TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Cache Management and Time-triggered Scheduling for Hard Real-time MPSoCs

Gang Chen, Biao Hu, Kai Huang, Alois Knoll

TUM-I148

Cache Management and Time-triggered Scheduling for Hard Real-time MPSoCs

Gang Chen
TU Munich, Germany
cheng@in.tum.de

Biao Hu
TU Munich, Germany
hub@in.tum.de

Kai Huang
TU Munich, Germany
huangk@in.tum.de

Alois Knoll
TU Munich, Germany
knoll@in.tum.de

***Abstract*—In hard real-time MPSoCs, shared cache has been considered as one of the major factors that degrade system predictability. How to manage the shared cache in order to optimize the system performance while guaranteeing the system predictability is still an open issue. State-of-the-art techniques on this topic use page coloring to partition the shared cache at OS level. In this paper, we present a cache management framework for hard real-time MPSoCs. The framework supports way-based cache partitioning at hardware level, building task-level time-triggered reconfigurable-cache MPSoCs. We evaluated the proposed framework w.r.t. different numbers of cores and cache modules and prototyped the constructed MPSoCs on FPGA. Experiment results based on FPGA measurements demonstrate the effectiveness of the proposed framework.**

I. Introduction

With increasing requirement for high-performance, multi-processor system-on-chip(MPSoC) architectures are believed to be the major solution for future embedded systems. To alleviate the high latency of the off-chip memory, MPSoC architectures are typically equipped with hierarchical cache subsystems. For instance, ARM Cortex-A15 series [2] and openSPARC series [27] all 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], [19], [11], [29], in recent years.

The main problem of cache hierarchy is that the behavior of shared cache is hard to predict and analyze statically [11], [1] 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 introduce significant worst-case timing penalties, 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 hard real-time systems is still an open issue [1], [30] and the difficulty actually prohibits an efficient use of the MPSoCs for hard real-time systems. For instance, to resolve the predictability problem for an MPSoCs, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems [29]. Being aware of this problem, this paper studies the problem of how to use the shared cache in a predictable and efficient manner under hard real-time requirements with the existence of cache interference.

To address this problem, most of the state-of-the-art techniques [29], [14], [19] on the multicore cache management for real-time systems use page coloring, i.e., a software 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 computing the color of a page. This timing overhead on the one hand prohibits a frequent change of the colors of pages [7], [16], 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, i.e., a time-triggered scheduling at application task level and a reconfigurable cache based on the schedule. With our customized reconfigurable cache component, the L2 cache is dynamically partitioned and allocated to application tasks according to the runtime schedule 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 is depended on the allocated cache size. On the other hand, the maximal cache budget that can be assigned to a task is depended on the cache sizes occupied by other tasks that are currently running on the other cores, i.e., depending on the scheduler. Besides, the performance of running tasks may have different requirements and may be sensitive to the amount of used cache. In principle, the task scheduling and the allocation of the size of the cache interrelate to each other with respect to the system performance, such as numbers of cache misses [7] and energy consumption [28]. Therefore, a sophisticated framework is needed to find the best trade-off between them in order to improve the system performance [28].

This paper tackles schedule-aware cache managment scheme for hard real-time MPSoCs. We present an integrated framework to study 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 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 of all tasks. With a customized reconfigurable cache component

and share-clock mutli-port timer component, our framework can generate MPSoCs with different numbers of cores and different cache modules (different cache configurations w.r.t. cache lines, size, associativity) and prototype on Altera FPGA. The contributions of our work are following:

- We proposed an integrated cache management framework that improves the execution predictability for hard real-time MPSoCs. The proposed framework can generate fully deterministic time-triggered non-preemptive schedule and cache configurations for a given tasks set with real-time constraints.
- We developed a parameterized reconfigurable cache 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 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 an ILP formulation that can model the time-triggered scheduling as well as the cache configuration for a given mapping of a task set on an MPSoC. With this ILP formulation, optimal task schedule and cache configuration can be computed to minimize the cache misses while preventing deadline misses of all tasks in the task set.
- We developed a share-clock mutli-port timer component that enables the precisely time-triggered schedule for the MPSoCs generated from our framework.
- We prototyped and evaluated the generated MPSoCs on Altera Statrix III FPGA using the real-time benchmark. We also analyze and discuss the experiment results under different hardware environment with respect to the number of cores and cache settings.

