared resources at any given time. The characteristics of this arbitration directly influence the timing behaviour of the system.

2.2.5 External Devices

The SoC is connected to the external environment and external devices through the I/O bridge. Such devices are, on the one hand, peripheral interfaces such as USB, Ethernet, PCI Express, and display controllers. On the other hand, the devices are interfaces like timers, network buses, sensors or actuators.

Generally, the devices are accessed as if they were memory addresses. This principle is called memory-mapped I/O. There are two ways that are used to communicate with the external devices. This is done either in programmed I/O mode, where a core explicitly reads or writes data to a device, or in direct memory access (DMA) mode, where data transfer occurs directly between the device and main memory without core intervention. In the latter case, the DMA controller acts like a core regarding memory access [9].

2.2.6 Virtual Memory

Modern systems typically hide the physical memory architecture from software and programmers. This is achieved by introducing virtual memory, which software then accesses via virtual addresses rather than raw physical memory. The virtual addresses are defined in a continuous memory space and mapped to the physical addresses by the operating system (OS) [4]. It uses specialised hardware, the so-called memory management unit (MMU), to translate virtual addresses into physical addresses. Every core is equipped with such an MMU, which is situated between a core and its private caches [9].

Instead of translating every possible single virtual address to a physical address, the translation is typically organised in blocks. Those blocks partition the virtual address space and are called pages. Page tables contain the mapping from the virtual memory address of a process to the physical page address and are used for lookup. When a page is not loaded in the memory, a page fault is fired during lookup, requiring the intervention of the OS. The amount of intervention depends on the specific hardware architecture.

In order to speed up the address translation, the MMU contains translation lookaside buffers (TLBs) that act like caches for recent translations from virtual to physical addresses. Similar to caches, TLBs can also be arranged in a hierarchical structure. A TLB miss triggers a page table walk, looking for a translation in the page table [9].

2.2.7 Other Architectures

Having examined the most common system on a chip (SoC) architecture, this section briefly summarises how other SoCs might be structured. While some of the aspects covered in later sections may be relevant for architectures presented here, this will not necessarily be the case for all of them.

The SoC discussed above employed a uniform memory access (UMA) architecture, where all cores share equal access latency to the main memory. In non-uniform memory access (NUMA) architectures, each core or group of cores instead has its local memory. This local memory yields lower latency for local cores [2]. Remote cores have a larger latency to that memory instead.

Some architectures employ scratchpad memories instead of traditional caches [3]. Here, instead of having a classical cache hierarchy, scratchpads provide fast memory that can be accessed by the cores. The writes and reads to that memory have to be explicitly specified by the programmer, requiring more manual programming work.

3 Interference Channels

Having examined the typical structure of a SoC in Section 2.2, this section focuses on deriving common interference channels from this structure and examining their potential impact on the performance and predictability of multi-core SoCs.

For timing interference to occur, two or more accessors (e.g. cores or DMA controllers) must attempt to access a shared resource simultaneously, but the resource cannot handle such concurrent access. Alternatively, the resource may have an internal state that is altered by one accessor's access, requiring the state to be refreshed before the other accessor can be handled.

Without complete knowledge of all concurrent accessors, these scenarios lead to unpredictable timing delays.

In general, three main types of interference channels can be identified in SoCs: shared cache, main memory, and system bus [2,3]. The causes and effects of the different types of interference channels are explained below.

3.1 Shared Cache

Caches introduce multiple sources of interference. Firstly, when running multiple tasks, even on one core, when a suspended task resumes, the contents of the caches have been altered since it was suspended. Thus, resuming the task leads to many more cache misses in multiple levels of the cache hierarchy than usual until the task has restored the cache state. This type of interference is called inter-task interference [3]. In a system with multiple accessors, this interference can even cause more interference when the system bus or the main memory is accessed to load the data for the task.

On shared caches, like the LLC, interference occurs when one accessor's cache operations evict or overwrite cache lines that another accessor is actively using. This phenomenon results in increased cache miss rates for the affected accessor, forcing it to fetch data from other levels of the cache hierarchy or the memory. Here, the cache placement policy and the cache replacement policy, if applicable, can strongly affect the severity of this interference. Additionally, this effect might even be amplified by the cache policies when both tasks are constrained to the same set of cache lines, not utilising the full available space, causing more frequent cache line evictions. This type of interference is called inter-core interference [3, 11].

