Journal of Circuits, Systems, and Computers
Vol. 25, No. 6 (2016) 1650062 (17 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126616500626

Dynamic Partitioned Cache Memory for Real-Time MPSoCs with Mixed Criticality*

Gang Chen[†,§], Kai Huang[†,‡,¶], Long Cheng[†,||], Biao Hu[†,**] and Alois Knoll[†,††]

[†]*Institute of Robotics and Embedded Systems, Technical University Munich, Boltzmannstr. 3, Garching 85748, Germany*

[‡]*School of Data and Computer Science, Sun Yat-sen University, Xiaoguwei Island, Panyu District, Guangzhou 510006, China*
[§]*cheng@in.tum.de*
[¶]*huangk@in.tum.de; huangk36@mail.sysu.edu.cn*
[||]*chengl@in.tum.de*
[**]*hub@in.tum.de*
[††]*knoll@in.tum.de*

Received 7 October 2015
Accepted 7 December 2015
Published 16 February 2016

Shared cache interference in multi-core architectures has been recognized as one of major factors that degrade predictability of a mixed-critical real-time system. Due to the unpredictable cache interference, the behavior of shared cache is hard to predict and analyze statically in multi-core architectures executing mixed-critical tasks, which will not only result in difficulty of estimating the worst-case execution time (WCET) but also introduce significant worst-case timing penalties for critical tasks. Therefore, cache management in mixed-critical multi-core systems has become a challenging task. In this paper, we present a dynamic partitioned cache memory for mixed-critical real-time multi-core systems. In this architecture, critical tasks can dynamically allocate and release the cache resourse during the execution interval according to the real-time workload. This dynamic partitioned cache can, on the one hand, provide the predicable cache performance for critical tasks. On the other hand, the released cache can be dynamically used by non-critical tasks to improve their average performance. We demonstrate and prototype our system design on the embedded FPGA platform. Measurements from the prototype clearly demonstrate the benefits of the dynamic partitioned cache for mixed-critical real-time multi-core systems.

Keywords: Reconfiguable cache; real-time MPSoCs; mixed criticality.

*This paper was recommended by Regional Editor Tongquan Wei.
¶Corresponding author.

1. Introduction

Multi-core systems have become one of the preferable choices in modern embedded systems to achieve more powerful computing ability while reducing the cost of the system at the same time. Safe-critical real-time embedded systems, such as electronic vehicles,[15] are also one of the promising domains which use the multi-core systems as their computing platforms, because growing number of applications require more powerful computing platform to provide more computing power. For example, the driver assistant system in modern automotive systems needs to process high-resolution video in real-time to track objects, which requires significant computing power.[27]

The multi-core architecture, however, poses a significant challenge in designing a safe-critical real-time embedded system due to timing unpredictability caused by shared resource interference. The shared cache has been recognized as one of the most important sources of unpredictability in multi-core systems.[6] The main problem of the shared cache is that the behavior of shared cache is hard to predict and analyze statically in multi-core architectures. For instance, a task running on one core may unpredictably evict useful cache space, which is used by one task in another core. These inter-core cache interferences are extremely difficult to analyze accurately, thus resulting in difficulty of estimating the worst-case execution time (WCET) of the application program.

Integrating tasks with different levels of safety requirements on a common computing platform has increasingly become a common trend in the design of real-time embedded systems. In general, not all tasks in embedded system are equally critical for the system.[1] In one computing platform where multiple tasks are executed, the typical case is that some of these tasks may be more critical to the entire system than others. For instance, in the system of unmanned aerial vehicles (UAV), the correct behavior on flight-control tasks is more important than photo capturing tasks and multimedia applications. One of the main problems brought by such a new design paradigm is that tasks with different criticalities running on the same computing platform will compete for shared resources. This competition will result in unpredictable resource interference on critical tasks. Shared cache in mixed criticality systems is one such source of interference that can increase the response time of critical tasks.[5] In the mixed criticality real-time systems, the timing predictability of critical tasks should be ensured for the correctness of the entire system. To achieve this, the cache resources of critical tasks should be strictly isolated to prevent the cache interference from non-critical tasks. At the same time, we also should maximize the average cache utilization of non-critical tasks to improve the quality-of-service (QoS) for the system.

In this paper, we present a dynamic partitioned cache memory for mixed-critical real-time multi-core system, in which cache resources are dynamically allocated to critical tasks. In this cache architecture, the cache resources of critical tasks are strictly isolated to prevent the cache interference of non-critical tasks. Therefore, the proposed cache can provide predictable cache performance for critical tasks. In addition, the proposed cache memory allows the critical tasks to dynamically occupy the cache resources from non-critical tasks according to real-time workload. To minimize the impacts of this dynamic cache resource occupation on the performance degradation of non-critical tasks, we proposed one scheme to determine the cache configurations of critical tasks. The generated solution can guarantee that the timing constraints of critical tasks are met, while trying to minimize the dynamic cache resource occupation. The main contributions of this paper can be summarized as follows:

- We present a dynamic partitioned cache memory for mixed-critical real-time multi-core systems and prototyped it on FPGA. The developed cache can provide predicable performance for critical tasks. Compared to the existing cache architectures,[24,29,13] the dynamic partitioned cache memory allows critical tasks to dynamically allocate the cache resources according to the real-time workload and leave as much as cache resources to non-critical tasks for their performance improvement, which enables us to utilize the cache resources more efficiently.
- We detail a hardware/software co-design approach to determine cache allocation for critical tasks that satisfies schedulabilty of critical tasks while minimizes performance impacts of non-critical tasks.
- We demonstrate the applicability of our technique by implementing a hardware–software prototype on FPGA and executing a set of memory-intensive real-time benchmarks. Compared to the existing techniques[24,29,13] which are devoted to analyze theoretical proposals and the simulation of caches design, we offer the physical implementation to verify the proposed cache. Measurements from the prototype clearly demonstrate the benefits for the multi-core system equipped with dynamic partitioned cache memory.

The rest of the paper is organized as follows. Section 2 reviews related work in the literature. Section 3 presents some background principles and the definition of the studied problem. Section 4 provides the details on the dynamic partitioned cache memory for mixed-critical multi-core system and cache resource management scheme for critical tasks. Experimental evaluation is presented in Sec. 5 and Sec. 6 concludes the paper.

2. Related Work

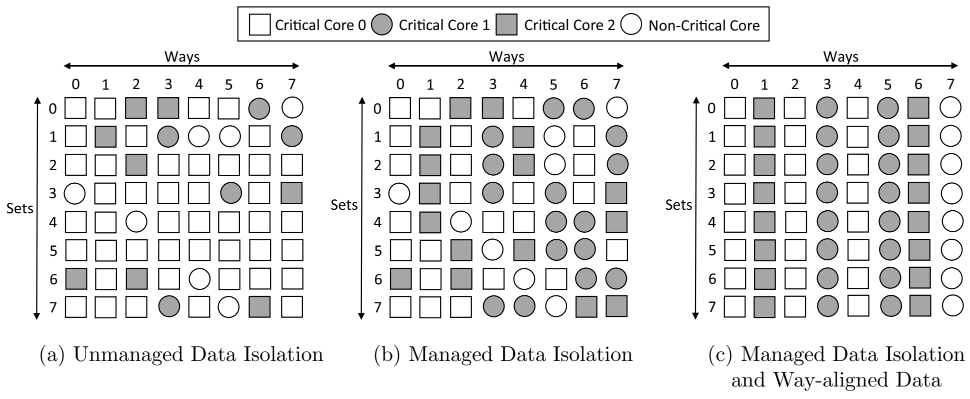
Cache partitioning techniques have been considered as one of promising techniques to improve the performance and predictability of embedded systems. By allocating private cache space to each task, cache interferences can be prevented. Much work has been done in general-purpose computing system to optimize different performance objectives by cleverly partitioning shared cache, including cache performance[21,22] and energy consumption.[31]

In the context of real-time systems, cache partitioning techniques have been explored mostly by using software-based page coloring techniques.[26,12,23] Software cache partitioning for uniprocessor real-time systems was proposed by Wolfe.[28,18] The off-chip memory mapping of the tasks is altered to guarantee the spatial isolation in the cache by using compiler technology. Bui *et al.*[2] exploited cache partitioning techniques to minimize the worst-case system utilization while considering the cache capacity constraint. For multi-core real-time systems, Kim *et al.*[12] proposed a coordinated cache management scheme to provide predictable cache performance. In this work, a portion of cache partitions is statically reserved for each core to prevent the inter-core cache interference. Tasks on each core share the reserved cache partitions and this sharing will result in intra-core cache interference. When performing the schedulability analysis, the penalties due to the sharing of cache partitions are bounded by accounting for cache warm-up delay and cache-related preemption delay (CPRD). Mancuso *et al.*[16] proposed a two-phase solution, which is based on page coloring technique, to prevent cache sharing interference. In the first phase, the real-time tasks are profiled to determine the memory access patterns. Cache resources are allocated to real-time tasks in the second phase by using colored lockdown allocation strategy. By using page coloring and real-time multiprocessor locking protocols together, Ward *et al.*[26] proposed a shared cache management scheme within the MC^2 scheduling framework[17] for mixed criticality real-time multi-core systems. In above state-of-the-art studies,[26,12,16] the shared cache is partitioned at OS-level by using page coloring techniques. Therefore, cooperating OS timing overhead also needs to be carefully considered in real-time systems. Besides, the research works[26,12,16] implement and evaluate the proposed approaches in a general-purpose operating system Linux (OS) patched with real-time extensions. Due to the complexity of the Linux kernel, the impacts of kernel activities, which have a considerable effect on real-time tasks, are hard to be predicted and evaluated.[8] Distinct to using software-based cache partitioning techniques, we present a dynamic partitioned cache architecture, which can execute dynamic cache partitioning at hardware level with minimal overhead, for mixed criticality real-time multi-core systems.

Little work has been done in the topic of cache architecture design for mixed-critical real-time multi-core system. A prioritized cache architecture is proposed in Refs. 24 and 29, where the access authorization of cache line is determined by the priority of tasks. The cache lines occupied by tasks with high priority cannot be evicted by tasks with low priority, while the cache lines owned by tasks with low priority can be evicted by tasks with high priority. Hence, the higher the task priority is, the less it suffers from inter-task conflicts. The prioritized cache can also be applied to critical and non-critical tasks. However, the prioritized cache is apt to the trashing of the whole cache by real-time critical tasks, which may result in bad performance of non-critical tasks. To avoid the trash of the whole cache, PRETI, a partitioned real-time cache scheme is presented in Ref. 13, reserves a fixed number of cache ways for critical tasks across all the sets based on prioritized cache scheme. How to determine this fixed number is not discussed in this proposal. Within the sets, data from each critical task can reside in *any* way. Based on this scheme, authors assume that the associated private space can be released when the task is terminated. This assumption is not realistic because it is difficult to release such distributed cache lines across all the sets. Above approaches are all evaluated by simulation and none of them is verified by a real hardware.

More recently, Chetan Kumar *et al.*[5] presented one cache design for mixed criticality real-time systems. The proposed cache design was implemented with a soft-core processor and verified in FPGA platform. Similar to prioritized cache architecture in Refs. 24, 29 and 13, the core of this cache design is based on the least critical cache replacement policy, in which critical cache lines cannot be evicted when there are non-critical or empty cache lines in one cache set. One different point comparing to prioritized cache architecture is that critical cache lines can be evicted only when all lines in a cache set are critical. Due to using similar cache replacement policy in prioritized cache architecture, most of cache lines are also easy to be occupied by real-time critical tasks, which will degrade the performance of non-critical tasks. Besides, this cache design is only designed for uniprocessor systems.

In contrast to existing work, we present dynamic way-based cache partitioning scheme and enforce data way-alignment for real-time critical tasks. Figure 1 gives an example of how our approach differs from existing work. As shown in Fig. 1(a), the prioritized cache[24,29] is opted to allocate the major cache lines to critical tasks. Thus, a few of cache lines can be used for non-critical tasks, which will degrade the performance of non-critical tasks. In addition, cache lines belonging to different tasks are entirely mixed across sets and ways in the prioritized cache. In Fig. 1(b), PRETI,[13] which is based on the prioritized cache, has been applied so that the number of ways



(a) Unmanaged Data Isolation (b) Managed Data Isolation (c) Managed Data Isolation and Way-aligned Data

Fig. 1. Data isolation among critical and non-critical tasks.

owned by each critical task is upper-bounded by a constant across all sets. However, within the sets, cache lines occupied by critical tasks are distributed in any way. These scattered cache lines are difficult to release for non-critical tasks' reusing. Our approach is shown in Fig. 1(c). We enforce data way-alignment so that a way is owned entirely by a critical task at a time. The main benefit is that the cache ways occupied by critical tasks can be dynamically allocated and released according to real-time workload. The released ways can be reused by non-critical tasks to further improve their performance. Besides, we physically implement and verify the proposed cache architecture on FPGA.

3. Background

This section introduces the notations and assumptions used in this paper. The characteristics of the mixed-critical multi-core architecture as well as the reconfigurable cache and real-time tasks are presented.

3.1. *Mixed-critical multicore system*

In this paper, we consider a multiprocessor architecture as shown in Fig. 2, where the cache subsystem is shared by every two cores. This multiprocessor architecture has been widely accepted by chip vendors. For example, in Intel Core 2 Quad Q8400 processor,[11] core 0 and core 1 share one cache and core 2 and core 3 share another cache. In each group of two cores which share the caches, we specify one core as *critical core* and another core as *non-critical core*. Critical and non-critical tasks are executed on *critical cores* and *non-critical cores*, respectively. This kind of mixed-critical hardware architecture has been widely adopted in much previous research work.[7,29,30] The DRAM controller is connected to the system bus. In this paper, we adopt real-time multi-core architecture presented in Ref. 19, where the accesses to

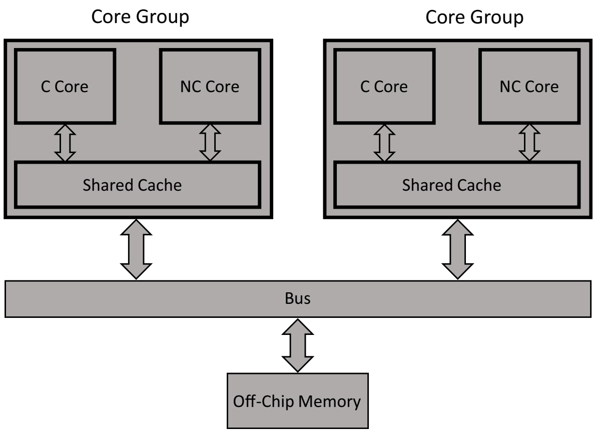


Fig. 2. Mixed-critical multi-core architecture.

shared memory bus are controlled by hierarchical arbiters. This architecture employs round-robin as the shared bus arbitration policy, thus the maximum bus access delay is bounded by the number of cores in a system.[19]

Note that, our cache design can also support a general case that several *critical cores* and *non-critical cores* can share one cache. In this general case, the system validation should be carefully considered. At any time, the sum of cache resources allocated to concurrently running tasks should be less than the cache capacity. To guarantee this property, we need to use complex cache-aware scheduling algorithms[4] to compute the cache allocations and scheduling. In this paper, we aim at demonstrating the applicability and effectiveness of dynamic partitioned cache memory in real-time multi-core systems with mixed criticality. To reduce the analysis complexity, we limit two cores as one group to share one cache in our multiprocessor architecture.

3.2. *Reconfigurable real-time cache*

Unlike traditional mixed-critical cache architectures which are based on priority cache,[24,29,13] our reconfigurable real-time cache supports dynamic way-based cache partitioning. As shown in Fig. 3, the shared L2 cache is partitioned in the ways. The cores can dynamically tune the number of ways at runtime. For example, core 2 can select the third and sixth ways by calling the cache reconfiguration APIs. This mix-critical real-time cache only allows critical cores to dynamically regulate their cache size according to real-time workload. We use the proposed reconfigurable real-time cache to provide performance isolation for the critical core running critical tasks and prevent unpredictable cache interference from the non-critical core running potentially unpredictable workload. Thus, this cache can prevent cache interference between the critical tasks and non-critical tasks and offer predicable cache behavior for the critical tasks.

In this work, we implement cache partitioning on the customized reconfigurable cache component and dynamically assign cache ways to tasks. The key difference between our cache and the existing priority cache for mix-critical systems is that our approach allows to borrow cache space from non-critical tasks when critical tasks are active and return the cache space of critical tasks back to non-critical tasks when critical tasks are terminated. In contrast, the cache space cannot be easily released due to distributed cache lines across sets in the priority cache architecture. This will

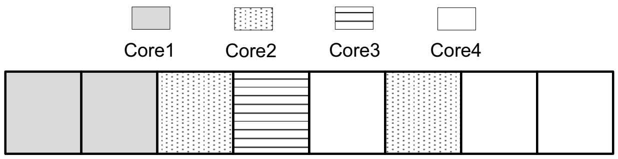


Fig. 3. Way-based cache partitioning.

result in bad performance of non-critical tasks, because the most part of cache space is seized by critical tasks.

3.3. *Task characteristics*

In this paper, we consider scheduling a set of n independent frame-based real-time tasks $\Gamma^c = \{\tau_1^c, \tau_2^c, \ldots, \tau_n^c\}$ on *critical cores*. The tasks share a common deadline D, which is also the period (or frame) of the task set. For example, in one period T which equals D, the tasks in kth frame are released at the same time instant $(k-1)\cdot T$ and share a common deadline $k \cdot T$. This task model has been widely used in Refs. 14, 25 and 20 and is a typical one which reflects various practical applications.[14] Note that we consider non-preemptive scheduling as it is widely used in industry practice, especially in the case of hard real-time systems.[10] Furthermore, non-preemptive scheduling eliminates CPRDs, and thus alleviates the need for complex and pessimistic CRPD estimation methods. The WCET of task τ_i^c under j ways of cache assignment is denoted as w_{ij}, which can be obtained from static analysis[9] or measurement-based approach.[12] On *non-critical cores*, the non-critical tasks are executed in a loop manner to achieve better performance, e.g., high-throughput.

4. Dynamic Partitioned Cache with Criticality Awareness

In this section, we present dynamic partitioned cache scheme for mixed criticality real-time systems. Our cache scheme allows cache accesses from critical and non-critical tasks to coexist. On the one hand, our cache scheme can guarantee the cache resources of critical tasks are strictly isolated to prevent the cache interference from non-critical tasks. Therefore, the timing predictability of critical tasks can be achieved. One the other hand, our cache scheme can dynamically control the release of the cache resources of the critical tasks according to real-time workload. These released cache resources can be reused by non-critical tasks to improve the average performance. Therefore, our cache scheme can minimize the slowdown of non-critical tasks by improving cache usages of non-critical tasks.

In the following, we first introduce system-level design of dynamic partitioned cache architecture which is presented in our previous research work in Ref. 3. Based on this description of dynamic partitioned cache, we present how the dynamic partitioned cache design can be extended for real-time multi-core systems with mixed criticality. Finally, we discuss how to determine cache configurations for the critical real-time tasks such that the predictability and timing constraints can be guaranteed while maximizing cache usages of non-critical tasks.

4.1. *Dynamic partitioned cache memory*

The implementation of dynamic partitioned cache scheme is based on our previous research work in Ref. 3. Currently, the dynamic partitioned cache in Ref. 3 can only

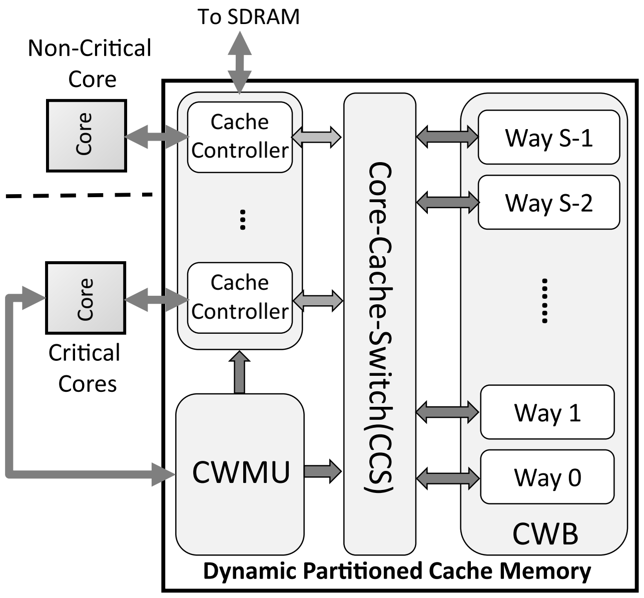


Fig. 4. System-level block diagram.

be applicable for real-time multi-core systems. In this paper, we extend the dynamic partitioned cache in Ref. 3 into real-time multi-core systems with mixed criticality.

In this subsection, we present an overview of dynamic partitioned cache in Ref. 3. Figure 4 depicts the system-level design on the architecture of a typical multiprocessor system equipped with dynamic partitioned shared cache memory. As depicted in Fig. 4, the cache memory consists of *cache ways management unit* (CWMU), *cache control unit* (CCU), *core to cache switch* (CCS) and *cache ways block* (CWB). In the architecture, CWMU controls the cache ways allocation according to the reconfiguration request of the cores. CCU manages the cache memory accesses by instantiating N *cache controllers* for a N-core system. CCS can dynamically connect cores to CWBs according to ways mask register of each core, which is maintained by CWMU according to the private cache ways pool of the cores. CWBs are memory blocks used for tag and data storage.

4.2. *Reconfiguration strategy between critical and non-critical cores*

As we mentioned in Sec. 3.2, to guarantee the predicable performance of the critical task, the dynamic partitioned cache memory *only* allows critical cores to dynamically manage the cache resources during the runtime. Non-critical cores can dynamically use the unassigned cache ways left in CWMU to improve the average performance. Thus, the reconfiguration ports of CWMU are only connected with critical cores, as shown in Fig. 4. In the dynamic partitioned cache memory, we only allow one critical core to regulate the cache resources occupied by itself. When one critical task on one critical core is scheduled to run, the critical core at first releases a certain number of cache resources from non-critical core for the execution of the critical task. Then, the critical core allocates itself the released cache resources to achieve the predicable execution of the critical task. After the critical task ends, the critical core releases the occupied cache resources from itself and reallocates these cache resources back to non-critical core. By using this reconfiguration strategy, we can, on the one hand, guarantee cache interference-free for critical tasks. On the other hand, the average performance of non-critical cores can be improved by reusing the dynamically released cache resources from critical cores.

4.3. *Implementation of reconfiguration strategy*

To illustrate how the cache reconfiguration strategy is implemented on the critical core, we list the example code as described in Listing 1. When the critical tasks are scheduled to execute in the critical core, the API function *EXE_CT*(*void* (**func*) (*void*), *int way_num*) in List 1 is called to implement cache reconfiguration strategy. The API function *EXE_CT* has two parameters: *func* and *way_num*. *func* parameter denotes the function pointer of the critical task routine, while *way_num* parameter denotes the number of cache ways allocated to critical tasks. In this API function, task routine is wrapped with cache configuration instructions (lines 3 and 4 and lines 6 and 7). Before the execution, *FreeWaysNC*(*way_num*) is called to ask *non-critical core* to release some ways (line 3) and *AlloWaysC*(*way_num*) is called to allocate the released ways for the *critical core* (line 4). Then, the critical task is called to execute under such isolated cache environment (line 5), in which cache interference can be prevented for the predicable execution. After the task is terminated, the critical core releases the occupied ways by calling *FreeWaysC*(*way_num*) (line 6) and reallocates these ways back to non-critical cores by calling *AlloWaysNC*(*way_num*) (line 7). According to implementation routines presented in List 1, the critical core can dynamically borrow ways from the non-critical core to guarantee the execution predictability of critical task on the critical core and return the borrowed cache ways back to non-critical cores to improve the performance of non-critical tasks when critical task ends.

```
1 void EXE_CT(void (*func)(void), int way_num)
2 {
3  FreeWaysNC(way_num);
4  AlloWaysC(way_num); //borrow ways
5  (*func)();
6  FreeWaysC(way_num); //return ways
7  AlloWaysNC(way_num);
8 }
```

Listing 1. Example code for cache reconfiguration.

4.4. *Determine cache configurations*

This subsection discusses how to decide cache configuration for critical real-time tasks to guarantee their real-time constraints while minimizing performance impact for non-critical tasks. As we mentioned in Sec. 3.1, we limit two cores in one group to share one cache in our multiprocessor architecture. In a core-group, one core is specified as *critical core* to execute real-time tasks and another core is specified as *non-critical core* to execute non-critical tasks. Because there is only one critical core in a core-group, we do not need to consider concurrent cache resources occupation from other critical tasks. Thus, the task schedule also do not correlate with cache size allocation, which reduces the complexity of system validation analysis. We consider the mapping specification that describes how critical tasks are mapped on core-groups is known as prior. In the following, we will present one light-weight cache allocation scheme to determine cache configurations for critical real-time tasks in one core-group. The generated cache configurations should guarantee the predictability and timing constraints of critical tasks while maximizing cache usages of non-critical tasks to improve the average performance.

To leave as much as cache resources to non-critical tasks for performance optimization, we use cache sensitivity index CSI_{τ^c} to represent how the WCETs of critical task τ^c are sensitive to the allocated caches. Cache sensitivity index CSI_{τ^c} is defined as follows:

$$CSI_{\tau^c}(i,j) = \frac{W_{\tau^c}(i) - W_{\tau^c}(j)}{j-i}, \quad i < j \leq s\,, \tag{1}$$

where $W_{\tau^c}(i)$ denotes the WCET of the critical task τ^c when i cache ways are allocated and s denotes the total number of ways of the shared cache. From the definition of CSI_{τ^c} in (1), we can see that the bigger the CSI_{τ^c} is, the more effective the cache allocation is on WCET reduction. In this scheme, cache sensitivity index CSI_{τ^c} is used to guide us to allocate the cache ways.

The pseudocode of the algorithm is depicted in Algorithm 1, which computes the cache configurations for critical real-time tasks under deadline constraints. The scheme starts to explore the cache configurations by assigning each critical task with one cache way (lines 1–4 in Algorithm 1). Considering that cache sensitivity index CSI_{τ^c} can be used to represent how efficient a cache configuration is, we greedily assigned the cache ways to critical tasks according to cache sensitivity index CSI_{τ^c} (lines 6–14 in Algorithm 1). In each iteration, we first find the maximum cache sensitivity index $CSI^{max}_{\tau^c_i}$ and the corresponding index $I_m(CSI^{max}_{\tau^c_i})$ for each critical task τ^c_i. Then, we compute the corresponding cache increase $\Delta^{max}_{\tau^c_i}$ for each critical task τ^c_i. Comparing $CSI^{max}_{\tau^c_i}$ among all critical tasks, the critical task τ^c_{max} with maximum cache sensitivity index $CSI^{max}_{\tau^c_i}$ is selected to increase its cache ways by $\Delta^{max}_{\tau^c_i}$ (lines 15 and 16 in Algorithm 1). When the sum of WCETs of

Algorithm 1 Determine cache configurations

```
Input: Deadline constraint $D$; a set of critical tasks $\Gamma^c = \{\tau_1^c, \tau_2^c, \ldots, \tau_n^c\}$, each of which has WCETs specifications $W_{\tau_i^c} = \{W_{\tau_i^c}(1), W_{\tau_i^c}(2), \ldots, W_{\tau_i^c}(s)\}$ under different cache configurations.
Output: Cache configurations $A = \{A_{\tau_1^c}, A_{\tau_2^c}, \ldots, A_{\tau_n^c}\}$
1: for Each critical task $\tau_i^c \in \Gamma^c$ do
2:     $A_{\tau_i^c} = 1$;
3: end for
4: Compute $Total\_WCET = \sum_{i=1}^n W_{\tau_i^c}(A_{\tau_i^c})$
5: while $Total\_WCET > D$ do
6:     for critical task $\tau_i^c \in \Gamma^c$ do
7:         if $A_{\tau_i^c} == S$ then
8:             $CSI_{\tau_i^c}^{max} = 0$;
9:             $\Delta_{\tau_i^c}^{max} = 0$;
10:        else
11:            Find $CSI_{\tau_i^c}^{max} = \max_{j=A_{\tau_i^c}+1}^{s}(CSI_{\tau^c}(A_{\tau_i^c}, j))$ and the corresponding index $I_m(CSI_{\tau_i^c}^{max})$;
12:            $\Delta_{\tau_i^c}^{max} = I_m(CSI_{\tau_i^c}^{max}) - A_{\tau_i^c}$;
13:        end if
14:    end for
15:    Find the task $\tau_{max}^c$ which has maximum $CSI_{\tau_i^c}^{max}$;
16:    $A_{\tau_{max}^c} = A_{\tau_{max}^c} + \Delta_{\tau_i^c}^{max}$;
17:    Update $Total\_WCET = \sum_{i=1}^n W_{\tau_i^c}(A_{\tau_i^c})$;
18:    if cache allocation of each task reaches $s$ then
19:        Report no cache configuration can be found;
20:    end if
21: end while
```

critical tasks τ^c is less than its deadline D, the iteration process ends and returns the minimal cache allocation to critical tasks to meet the deadline constraints. If the cache allocation of each critical task reaches the cache capacity s, the iteration process also ends and reports no cache configuration can be found because the process has explored all possible cache configurations in the searching space (lines 18–20 in Algorithm 1).

5. Experimental Evaluations

In this section, we show the effectiveness of our criticality-aware cache design in a real hardware platform. We first describe our experimental setup and then present the results obtained from the physical implementation.

5.1. *Experimental setup*

We implement the proposed cache reconfigurable multi-core system on the Altera DE2-115 board equipped with Cyclone IV FPGA, which is based on the NIOS II multi-core. Two cores in a core-group are shared with the unified cache, which is an instance of the dynamic partitioned cache memory proposed in Ref. 3. In our evaluation, the dynamic partitioned cache memory is configured as 16 KB cache with eight configurable cache ways. In our physical implementation, each NIOS core runs at 50 MHz.

To evaluate the performance of the dynamic partitioned cache memory, we use 10 benchmark programs as critical real-time tasks, which are selected from CHStone (Adpcm, Sha), MiBench (Crc, FFT), DSPstone (N_complex_update, LMS), Verabench (Corner_turn) and other research work (Sobel, ACC, Fdct). To avoid the selected task from saturating fast, we make some adaptations to the input scales of some benchmarks, such that they comply with the specified cache size. Table 1 lists the real-time task sets used in our experiments for one core-group, which are combinations of the selected benchmarks. Regarding the non-critical task, we use the same experiment settings in Ref. 31. The same program in Ref. 31 is used as non-critical task and is executed in a loop manner.

Due to hardware resource limitation, we are only allowed to put one core-group into FPGA platform. Therefore, the two-core system with one core-group is investigated in this experiment. Considering the hardware architecture presented in Sec. 3.1, the cache is shared in one core-group which contains a critical core and a non-critical core. Therefore, core-group can be considered as an independent research unit for the evaluation. We think the investigation on one core-group is enough to demonstrate the concept of applying dynamic partitioned cache into mixed-critical multi-core systems.

5.2. *Results*

To evaluate the performance of dynamic partitioned cache memory in real-time multi-core systems with mixed criticality, we compare our proposed cache design with static cache partitioning scheme. Static cache partitioning scheme uses the similar experiment settings as PRETI cache,[13] where multiple real-time tasks mapped on the same critical core have the same number of cache partitions. However, since cache

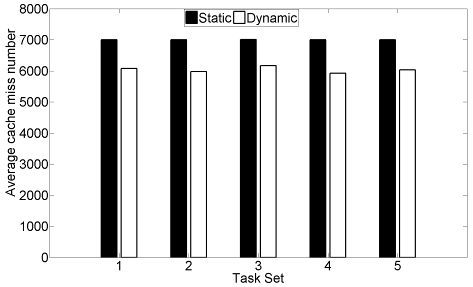
Table 1. Benchmark sets for critical real-time tasks.

Set 1	LMS, Crc, ACC
Set 2	Sobel, ACC, Fdct
Set 3	Adpcm, Crc, N_complex_update
Set 4	Corner_turn, Sha, Fdct, ACC
Set 5	FFT, N_complex_update, Crc

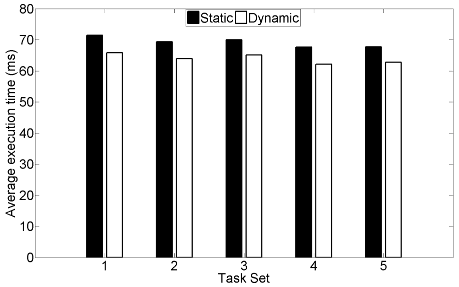
lines in PRETI cache[13] are distributed across all the sets, it is difficult to release these scattered cache lines in practice. In this experiment, we use this static cache partitioning scheme to emulate PRETI cache.[13] We use the similar approach in Ref. 13 to determine the cache allocation for each core. To evaluate the performance of non-critical tasks, we record both cache miss numbers and execution time for 300 task invocations and report the average performance for both approaches. All benchmark codes are executed on the constructed multi-core system which is implemented on FPGA and the results are collected from this FPGA implementation.

Figure 5 shows average cache miss number and average execution time of non-critical tasks for both approaches. From the result measured by real hardware, we can see that dynamic partitioned cache memory can improve system performance of non-critical tasks for all benchmark sets when compared to static cache partitioning scheme. This is expected due to the following reasons: (i) Real-time tasks might have different requirements and sensitivities to the allocated cache ways. Dynamic partitioned cache memory can efficiently assign the cache ways according to these different requirements of tasks, which could meet the features of the tasks better when compared to static cache partitioning scheme. (ii) In dynamic partitioned cache memory, cache ways occupied by critical tasks can be dynamically released when critical tasks are terminated. The released ways can be reused by non-critical tasks to further improve their performance. Comparing to static cache partitioning scheme, dynamic partitioned cache can on average achieve 13.8% on cache miss reduction and 7.62% on execution time reduction for non-critical tasks, respectively.

Next, we will show how the performance of non-critical tasks is dynamically varied under two-cache architectures. Figure 6 shows measured execution time and cache miss behavior for the non-critical task invocation under two-cache architectures while the critical core runs critical tasks in task set 4. From the results, we can make the following observations: (i) Cache miss behavior of non-critical task on static partitioned cache is steady. This is caused by the reason that the number of cache ways allocated to the non-critical core remains constant during the run-time due to static partition policy. (ii) As a comparison, cache miss behavior of

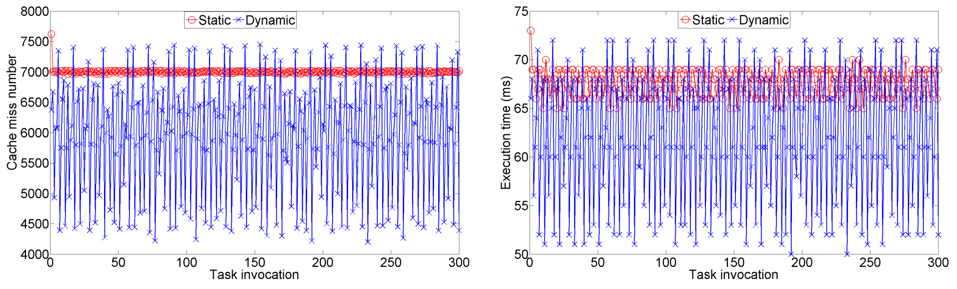


(a) Average cache miss number.



(b) Average execution time.

Fig. 5. Performance improvement of non-critical tasks.



(a) Cache miss number. (b) Execution time.

Fig. 6. Performance improvement of non-critical tasks.

non-critical task on dynamic partitioned cache varies significantly. This is expected due to the cache reconfiguration strategy we used. According to the cache reconfiguration strategy presented in Sec. 4.3, the critical core can dynamically occupy the cache ways from the non-critical core according to real-time workload. The remaining cache resources which are not used by the critical core are dynamically used by the non-critical core. This dynamical usage of cache resources results in this significant performance fluctutation on non-critical core. (iii) As shown in Fig. 6, most of non-critical task invocations on dynamic partitioned cache can benefit from the features of dynamic partition policy during the runtime. Most of task invocations on dynamic partitioned cache can achieve less cache miss and execution time when compared to static partition policy.

6. Conclusion

Nowadays, the integration of tasks with different criticalities into a common computing platform is becoming an important trend in the design of real-time embedded systems. Cache management in multi-core systems executing tasks with different criticalities has become a challenging task due to inter-core cache interference. To guarantee the timing predictability of safety-critical tasks, the cache management policy should provide performance isolation for safety-critical tasks and prevent cache interference from non-critical tasks with unpredictable workload. At the same time, we should maximize the average cache utilization of non-critical tasks to ensure high performance.

In this paper, we present a dynamic partitioned cache memory for mixed-critical real-time multi-core systems. By using the proposed cache memory, the cache resource of critical tasks can be strictly isolated to prevent the cache interference from non-critical tasks. Furthermore, compared to the existing cache architectures in Ref. 24, 29 and 13, the proposed cache memory allows the critical task to dynamically allocate and release the cache resources according to the real-time workload. The remaining cache resources which are not used by critical tasks are dynamically used by non-critical tasks. Based on this cache design, we also present one cache management scheme to determine cache configurations for critical tasks such that cache usages of non-critical tasks can be maximized. Experimental results obtained from the physical FPGA implementation demonstrate the effectiveness of the proposed cache design.

References

1. S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow and L. Stougie, Scheduling real-time mixed-criticality jobs, *IEEE Trans. Comput.* **61** (2012) 1140–1152.
2. B. D. Bui, M. Caccamo, L. Sha and J. Martinez, Impact of cache partitioning on multi-tasking real time embedded systems, *Proc. 2008 14th IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)* (2008).
3. G. Chen, B. Hu, K. Huang, A. Knoll, D. Liu and T. Stefanov, Automatic cache partitioning and time-triggered scheduling for real-time MPSoCs, *2014 Int. Conf. Reconfigurable Computing and FPGAs (ReConFig)* (2014).
4. G. Chen, K. Huang, J. Huang and A. Knoll, Cache partitioning and scheduling for energy optimization of real-time MPSoCs, *Proc. 24th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors (ASAP)* (2013).
5. N. Chetan Kumar, S. Vyas, R. Cytron, C. Gill, J. Zambreno and P. Jones, Cache design for mixed criticality real-time systems, *2014 32nd IEEE Int. Conf. Computer Design (ICCD)*, October 2014, pp. 513–516.
6. D. Dasari, B. Akesson, V. Nelis, M. Awan and S. Petters, Identifying the sources of unpredictability in cots-based multicore systems, *Proc. 2013 8th IEEE Int. Symp. Industrial Embedded Systems (SIES)* (2013).
7. L. Ecco, S. Tobuschat, S. Saidi and R. Ernst, A mixed critical memory controller using bank privatization and fixed priority scheduling, *2014 IEEE 20th Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2014, pp. 1–10.
8. G. Gracioli and A. Frohlich, An experimental evaluation of the cache partitioning impact on multicore real-time schedulers, *2013 IEEE 19th Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2013, pp. 72–81.
9. N. Guan, X. Yang, M. Lv and W. Yi, FIFO cache analysis for WCET estimation: A quantitative approach, *Proc. Design, Automation Test in Europe Conf. Exhibition (DATE)* (2013).
10. N. Guan, W. Yi, Z. Gu, Q. Deng and G. Yu, New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms, *Proc. 2008 Real-Time Systems Symp. (RTSS)* (2008).
11. Intel Q8400 processor (2014), http://ark.intel.com/de/products/.
12. H. Kim, A. Kandhalu and R. Rajkumar, A coordinated approach for practical OS-level cache management in multi-core real-time systems, *Proc. 2013 25th EuroMicro Conf. Real-Time Systems (ECRTS)* (2013).
13. B. Lesage, I. Puaut and A. Seznec, PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems, *Proc. 20th Int. Conf. Real-Time and Network Systems (RTNS)*, November 2012, pp. 171–180.
14. D. Li and J. Wu, Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms, *2012 41st Int. Conf. on Parallel Processing (ICPP)*, September 2012, pp. 430–439.

15. M. Lukasiewycz, S. Steinhorst, F. Sagstetter, W. Chang, P. Waszecki, M. Kauer and S. Chakraborty, Cyber-physical systems design for electric vehicles, *Proc. 2012 EuroMicro Conf. Digital System Design (DSD)* (2012).
16. R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo and R. Pellizzoni, Real-time cache management framework for multi-core architectures, *Proc. 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symp. (RTAS)* (2013).
17. M. Mollison, J. Erickson, J. Anderson, S. Baruah and J. Scoredos, Mixed-criticality real-time scheduling for multicore systems, *2010 IEEE 10th Int. Conf. Computer and Information Technology (CIT)*, June 2010, pp. 1864–1871.
18. F. Mueller, Compiler support for software-based cache partitioning, *Proc. ACM SIGPLAN Workshop Language, Compiler, and Tool Support for Real-Time Systems* (1995).
19. M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat and M. Valero, Hardware support for WCET analysis of hard real-time multicore systems, *Proc. 36th Annu. Int. Symp. Computer Architecture (ISCA)*, June 2009, pp. 57–68.
20. X. Qi, D. Zhu and H. Aydin, Global scheduling based reliability-aware power management for multiprocessor real-time systems, *Real-Time Syst.* **47** (2011) 109–142.
21. M. Qureshi and Y. Patt, Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches, *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, December 2006, pp. 423–432.
22. D. Sanchez *et al.*, Vantage: Scalable and efficient fine-grain cache partitioning, *Proc. 2011 38th Annu. Int. Symp. Computer Architecture (ISCA)* (2011).
23. N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein and R. Rajkumar, Coordinated bank and cache coloring for temporal protection of memory accesses, *Proc. 2013 IEEE 16th Int. Conf. Computational Science and Engineering (ICESS)* (2013).
24. Y. Tan, A prioritized cache for multi-tasking real-time systems, *Proc. 11th Workshop Synthesis and System Integration of Mixed Information technologies (SASIMI)*, 2003, pp. 168–175.
25. W. Wang, P. Mishra and S. Ranka, Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems, *Proc. 2011 48th ACM/EDAC/IEEE Design Automation Conf. (DAC)* (2011).
26. B. Ward, J. Herman, C. Kenna and J. Anderson, Making shared caches more predictable on multicore platforms, *Proc. 2013 25th EuroMicro Conf. Real-Time Systems (ECRTS)* (2013).
27. J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar and B. Litkouhi, Towards a viable autonomous driving research platform, *Proc. 2013 IEEE Intelligent Vehicles Symp. (IV)* (2013).
28. A. Wolfe, Software-based cache partitioning for real-time applications, *J. Comput. Softw. Eng.* (1994) 315–327.
29. J. Yan and W. Zhang, Time-predictable multicore cache architectures, *2011 3rd Int. Conf. Computer Research and Development (ICCRD)*, March 2011.
30. H. Yun, G. Yao, R. Pellizzoni, M. Caccamo and L. Sha, Memory access control in multiprocessor for real-time systems with mixed criticality, *2012 24th EuroMicro Conf. Real-Time Systems (ECRTS)*, July 2012, pp. 299–308.
31. H. Yun, G. Yao, R. Pellizzoni, M. Caccamo and L. Sha, Memory access control in multiprocessor for real-time systems with mixed criticality, *2012 24th EuroMicro Conf. Real-Time Systems (ECRTS)*, July 2012, pp. 299–308.
32. C. Zhang, F. Vahid and W. Najjar, A highly configurable cache for low energy embedded systems, *ACM Trans. Embedded Comput. Syst.* (2005) 363–387.