The rest of the paper is organized as follows: Section II reviews related work in the literature. Section III presents some background principles. Section IV overviews the proposed framework and Section V describes the proposed synthesis approach. Section VI illustrates the hardware infrastructures and Section VII explains how the proposed framework works. Experimental evaluation is presented in Section VIII and Section IX concludes the paper.

II. Related Work

Cache Partitioning: Shared cache interference in multicore system has been recognized as one of major factors that degrade the average performance [7], [16], as well as predictability of system [29], [11]. Much work has been done in general-purpose computing to optimize different objectives by cleverly partitioning shared cache, including cache performance [24], [26] and energy consumption [20]. However, most existing studies are evaluated by simulation. Simulation-based study may only simulate a few billion instructions for a program, which is equivalent to about a few seconds of execution on a real machine. Besides, evaluations on simulators are prone to inaccuracy. Using page coloring technique in OS level, Lin et al. [16] evaluate a dynamical cache partitioning scheme on an Intel 5160 processor. Recently, Cook et al. [7] evaluate the potential of the hardware cache partitioning mechanism and policy to improve system performance with the use of Intel's Sandy Bridge client quad-core chip. This chip only supports coarse-grained way-based cache partitioning on 12-ways associative 6MB LLC, which is shared by all cores via a shared bus. Due to this coarse-grained cache partitioning, authors in [7] report that most parallel applications do not need to adjust their cache size. In contrast to [7], we implement cache management scheme on the proposed FPGA-based multicore system, which allows us to evaluate the efficiency of our scheme in different cache partitioning scale. Furthermore, the above cache management schemes cannot be applied to real-time system.

Little work in the literature has been done in the context of hard real-time system. The major challenge is that no real-time scheduling policy taking into account cache space demands is established [1]. Given a task-level cache partitioning, the authors in [11] develop a sufficient schedulability test for non-preemptive fixed priority scheduling for multicores. However, the work does not consider how to partition the cache size to individual tasks. How to choose cache partition size to optimize system performance while guaranteeing the system predictability is still an open problem [1]. Recently, the state-of-art techniques [29], [14], [19] on the multicore cache management in the context of real-time system has been proposed by using page coloring, which partitions cache by sets in OS-level. However, page coloring based techniques are used to suffer from a significant timing overhead inherent to changing the color of a page, which results that making decision of changing the color of a page cannot be frequent. The authors in [16] report the observed overhead of page coloring based dynamic cache partitioning reaches 7% of the total execution time even after the optimization. In contrast to the page coloring techniques above, we present cache management framework to improve the system predictability for the proposed FPGA-based time-triggered multi-core system, which execute way-based cache partitioning in hardware level. Our approach can dynamically change cache size with minimal overhead (scaling to cycles).

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 vehicle [17] and avionics systems [15], favors a time-triggered approach [3]. Sagstetter et al. [25] present an approach for schedule integration of time-triggered systems tailored to the automotive domain. In [10], Ayman et al. describe a two-stage search technique which is intended to support the configuration of time-triggered schedulers for single-processor embedded systems. However, none of them apply time-triggered scheduling and cache management jointly on real multicore platform to achieve timing predictability.

III. Background

A. Way-based Cache Partitioning

Our cache management scheme implements way-based cache partitioning on FPGA. As shown in Fig. 1, the L2 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 cache reconfiguration APIs. In this work, we dynamically assign cache ways to tasks.

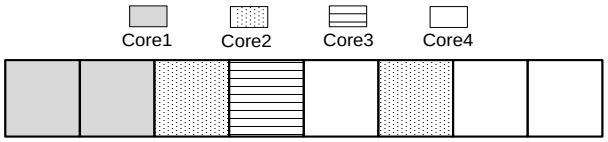


Fig. 1. Way-based Cache Partitioning.

