Automatic Cache Partitioning and Time-triggered Scheduling for Real-time MPSoCs

Gang Chen[1], Biao Hu[1], Kai Huang[1,2], Alois Knoll[1], Kai Huang[3], Di Liu[4], Todor Stefanov[4]
[1]Tech. Univ. Muenchen TUM, [2]Sun Yat-sen University, [3]Zhejiang University, [4]Leiden University
{cheng,hub,huangk,knoll}@in.tum.de, huangk@vlsi.zju.edu.cn, {d.liu,t.p.stefanov}@liacs.leidenuniv.nl

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 still an open issue. In this paper, we present a framework that can exploit cache management for real-time MPSoCs. The framework supports dynamic way-based cache partitioning at hardware level, building task-level time-triggered reconfigurable-cache MPSoCs. It automatically determines time-triggered schedule and cache configuration for each task to improve the system performance while guarantee 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. Experiment results based on FPGA implementation demonstrate the effectiveness of the proposed framework over the state-of-the-art cache management strategies when tested 27 benchmark programs on the constructed MPSoCs.**

I. Introduction

Computing systems are increasingly moving towards multi-core platforms. To alleviate the high latency of the off-chip memory, multi-processor system-on-chip (MPSoC) architectures are typically equipped with hierarchical cache subsystems. For instance, ARM Cortex-A15 series [4] use small L1 caches for individual cores and a relatively large L2 cache shared among different cores. Due to this inherent complex cache hierarchy, the analysis of shared cache subsystem has received much attention [14], [17], [31], in recent years.

The main problem of cache hierarchy is that the behavior of shared cache is hard to predict and analyze statically [1], [14] in MPSoCs. For instance, a task running on one core may evict useful L2 cache space, which is used by another task in another core. These inter-core cache interferences will cause an increase in the miss rate [34], leading to a corresponding decrease in performance. In addition, inter-core cache interferences are extremely difficult to analyze accurately [14], thus resulting in difficulty of estimating the worst-case execution time (WCET) of the application program. Therefore, how to tackle the shared cache in the context of real-time systems is still an open issue [1], [34] and the difficulty actually prohibits an efficient use of the MPSoCs for real-time systems. For instance, to resolve the predictability problem for MPSoCs, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems [31]. The work in [17] also report 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. Being aware of this, this paper studies the problem of how to use the shared cache in a predictable and efficient manner under real-time requirements with the existence of cache interference.

To address this problem, most of the state-of-the-art techniques [17], [27], [31] on the multi-core cache management for real-time systems use page-coloring, i.e., a software cache partitioning approach in the OS level, to partition the cache by sets. The 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 [18], on the other hand makes color changes of tasks whose execution time is less than the page-change overhead not worthy. To tackle these problems, we consider task-level schedule-aware cache partitioning and implement cache partitioning in our customized reconfigurable cache hardware component with minimal timing overhead.

Combining real-time task scheduling and cache size 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 of tasks may have different sensitivity to the assigned cache size. In principle, the task scheduling and the cache size allocation interrelate to each other with respect to the system performance, such as cache misses and energy consumption [30]. Therefore, a sophisticated framework is needed to find the best trade-off between them in order to improve the system performance.

This paper tackles schedule-aware cache managment scheme for real-time MPSoCs. We present an integrated framework to exploit and verify the interactions between the task scheduling and the shared L2 cache interference. For a given set of tasks and a mapping of the tasks on an MPSoC, our approach 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 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 overflows. With a customized reconfigurable cache component and share-clock multi-port timer component, our framework can generate MPSoCs with different numbers of cores and different cache modules (different cache configurations with respect to cache lines, size, and associativity) and prototype on Altera FPGA. The contributions of our work are as follows:

- We proposed 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 system performance under real-time constraints.
- We developed a parameterized reconfigurable cache

This work has been partly funded by German BMBF projects ECU (grant number: 13N11936) and Car2X (grant number: 13N11933).

memory and prototyped it on FPGA. 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 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.

- We developed a share-clock multi-port timer component that enables the time-triggered schedule to be implemented on the MPSoCs generated from our framework.
- We prototyped and evaluated the generated MPSoCs on Altera Statrix V FPGA using 27 real-time benchmarks. We also analyze and discuss the experiment results under different hardware environment with respect to the number of cores and cache settings.

II. Related Work

Real-Time Cache Partitioning: Shared cache interference in a multi-core system has been recognized as one of major factors that degrade the average performance [18], as well as predictability of a system [31], [14]. Many works have been done in general-purpose computing to optimize different performance objectives by cleverly partitioning shared cache, including cache performance [23], [24] and energy consumption [33]. In the context of real-time systems, cache partitioning technique have been explored mostly by using software-based solution [32], [20], [31], [17], [27]. In [32], [20], 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's 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 [31], [17], [27] 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 [18] report 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 [3], [33]. Therefore, our way-based reconfigurable cache can also bring benefits for low-power design.

Reconfigurable Cache: Numbers of reconfigurable cache architectures have been proposed in the literature. Most of them [3], [26], [8] are devoted to the analysis of theoretical proposals and the simulation of reconfigurable caches, only a few are devoted to the physical implementation of the proposed cache models. Zhang et al. [33] proposed a reconfigurable cache architecture where the cache ways configuration could be tuned via the combination of configuration register and physical address bits. 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. In addition, the number of 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. [12] presented one general-purpose reconfigurable cache design *only* for uni-processor systems, which was implemented on FPGA. It cannot be easily extended to multi-core system. In this paper, we propose a parametrized 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 the cache resource is strictly isolated among real-time applications to prevent the cache interference. This 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.

III. Background

A. Way-based Cache Partitioning

Our cache management scheme implements dynamic way-based cache partitioning on FPGA. As shown in Fig. 1, 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.

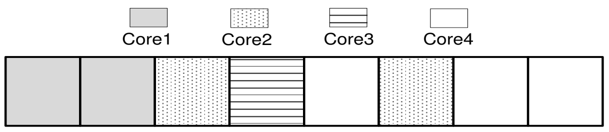


Fig. 1. Way-based Cache Partitioning.

B. 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 the worst case execution time. 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 [13]. 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. The deadline d_i of the task T_i is equal to its period h_i.

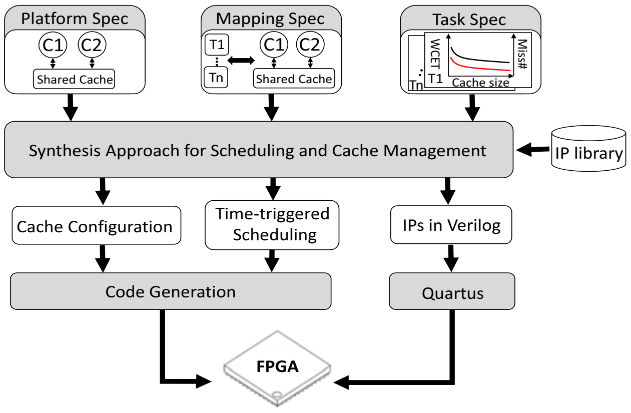


Fig. 2. System Design Framework.

IV. Framework Overview

In this section, we give an overview of our system design framework depicted in Fig. 2, 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. 2, the input specifications of the proposed framework consists 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, i.e., period and deadline, 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 VII.

As output, the synthesis approach can generate cache size allocation and time-triggered scheduling for each task according to the *input specification*, 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*.

V. Synthesis Approach for Scheduling and Cache Management

This section presents the synthesis approach for timing schedule and cache management. We reuse the approach in [6] 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.

A. 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: $z_{p\widetilde{p}}^{ij} = 1$ if the i-th instance of task T_p finishes before the start of j-th instance of $T_{\widetilde{p}}$, and 0 otherwise. H_r and $H_{p\widetilde{p}}$ denote the hyper-period of all tasks and the hyper-period of only task T_p and $T_{\widetilde{p}}$ (i.e., LCM of periods of T_p and $T_{\widetilde{p}}$), 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. $\forall T_p, T_{\widetilde{p}} \in TS(T_p), i = 0, ..., (\frac{H_{p\widetilde{p}}}{h_p} - 1), j = 0, ..., (\frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} - 1)$:

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

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

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

B. 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 [6], 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.

According to features of time-triggered scheduling, 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:

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

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.
$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

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

The term of $PSWF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{u} c_{\widetilde{p}k} \cdot k$ represents cache requirements of the task $T_{\widetilde{p}}$ 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 [6]. 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^{ij}_{cache} represent the cache capacity (in the number of ways) and the cache miss of task T_i under j-way cache configuration, respectively.