A third source of interference is induced by cache coherence. On the one hand, cache coherence generates extra traffic on the system bus to synchronise cache states. On the other hand, writes by other tasks can invalidate cache lines in some private caches, requiring the affected lines to be refreshed. Here, write-back is not an option for real-time systems. This is because the WCET cannot be analysed due to decoupled memory writes [3].

When analysing the sources of interference for caches, it stands out that several of those sources can combine and trigger each other. This is especially significant if the system has multiple accessors that are running multiple tasks in parallel. Then the cache misses can cascade and add even more misses in the cache hierarchy, adding multiple delays to the cache requests. Additionally, the other interference channels are also affected by interference caused in caches. This is because, as cache misses cascade through the hierarchy, they induce system bus access and main memory requests that must be handled.

3.2 Main Memory

Since the main memory is shared between all cores and the DMA controllers in the SoC and accessed by a single memory controller, it poses as a bottleneck and a source of interference in the system. When multiple requests are issued at once, the memory controller has to serialise them. The order of that serialisation is then decided by the employed scheduling policy. Since the scheduling policy reorders the requests issued to the memory, the response time can now vary based on the prioritised requests. This type of interference is called inter-bank interference [3].

Another type of interference is caused by access to memory banks through a row buffer. If several tasks read from the same bank, but require access to different rows, two possible sources of time delay are introduced. The first delay is caused by the tasks unloading each other's currently used rows, leading to a longer access time for each memory operation. The second delay is introduced by requests being delayed for requests of a higher priority. This type of interference is called intra-bank interference [3].

3.3 System Bus

Since all memory-related traffic is routed through the system bus, there is some potential for interference due to bus contention. When multiple accessors issue transactions simultaneously, the interconnect's arbitration logic determines which transaction proceeds first. Here, arbitration schemes such as Fixed Priority, Round Robin, or First Come First Served (FCFS) based policies are used. The arbitration scheme can heavily influence the timing delay and its predictability, as well as the usable bandwidth. While a simple bus only allows for one concurrent communication, even more sophisticated interconnects have a limit to the number of possible connections. Additionally, bus transactions can differ in size and type (read or write), and the induced delay can vary widely [2, 3].

Due to its central placement in the SoC, the system bus can heavily influence the system performance and timing predictability. Additionally, since it is involved in the fetching of data after cache misses and accessing the main memory, it can directly influence the interference delays induced by the other two sources of interference. This makes bus-level contention a critical factor in the overall timing behaviour of shared memory systems.

4 Interference Mitigation

Now that common interference channels of SoCs have been investigated, they can be analysed for their mitigation techniques. To achieve this, a coarse overview of techniques for each interference channel is given. Then, one of those approaches to mitigate the interference channel is discussed in detail for each interference channel. All methods below are software-based and do not require hardware features that are not present in most modern SoCs.

4.1 Shared Cache

The approaches for cache interference mitigation can be divided into the following categories: cache partitioning techniques, cache locking techniques and predictable cache coherence protocols [3].

Cache partitioning focuses on subdividing the cache into a partition per task or core. The accessor then only uses their assigned part of the cache, avoiding interference. Those techniques can either be static or dynamic, trading off predictability against cache utilisation. Cache locking techniques, on the other hand, can block sections of the cache from eviction. This increases access predictability, but requires careful evaluation of which parts should be locked. The final technique focuses on designing more predictable cache coherence protocols.

In the following, the cache partitioning technique presented by [11] is discussed in detail.

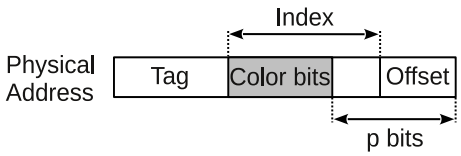


Figure 4: Address With Page Colour Bits [11].

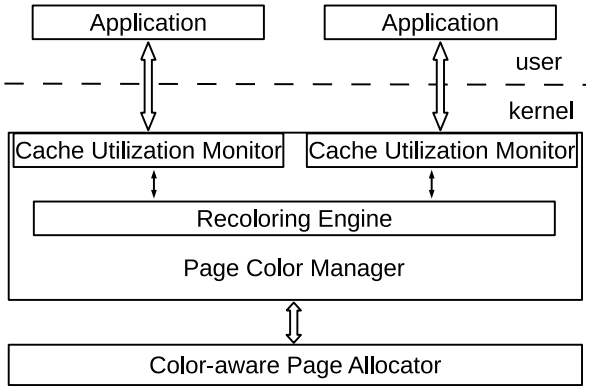


Figure 5: COLORIS Architecture Overview [11].