B. Task Model

We consider the functionality of the entire system as an implicit-deadline periodic task set $\tau = \{T_1, ..., T_n\}$, which consists of a set of independent periodic tasks. The deadline D of the task is equal to its period. We adopt the same assumption as [14] and assume that the worst case execution time (WCET) of each task T_i with a specific L2 cache size can be determined ahead of time. Measurement-based WCET estimate technique is used to determine the worst case execution time (WCET), which will be described in more details in Section VII. We use w_{ij} to denote the WCET of task $T_i \in \tau$ with j ways L2 cache allocated and $W_i = \{w_{i1}, w_{i2}, ..., w_{is}\}$ to denote the WCET profile of task T_i, where s is the total number of ways in the L2 cache (cache capacity). Timing predictability is highly desirable for safety-related applications. In this paper, we consider a periodic time-triggered non-preemptive scheduling policy. 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 T_i. The start time s_i is an unknown variable, which is determined by *scheduling* S.

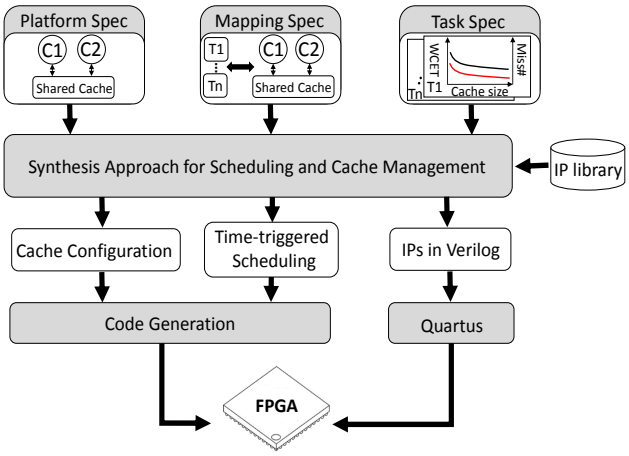


Fig. 2. System Design Framework.

IV. System Design 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 consist of the following three parts.

1) *Platform Specification* describes the settings of a multiprocessor platform, such as the core number of the system, 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, our synthesis approach can generate cache size allocation and time-triggered scheduling for each task according to *input specification*, by which 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 real-time OSs. At the same time, parameterized reconfigurable cache IP and share-clock mutli-port timer IP can be generated according to settings in *platform specification*.

V. Synthesis Approach for Scheduling and Cache Management

This section presents our synthesis approach for timing schedule and cache management. To minimize the cache miss of the system, the problem is formulated as integer linear programming (ILP). With this formulation, cache size allocation and time-triggered scheduling for each task can be generated automatically. We start with an ILP formulation that only focuses on the scheduling problem. Then, the constraints of cache capacity are integrated. In this phase, we use periodical square wave function (PSWF) to capture the cache demand of every task and effectively model the interference between tasks.

A. Time-Triggered Task Scheduling

Time-triggered non-preemptive schedule is considered in this paper to achieve fully predictability of 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 $\sum_{j=1}^{s} c_{ij} w_{ij}$, where s 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:

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

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.
$\forall T_p, T_{\widetilde{p}} \in TS(T_p), i = 0, ..., (\frac{H_{p\widetilde{p}}}{h_p} - 1), j = 0, ..., (\frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} - 1)$:

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

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} - z_{p\widetilde{p}}^{ij} H_r + \xi \leq i \cdot h_p + s_p \tag{3}$$

The constraints (2) and (3) 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 constraints described above guarantee a valid time-triggered schedule. The next step is to add the cache management constraints. The goal here is to 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 state following lemma, 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 start instant of any task within one hyper-period, the cache never overflows.
Proof: Note that the amount of cache allocated to a task is constant during its execution interval. It acquires the resources at the start instant and releases the resources at the finish instant. Hence, cache overflow will not occur if the available resources fulfill the requirement of tasks at its beginning. □

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. As we conduct cache management in task level, this feature also can be used to indicate the resource demand of task in the time domain. For task T_p with start time s_p and execution time e_p, the cache demand at instant t can be defined as:

$$PSWF(t, T_p) = \left\lfloor \frac{t - s_p}{h_p} \right\rfloor + 1 - \left\lceil \frac{t - s_p - e_p}{h_p} \right\rceil \tag{4}$$

The PSWF has several mathematic properties that are beneficial to model the interference between tasks.

Prop. 1: $PSWF(t, T_p) \in \{0, 1\}$. The PSWF with 1 is the task T_p that requires the cache at time instant t and 0 otherwise.

Prop. 2: $PSWF(t, T_p) = PSWF(mod(t, h_p), T_p)$.