VI. Proposed Hardware Infrastructure

In this section, we present the 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. 3 illustrates the proposed multi-core system on FPGA, where the L2 cache is shared among cores. We adopt the NIOS II fast core in the system. Modules highlighted with white color in Fig. 3 indicate the hardware components specifically designed and implemented for our framework. The system consists of several NIOS II cores with private L1 cache (both instruction and data cache), along with reconfigurable cache IP which supports dynamic cache partitioning and share-tick timer IP for the time-triggered scheduling.

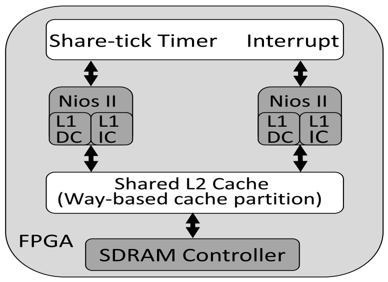


Fig. 3. System Architecture.

A. 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 [22], 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 [22]. 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 [22]. In this paper, we mainly focus on studying the interaction between scheduling and cache management, 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 [2]. Note that inter-core cache interference still exists although software on each core runs in different regions of the same off-chip memory. 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. 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 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 constraints. This will result in that the cache ways occupied by one core might not be adjacent to each other. As shown in Fig. 1, 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.

B. 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. 4, 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*

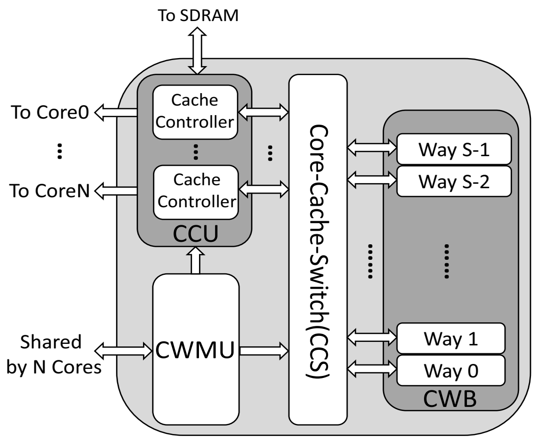


Fig. 4. Reconfigurable Cache Architecture.

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*. *Cache ways blocks (CWB)* are memory blocks used for tag and data storage.

C. Cache Ways Management Unit (CWMU)

The *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. 5, 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. I). 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 not enough ways exist in the pool, *cache overflow* error will be returned to the distinguished core. Note that our synthesis approach depicted in Section V 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.

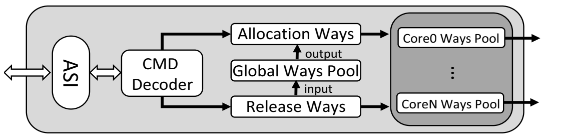


Fig. 5. Cache Ways Managment Unit (CWMU).

D. 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 a write-through cache owing to its simplicity. Fig. 6 depicts the block diagram of *cache controller*. Transactions from L1 cache of NIOS core are injected through L2 cache port, 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 L1 and L2 cache line size, respectively. Thus, muxs and demuxs in *ASI* and *AMI* are used to packet and de-packet 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 VI-E.

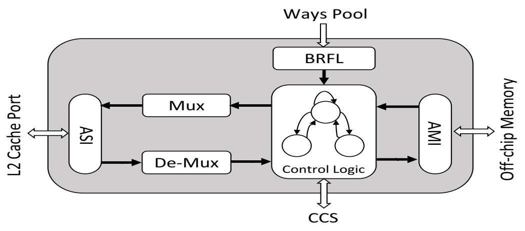


Fig. 6. Cache Controller (CC).

E. 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 compared to LRU replacement policy [15], we consider FIFO cache replacement policy in this paper. In addition, the FIFO replacement policy has been widely used in ARM 11 processor and Intel X86 processor [15].

As mentioned in Section VI-A, 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), shown in Fig. 7, 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 one clock. Unfortunately, no FPGA can support this feature. In this paper, we propose one solution 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 the toggle to determine 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 into SRM and VBM, making that ways are selected in FIFO replacement manner. Note that write control logic (WCL) can also be easily extended for other replacement policies, e.g., LRU. BRFL outputs a valid reference and the victim can be referred from the ways pool.

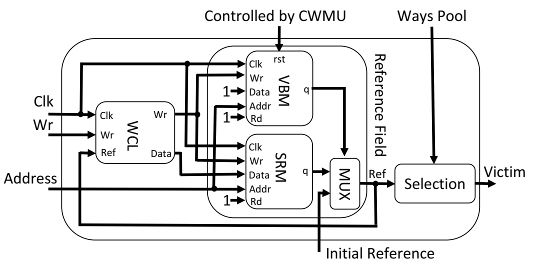


Fig. 7. Block Reference Field Logic (BRFL).

F. 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.

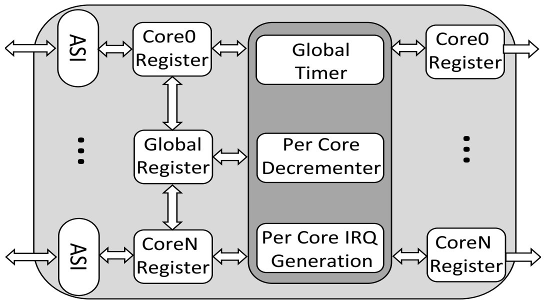


Fig. 8. Share-clock Timer IP.

Fig. 8 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.

VII. Task Profiling and Software Implementation

The aim of the task profiling is to identify the worst-case execution time (WCET) and cache miss number with different cache size for a given task set. According to the system architecture shown in Fig. 3, 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 [25] to estimate the WCET of a task. In this paper, we use measurement-based approach in [17] 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. I.

TABLE I
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. I lists all the atomic APIs currently supported by reconfigurable cache IP. We refer to the implementation of time-triggered scheduler in [11] 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 V 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. I) in each task invocation. High performance code can be generated by this approach.

VIII. 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 [9] solver is used to solve the ILP problems for our synthesis approach. We set the overhead of the task switch to 0.1ms by experiments, which is big enough to execute the task switch.

A. Experimental Setup

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. 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 [11]. 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).

To evaluate the effectiveness of our framework and hardware platform, we use 27 benchmark programs selected from MiBench [16] (Qsort, Dijkstra, Pbmsrch, FFT), CHStone [7] (Adpcm, Aes, Gsm, Sha, Mpeg2), DSPstone [10] (Dot_product, Fir2dim, Fir, Biquad, Lms,

TABLE III
BENCHMARK SETS FOR FOUR-CORE SYSTEM

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

TABLE II
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

Matrix, N_complex_update), PARSEC [5] (Blackscholes), UTDSP [28] (Histogram, Spectral, Lpc, Decode), Verabench [29] (Beamformer, Corner_turn) and some other research works [21], [19] (Sobel,Nsichneu,Qurt,Fdct). The input scales of some benchmarks used in this study are too small to be memory-intensive tasks for the specified cache size. To avoid the selected task to saturate fast, we made some adaptations to the input scales of some benchmarks, such that they comply with the specified cache size. Tab. II and Tab. III 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 [30], we specify the task mappings based on the rule that the total execution time of each core is comparable.

B. 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 to 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. IV 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 IV
SPEED AND AREA MEASUREMENTS ON STRATIX V FPGA

Port Number	Cache Settings	Combinational ALUTs	Total Registers	$\mathbf{F_{max}}$ (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 VI-E, 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.

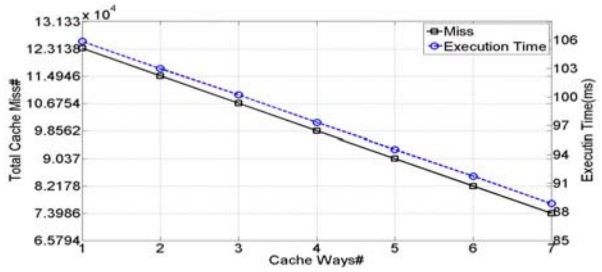


Fig. 9. # Cache Miss and Execution Time for memory reuse code.

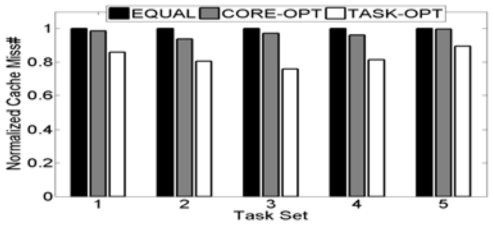
Listing 1. The Code for Functionality Verification
```
unsigned int b[Cache_Depth*Ways_Num][Line_Size];
unsigned int i,temp;
// Load data into cache
for(i=0;i<Cache_Depth*Ways_Num;i++){
 temp=b[i][0];
}
//start to reuse cache
while(i>0){
 temp=b[i][0];
 i--;
}
```

C. Functionality Verification

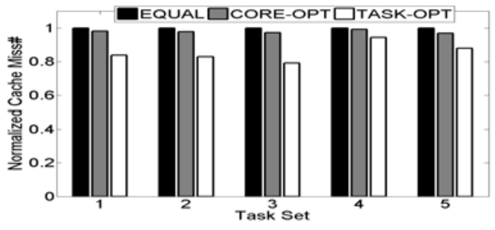
We implemented a functional test to verify the correctness of the reconfigurable cache prototype implementation. This verification is based on memory reuse code, as shown in Listing 1, which can mimic the behavior of L2 cache. This functional test is conducted on the two-core system with 2MB reconfigurable shared L2 cache (8 ways, 8192 cache depth, 256 bit line size). By calling cache reconfiguration listed in Tab. I, we implement memory reuse code under different cache ways. Fig. 9 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.

D. Runtime Performance

Finally, 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)



(a) # Cache Miss on Two-core System



(b) # Cache Miss on Four-core System

Fig. 10. # Cache Miss Reduction on Different Hardware Platform.

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. II and Tab. III 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 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 [17], 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 [17] into our framework to generate optimal cache reservations for each core.
- `TASK-OPT`: Our synthesis approach.

Fig. 10 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.

IX. Conclusion

This paper presents a cache management framework for real-time MPSoCs. The framework optimally integrates time-triggered scheduling and cache partitioning such that the shared cache can be used in a predictable and efficient manner. In contrast to software-based cache partitioning techniques in the literature, we conduct cache partitioning at hardware level and prototyped an implementation on FPGA. Experimental results in the FPGA using a diverse set of applications demonstrate the effectiveness of the proposed framework. For the next step, we are interested in integrating cache coherency protocol in our cache architecture. Furthermore, another interesting future work would be to extend the proposed cache to support mixed-critical real-time system.

References

[1] A. Abel et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR 2013- Concurrency Theory*. 2013.
[2] Adapteva Parallella. `http://www.adapteva.com/parallella/`.
[3] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO*, 1999.
[4] ARM Cortex-A15 series. `http://www.arm.com/products`.
[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.
[6] G. Chen et al. Cache partitioning and scheduling for energy optimization of real-time mpsocs. In *ASAP*, 2013.
[7] CHStone. `http://www.ertl.jp/chstone/`.
[8] L. Coutinho et al. Dynamically reconfigurable split cache architecture. In *ReConFig*, 2008.
[9] IBM ILOG CPLEX. `http://www.ibm.com/software/`.
[10] Dspstone. `http://www.ice.rwth-aachen.de/`.
[11] A. Gendy et al. Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems. *IEEE Transactions on Industrial Informatics*, 2008.
[12] A. Gil et al. Reconfigurable cache implemented on an fpga. In *ReConFig*, 2010.
[13] N. Guan et al. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *RTSS*, 2008.
[14] N. Guan et al. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, 2009.
[15] N. Guan et al. Fifo cache analysis for wcet estimation: A quantitative approach. In *DATE*, 2013.
[16] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.
[17] H. Kim et al. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *ECRTS*, 2013.
[18] J. Lin et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.
[19] Malardalen real-time research center. `http://www.es.mdh.se/`.
[20] F. Mueller. Compiler support for software-based cache partitioning. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1995.
[21] 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.
[22] Creating multiprocessor nios systems tutorial. `http://www.altera.com`.
[23] M. Qureshi et al. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.
[24] D. Sanchez et al. Vantage: Scalable and efficient fine-grain cache partitioning. In *ISCA*, 2011.
[25] H. Shah et al. Weighted execution time analysis of applications on cots multi-core architectures. Technical Report TUM-I1339, 2013.
[26] G. E. Suh et al. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 2004.
[27] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.
[28] UTDSP. `http://www.eecg.toronto.edu/UTDSP.html`.
[29] Versabench. `http://groups.csail.mit.edu/versabench/`.
[30] W. Wang et al. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *DAC*, 2011.
[31] B. Ward et al. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.
[32] A. Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, 1994.
[33] C. Zhang et al. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computing Systems*, 2005.
[34] S. Zhuravlev et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 2012.