4.1.1 Dynamic Page Colouring

COLORIS, the cache interference mitigation method presented in [11], is structured into two components: a Page Colour Manager and a Colour-aware Page Allocator. It aims to mitigate LLC interference by partitioning the cache and assigning partitions to processes using page colouring. An overview of the structure of the method is given in Figure 5.

A colour is defined by the index bits of the physical address, which are not used by the OS as the page byte offset, as shown in Figure 4. Pages that have the same colour are mapped to the same set in a set-associative cache. This phenomenon is used by COLORIS and other page colouring techniques to control which cache lines are used by which tasks or cores by assigning them pages of specific colour.

The colour-aware page allocator is responsible for allocating pages of specific colours. To achieve this, the allocator keeps a set of free pages in a memory pool. Here, pages with the same colour are linked together. When receiving a request from a process, the allocator determines the colours assigned to the process by polling the page colour manager. A page of one of those colours, which is selected by round-robin, if the resulting colour is available, is then returned. If no pages with any of those colours are available, the allocator requests new pages of different colours from the OS. Freed pages are returned to the page pool unless the pool is already full.

The page colour manager controls which colours are assigned to which processes. This allocation can be static or dynamic.

In the static allocation case, the available cache space is divided into N sections in a system with N cores, assigning each to a core. With C colours available, each section and its corresponding processes are then assigned $\frac{C}{N}$ colours. Now, a process running on a core can utilise

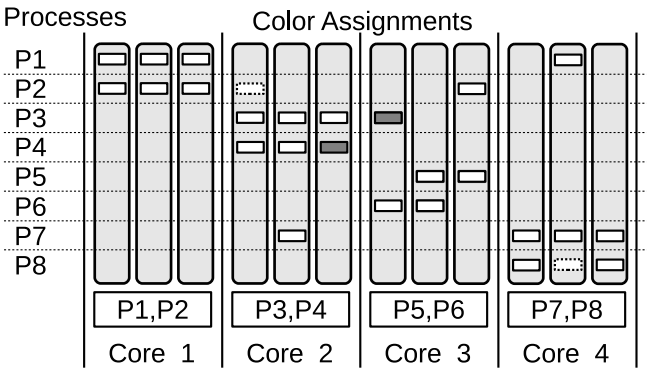


Figure 6: Example of the COLORIS Page Colouring Scheme [11].

the whole cache space assigned to that core. This partitioning is visualised in Figure 6.

Since this scheme can still lead to sub-optimal cache utilisation, COLORIS further implements a dynamic recolouring scheme. The partitioning is initialised with the static allocation scheme. The page colour manager monitors the cache miss rate of each process using HPCs. The algorithm used for monitoring is shown in Algorithm 1.

To achieve this, the function `monitor` is executed for each process in a set interval. The cache miss rate (*cmr*) of the process (*current*) is given as an argument.

It is then compared to the two configurable thresholds $HighThreshold$ and $LowThreshold$. If the miss rate of the process exceeds the $HighThreshold$, the manager initiates the allocation of new colours for the process by calling the function `alloc_colors`. Additionally, if the process is not marked as cold, it is marked as hot; otherwise, the mark as cold is removed. If the miss rate falls below $LowThreshold$ and the task is marked as hot, some of the assigned colours can be reclaimed and redistributed. This is done by calling the function `pick_victims`. $UNIT$ is a parameter describing the number of colours to be allocated or removed in a recolouring step.

To decide which pages to recolour, COLORIS uses the concept of 'hotness', which is defined as the number of processes currently sharing a colour. New pages are allocated to cold colours, while hot colours are selected for reclaiming. An additional factor that is considered is whether a colour is local or remote. It is considered local when it was originally assigned to the core running the process; otherwise, it is called remote. When allocating new colours, local colours are considered first. Only if no local colour is available, remote colours are allocated. Conversely, when assigned colours are removed, remote colours are removed first.

This prioritisation and the recolouring step are also visualised in Figure 6. White blocks are colours owned by the respective process before and after recolouring. Dashed white blocks are colours lost after recolouring, while black blocks are colours gained by recolouring by the respective process.

The recolouring step itself is performed by the recolouring engine. For removing colours, a lazy recolouring scheme is used. Here, all pages that have those colours are marked for later recolouring. When a marked page is accessed by a process, a page fault occurs. A new page is then allocated in one of the remaining colours, and the data of the marked page is copied to it.

For adding new cache colours, two recolouring policies are implemented: selective moving and

Algorithm 1: COLORIS Cache Utilisation Monitor, taken from [11]

```
1  function monitor(cmr) begin
2      assignment ← assignment_of(current);
3      if cmr > HighThreshold then
4          if isCold = false then
5              isHot ← true;
6              return;
7          end
8          new ← alloc_colors(UNIT);
9          assignment += new;
10         isCold ← false;
11     else if cmr < LowThreshold then
12         if isHot = true then
13             isHot ← false;
14             isCold ← true;
15             victims ← pick_victims(UNIT);
16             assignment −= victims;
17         end
18     end
19 end
20 function alloc_colors(num) begin
21     new ← ∅;
22     while num > 0 do
23         if need_remote() then
24             new += pick_coldest_remote();
25         else
26             new += pick_coldest_local();
27         end
28         num ← num − 1;
29     end
30     return new;
31 end
32 function pick_victims(num) begin
33     victims ← ∅;
34     while num > 0 do
35         if has_remote() then
36             victims += pick_hottest_remote();
37         else
38             victims += pick_hottest_local();
39         end
40         num ← num − 1;
41     end
42     return victims;
43 end
```

redistribution. In selective moving, the characteristic of n-way set-associativity of the LLC is utilised. The entire page table of the current process is traversed, and every $n + 1$ pages of one colour, one page is recoloured, using the round robin scheme of the colour-aware page allocator. When using the redistribution approach, COLORIS makes use of an access bit that is present per page table entry. All access bits are first reset. Then, after waiting for a fixed time (WIN), all pages of which the access bit was set in that time frame are then recoloured. The pages found by this process are then recoloured by using the same lazy recolouring scheme that was used for removing colours. In experiments conducted by [11], redistribution, despite requiring a higher overhead, resulted in a better overall system performance.

4.2 Main Memory

In general, main memory interference mitigation techniques can be distinguished: spatial isolation, temporal isolation, improved DRAM controller protocols and memory channel partitioning [3].

Spatial isolation techniques are characterised by assigning memory banks exclusively to a task or core. Temporal isolation techniques, on the other hand, focus on partitioning the memory access in time instead of space. This is achieved by allowing memory access only at specific times per task. The next approach aims to implement DRAM controllers that prioritise small latency. The last technique utilises that modern memory is equipped with more than one memory channel. The access to specific channels is then restricted to specific tasks.

Next, the memory channel partitioning technique discussed in [4] is analysed in detail.

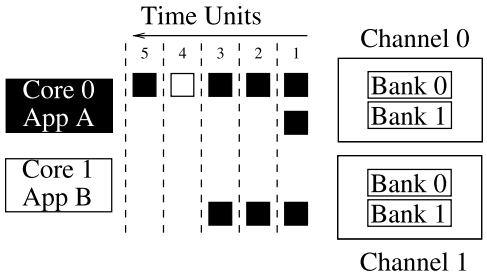
4.2.1 Memory Channel Partitioning

The Memory Channel Partitioning (MCP) approach presented in [4] aims to reduce interference by separating applications that are likely to interfere with each other into different memory channels.

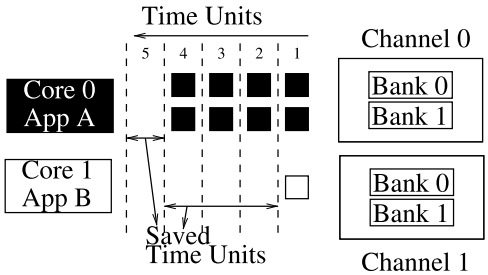
Two characteristics of applications emerge here that influence the likelihood of interference between two applications. The Memory Access Intensity is the rate at which an application produces cache misses on the LLC. It is measured in Misses per Kilo Instructions (MPKI). The other attribute is Row-Buffer Locality, which is calculated as the average Row-Buffer Hit Rate (RBH). It describes the fraction of how many requests fire row-buffer hits.

On one side, applications with high memory access intensity are more likely to interfere with applications with low intensity. This is visualised in Figure 7. In the conventional case, application B reads from memory bank 0 of channel 0, which delays the last access of app A, requiring 5 time units overall. When partitioning A and B into two different channels, this interference cannot occur, and both applications finish their accesses earlier.

On the other side, applications with high row-buffer locality and apps with low row-buffer locality are more likely to interfere with each other. This effect is shown in Figure 8. In the conventional case here, following FR-FCFS, both accesses of app B to bank 1 in channel 0 are delayed, because they result in a row-buffer miss, while the row required by A is already in the buffer. This then requires 6 overall time units for the whole access sequence. When separating

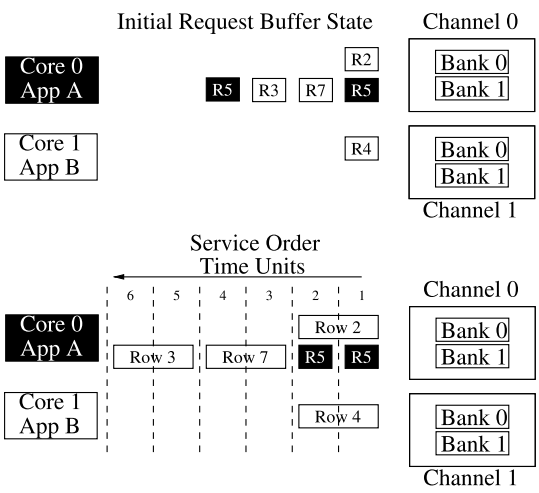


(a) Conventional Page Mapping

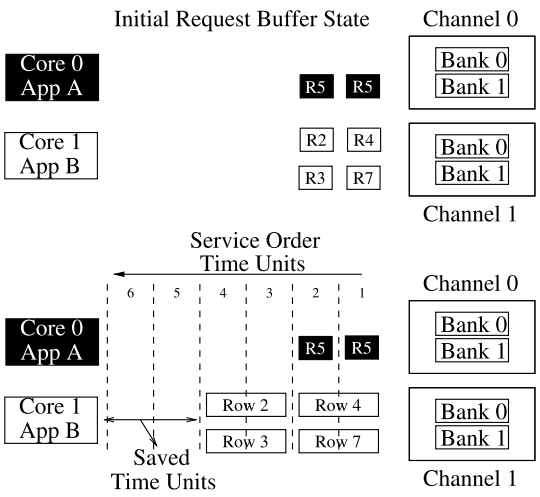


(b) Channel Partitioning

Figure 7: Comparison between conventional page mapping (a) and MCP (b), showing the advantages of mapping low and high-memory-intensity applications to different channels. The figure was taken from [4].



(a) Conventional Page Mapping



(b) Channel Partitioning

Figure 8: Comparison between conventional page mapping (a) and MCP (b), showing the advantages of mapping low and high row-buffer hit rate applications to different channels. The figure was taken from [4].

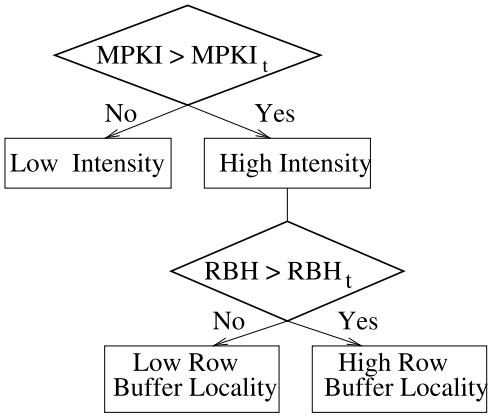


Figure 9: MCP Application Grouping Procedure, taken from [4].

A and B in case (b), app B does not experience interference anymore and finishes after 4 time units, while the time required by A is not changed.

Using the insights gained from the application characteristics described above, the MCP algorithm is described below. MCP runs in set periodic intervals. During each interval, all running applications are profiled by measuring their MPKI and RBH.

At the end of every interval, each application is categorised in one of three groups by comparing its MPKI with the threshold MPKI_t and RBH with the threshold RBH_t. This step is visualised in Figure 9. The threshold value of MPKI_t is dynamically set by averaging the MPKI values of all applications. This average is then multiplied by a scaling factor. The RBH_t value is set statically to 50%.

Each of the three groups of applications is now assigned a number of memory channels. At first, the memory channels are split into high and low-intensity channels proportional to the number of apps in the groups. Then, the high intensity channels are split into low and high locality, now proportional to the sums of MPKI. Within each group, the applications are then assigned to a specific memory channel. This is done by starting from the app with the smallest MPKI, assigning the app to one of the group channels until the sum of MPKI of apps in the channel is the sum of MPKI of apps in the group divided by the number of channels assigned to the group.

Now that each app has been assigned to a preferred channel, the MCP attempts to enforce this. This is achieved by allocating pages to the preferred channels. If a page fault occurs, this indicates that the page is not yet resident in any memory channel. Then, if a free page is found, the new page is allocated. Otherwise, a search lasting N steps, starting from the oldest unallocated page, is conducted. If an unallocated page in the preferred channel is found, it is used; otherwise, the oldest unallocated page is used instead. Even if a page hit is not in the preferred memory channel, it is still used (although implementing page migration would be possible).

4.3 System Bus

Interference mitigation of system bus interference can be classified in memory bandwidth regulator approaches, phased execution models, offline scheduling and hardware isolation [3].

Memory bandwidth regulation techniques divide the total memory bandwidth into partitions and allocate them to a core each. Approaches categorised as phased execution models extend the execution model of the tasks. Each task is modelled to have multiple phases in which it has different access rights. A scheduling algorithm is then used to schedule the tasks in such a way that eliminates contention. In offline scheduling, a precomputed static table is used to restrict access to the bus. Finally, hardware isolation techniques reduce interference by using predictable arbitration to isolate the bus to specific cores at a time.

The memory bandwidth regulation approach discussed in [12] is presented below.

4.3.1 Memory Bandwidth Regulation

The solution, called MemGuard, addresses system/memory bus interference, as presented in [12]. The mitigation is achieved by reserving a part of the minimum bandwidth of the bus for each core. Unused bandwidth is then made available to all cores. The method assumes that the LLC is partitioned so that no interference is introduced by the cache.

A high-level overview of the algorithm is given in Algorithm 2. It uses a bandwidth regulator per core that runs periodically in a set period P, by calling the function the function `periodic_timer_handler`. The overall minimum service rate budget is r_{min}, which is determined by a previous benchmark run on that system. Each core i is statically assigned a memory access budget Q_i for a period. The sum of individual budgets Q_i must not exceed the minimum service rate budget.

First, the function `periodic_timer_handler` predicts the bandwidth usage $Q_i^{predict}$ of the core. Then, the bandwidth usage at which a HPC overflow interrupt is called q_i is the to $min\{Q_i^{predict}, Q_i\}$. This is done to not use unavailable budget when $Q_i^{predict} > Q_i$ and to be able to donate budget that it predicts that it doesn't need. The donated budget of all cores is tracked by the global budget G, which is reset at the beginning of each period. Finally, the HPC is set to trigger its interrupt at q_i, and the core is enabled to resume its tasks.

Now, when the HPC triggers its interrupt, the function `overflow_interrupt_handler` is called. At first, the budget u_i, which was used in the current period, is determined. Then, if some global budget is still available, a new budget q_i is set at which the HPC interrupt is triggered for that core. Here, q_i is either set to the difference between the minimum budget Q_i and the used budget u_i or a static small budget Q_{min}, when the core has already used more than its allocated budget Q_i. Then G is reduced by the reclaimed budget, the HPC interrupt is programmed at q_i, and the core is allowed to continue. It is also ensured that only the budget available in G is used.

When the global budget has already been consumed, three cases are possible: If the core has not fully used its budget Q_i yet, it is allowed to continue regardless. This is the case when the prediction for that period was wrong, the predictor is then notified and compensates for the difference in the next period. In the second case, all cores have used their assigned budgets part of r_{min}, meaning the bus has additional bandwidth, exceeding r_{min}, left. Then the bus is

Algorithm 2: MemGuard Implementation, taken from [12]

```
1  function periodic_timer_handler begin
2      Q_i^predict ← output of usage predictor;
3      Q_i ← user assigned static budget;
4      if Q_i^predict > Q_i then
5          q_i ← Q_i;
6      else
7          q_i ← Q_i^predict;
8      end
9      G += max{0, Q_i − q_i};
10     program PMC to cause overflow interrupt at q_i;
11     re-schedule all dequeued tasks;
12 end
13 function overflow_interrupt_handler begin
14     u_i ← used budget in the current period;
15     if G > 0 then
16         if u_i < Q_i then
17             q_i ← min{Q_i − u_i, G};
18         else
19             q_i ← min{Q_min, G}};
20         end
21         G −= q_i;
22         program PMC to cause overflow interrupt at q_i;
23         return;
24     end
25     if u_i < Q_i then
26         return;
27     end
28     if ∑ u_i = r_min then
29         wake up all cores;
30         return;
31     end
32     de-schedule tasks in the CPU run-queue;
33 end
```

opened to best-effort access to be able to use that bandwidth, all cores are notified and are free to resume until the next period tick. In the third case, the core has used its full budget Q_i, while the bus is still in the guaranteed bandwidth mode. Then the core is paused by de-scheduling all tasks in its run queue. It resumes either when the next period starts or when the bus switches into the best-effort mode.

When running hard real-time tasks, a core must deactivate the bandwidth sharing feature by using $q_i = Q_i$ in the function `periodic_timer_handler`. This prevents the case where the predictor underestimates the required bandwidth, but other cores have already reclaimed it for themselves. The more tasks use the bandwidth reclaiming and sharing feature, the better the overall bandwidth usage of the whole system. Another consideration is the period MemGuard regulates. Larger periods lead to less accurate predictions, while smaller periods increase interrupt handling and performance overhead required for monitoring the cores.

5 Related Work

This section summarises and highlights literature and related topics that were not covered by this report. They can be divided into methods for interference analysis, interference-aware scheduling, further types of interference and other interference mitigation methods.

5.1 Methods for Analysing Interference Channels

Methods for analysing interference channels can mainly be categorised into empirical and formal methods. An overview of both approaches is given in the following.

5.1.1 Empirical Analysis

The focus of research in this field is on empirically measuring possible interference in the system and its impact on predictability and performance. Two approaches in this field are, for example, those described in [1], [2] and [8]. For measurement purposes, two types of tasks are introduced: attacker tasks and victim tasks. Victim tasks behave normally and are used to measure the effect of interference. Attacker tasks, on the other hand, are programmed to cause maximum interference by accessing shared resources erratically and as frequently as possible. Attackers are typically optimised for specific interference channels. Additionally, attackers employ adaptive parameter tuning to maximise the interference caused.

5.1.2 Formal Analysis

Methods in this category take a more formal approach to interference analysis. For example, in [7], the bus contention is analysed. This work uses a formal system model to estimate the WCET for such a system. Additionally, introducing a cache persistence model to multi-core results in a much more precise calculated WCET in the experiments conducted. The bounds resulting from such a formal analysis can then be used in schedulability analysis (see Section 5.2).

5.2 Schedulability Analysis

Some other works, for example [10] and [5], have also explored interference channels from a different angle. Rather than attempting to mitigate the effects of interference, these works have integrated them into schedulability analysis. Using formal models of shared resource interference leads to more accurate WCET and WCRT predictions. This means that more executions are deemed schedulable because they do not have to compensate for delays that are unpredictable without a proper shared resource model. This leads to higher utilisation of the hardware and the possibility of executing more tasks, while also guaranteeing that a system is real-time capable for a given scheduler.

5.3 Further Types of Interference

Timing interference is not the only type of interference that can occur. In safety-relevant systems such as avionics, data interference is also an important factor. While timing interference causes unpredictable response times, the interference analysed in [9] can directly alter the behaviour of the system in unpredictable ways. This work focuses on avionics and analyses data access in shared resources in terms of which processes have access rights. It examines the potential for unauthorised data alteration and presents data isolation techniques.

5.4 Other Mitigation Methods

In addition to the interference mitigation approaches presented in this report, many more can be found in the literature. These methods target different use cases and interference channels or are specialised for specific system architectures. Additionally, some methods focus on overall performance, disregarding the response times of individual tasks, while others focus exclusively on hard real-time. A comprehensive review of various interference mitigation methods can be found in [3].

Some work also combines mitigation approaches in order to mitigate multiple channels or to maximise the hardware performance. Such an example is given in the work of [4], which has also introduced the MCP mitigation method discussed in Section 4.2.1. The integrated memory partitioning and scheduling (IMPS) method combines MCP and memory scheduling. It prioritises tasks that are very low memory-intensive before others, allowing them to use any memory channel. It then uses MCP for all other tasks.

6 Conclusion

This report investigated the problem of timing interference in multi-core system-on-chip devices, which is a key challenge for modern embedded and real-time systems. As SoCs evolve to meet performance demands under tight SWaP-C constraints, sharing critical hardware resources between multiple users inevitably leads to unpredictable execution delays if not adequately compensated for. This unpredictability undermines the primary requirement of real-time systems, which is to provide timely and reliable responses in all circumstances.

Analysis of a typical SoC architecture identified three major sources of interference: shared caches, main memory and the system bus. These shared resources introduce contention mechanisms that can degrade performance and introduce unpredictability. It was also observed that the effects of interference often cascade across multiple levels of the memory hierarchy and can interact with other interference channels. While no universal solution exists, the study of mitigation strategies has shown that targeted approaches can reduce timing variability. Techniques such as dynamic page colouring, memory channel partitioning and memory bandwidth regulation demonstrate how interference can be contained through software-level control, eliminating the need for specialised hardware.

A recurring observation is the trade-off between predictability and resource utilisation. Approaches that strictly isolate resources can provide high timing guarantees since they can entirely circumvent the interference source, but they may also leave significant hardware capacity unused. Conversely, approaches that employ more dynamic schemes improve utilisation at the cost of increased complexity and monitoring overhead. For safety-critical tasks, conservative strategies may be necessary, but these approaches often allow strict guarantees to be enabled for certain tasks while maximising hardware utilisation for soft or non-real-time tasks, creating hybrid approaches.

In conclusion, it is essential to understand and mitigate interference to deploy multi-core SoCs effectively in real-time domains such as avionics, automotive safety and autonomous systems. Although this report has focused on software-based mitigation techniques, interference-aware scheduling and analytical models, as well as hardware features, are also viable mitigation methods. Furthermore, these methods can be combined to maximise the mitigation effect of each. These methods are crucial for fully harnessing the computational power of multi-core SoCs while maintaining the high level of predictability demanded by real-time applications.

References

[1] Dan Iorga, Tyler Sorensen, John Wickerson & Alastair F. Donaldson (2020): *Slow and Steady: Measuring and Tuning Multicore Interference*. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 200–212, doi:10.1109/RTAS48715.2020.000-6.

[2] Ao Li, Marion Sudvarg, Han Liu, Zhiyuan Yu, Chris Gill & Ning Zhang (2022): *PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference*. In: *2022 IEEE Real-Time Systems Symposium (RTSS)*, pp. 225–239, doi:10.1109/RTSS55097.2022.00028.

[3] Tamara Lugo, Santiago Lozano, Javier Fernández & Jesus Carretero (2022): *A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms*. *IEEE Access* 10, pp. 21853–21882, doi:10.1109/ACCESS.2022.3151891.

[4] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir & Thomas Moscibroda (2011): *Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning*. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, Association for Computing Machinery, New York, NY, USA, pp. 374–385, doi:10.1145/2155620.2155664.

[5] Luis Ortiz, Ana Guasque, Patricia Balbastre, José Simó & Alfons Crespo (2024): *Schedulability Analysis in Fixed-Priority Real-Time Multicore Systems with Contention*. *Applied Sciences* 14(10), p. 4033, doi:10.3390/app14104033.

[6] Luis Miguel Pinho, Eduardo Quinones & Marko Bertogna, editors (2022): *High Performance Embedded Computing*. River Publishers, New York, doi:10.1201/9781003338413.

[7] Syed Aftab Rashid, Geoffrey Nelissen & Eduardo Tovar (2020): *Cache Persistence-Aware Memory Bus Contention Analysis for Multicore Systems*. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 442–447, doi:10.23919/DATE48585.2020.9116265.

[8] William Vance, Roshini Ashok, John Ross, Bruce Jacobs, Tuan Bui & Mark Wotell (2024): *Analysis of Cache Memory Interference on Multicore Systems Utilizing Shared Memory Aggressor Applications*. In: *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*, pp. 1–6, doi:10.1109/DASC62030.2024.10749008.

[9] Steven H. VanderLeest, Jesse Millwood & Christopher Guikema (2018): *A Framework for Analyzing Shared Resource Interference in a Multicore System*. In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pp. 1–10, doi:10.1109/DASC.2018.8569651.

[10] Jun Xiao, Sebastian Altmeyer & Andy D. Pimentel (2020): *Schedulability Analysis of Global Scheduling for Multicore Systems With Shared Caches*. *IEEE Transactions on Computers* 69(10), pp. 1487–1499, doi:10.1109/TC.2020.2974224.

[11] Ying Ye, Richard West, Zhuoqun Cheng & Ye Li (2014): *COLORIS: A Dynamic Cache Partitioning System Using Page Coloring*. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, Association for Computing Machinery, New York, NY, USA, pp. 381–392, doi:10.1145/2628071.2628104.

[12] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo & Lui Sha (2013): *MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-Core Platforms*. In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 55–64, doi:10.1109/RTAS.2013.6531079.