Prop. 3: Define intermediate variables $X_{t,T_p} = \left\lfloor \frac{t - s_p}{h_p} \right\rfloor$ and $Y_{t,T_p} = \left\lceil \frac{t - s_p - e_p}{h_p} \right\rceil$, then PSWF can be linearized as $PSWF(t, T_p) = X_{t,T_p} + 1 - Y_{t,T_p}$ with two extra constraints $\frac{t-s_p}{h_p} - 1 < X_{t,T_p} \leq \frac{t-s_p}{h_p}$ and $\frac{t-s_p-e_p}{h_p} \leq Y_{t,T_p} < \frac{t-s_p-e_p}{h_p} + 1$.

According to Lem. 1, we can guarantee to avoid *cache overflow* by checking start instant of any task within one hyper-period. Thus, we can formulate cache management constraints as following.
$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{s} c_{pk} \cdot k + \sum_{T_{\widetilde{p}} \notin TS(T_p)} PSWF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k \leq s \tag{5}$$

One may notice that there are non-linear items in (5), i.e., $PSWF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. However, we can transform this non-linear term into a set of linear constraints using the approach presented in [5] (see Lem. 2). Here, we define an intermediate variable $a_{p\widetilde{p}}^{i} = PSWF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. According to Prop. 1 and Prop. 3, $PSWF(s_p + i \cdot h_p, T_{\widetilde{p}})$ could be linearized as $X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}$ with $(X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}) \in \{0, 1\}$. Based on Lem. 2, the non-linear item $PSWF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$ in (5) can also be linearized.

Lem. 2: Given a constant $s > 0$ and two constraint spaces $P_1 = \{[t, b, x] | t = b \cdot x, 0 \leq x \leq s, b \in \{0, 1\}\}$ and $P_2 = \{[t, b, x] | 0 \leq t \leq b \cdot s,\ t \leq x, t - b \cdot s - x + s \geq 0, b \in \{0, 1\}\}$, then $P_1 \rightleftharpoons P_2$

Besides, each task must have exactly one cache configuration.

$$\sum_{k=1}^{s} c_{ik} = 1 \tag{6}$$

Up to now, we have presented the formulation for the task scheduling and cache partitioning. To minimize the cache miss number in one hyper-period, the following objective function is used:

$$CM = \sum_{\forall T_i} \frac{H_r}{h_i} \sum_{j=1}^{s} c_{ij} CM^{ij} \tag{7}$$

where s 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.

VI. Proposed Hardware Infrastructure

In this section, we present the FPGA-based multi-core system which supports dynamical cache partitioning and time-triggered scheduling. A major benefit of choosing FPGA for prototyping our multicore system is the high configurability of this processor. This allows us to evaluate the proposed

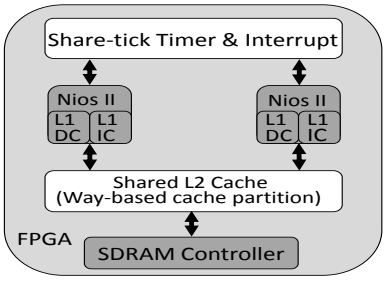
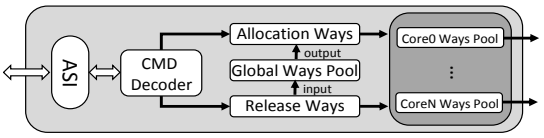
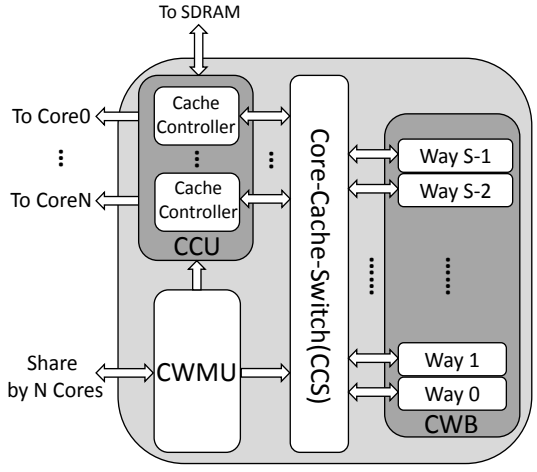


Fig. 3. System Architecture.

integrated scheduling and cache management framework under various hardware configurations with different cache size and varied arithmetic units. Fig. 3 illustrates proposed hardware infrastructure, our multicore NIOS II system on FPGA. We adopt 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 instrument and data cache), along with reconfigurable cache IP which supports dynamically cache partitioning and share-tick timer IP for time-triggered scheduling.



(a) Cache Ways Managment Unit (CWMU)



(b) Reconfigurable Cache Diagram

Fig. 4. Reconfigurable Cache.

A. Reconfigurable Cache IP

Two significant parts need to be considered for way-based cache partition infrastructure, i.e, how to deal with cache coherency and atomic operations. According to 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 the cache coherency [23]. 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 [23]. In this paper, we focus on studying the interaction between timing scheduling and cache management, and follow this official recommend from Altera to create multicore system. Note that inter-core cache interference still exists although software on each core runs in different regions of the same off-chip memory. Besides, in contrast to single port shared cache in Intel's Sandy Bridge chip [7], the current shared cache architecture is multi-port cache, which allows NIOS cores to access cache concurrently. Note that, regarding the cores which offer hardware cache coherency, like ARM processor, our cache can also be extended to single port shared cache and be used to support cache coherency by updating all the hit cache lines in each write operation. Another significant part that should be carefully considered is atomic operations. When cores change their cache size, the APIs for adjusting cache size should be guaranteed to be atomic for implementing synchronization primitives. In this paper, we develop a component, called *cache ways management unit (CWMU)* as shown in Fig. 4(a), to execute cache ways allocation and release, which grantees the offered APIs atomic.

Fig. 4(b) shows the block diagram of reconfigurable shared cache, which allows cores to dynamically change the number of owned cache ways. 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)*. The *cache ways management unit (CWMU)*, by which each core can send command to require or release cache ways, is connected to N NIOS cores by *avalone 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. 4(a), when *CWMU* receives one command from one NIOS core, *CMD decoder* component can distinguish 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 ID will be fetched from *global ways pool*. Then, the fetched ways ID is put into the cache ways pool of the distinguished core. Ways occupied by the distinguished core and replacement information are also updated at the same time. Before fetching ways ID from *global ways pool*, the logic will check whether there is enough ways in the pool. If no enough ways exist in the pool, allocation error will be returned to the distinguished core. In contrast to the procedure of allocation ways command, release ways command will fetch ways ID 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. *Cache control unit (CCU)* instantiates N *cache controller* for 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 cache concurrently. Note that

conventional uni-processor cache can be reused for *cache controller* with tiny changes. In *cache controller*, we adopt FIFO cache replacement policy, which has been widely used in ARM 11 processor and Intel X86 processor [12]. When *CWMU* receives command to change ways number of one core, *CWMU* will update its cache replacement information, i.e, replacement ways pool, for cache controller. Especially for the case of releasing ways, *cache controller* will flush the released ways automatically when it remove cache ways from the replacement ways pool. *Core to cache switch (CCS)* can dynamically connect cores to cache ways block according to ways mask register of each core, which is maintained by *CWMU* according the private cache ways pool of the cores. *Cache ways blocks (CWB)* are some memory blocks used for tag and data store.

B. Share-clock Mutli-port Timer IP

To support the dynamic timekeeping functionality in 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 synchronization problem. In multicore system, we should guarantee that all the cores in the system are triggered in one global timer. Only in that way, the task on different cores can be precisely triggered and well synchronized.

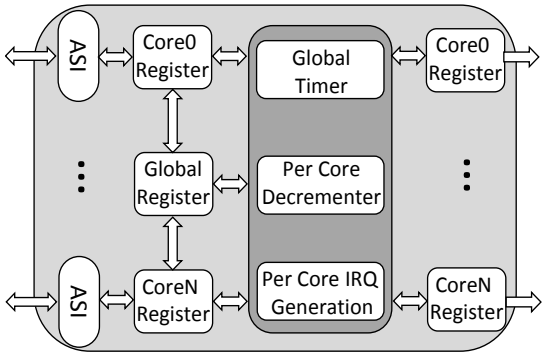


Fig. 5. Share-clock Timer IP.

Fig. 5 shows the block diagram of share-clock mutli-port timer, in which each port is connected to one NIOS core by avalone slave interface (ASI). Share-clock mutli-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 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. Global register is used to synchronize the cores to launch at the same point. Only when all cores call APIs to start timer, global register is set as 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 worst-case execution time (WCET) and cache miss number with different cache size for a given task set. According to the system architecture as shown in Fig. 3, off-chip memory is shared by all cores, and the shared off-chip memory is accessed when a cache miss occurs. The contention on the shared memory delays service to this memory access. In this section, we present one measurement based approach to estimate WCET of task which takes the contention on the shared memory into account.

To guarantee the schedulability of tasks, we need to get the worst case execution time of each task T_i in a safety way. To get the worst-case contention on the shared off-chip memory, we set one core to measure the execution time of a task with other cores running microbenchmark [9], a task that can mimic the behavior of L2-bounded application and inject synthetic traffic on the off-chip memory. In microbenchmark, each iteration of this program executes memory instructions that result miss in L2 cache and access in off-chip memory [9]. In this way, we can get the worst-case contention among cores and guarantee the safety of the measured WCET. Regarding cache miss, we can obtain it from the customized performance counter by calling the related APIs in Tab. I.

Tab. I lists all the atomic APIs currently supported by reconfigurable cache IP. We refer to the implementation of time-triggered scheduler in [10] and implement time-triggered scheduler with the share-clock mutli-port timer on the NIOS-based multicore system. To minimize the cache miss of the system, the synthesis approach in Section V can generate 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.

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 occupied ways, ways# in pool, error state

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.

A. Experiment Setup

To evaluate the effectiveness of our framework and hardware platform, we adopt 24 benchmarks selected from MiBench [13] (bitcount, qsort, dijkstra, Pbmsrch, FFT), CH-Stone [6] (adpcm, blowfish, aes, gsm, sha, mpeg2), DSP-stone [8] (convolution, dot_product, fir2dim, fir, biquad, lms,

matrix1, n_complex_update), PARSEC [4] (blackscholes), and some research work [21], [18] (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 to make them compliant 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 cores system, which are combinations of the selected benchmarks. According to [28], we specify the task mappings based on the rule that the total execution time of each core is comparable.

TABLE II
BENCHMARK SETS FOR TWO CORES SYSTEM

	Core 1	Core 2
Set 1	Fir2dim, Pbmsrch	Blackscholes, Fir
Set 2	Lms, Bitcount	FFT, Biquad
Set 3	FFT, Bitcount, Lms	Sobel, Aes, Fir
Set 4	Nsichneu, Fir, Aes, Matrix1	Dijkstra, Blowfish, Lms, Fir2dim

Our proposed time-triggered multicore system is implemented on Altera development board equipped with Statrix III FPGA, which is based on NIOS II based multicore architecture. In the multicore architecture, we adopt fast NIOS II core equipped with 512 bytes priviate L1 instruction cache and 512 bytes priviate L1 data cache. All cores are shared with the unified L2 cache, which is instanced by the proposed reconfigurable cache IP. By cooperating with the proposed share-clock mutli-port timer, we implement partitioned time-triggered scheduling on each core according to [10]. The global tick of shared clock timer is 1ms. We set the overhead of task switch as 0.1ms.

B. Result

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 to Altera Stratix III 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, 6/128/256 indicates 6-ways cache architecture with 128 address depth and 256-bit line size. F_{max} indicates the maximum frequency that the constructed multicore system can run on. We also list the maximum frequency for the case that system do not equip our shared configurable cache (*only with L1 cache*).

For increase in depth address and ways number, the number of combinational ALUTs and registers also increase. To flush cache ways 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 consume more logic resource in combinational ALUTs and registers. For the ways number, the contributing factors are the core-cache-switch circuitry, FIFO replace policy circuitry, and wide logical OR, all of which grow with the increased ways number. Regarding the maximum operating frequency F_{max}, we can see that, even for the system *only* equipped with L1 cache and without L2 cache, the constructed multicore systems can only run at about 130MHz. After the system equips with the configurable cache, the multicore system can still run at about 100MHz. As expected, we notice that the 8-ways cache is faster than 16-ways cache.

TABLE IV
SPEED AND AREA MEASUREMENTS ON STRATIX III CHIPS

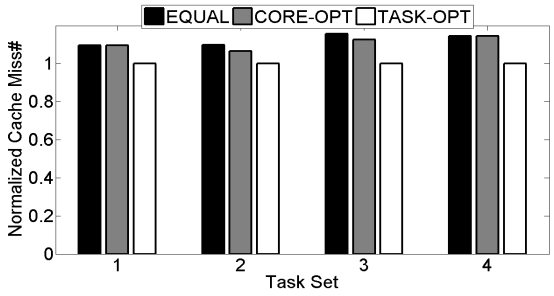
Port Number	Cache Settings	Combinational ALUTs	Total Registers	$\mathbf{F_{max}}$ (MHz)
Two Core	6/128/256	13763	9915	114/139
	6/256/256	16220	13245	114/139
	12/128/256	18659	11559	111/139
	12/256/256	22489	16425	109/139
Four Core	8/128/256	27847	19240	110/131
	8/256/256	32013	25388	108/131
	16/128/256	36599	22054	103/131
	16/256/256	42737	29966	101/131

2) Runtime Measurement: To evaluate the effectiveness of our framework, 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 two hardware platforms, each NIOS core runs at 100MHz. Tab. II and Tab. III respectively list the task sets used in our experiments and task mapping information for two cores system and four cores system. We compare three approaches , i.e., equal partitioning cache on cores(`EQUAL`), the core-based cache partitioning approach (`CORE-OPT`) from [28], and our synthesis approach (`TASK-OPT`).

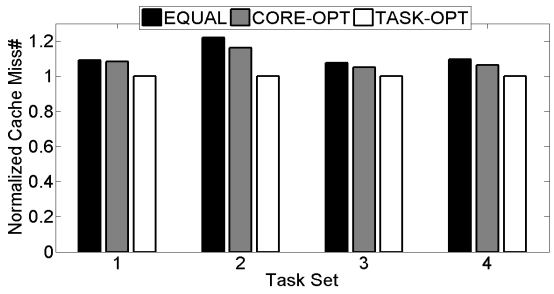
Fig. 6 shows the total cache miss number in one hyper-period of the approaches normalized w.r.t `TASK-OPT`. All results are collected by implementing the cache management scheme and scheduling obtained from the corresponding approach on the proposed multicore system. From the result measured by real hardware, we can see the core-based cache partitioning approach (`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 core-based cache partitioning approach (`CORE-OPT`), our synthesis approach (`TASK-OPT`) partition the cache in task level and integrates cache partitioning globally with scheduling. we can observe that our synthesis approach (`TASK-OPT`) outperforms core-based cache partitioning approach (`CORE-OPT`). Our approach (`TASK-OPT`) can on average reduce 10.8% (up to

TABLE III
BENCHMARK SETS FOR FOUR CORES 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	Biquad, Qurt, Dijkstra	Lms, Qsort, Gsm	Fdct, Sobel
Set 3	Matrix1, Adpcm	Fir2dim, Sobel	Biquad, Sha	Nsichneu, FFT
Set 4	Fir2dim, Convolution, Dot_product	Matrix1, Blowfish, Mpeg2	Biquad, Dijkstra, Blackscholes	Lms, Sobel, Qsort, Fir



(a) # Cache Miss on Two Cores System



(b) # Cache Miss on Four Cores System

Fig. 6. # Cache Miss Reduction on Different Hardware Platform.

14.5%) and 9.0% (up to 16.1%) cache miss with respect to `CORE-OPT` on 2-core and 4-core architectures.

IX. CONCLUSION

This paper presents a cache management framework for hard real-time MPSoCs. The framework optimally integrates time-triggered scheduling and cache partitioning such that the shared cache is used in a predictable and efficient manner. In contrast to software-based cache partitioning techniques in the literature, we conduct cache partition at hardware level and prototyped an implementation on FPGA. Experiment results in the FPGA using a diverse set of applications and different numbers of cores and cache modules demonstrate the effectiveness of the proposed framework.

REFERENCES

[1] A. Abel, F. Benz, J. Doerfert, B. Drr, S. Hahn, F. Haupenthal, M. Jacobs, A. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR 2013 - Concurrency Theory*. 2013.

[2] ARM Cortex-A15 serious. `http://www.arm.com/products`.

[3] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[5] G. Chen, K. Huang, and A. