Reconfigurable Cache for Real-time MPSoCs: Scheduling and Implementation

Gang Chen, Biao Hu, Kai Huang , Alois Knoll, Kai Huang, Di Liu, Todor Stefanov, and Feng Li

Abstract

Shared cache in modern multi-core systems has been considered as one of the major factors that degrade system predictability and performance. How to manage the shared cache for real-time multi-core systems in order to optimize the system performance while guaranteeing the system predictability is an open issue. In this paper, we present a reconfigurable cache architecture which supports dynamic cache partitioning at hardware level and a framework that can exploit cache management for real-time MPSoCs. The proposed reconfigurable cache allows cores to dynamically allocate cache resource with minimal timing overhead while guaranteeing strict cache isolation among the real-time tasks. The cache management framework automatically determines time-triggered schedule and cache configuration for each task to minimize cache misses while guaranteeing the real-time constraints. We evaluate the proposed framework with respect to different numbers of cores and cache modules and prototype the constructed MPSoCs on FPGA. Our experiments show that, our automatic framework brings significant benefits over the state-of-the-art cache management strategies when testing 27 benchmark programs on the constructed MPSoCs.

Keywords:
Cache interference, dynamic cache partitioning, scheduling, real-time multi-core systems.

1. Introduction

Over the past few decades, both the speed and the number of transistors in a dense integrated circuit of processors doubled approximately every two years. This trend is commonly known as the Moore's Law. However, the access speed of the off-chip memory did not follow the same trend. To bridge the performance gap between the off-chip memory and processor, the cache component is included in nearly all processors to transparently store frequently accessed instructions and data. Since the access speed of the cache component is much faster than the off-chip memory, the cache component can effectively alleviate the performance gap between the processor and the off-chip memory by exploiting the temporal and spatial locality properties of programs.

Nowadays, the computing systems are increasingly moving towards multi-core platforms for the next computing performance leap. Increasing the number of cores increases the demanded memory access speed. The performance gap between memory and processor is further broadened in multi-core platforms. To alleviate the increasing high latency of the off-chip memory, multi-processor system-on-chip (MPSoC) architectures are typically equipped with hierarchical cache subsystems. For instance, both ARM Cortex-A15 series [1] and openSPARC series [2] all use small L1 caches for individual cores and a relatively large L2 cache shared among different cores. In such hierarchical cache subsystems, the shared cache can be accessed by all cores so that several important advantages can be achieved, such as increased cache space utilization and data-sharing opportunities.

At the same time, the shared caches also bring several drawbacks. The main disadvantage of shared caches is that uncontrolled cache interference can occur among cores, because all cores are allowed to freely access the entire shared caches. A graphic example of uncontrolled cache interference is illustrated in Fig. 1. In this example, the data element b_0 is loaded into shared cache when core 1 needs to access the data element b_0. One cache line is occupied by core 1 for future usage. Later, when core 2 needs to access another data element b_1 which is mapped in the same place of b_0, the cache line occupied by b_0 is replaced by b_1. This will result in a cache miss for the later access of b_0 on core 1. As a result, scenarios may occur where one core may constantly evict useful cache lines belonging to another core, while such cache evictions cannot bring a significant improvement for itself. Such cache interferences will cause the increase in the miss rate [3], leading to a corresponding decrease in the performance. In addition, uncontrolled cache interferences also result in unfairness [4] and the lack of Quality-of-Service (QoS) [5]. For example, a low priority application running on one core may rapidly occupy the entire shared cache and evict most of the cache lines of higher priority applications co-executed on another core.

Multi-core platforms have been used to realize a wealth of new products and services across many domains due to the average high performance. However, safety-critical real-time embedded systems failed to be benefited by this trend. In safety-critical real-time embedded systems including avionic and automotive systems, failures may lead

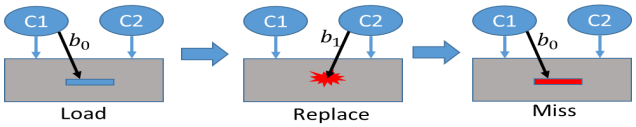


Figure 1: A graphic example of cache interference.

to disastious consequences, such as losses of lifes. Therefore, the safety-critical systems must be certified to ensure their reliability before being applied. System predictability is one of the most important principles for the development of the certifiable computing platforms [6]. In addition, system predictability is also one of the foundamental requirements for the real-time correctness. The timing correctness of real-time systems usually depends on worst-case execution time (WCET) analysis of programs. In the modern real-time computing theory, WCETs of individual tasks can be calculated as a prior to compute the schedulability of the complete system. Unfortunately, this assumption is not even true in a modern multi-core platform equipped with a shared cache. The main problem is that the behavior of shared cache is hard to predict and analyze statically [7, 8] in multi-core systems. Cache interferences as shown in Fig. 1 are extremely difficult to accurately analyze[8], thus resulting in difficulties of estimating the WCETs of the application program. How to tackle the shared cache in the context of real-time systems is still an open issue [7] and the difficulty actually prohibits an efficient use of multi-core computing platfoms for real-time systems. For instance, to resolve the predictability problem for multi-core computing platfoms, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems [9, 6]. The work in [10] also reports that inter-core cache interferences on a state-of-the-art quad-core processor increased the task completion time by up to 40%, compared to when it runs alone in the system. Therefore, it is crucial to design an interference-free shared cache memory component to improve the performance and predictability of multi-core systems.

Cache partitioning is a promising technique to tackle the aforementioned problem [3, 11, 12], which partitions the shared L2 cache into separate regions and designates one or a few regions to individual cores. Cache partitioning also has the advantage that it can provide spatial isolation of the cache, which is required by safety standards such as ARINC 653 in the avionic domain. According to [3], cache partitioning technique can be classified as software-based and hardware-based approach. The software-based approach, which is also known as *page coloring*, assigns different cache sets to different partitions by exploiting the translation from virtual to physical memory addresses. Although the software-based approach has been extensively studied in the community and can derive some promising results to improve the system performance for general purpose computing systems [13, 14, 15, 16, 17] and guarantee system prediction for safety real-time computing systems [6, 10, 18, 19], it has three important limitations: first, it requires significant modifications of the virtual memory system, a complex component of the OS. Second, one main problem for page-coloring based techniques is the significantly large timing overhead when performing recoloring. This timing overhead on the one hand prohibits a frequent change of the colors of pages [20, 13], on the other hand makes color changes of tasks whose execution time is less than the page-change overhead not worthy. Thus, software cache partitioning approach can only work well when recoloring is performed infrequently [3]. Third, the page-coloring techniques [6, 10, 19] partition the cache by sets at OS-level, cooperating OS timing overhead also needs to be carefully considered in real-time systems. Besides, the state-of-the-art studies [6, 10, 18, 19] 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. In contrast, hardware-based approach usually assigns cache ways within each cache set to different partitions with minimal timing overhead. However, most of the hardware-based cache partitioning approaches in the literature can only be used in uni-processor systems [21, 22, 23] or cannot strictly guarantee the cache space isolation among real-time applications [24].

Combining real-time task scheduling and task-level cache partitioning allocation is however more involved. On the one hand, the WCET of a task depends on the allocated cache size. On the other hand, the maximal cache budget that can be assigned to a task depends on the cache sizes occupied by other tasks that are currently running on the other cores, i.e., depending on the scheduler. Furthermore, the performance (e.g., cache miss, energy consumption, and execution time) of running tasks may have different requirements and may be sensitive to the amount of used cache because memory access patterns of tasks vary greatly from task to task. In principle, the task scheduling and the cache size allocation interrelate to each other with respect to the system performance, such as cache misses [20] and energy consumption [25]. Therefore, a sophisticated framework is needed to find the best trade-off between them in order to improve the system performance [25].

In this paper, we present a dynamic partitioned cache memory for multi-core systems and implement dynamic cache partitioning on top of our customized reconfigurable cache hardware component. This paper summarizes and extends the results built in [26]. In this cache architecture, the cache resources are strictly isolated to prevent the cache interference among cores. Therefore, the proposed cache can provide predictable cache performance for real-time applications. To efficiently use cache resources, the proposed cache allows cores to dynamically allocate cache resource according to the demand of applications. Based on the proposed dynamic partitioned cache memory, we

tackle schedule-aware cache management scheme for real-time multi-core system. We present an integrated framework to study and verify the interactions between the task scheduling and the shared cache interference. For a given set of tasks and a mapping of the tasks on a multi-core system, our framework can generate a fully deterministic time-triggered non-preemptive schedule and a set of cache configurations during the compilation time. During runtime, the cache is reconfigured by the scheduler according to offline computed configurations. The generated schedule and the cache configurations together minimize the cache miss of the cache subsystem while preventing deadline misses and cache overflow. With the customized reconfigurable cache component and share-clock multi-port timer component, our framework can generate multi-core system with different cache modules (different cache configurations with respect to cache lines, size, associativity) and prototype on Altera FPGA. The contributions of our work are as follows:

- A parameterized dynamic patitioned cache memory is developed for the real-time multicore systems. The cache size, line size, and associativity of the cache memory can be parameterized during compile time while the partition of the cache can be reconfigured in a flexible manner during runtime. We also design a complete set of APIs with atomic operation, such that the application tasks can reconfigure their cache sizes during runtime. In contrast to most existing work [27, 28, 29, 30, 31, 25] in the literature, which is devoted to analyze theoretical proposals and the simulation of reconfigurable caches, the proposed cache is physically implemented and prototyped on FPGA.

- We conduct the front-end chip design for the proposed reconfigurable cache by using Synopsys design compilers [32] with the SMIC 130nm standard technology library [33], and find the implementation of the proposed cache is practical in terms of chip area and power consumption.

- We propose an integrated cache management framework that improves the execution predictability for real-time MPSoCs. The proposed framework can automatically generate fully deterministic time-triggered non-preemptive schedule and cache configurations to optimize the system performance under real-time constraints. We develop a share-clock multi-port timer component that enables the time-triggered schedule to be implemented on the MPSoCs generated from our framework.

- We prototype and evaluate the generated MPSoCs on Altera Statrix V FPGA using 27 real-time benchmarks. We also analyze and discuss the experimental results under different hardware environments with respect to the number of cores and cache settings.

The rest of the paper is organized as follows: Section 2 reviews related work in the literature. Section 3 presents some background principles. Section 4 overviews the proposed framework and Section 5 describes the proposed synthesis approach. Section 6 illustrates the proposed hardware infrastructures and Section 7 explains how the proposed framework works. Experimental evaluations are presented in Section 8 and Section 10 concludes the paper.

2. Related Work

Real-Time Cache Partitioning: Shared cache interference in a multi-core system has been recognized as one of the major factors that degrade the average performance [13], as well as predictability of a system [6, 8]. Many works have been done in general-purpose computing to optimize different performance objectives by cleverly partitioning shared cache, including cache performance [34, 35] and energy consumption [24]. In the context of real-time systems, cache partitioning technique have been explored mostly by using software-based solution [36, 37, 6, 10, 19]. In [36, 37], the off-chip memory mapping of the tasks is altered to guarantee the spatial isolation in the cache by using compiler technology. However, altering tasks' mapping in the off-chip memory is far from trivial, which requires significant modifications of the compilation tool chain. In addition, the partitioning of the task can only be statically suppressed in fixed cache set regions due to the pre-decided memory mapping, which also prevents the efficient usage of the limited cache resource. Recently, the techniques [6, 10, 19] on the multi-core cache management in the context of real-time systems have been proposed by using page-coloring, which partitions the cache by sets at OS-level. However, page-coloring based techniques usually suffer from a significant timing overhead inherent to changing the color of a page, which results in that making decision of changing the color of a page cannot be frequent. The authors in [13] reported that the observed overhead of page-coloring based dynamic cache partitioning reaches 7% of the total execution time even after conducting the optimization to reduce the recoloring times. Distinct to above set-based cache partitioning techniques, we present a reconfigurable cache architecture to execute dynamic way-based cache partitioning in hardware level. Our approach can dynamically change the cache size with minimal overhead (scaling to cycles). Besides, compared to set-based cache partitioning techniques, our way-based reconfigurable cache can turn off the whole unused ways to save static energy [27, 24]. Therefore, our way-based reconfigurable cache can also bring benefits for low-power design.

Reconfigurable Cache: Numbers of general or application specific reconfigurable cache architectures have been proposed in the literature. Albonesi et al. [27] proposed a selective ways cache architecture for uni-processor system, which can disable a subset of the ways in a set associative cache during periods of modest cache activity and

enable the full cache to remain operational for more cache-intensive periods. By collecting cache performance of applications on runtime, Suh at al. [28] proposed a general dynamic partitioning scheme for the set associative cache. The simulation based evaluation shows the potentials for performance improvement. Benitez et al. [29] proposed amorphous cache aimed at improving performance as well as reducing energy consumption. As opposed to the traditional cache architectures, the proposed cache architecture uses homogenous sub-caches which can be selectively turn-off according to the workload of the application and reduce both its access latency and power consumption. Based on the cache architecture in [24], Sundararajan et al. [30] presented a set and way management cache architecture for efficient run-time reconfiguration.

Most of above work [27, 28, 29, 30] is devoted to analyze theoretical proposals and the simulation of reconfigurable caches. Thus, their systems are only tailored at simulation. Only few research work [24, 21, 22, 23] is devoted to the physical implementation of the proposed cache models. Zhang et al. [24] proposed a reconfigurable cache architecture where the cache ways configuration could be tuned via the combination of configuration register and physical address bits. Fig. 2 illustrates a four-way set-associative reconfigurable cache architecture proposed in [24]. In this architecture, the cache ways selection during the reconfiguration is related to the address bits of the application, which cannot guarantee the strict cache isolation among real-time applications. As shown in Fig. 2, one way is selected when Reg0= 0 and Reg1= 0. However, which exact one way is selected is also determined by two physical address bits $A18$ and $A19$. The overlapped address mapping of the real-time applications on these two physical address bits $A18$ and $A19$ will result in cache interference. In addition, the number of the allocated cache ways can *only* be configured to be a power of two, which prevents the efficient usage of the limited cache ways. Gil et al. [21, 22] presented one general-purpose reconfigurable cache design *only* for uni-processor systems to be implemented on FPGA. Besides, the proposed reconfigurable cache design [21, 22] can only work as direct mapped cache or 2-way set associative cache. Thus, this cache design is quite limited for usage. Motorola M*CORE processor [23] supports a configurable unified set-associative cache whose four ways could be individually shutdown to reduce dynamic power during cache accesses. Besides, the cache in M*CORE processor [23] can be configured as different functional cache (instruction cache, data cache, or unified cache). However, M*CORE processor is developed for uni-processor systems. It is not easy to extend such reconfigurable cache into multi-core systems due to synchronization and atomic operation issues as stated in Section 6.1.

In this paper, we propose a parameterized reconfigurable cache architecture for real-time multi-core system and physically implement it on FPGA. In this architecture, cache ways can be tuned without constraints and can be efficiently and dynamically partitioned and allocated to applications, which can guarantee that the cache resource is strictly isolated among real-time applications to prevent the cache interference. Besides, our reconfigurable cache memory supports parameterized design. The cache size, line size, and associativity of the cache memory can be parameterized during compile time. The reconfigurable cache memory can be automatically generated by setting the parameters, e.g., cache size, line size, and associativity. Thus, the proposed reconfigurable cache memory supports hardware generation. The dynamic partitioned cache memory can be interfaced and executed with CPUs for embedded systems such as Altera NIOS II processor. We provide one physicial prototype on FPGA and this prototype will serve us a real (not simulation) reconfigurable cache for studying and validating cache management strategies on the real-time multi-core system under different cache configurations.

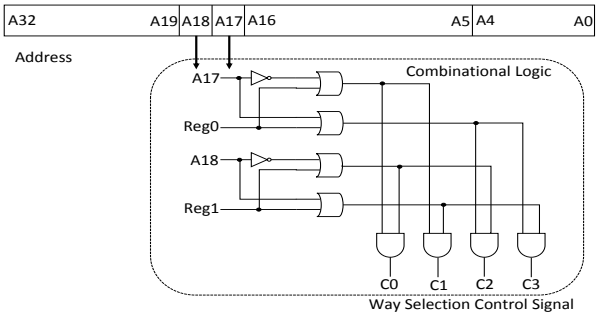


Figure 2: Illustration of four-way set-associative reconfigurable cache architecture in [24].

Time-triggered Scheduling: Time-triggered execution models can offer a fully deterministic real-time behavior for safety-critical systems. Current practice in many safety-critical system domains, such as electric vehicles [38] and avionics systems [39], favors a time-triggered approach [40]. Sagstetter et al. [41] presented a schedule integration framework for time-triggered distributed systems tailored to the automotive domain. The proposed framework uses two-step approach, where a local schedule is computed first for each cluster and the local schedules are then merged to the global schedule, to compute the schedule for the entire FlexRay network and task schedule on ECUs. To optimize the control performance of distributed time-triggered automotive systems, Goswami et al. [42] presented an automatic schedule synthesis framework, which generates time-triggered scheduling for tasks on processor and messages on bus. Nghiem et al. [43, 44] presented an implementation of PID controller using time-triggered scheduling paradigm and showed the effectiveness of such time-triggered implementation. Based on time-triggered scheduling, Jia et al. [45] presented an approach to compute message scheduling based on Satisfiability Modulo Theories (SMT) for Time-Triggered Network-on-Chip. All above techniques are evaluated by simulation. In [46], Ayman et al. describe a two-stage search technique which is intended to support the con-

figuration of time-triggered schedulers for single-processor embedded systems. However, none of them apply time-triggered scheduling and cache management jointly on real-time multi-core platform in order to achieve timing predictability and system performance.

3. Background

3.1. Way-based Cache Partitioning

Our cache management scheme implements dynamic way-based cache partitioning on FPGA. As shown in Fig. 3, the shared cache is partitioned in the ways. Each core can dynamically tune the number of selective-ways. For example, core 2 can select the 3rd and 6th way by calling the cache reconfiguration APIs. In this work, we implement cache partitioning on our customized reconfigurable cache component and dynamically assign cache ways to tasks.

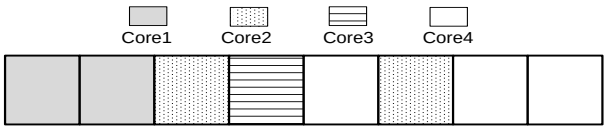


Figure 3: Way-based Cache Partitioning.

3.2. Task Model

We consider the functionality of the entire system as a task set $\tau = \{T_1, ..., T_n\}$, which consists of a set of independent periodic tasks. We use w_{ij} to denote the worst case execution time (WCET) of task $T_i \in \tau$ with j ways shared cache allocated and $W_i = \{w_{i1}, w_{i2}, ..., w_{iu}\}$ to denote the WCET profile of task T_i, where u is the total number of ways in the shared cache (cache capacity). In this paper, a measurement-based WCET estimate technique is used to determine WCET. Timing predictability is highly desirable for safety-related applications. We consider a periodic time-triggered non-preemptive scheduling policy, which can offer a fully deterministic real-time behavior for safety-critical systems. Note that we consider non-preemptive scheduling as it is widely used in industry practice, especially in the case of hard real-time system [47]. Furthermore, non-preemptive scheduling eliminates the cache-related preemption delays (CPRDs), and thus alleviates the need for complex and pessimistic CRPD estimation methods. We use R to denote the set of the profiles for all tasks in task set τ. A task profile $r_i \in R$ is defined as a tuple $r_i =< W_i, s_i, h_i, d_i >$, where s_i, h_i, d_i are respectively the start time, period, and deadline of the task T_i. We consider implicit-deadline systems [48, 49] where the deadline of each real-time task is equal to its period. This classic system setting has been widely used and studied in the real-time community [49, 50, 51].

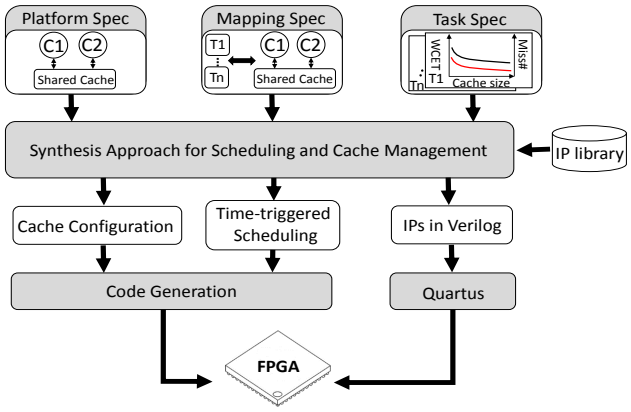


Figure 4: System Design Framework.

4. Framework Overview

In this section, we give an overview of our system design framework depicted in Fig. 4, which takes both real-time scheduling and cache partitioning into consideration to study and verify the interactions between the multi-core real-time scheduling and shared cache management. As shown in Fig. 4, the input specifications of the proposed framework consist of the following three parts:

1. *Platform Specification* describes the settings of a multiprocessor platform, such as the number of cores, the settings of L2 cache with respect to cache size, line size and associativity.
2. *Mapping Specification* describes the relation between all tasks in the *task specification* and all cores in the *platform specification*. The mapping specifications can be written by hand or automatically generated by design space exploration tools.
3. *Task Specification* describes task timing requirements and task profile information (i.e., the WCETs and cache miss number under different cache size). We describe the details about how to profile each task in Section 7.

As output, the synthesis approach can generate cache size allocation and time-triggered scheduling for each task according to the *input specifications*, by which the total cache miss number is minimized. Based on this optimal schedule and cache allocation, tasks can be scheduled with insertion of cache size allocation instructions. Task code can be generated by integrating this optimal approach into real-time scheduler. At the same time, parameterized reconfigurable cache IP and share-clock mutli-port timer IP can be generated according to the settings in *platform specification*.

5. Synthesis Approach for Scheduling and Cache Management

This section presents the synthesis approach for timing schedule and cache management. We reuse the approach in [11] to model the scheduling and cache interference,

and formulate the problem as integer linear programming (ILP) to minimize the cache miss of the system. With this formulation, the cache size allocation and time-triggered scheduling for each task can be generated automatically, which could avoid deadline miss and cache overflow.

5.1. Time-Triggered Task Scheduling

Time-triggered non-preemptive schedule is considered in this paper to achieve full predictability of the system. For each task T_i with the profile $< W_i,\ s_i,\ h_i,\ d_i >$, the k-th instance of task T_i starts at $s_i + k \cdot h_i$. W_i contains the WCETs of the task with different cache configurations. We use a set of binary variables c_{ij} to describe the amount of cache allocated to the task T_i: $c_{ij} = 1$ if exactly j cache ways are allocated to T_i and $c_{ij} = 0$ otherwise. In this case, the actual WCET of T_i can be obtained as $\sum_{j=1}^{u} c_{ij} w_{ij}$, where u is the total number of ways of the shared cache. To formulate the scheduling problem by means of ILP, we have to gurantee the following timing constraints.

For deadline constraint, task T_i has to finish no later than its deadline:

$$s_i + \sum_{k=1}^{u} c_{ik} w_{ik} \leq d_i$$

The non-preemptive constraint requires that any two tasks mapped to the same core must not overlap in time. Let binary variable denote the execution order of task T_i and T_j: $z_{p\tilde{p}}^{ij} = 1$ if the i-th instance of task T_p finishes before the start of j-th instance of $T_{\tilde{p}}$, and 0 otherwise. H_r and $H_{p\tilde{p}}$ denote the hyper-period of all tasks and the hyper-period of only task T_p and $T_{\tilde{p}}$ (i.e., LCM of periods of T_p and $T_{\tilde{p}}$), respectively. $TS(T_p)$ denotes the set of tasks that are mapped to the same core as T_p does. ξ denotes the overhead of task switch. The non-preemption constraint can thereby be expressed as follows.
$\forall T_p, T_{\tilde{p}} \in TS(T_p), i = 0, ..., (\frac{H_{p\tilde{p}}}{h_p} - 1), j = 0, ..., (\frac{H_{p\tilde{p}}}{h_{\tilde{p}}} - 1)$:

$$i \cdot h_p + s_p + \sum_{k=1}^{u} c_{pk} w_{pk} - (1 - z_{p\tilde{p}}^{ij}) H_r + \xi \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} \quad (1)$$

$$j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \sum_{k=1}^{u} c_{\tilde{p}k} w_{\tilde{p}k} - z_{p\tilde{p}}^{ij} H_r + \xi \leq i \cdot h_p + s_p \quad (2)$$

The constraints (1) and (2) ensure that either the instance of T_p runs strictly before the instance of $T_{\tilde{p}}$, or vice verse.

5.2. Cache Management Constraints

The next step is to add the cache management constraints, which guarantee the feasibility of cache management, i.e., at any point in time, the sum of cache ways allocated to the tasks currently being executed does not exceed the cache capacity. To avoid *cache overflow*, we recall the following lemma in [11], which indicates that a finite number of time instants, i.e., at the start of any task, should be checked for the *cache overflow*.

Lem. 1. *If the cache does not overflow at the start instant of any task within one hyper-period, the cache never overflows.*

By using the similar approach in [11], periodical square wave function (PSWF) is used to model the resource demand of task in the time domain. According to [11], we can use periodical square wave function (PSWF) to indicate if the task is running at the specific time instance. For task T_p with start time s_p and execution time e_p, the cache demand at the instant t can be defined as:

$$PSWF(t, T_p) = \left\lfloor \frac{t - s_p}{h_p} \right\rfloor + 1 - \left\lceil \frac{t - s_p - e_p}{h_p} \right\rceil$$

The PSWF above indicates that task T_p requires the cache only in interval $[s_p + i \cdot h_p,\ s_p + e_p + i \cdot h_p]$. According to Lem. 1, we can guarantee to avoid *cache overflow* by checking the start instant of any task within one hyper-period. Thus, we can formulate cache management constraints as follows.
$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{u} c_{pk} \cdot k + \sum_{T_{\tilde{p}} \notin TS(T_p)} PSWF(s_p + i \cdot h_p, T_{\tilde{p}}) \sum_{k=1}^{u} c_{\tilde{p}k} \cdot k \leq u$$

The term of $PSWF(s_p + i \cdot h_p, T_{\tilde{p}}) \sum_{k=1}^{u} c_{\tilde{p}k} \cdot k$ represents cache requirements of the task $T_{\tilde{p}}$ at the start time of T_p. One may notice it is non-linear term. We can transform this non-linear term into a set of linear constraints using the approach presented in [11]. Besides, each task must have exactly one cache configuration.

$$\sum_{k=1}^{u} c_{ik} = 1$$

To minimize the cache miss number in one hyper-period, the following objective function is used:

$$CM = \sum_{\forall T_i} \frac{H_r}{h_i} \sum_{j=1}^{u} c_{ij} CM^{ij}$$

where u and CM_{cache}^{ij} represent the cache capacity (in the number of ways) and the cache miss of task T_i under j-way cache configuration, respectively.

6. Proposed Hardware Infrastructure

In this section, we present one FPGA-based multi-core system which supports dynamic cache partitioning and time-triggered scheduling. A major benefit of choosing FPGA for prototyping our multi-core system is the high configurability of the processor. This allows us to evaluate the proposed integrated scheduling and cache management framework under various hardware configurations with different cache sizes and varied arithmetic units. Fig. 5 illustrates the proposed multi-core system on FPGA, where the cache is shared among cores. We adopt the NIOS II core in the system. Modules highlighted with white color

in Fig. 5 indicate the hardware components specifically designed and implemented for our framework. The system consists of several NIOS II cores along with reconfigurable cache IP which supports dynamic cache partitioning and share-tick timer IP for time-triggered scheduling.

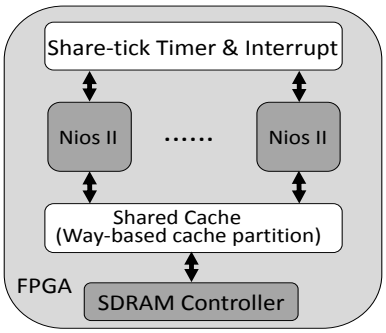


Figure 5: System architecture.

6.1. Design Consideration and Challenge

Cache coherency problem is one of critical design considerations for the dynamic way-based cache partition infrastructure. According to the Altera NIOS II datasheet [52], the current NIOS architecture does not provide hardware cache coherency. When creating multiprocessor systems, software for each processor is required to locate in its own unique region of off-chip memory to avoid to implement cache coherency [52]. NIOS II SBT provides a simple scheme of memory partitioning that allows multiple processors to run their software from different regions of the same off-chip memory [52]. Besides, according to the state-of-the-art research work in [53], current cache coherence strategies are not suitable for real-time systems. In this paper, we mainly focus on studying the cache interference among the cores, and follow this official design from Altera to create our multi-core system. Actually, this kind of memory architecture known as Partitioned Global Address Space (PGAS) has been widely accepted in the embedded community for efficiency reasons and real-life examples come from Adapteva Parallella multi-core chip E16G301 and E64G401 [54]. Note that inter-core cache interference still exists although software on each core runs in different regions of the same off-chip memory[1]. Besides, the proposed shared cache architecture is multi-port cache, which allows NIOS cores to access the cache concurrently.

Another important part that should be carefully considered is atomic operations. In general, to adaptively change the cache size, one core needs a two-phase operation, i.e., inquiry and allocation (as shown in Fig. 6). In the inquiry phase, the core needs to check which ways are available at the current moment. Then, based on the inquiry results, the core can acquire cache resource in the

[1]Unique region of each processor on off-chip memory is larger than the total cache size

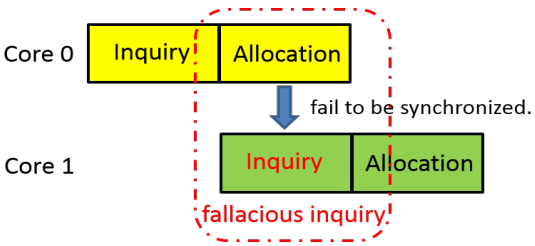


Figure 6: Atomic operations.

allocation phase. Normally, this procedure works well in a uni-processor system due to no core interference. However, in multi-core systems, when one core is checking the cache resource state, the cache management logic might be conducting cache allocation for other cores. This may lead to the fallacious cache resource state inquiry, because the results of the on-going cache allocation fail to be synchronized to the current cache resource state. Therefore, in a multi-core system, the APIs for adjusting the cache size should be guaranteed to be atomic for implementing synchronization primitives. Hence, we develop a component, called *cache ways management unit (CWMU)* to execute cache ways allocation and release, which grantees the offered APIs atomicity.

The implementation of the replacement policy for the way-based partitioning cache is another design challenge. To efficiently use the limited cache resource, the proposed cache architecture allows each core to dynamically tune its cache ways without any constraint. This will result in that the cache ways occupied by one core might not be adjacent to each other. As shown in Fig. 3, the 3rd and 6th ways are occupied by core 2. Therefore, standard replacement policies cannot be applied. In this paper, we develop block reference field logic (BRFL) to maintain this discontinuous cache ways distribution.

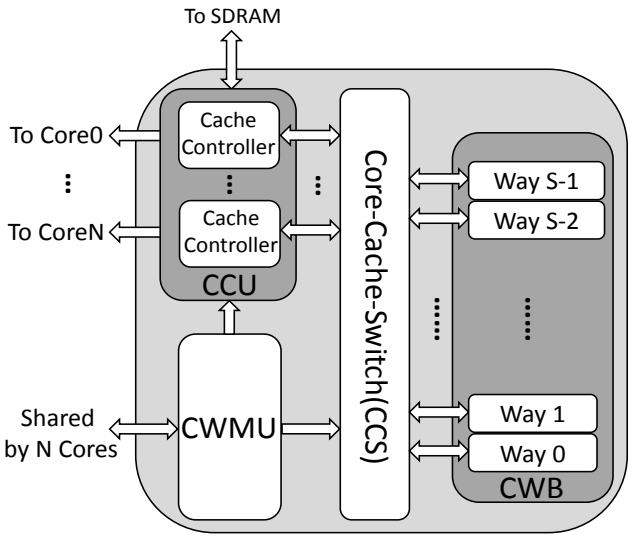


Figure 7: Reconfigurable cache architecture.

6.2. Reconfigurable Cache Architecture

This section presents an overview of the proposed reconfigurable shared cache architecture. The reconfigurable shared cache component allows cores to dynamically change the number of owned cache ways. As depicted in Fig. 7, the proposed reconfigurable shared cache consists of *cache ways management unit (CWMU)*, *cache control unit (CCU)*, *core to cache switch (CCS)*, and *cache ways block (CWB)*. In the proposed architecture, *cache ways management unit (CWMU)* controls the cache ways allocation according to the reconfiguration request of the cores. The reconfiguration port of *CWMU* is shared by all cores. *Cache control unit (CCU)* manages the cache memory accesses by instantiating N *cache controllers* for N-core system. *Core to cache switch (CCS)* can dynamically connect cores to cache ways blocks according to ways mask register of each core, which is maintained by *CWMU* according to the private cache ways pool of the cores. *Cache ways blocks (CWB)* are memory blocks used for tag and data storage.

6.3. Cache Ways Management Unit (CWMU)

Cache ways management unit (CWMU) is used to manage cache ways in a centralized manner, by which each core can send reconfiguration command to dynamically regulate its cache ways. *CWMU* is connected to N NIOS cores by *avalon slave interface (ASI)* and a round-robin arbiter is automatically created between N NIOS cores and CWMU by Altera SOPC builder. As shown in Fig. 8, when *CWMU* receives one command from one NIOS core, the *CMD decoder* component can distinguish the core ID (i.e., identity which core sends this command) and its command type (i.e., identity command types in Tab. 1). If it is allocation ways command, ways IDs will be fetched from the *global ways pool*. Then, the fetched ways IDs are put into the cache ways pool of the distinguished core. Then, *core to cache switch (CCS)* is controlled to connect cache ways to the distinguished core according to the cache ways pool. Before fetching ways IDs from *global ways pool*, the logic will check whether there are enough ways in the pool. If no enough ways exist in the pool, *cache overflow* error will be returned to the distinguished core. Note that the approach in [11] can be applied to calculate one safe cache configuration for real-time applications, which can guarantee that *cache overflow* error will never occur. In contrast to the procedure of allocation ways command, release ways command will fetch ways IDs from the cache ways pool of the distinguished core to the *global ways pool*. Ways occupied by the distinguished core and replacement information are correspondingly updated at this point. Note that due to this centralized management scheme, cores do not need to inquiry the cache state any more before the allocation operation. Therefore, the APIs for cache reconfigurations are atomic.

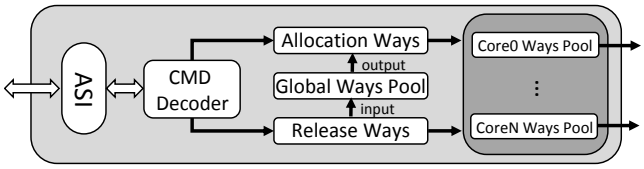


Figure 8: Cache ways managment unit (CWMU).

6.4. Cache Control Unit (CCU)

Cache control unit (CCU) instantiates N *cache controllers* for an N-core system, where each core owns one *cache controller*. *Cache controller* is used to maintain the access for its corresponding NIOS core. Thus, this shared cache allows NIOS cores to access the cache concurrently. For *cache controller*, we employ the write-through policy for each write operation. Cache write-through policy is inherently tolerant to soft errors due to its immediate update feature [55]. The cache architecture with write-through policy has been adopted in many real-life high-performance processors such as Niagara processor [56], IBM POWER5 processor [57], and Itanium processor [58].

Fig. 9 depicts the block diagram of *cache controller*. Transactions from NIOS cores are injected through the cache ports, which is instantiated as *avalon slave interface (ASI)*. Evictions, refills and write-through are asserted from off-chip memory port, which is instantiated as *avalon master interface (AMI)*. The data-width of both *ASI* and *AMI* in our case is 32 bit. The supported maximum burst of both ports depends on the cache line size. Thus, muxs and demuxs in *ASI* and *AMI* are used to packet and depacket bytes in the corresponding cache line size. The control logic performs hit/miss check, returns the read data, and asserts evictions and refills. The victim cache line is selected by the block reference field logic (BRFL) during the refill phase. The implementation of the partitioned replacement policy is presented in Section 6.5.

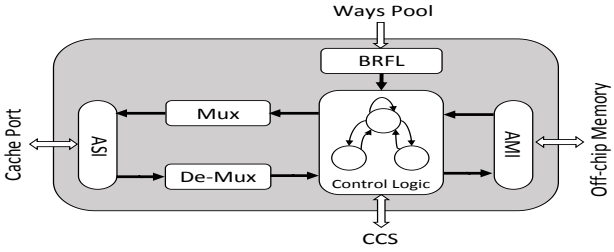


Figure 9: Cache controller (CC).

6.5. Implementation of Partitioned FIFO Replacement Policy

When a new data must be stored in a cache memory and all cache ways have been occupied, one of the existing cache line must be selected for replacement. Standard replacement policies include LRU, FIFO, etc. As the cache with the FIFO replacement policy could support accurate

quantitative WCET estimations [59] and prevent timing anomalies [60] for the real-time applications, we consider FIFO cache replacement policy in our design. In addition, the FIFO replacement policy has been widely used in the state-of-the-art processors such as ARM 11 processor and Intel X86 processor [59].

As mentioned in Section 6.1, dynamic cache partitioning may result in that cache ways occupied by one core might not be adjacent to each other. To maintain the discontinuous cache ways distribution, the block reference field logic (BRFL), as shown in Fig. 10, is proposed to perform victim selection for cache write operations. The reference field contains selection reference memory (SRM) and valid bits memory (VBM). The references of the next selection of victim cache lines are stored in the selection reference memory (SRM). SRM can be instantiated by one FPGA dual port memory block with the depth Q and width $Log_2(u)$, where Q and u denote cache depth and cache associativity, respectively. When the core release ways, all the contents of SRM should be cleared to initial reference. In general, we can clean the content of SRM one by one. Assuming each clean operation will cost one clock, cleaning all the content of SRM will cost Q clocks. Therefore, this solution will significantly increase the timing overhead of reconfiguration. To minimize timing overhead of cache reconfiguration, we propose one solution in this paper to reset SRM by using VBM, which can be instantiated as Q-bit register and be cleared in one clock. By using this similar approach, the cache ways can be flushed in one clock when the core release the ways. We use one bit valid register to associate with each reference in SRM. When we read a reference from one location of SRM, the valid bit register acts as a toggle to determine the output. Based on the current reference, the write control logic (WCL) updates the write data for reference field on each cache write operation and write the next selection to reference field of SRM and VBM, making that ways are selected in FIFO replacement manner.

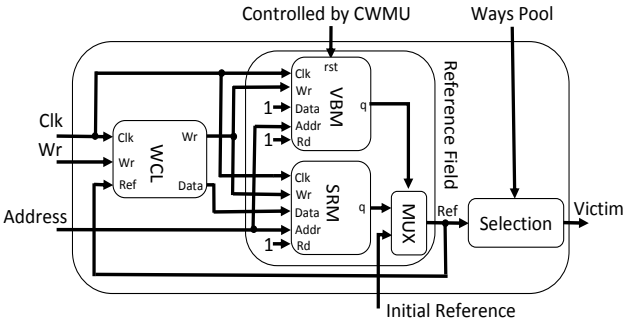


Figure 10: Block reference field logic (BRFL).

6.6. Share-clock Multi-port Timer IP

To support the dynamic timekeeping functionality in the time-triggered scheduling, a free-running counter and timers per processor are required. For the single processor system, this role is adequately served by the NIOS timer peripheral. While this is sufficient for a single core system, it does not work well with multiple processors due to a synchronization problem. In a multi-core system, we should guarantee that all the cores in the system are triggered in one global timer. Only in that way, the tasks on different cores can be precisely triggered and well synchronized.

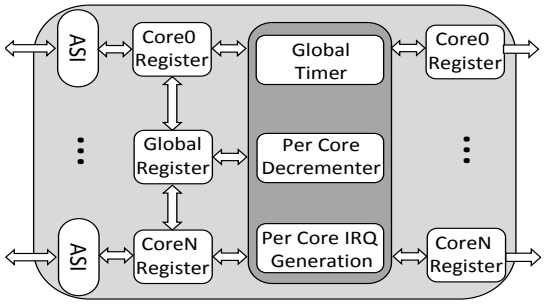


Figure 11: Share-clock Timer IP.

Fig. 11 shows the block diagram of the share-clock multi-port timer, in which each port is connected to one NIOS core by avalon slave interface (ASI). The share-clock multi-port timer provides each core with a dedicated 32-bit decrementer, which decrements based on the shared global timer. Here, the shared global timer expires every constant time (e.g., 1ms), which triggers each decrementer to decrement once. When one decrementer expires, an interrupt is generated to the corresponding core. Each core can dynamically control the period by setting its register, which triggers the task in different point. The global register is used to synchronize the cores to be launched at the same point. Only when all cores call the APIs to start timer, the global register is set to 1. Each core keeps waiting until this global register is active.

7. Task Profiling and Software Implementation

The aim of the task profiling is to identify the WCET and cache miss number with different cache size for a given task set. According to the system architecture shown in Fig. 5, the bus for accessing the off-chip memory is shared by all cores via the round-robin arbiter. This shared bus interference under the round-robin arbiter can be efficiently analyzed by techniques in [61] to estimate the WCET of a task. In this paper, we use measurement-based approach in [10] to estimate the WCET of a task. Regarding cache miss, we can obtain it from the customized performance counter by calling the related APIs in Tab. 1.

Table 1: APIs Supported by Reconfigurable Cache

allo_ways(way_num)	Allocate cache ways to cores
rel_ways(way_num)	Release cache ways from cores
clc_perf_cnt()	Clear the performance counter
get_hit_cnt()	Get the value of cache hit counter
get_miss_cnt()	Get the value of cache miss counter
get_state()	Return ways state, error state

Tab. 1 lists all the atomic APIs currently supported by reconfigurable cache IP. We refer to the implementation of time-triggered scheduler in [46] and implement the time-triggered scheduler with the share-clock multi-port timer on the NIOS-based multi-core system. To minimize the cache miss of the system, the synthesis approach in Section 5 can generate the task-level cache size configurations and time-triggered scheduler. According to the generated configurations, tasks can be scheduled with inserting cache configuration instructions (see Tab. 1) in each task invocation. High performance code can be generated by this approach.

8. Experimental Evaluations

In this section, we present the results obtained with an implementation of the proposed framework, as well as the performance of the proposed hardware platform. In our framework, the CPLEX [62] solver is used to solve the ILP problems for our synthesis approach. We implement the proposed time-triggered cache reconfigurable multi-core system on the Altera DE5 board equipped with Statrix V FPGA, which is based on the NIOS II multi-core architecture. In the multi-core architecture, we adopt the fast NIOS II core equipped with 512 bytes private L1 instruction cache and 512 bytes private L1 data cache. The private L1 cache module is provided by Altera and integrated in NIOS processor. All cores are shared with the unified L2 cache, which is an instance of the proposed reconfigurable cache IP. By cooperating with the proposed share-clock mutli-port timer, we implement the partitioned time-triggered scheduling on each core according to [46]. Time-triggered scheduling on each core is implemented in a bare-metal manner. The global tick of the shared clock timer is 1ms. To guarantee the predictability of the implementation of the scheduler, we reserve 1 fixed way for each core for the scheduler implementation (e.g., task switch).

Table 2: Benchmark sets for two-core system

	Core 1	Core 2
Set 1	Sobel, Fir	Histogram, Lms
Set 2	Fir2dim, Pbmsrch	Blackscholes, Fir
Set 3	Lms, FFT	Nsichneu, Sobel
Set 4	Lms, Histogram, FFT	Fir, Aes, Sobel
Set 5	Lms, Histogram Corner_turn,Pbmsrch	FFT, Sobel Nsichneu, Fir

To evaluate the effectiveness of our framework and hardware platform, we use 27 benchmark programs selected from MiBench [63] (Qsort, Dijkstra, Pbmsrch, FFT), CHStone [64] (Adpcm, Aes, Gsm, Sha, Mpeg2), DSP-stone [65] (Dot_product, Fir2dim, Fir, Biquad, Lms, Matrix, N_complex_update), PARSEC [66] (Blackscholes), UTDSP [67] (Histogram, Spectral, Lpc, Decode), Verabench [68] (Beamformer, Corner_turn), and some other research works [69, 70] (Sobel, Nsichneu, Qurt, Fdct). To avoid the selected tasks to saturate fast, we made some adaptations to the input scales of some benchmarks, such that they comply with the specified cache size. Tab. 2 and Tab. 3 respectively list the task sets used in our experiments for two-cores system and four-core system, which are combinations of the selected benchmarks. According to [25], we specify the task mappings based on the rule that the total execution time of each core is comparable.

8.1. Speed and Area Measurements

First of all, we compare the different types of caches with respect to their maximum operating frequency and area in terms of logic and memory usage. Different types of caches are synthesized on Altera Stratix V FPGA with Quartus II (version 13.0) to obtain area and critical path delay (maximum operating frequency F_{max}) numbers. The effect of increased cache depth, associativity, line size, and port number will be examined for all cache types. Tab. 4 summarizes the results for different types of caches. The 'cache settings' column is organized as form of *associativity/depth/line size*. For example, 4/128/256 indicates 4-ways cache architecture with 128 cache depth and 256-bit line size. F_{max} indicates the maximum frequency that the constructed multi-core system can run on.

Table 4: Speed and Area Measurements on Stratix V FPGA

Port Number	Cache Settings	Combinational ALUTs	Total Registers	F_{max} (MHz)
Two Core	4/256/256	11510	8899	168.41
	4/512/256	14453	11461	159.41
	8/256/256	17619	10506	151.10
	8/512/256	21609	14604	152.14
Four Core	8/256/256	29809	18683	140.29
	8/512/256	36074	24831	134.34
	16/256/256	39821	22014	126.90
	16/512/256	49225	31234	125.83

For increase in depth address and ways number, the number of combinational ALUTs and registers also increases. As explained in Section 6.5, to flush cache ways and reset the replacement reference in one cycle, we separate the valid bit of each line from memory block and implement it in customized memory block which supports clearing contents globally. Thus, the increment of address depth will result in the increment of the number of valid bit, which leads to more logic resource in combinational ALUTs and registers. Regarding the ways number, the contributing factors are the core-cache-switch circuitry, FIFO replacement policy circuitry, and wide logical OR, all of which grow with the increased ways number. Regarding the maximum operating frequency F_{max}, we notice that 2-core cache is faster than 4-core cache and the cache architecture with less associativity is faster than the one with more associativity.

8.2. Physical Chip Synthesis Results

In this section, we report physical chip synthesis results for the proposed dynamic partitioned cache memory. The proposed cache memory is implemented in synthesizable

Table 3: Benchmark sets for four-core system

	Core 1	Core 2	Core 3	Core 4
Set 1	Lms,FFT	Fir2dim,Pbmsrch	Matrix1,N_complex	Fir,Biquad
Set 2	Fir,Mpeg2 Histogram	Biquad Qurt	Lms,Gsm Qsort	Fdct,Sobel Dijkstra,Aes
Set 3	Matrix,FFT Spectral_estimation	Fir2dim Sobel	Biquad Decode	Beamformer Histogram
Set 4	Corner_turn Dotproduct	Fir Sha	Histogram Nsichneu	Nsichneu Lms
Set 5	Fdct, Lpc Fir2dim	Histogram,Sha Sobel,decode	FFT,Adpcm Corner_turn	Blackscholes Fir

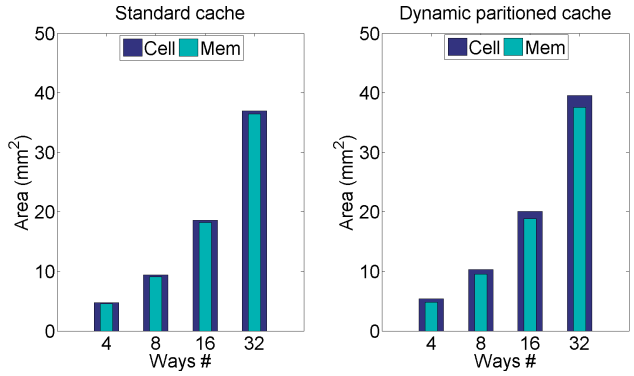


Figure 12: Chip area for dual-core caches with the varying cache way numbers.

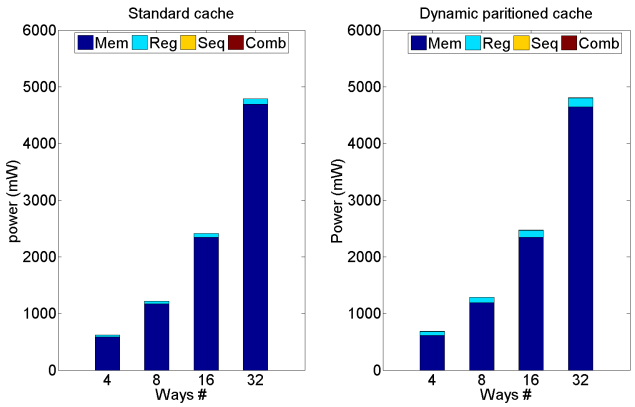


Figure 13: Power consumption for dual-core caches with the varying cache way numbers.

Verilog HDL code and synthesized by using Synopsys design compilers [32] with the SMIC 130nm standard technology library [33]. We use ARM Artisan 130nm memory IPs [71] to generate RAM blocks for our cache. Considering that the proposed cache memory supports way-based dynamic cache partitioning, we mainly focus on studying how the cache way numbers impact chip design process in terms of chip area and power consumption. We conduct the experiments to report the chip area and power consumption [2] of the proposed cache memory under different configurations. In the experiment, we implemented 4 different configurations for the dual-core caches memory, where the cache way numbers are varied from 4 to 32. The cache depths and cache lines are fixed as 1024 and 128, respectively. For comparison, a standard shared cache without dynamic partitioning functionality is also developed and verified by using the same experiment setups. Considering chip manufacturing technology we used (i.e., 130nm technology), we restrict the frequency of all cache designs at 400MHz and report the chip area and power consumption under this speed level.

Fig. 12 and Fig. 13 illustrate the chip area and power consumption for different types of caches, respectively. As shown in Fig. 12, the chip density is mainly contributed by the memory blocks in both cache architectures because the cache is mainly composed by the memory blocks. Comparing to the standard cache without dynamic cache partitioning, the total density overhead of our cache implementation ranges from 7% to 13% and maily comes from memory and combinational blocks. This density overhead is introduced with the addition of selection reference memory (SRM) in FIFO replacement policy circuitry and the routing logic in core-cache-switch circuitry. Another important observation is that the chip area is nearly increased linearly with the ways configurations. The cache with 32-ways configuration occupies 7X chip area than the cache with 4-ways configuration. Fig. 13 depicts the power consumption for both cache architectures under the different cache ways configurations. The main power overhead is caused by the increase of registers for cache controller. The power overhead of our cache design ranges from 0.3% to 10%. Thus, our cache design has a close power consumptions with respect to the standard cache design. Besides, the more cache ways we configure, the more power the cache memory will consume. From the results, we can see that reducing one more cache ways can on average reduce 148 mW power consumption. This means turning off cache ways can significantly reduce the power consumption of the system. This brings another potential research di-

[2] Considering that the workload of cache subsystem is application-specific and it is difficult to develop one specific test bench to obtain switching information of the cache component, we assume the switch percentage of the devices in our design is 100%.

rection about how to dynamically manage the cache ways resource to achieve energy efficiency for the cache subsystem.

8.3. Functionality Verification

We implemented a functional test to verify the correctness of the reconfigurable cache prototype implementation. This verification is based on memory reuse code, as shown in Fig. 14, which can mimic the behavior of cache access behavior. According to the test presented in Fig. 14, the program firstly access the array $b[Cache\_Depth * Ways\_Num][Line\_Size]$, whose size equals the predefined cache, in the first *for* loop. The parameter $Cache\_Depth$, $Ways\_Num$, and $Line\_Size$ are denoted as the cache depth, cache way number, and the word number of cache line, respectively. After the first loop, the assigned N-ways cache ($N < Ways\_Num$) will remain the last visted $N \times Cache\_Depth \times Line\_Size$ array data elments. For example, if we assign one cache way to this functional test program, this one-way assigned cache will be occupied by the array data elments from $b[Cache\_Depth * (Ways\_Num - 1)][0]$ to $b[Cache\_Depth * Ways\_Num - 1][Line\_Size - 1]$. In the second *while* loop, the array $b[Cache\_Depth * Ways\_Num][Line\_Size]$ is revisted in the reverse order for the sake of cache resue. The more the cache is assigned, the more cache resue can be achived which in turn can lead to less cache miss.

```
1  unsigned int b[Cache_Depth*Ways_Num][Line_Size];
2  unsigned int i,temp;
3  // Load data into cache
4  for(i=0;i<Cache_Depth*Ways_Num;i++){
5   temp=b[i][0];
6  }
7  //start to reuse cache
8  while(i>0){
9   temp=b[i][0];
10  i--;
11 }
```

Figure 14: The code for functionality verification.

This functional test is conducted on the two-core system with 2MB reconfigurable shared cache (8 ways, 8192 cache depth, 256 bit line size), which is implemented on the Altera DE5 development board equipped with Statrix V FPGA. The results as shown in Fig.15 are obtained by real measurements on FPGA implementation. By calling cache reconfiguration listed in Tab. 1, we implement memory reuse code under different cache ways. Fig. 15 shows cache miss numbers and execution times under different cache ways. We can see that both cache miss numbers and execution times predictably decrease linearly with reconfigured cache ways. By increasing one way, cache miss numbers decrease linearly with step 8192 (i.e., cache depth). This is expected since 8192 more cache lines are buffered for memory reuse when increasing one way.

Lets give a quantitative analysis to this result. According to the test in Fig. 14, each cache access in the first

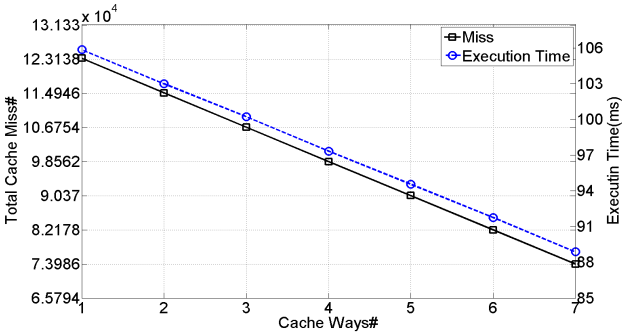


Figure 15: # Cache miss and execution time for memory reuse code.

for loop always result in cache miss. Thus, there should be 8192×8 cache misses to happen during the data load phase (i.e., the first *for* loop in Line 4-6). According to the analysis we state above, only $N \times 8192$ cache lines can be reused during the cache reuse phase (i.e., the second *while* loop in Line 8-11). Thus, we will get another $(8-N) \times 8192$ cache misses during the cache reuse phase. Totally, we are expected to get $(16 - N) \times 8192$ cache misses if we assign N cache ways to this test program. From this analysis, we can see the cache miss number should decrease linearly with reconfigured cache ways.

It is worthy noting that our cache works as a unified shared cache in the experiment setup. Instruction access will also result in additional cache miss numbers. Thus, the measured cache miss number is a rough number which do not take instruction access into account. To eliminate the impact of the cache miss caused by instruction access, we use the array with the large size (2M byte) in the test program and set our cache with the large size in this experiment to relieve the impact of instruction access and make our verification more accurate. By these settings, the cache miss caused by instruction access can be ignored comparing to the cache miss caused by data access. From the result as shown in Fig. 15, we can see that cache miss numbers are expected to decrease linearly with reconfigured cache ways. We also calculate the cache miss differences between the expectation and measurement, which is caused by instruction access. The maximum cache miss difference normalized with respect to the expected cache miss is up to 0.39%, which is very small. This means our cache works as the expected manner and the reconfiguration functionality of the designed cache is correct.

8.4. Timing Predictability

The purpose of this experiment is to verify how effective the proposed framework is in avoiding cache interference. In this experiment, we evaluate the system timing predictability on the two-core platform with 256KB cache (8 ways with 32KB size for each way, 256 bit line size). The specified multi-core platform is physically implemented on Altera DE5 development board equipped with Statrix V FPGA. All the results are collected by real measurements on FPGA implementation. We run 4 tasks on different

cores simultaneously (Pmbsrch and Lms are on core 1, while Sobel and Ncomplex are on core 2). For comparison, we also developed one single port standard shared cache without cache partitioning, which is shared by all core. For this cache architecture, the entire cache space is competitively used by all tasks. For our reconfigurable cache, the schedule and cache configuration are automatically generated by our synthesis approach to optimize the cache miss: 1 way for Pmbsrch, 7 ways for Lms, 7 ways for Sobel, 1 way for Ncomplex.

Fig. 16 shows the observed execution time and cache miss of each task invocations for the four tasks for two cache architectures. From the results, we can make the following observations: (1) All tasks on our proposed cache run in a stable manner and the execution time of all task do not exceed their WCETs that are estimated with cache space isolation. The execution time and cache miss of all tasks on our proposed cache are steady. It means that the timing of tasks on our proposed cache can be well predicted. As one comparison, we can see the execution time and cache miss of all tasks on standard shared cache vary significantly. Without cache isolation, tasks compete for the shared cache and useful cache lines for one task on one core may be evicted by one task on another core. This cache interference will result in poor timing predictability. (2) Because only one way is assigned to Pmbsrch and Ncomplex, we get a direct-mapped cache during the execution of Pmbsrch and Ncomplex. On standard shared cache, Pmbsrch and Ncomplex can still use the whole cache size although inter-core cache interferences exist, which may lead to less cache miss compared to direct-mapped cache. Note that, the system predictability is the prerequisite in real-time systems. Only when the system predictability is guaranteed, we can then consider how to improve performance. In this experiment, we aim to evaluate the system timing predictability. One interesting observation is that, even with smaller cache miss, the execution time of pmbsrch and ncomplex on standard shared cache is still greater than the one on our proposed cache. This may be caused by the fact that, all cores share standard cache via only one port, which will degrade the performance. In contrast, our proposed cache is a multi-port cache, which allows cores to access cache concurrently. Note that the scope of this experiments is to verify the proposed cache architecture can avoid the cache interference. The stability of task execution as shown in Fig. 16 has presented how the proposed cache architecture can avoid the cache interference to achieve the system predictability.

8.5. Runtime Performance

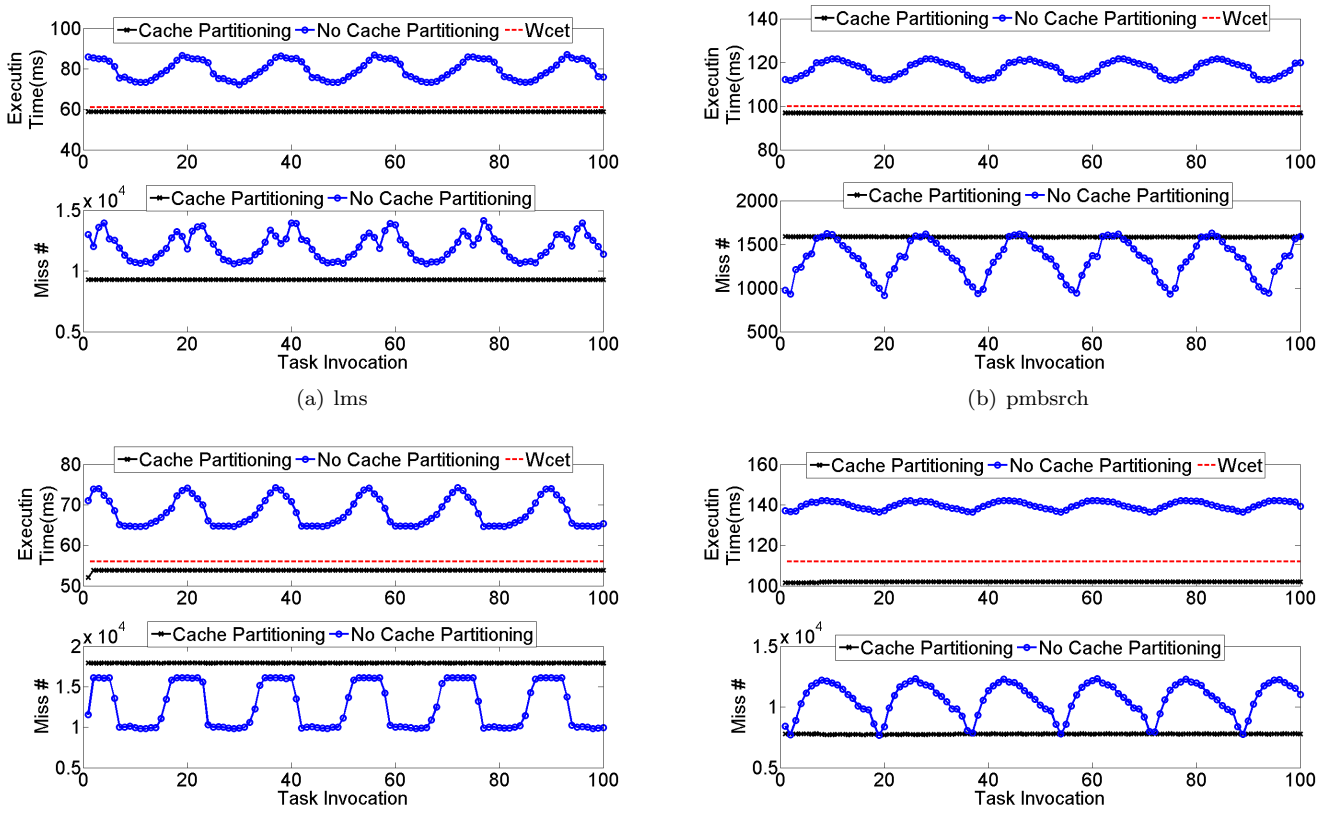
Then, we evaluate the effectiveness of the proposed automatic cache management framework under timing predictability requirement. In this experiment, we implement the cache management scheme and scheduling on two hardware platforms: two-core system with 256KB shared unified L2 cache (8 ways with 32KB size for each way, 256 bit line size) and four-core system with 256KB shared unified L2 cache (16 ways with 16KB size for each way, 256 bit line size). In the two hardware platforms, each NIOS core runs at 125Mhz. Tab. 2 and Tab. 3 list the task sets used in our experiments and the task mapping information for the two-core system and the four-core system, respectively. We physically implement two hardware platforms on FPGA and execute benchmark code on speciflied hardware platform. We compared the cache miss numbers with the following technique:

- `EQUAL`: Equal partitioning cache on cores.
- `CORE-OPT`: According to the cache reservation technique in the state-of-the-art work [10], a portion of cache partitions are statically reserved for each core to prevent inter-core cache interference. For fairness comparison, we integrate this cache reservation technique [10] into our framework to generate optimal cache reservations for each core.
- `TASK-OPT`: Our synthesis approach.

Fig. 17 shows the total cache miss number in one hyperperiod of the approaches normalized w.r.t `EQUAL`. All results are collected by implementing the cache management scheme and scheduling obtained from the corresponding approach on the proposed multi-core system. From the result measured by real hardware, we can see cache reservation technique (`CORE-OPT`) fails to improve system performance of most benchmark sets. This is because tasks assigned on the same core might have different requirements and sensitivity to the allocated cache amount, and a designed region with a constant size to individual cores cannot fully meet the features of the tasks. In contrast to the cache reservation technique (`CORE-OPT`), our synthesis approach (`TASK-OPT`) partitions the cache in task level and integrates cache partitioning globally with scheduling. We can observe that our synthesis approach (`TASK-OPT`) outperforms the cache reservation technique (`CORE-OPT`). Our approach (`TASK-OPT`) can on average reduce 14.93% (up to 22.03%) and 12.56% (up to 18.6%) cache miss with respect to `CORE-OPT` on 2-core and 4-core architectures, respectively.

8.6. Reconfiguration Overhead Measurement

Finally, we conduct experiments to measure the timing overhead for cache reconfiguration operations. According to Section 6.3, the port of cache ways management unit (CWMU) is shared by all cores. To inject traffic on the shared bus, we implement allocation and release cache configuration instructions in infinite loop concurrently on the interference core. To measure the timing overhead, allocation and release cache configuration instructions are implemented for 10000 times on the target core. In each iteration, we implement allocation-release cache configuration instruction pair to avoid the cache overflow. And we directly read the time stamp counter and report the average latency as the timing overhead of allocation-release



(a) lms

(b) pmbsrch

(c) ncomplex

(d) sobel

Figure 16: Cache partition and no cache partition.

instruction pair. According to our experiment, the average timing overhead of one allocation-release cache configuration instruction pair is 16 cycles, which is ignorable when comparing to OS-based cache partitioning.

9. Discusssion

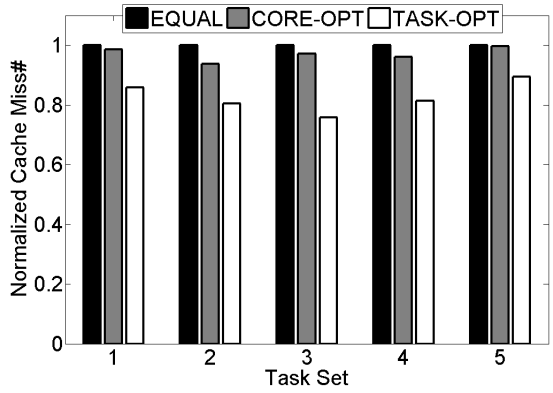
According to the state-of-the-art survey in [7, 3], how to manage the shared cache in a predictable and efficient manner under real-time requirements is still an open issue. As one of the uniqueness of our approaches, we provide not only a reconfigurable cache architecture, which enables us to use the shared cache in a predictable and efficient manner, but also one schedule-aware cache management scheme. Besides, we also provide a physical implementation on both of hardware and software to evaluate the usability of our approaches. In this section, we summarizes the features that is currently supported and also discuss the next steps for our approaches.

We propose a parameterized reconfigurable cache architecture, so called dynamic partitioned cache memory, for real-time multi-core system and physically implement it on FPGA. The dynamic partitioned cache memory is interfaced to Altera NIOS II based multi-core system. In principle, our cache can be implemented at any level of caches (L1 or L2) in the cache hierarchy. Due to the technology limitations stemmed from Altera NIOS II soft-core processor, we currently do not implement cache coherency protocol on the proposed cache. Besides, according to the state-of-the-art research work in [53], current cache coherence strategies are not suitable for the real-time system.

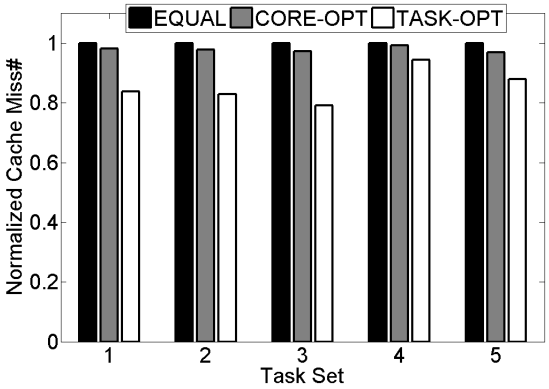
Another aspect for improvement is to enable write-back policy on the proposed dynamic partitioned cache memory. Currently, the proposed shared cache architecture is multi-port cache with using write-through policy, which allows NIOS cores to access the cache concurrently. By using write-through policy, the data in cache is always consistent to the off-chip memory. Thus, the cache ways can be released immediately and we can conduct the cache reconfiguration with the minimal timing overhead. However, if write-back policy is adopted, all dirty data in the released cache ways need to write back to off-chip memory during the cache reconfiguration phase. This will result in a significant timing overhead for the cache reconfiguration. Therefore, how to integrate write-back policy on the proposed dynamic partitioned cache memory would be one interesting future work.

10. Conclusion

In this paper, we present a reconfigurable cache architecture which supports dynamic cache partitioning at hard-



(a) # Cache Miss on Two-core System



(b) # Cache Miss on Four-core System

Figure 17: # Cache Miss Reduction on Different Hardware Platform.

ware level and a framework that can exploit cache management for real-time MPSoCs. By using the proposed cache, the cache resource can be strictly isolated to prevent the cache interference among cores. Furthermore, the proposed cache supports dynamic cache partitioning and allows cores to dynamically allocate cache resource according to the demand of applications, which will enable us to efficiently use cache resources. In contrast to most existing work [27, 28, 29, 30, 31, 25] in the literature, which is devoted to analyze theoretical proposals and the simulation of reconfigurable caches, the proposed cache is physically implemented and prototyped on FPGA. This prototype will bridge the gap between simulation and real systems, and will serve us a real (not simulation) reconfigurable cache for studying and validating cache management strategies on the real-time multi-core system under different cache settings. The proposed framework optimally integrates time-triggered scheduling and dynamic cache partitioning such that the shared cache can be used in a predictable and efficient manner. Experimental results in the FPGA using a diverse set of applications and different number of cores and cache modules demonstrate the effectiveness of the proposed framework.

References

[1] ARM Cortex-A15 series, `http://www.arm.com/products`.

[2] OpenSPARC, `http://www.opensparc.net/`.

[3] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, M. Prieto, Survey of scheduling techniques for addressing shared resources in multicore processors, ACM Computing Surveys (2012) 4:1–4:28.

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

[5] R. Iyer, Cqos: A framework for enabling qos in shared caches of cmp platforms, in: Proceedings of the 18th Annual International Conference on Supercomputing, 2004.

[6] B. Ward, J. Herman, C. Kenna, J. Anderson, Making shared caches more predictable on multicore platforms, in: Proceedings of 2013 25th Euromicro Conference on Real-Time Systems (ECRTS), 2013.

[7] A. Abel, F. Benz, J. Doerfert, B. Drr, S. Hahn, F. Haupenthal, M. Jacobs, A. Moin, J. Reineke, B. Schommer, R. Wilhelm, Impact of resource sharing on performance and performance prediction: A survey, in: Proceedings of 24th Conference on Concurrency Theory (CONCUR), 2013.

[8] N. Guan, M. Stigge, W. Yi, G. Yu, Cache-aware scheduling and analysis for multicores, in: Proceedings of 2009 ACM International Conference on Embedded Software (EMSOFT), 2009.

[9] S. Fisher, Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4, White paper, SYSGO AG (2014).

[10] H. Kim, A. Kandhalu, R. Rajkumar, A coordinated approach for practical os-level cache management in multi-core real-time systems, in: Proceedings of 2013 25th Euromicro Conference on Real-Time Systems (ECRTS), 2013.

[11] G. Chen, K. Huang, J. Huang, A. Knoll, Cache partitioning and scheduling for energy optimization of real-time mpsocs, in: Proceedings of 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2013.

[12] G. Chen, B. Hu, K. Huang, A. Knoll, K. Huang, D. Liu, Shared l2 cache management in multicore real-time system, in: Proceedings of 22nd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, P. Sadayappan, Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems, in: Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture (HPCA), 2008.

[14] S. Cho, L. Jin, Managing distributed, shared l2 caches through os-level page allocation, in: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2006.

[15] B. N. Bershad, D. Lee, T. H. Romer, J. B. Chen, Avoiding conflict misses dynamically in large direct-mapped caches, ACM SIGOPS Operating Systems Review (1994) 158–170.

[16] W. Jing, R. Fan, The research of hibernate cache technique and application of ehcache component, in: Proceedings of 2011 IEEE 3rd International Conference on Communication Software and Networks (ICCSN), 2011, pp. 160–162.

[17] L. Zhang, E. Speight, R. Rajamony, J. Lin, Enigma: Architectural and operating system support for reducing the impact of address translation, in: Proceedings of 2010 24th ACM International Conference on Supercomputing (ICS), 2010.

[18] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, R. Pellizzoni, Real-time cache management framework for multi-core architectures, in: Proceedings of 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[19] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, R. Rajkumar, Coordinated bank and cache coloring for temporal protection of memory accesses, in: Proceedings

of 2013 IEEE 16th International Conference on Computational Science and Engineering (ICESS), 2013.
[20] H. Cook, M. Moreto, S. Bird, K. N. Dao, D. Patterson, K. Asanovic, A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness, in: Proceedings of 40th ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013.
[21] A. Gil, J. Benitez, M. Calvino, E. Gomez, Reconfigurable cache implemented on an fpga, in: Proceedings of 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2010, pp. 250–255.
[22] A. Santana Gil, F. Quiles Latorre, M. Hernandez Calvino, E. Herruzo Gomez, J. Benavides Benitez, Optimizing the physical implementation of a reconfigurable cache, in: Proceedings of 2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2012, pp. 1–6.
[23] A. Malik, B. Moyer, D. Cermak, A low power unified cache architecture providing power and performance flexibility, in: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED), 2000, pp. 241–243.
[24] C. Zhang, F. Vahid, W. Najjar, A highly configurable cache for low energy embedded systems, ACM Transactions on Embedded Computing Systems (2005) 363–387.
[25] W. Wang, P. Mishra, S. Ranka, Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems, in: Proceedings of 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 2011.
[26] G. Chen, B. Hu, K. Huang, A. Knoll, D. Liu, T. Stefanov, Automatic cache partitioning and time-triggered scheduling for real-time mpsocs, in: Proceedings of the 2014 9th International Conference on Reconfigurable Computing and FPGAs (ReConfig), 2014.
[27] D. Albonesi, Selective cache ways: on-demand cache resource allocation, in: Proceedings of 1999 32nd Annual International Symposium on Microarchitecture (MICRO), 1999, pp. 248–259.
[28] G. E. Suh, L. Rudolph, S. Devadas, Dynamic partitioning of shared cache memory, Journal of Supercomputing 28 (1) (2004) 7–26.
[29] D. Benitez, J. Moure, D. Rexachs, E. Luque, A reconfigurable cache memory with heterogeneous banks, in: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2010, pp. 825–830.
[30] K. T. Sundararajan, T. M. Jones, N. Topham, A reconfigurable cache architecture for energy efficiency, in: Proceedings of the 8th ACM International Conference on Computing Frontiers (CF), 2011.
[31] S. Mittal, Z. Zhang, J. Vetter, Flexiway: A cache energy saving technique using fine-grained cache reconfiguration, in: Proceedings of 2013 IEEE 31st International Conference on Computer Design (ICCD), 2013.
[32] Synopsys Design Compilers, `http://www.synopsys.com`.
[33] Semiconductor Manufacturing International Corporation, `http://www.smics.com`.
[34] M. Qureshi, et al., Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches, in: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2006.
[35] D. Sanchez, et al., Vantage: Scalable and efficient fine-grain cache partitioning, in: Proceedings of 2011 38th Annual International Symposium on Computer Architecture (ISCA), 2011.
[36] A. Wolfe, Software-based cache partitioning for real-time applications, Journal of Computer and Software Engineering (1994) 315–327.
[37] F. Mueller, Compiler support for software-based cache partitioning, in: Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, 1995.
[38] M. Lukasiewycz, S. Steinhorst, F. Sagstetter, W. Chang, P. Waszecki, M. Kauer, S. Chakraborty, Cyber-physical systems design for electric vehicles, in: Proceedings of 2012 Euromicro Conference on Digital System Design (DSD), 2012.
[39] C. Lin, H.-M. Yen, Y.-S. Lin, Development of time triggered hybrid data bus system for small aircraft digital avionic system, in: Proceedings of IEEE/AIAA 26th Digital Avionics Systems Conference (DASC), 2007.
[40] S. Baruah, G. Fohler, Certification-cognizant time-triggered scheduling of mixed-criticality systems, in: Proceedings of 2011 IEEE 32nd Real-Time Systems Symposium (RTSS), 2011.
[41] F. Sagstetter, M. Lukasiewycz, S. Chakraborty, Schedule integration for time-triggered systems, in: Proceedings of 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), 2013.
[42] D. Goswami, M. Lukasiewycz, R. Schneider, S. Chakraborty, Time-triggered implementations of mixed-criticality automotive software, in: Proceedings of the 15th Conference for Design, Automation and Test in Europe (DATE), 2012.
[43] T. Nghiem, G. J. Pappas, R. Alur, A. Girard, Time-triggered implementations of dynamic controllers, ACM Transactions on Embedded Computing Systems (TECS) (2012) 58:1–58:24.
[44] T. Nghiem, G. J. Pappas, R. Alur, A. Girard, Time-triggered implementations of dynamic controllers, in: Proceedings of the 6th ACM/IEEE International Conference on Embedded Software (EMSOFT), 2006.
[45] J. Huang, J. Blech, A. Raabe, C. Buckl, A. Knoll, Static scheduling of a time-triggered network-on-chip based on smt solving, in: Proceedings of the 15th Design, Automation Test in Europe Conference Exhibition (DATE), 2012.
[46] A. Gendy, M. Pont, Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems, IEEE Transactions on Industrial Informatics (2008) 37–46.
[47] N. Guan, W. Yi, Z. Gu, Q. Deng, G. Yu, New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms, in: Proceedings of 2008 Real-Time Systems Symposium (RTSS), 2008.
[48] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) (1973) 46–61.
[49] I. Lee, J. Y.-T. Leung, S. H. Son, Handbook of Real-Time and Embedded Systems, CRC Press, 2007.
[50] S. Baruah, M. Bertogna, G. Buttazzo, Multiprocessor Scheduling for Real-Time Systems, Springer, 2015.
[51] J. Lee, K. G. Shin, I. Shin, A. Easwaran, Composition of schedulability analyses for real-time multiprocessor systems, IEEE Transactions on Computers (TC) (2015) 941–954.
[52] Creating multiprocessor nios systems tutorial, `http://www.altera.com`.
[53] A. Pyka, M. Rohde, S. Uhrig, A real-time capable first-level cache for multi-cores, in: Proceedings of 2013 1st Workshop on High-performance and Real-time Embedded Systems (HiRES), 2013.
[54] Adapteva Parallella, `http://www.adapteva.com/parallella/`.
[55] J. Dai, L. Wang, An energy-efficient l2 cache architecture using way tag information under write-through policy, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 21 (1) (2013) 102–112.
[56] P. Kongetira, K. Aingaran, K. Olukotun, Niagara: a 32-way multithreaded sparc processor, IEEE Micro 25 (2) (2005) 21–29.
[57] D. H. Jim Mitchell, G. Ahrens, Ibm power5 processor-based servers: A highly available design for business-critical applications, White paper, IBM (2005).
[58] N. Quach, High availability and reliability in the itanium processor, IEEE Micro 20 (5) (2000) 61–69.
[59] N. Guan, X. Yang, M. Lv, W. Yi, Fifo cache analysis for wcet estimation: A quantitative approach, in: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2013.
[60] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, M. Valero, Hardware support for wcet analysis of hard real-time multicore systems, in: Proceedings of 2009 36th Annual International Symposium on Computer Architecture (ISCA), 2009.

[61] H. Shah, K. Huang, A. Knoll, Weighted execution time analysis of applications on cots multi-core architectures, Tech. Rep. TUM-I1339 (2013).
[62] IBM ILOG CPLEX, http://www.ibm.com/software/.
[63] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Proceedings of 2001 IEEE International Workshop on Workload Characterization (WWC), 2001.
[64] CHStone, http://www.ertl.jp/chstone/.
[65] Dspstone, http://www.ice.rwth-aachen.de/.
[66] C. Bienia, Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (2011).
[67] UTDSP, http://www.eecg.toronto.edu/UTDSP.html/.
[68] Versabench, http://groups.csail.mit.edu/versabench.
[69] H. Nikolov, et al., Systematic and automated multiprocessor system design, programming, and implementation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2008) 542–555.
[70] Malardalen real-time research center, http://www.es.mdh.se/.
[71] ARM Artisan Physical IP Solutions, http://www.artisan.com.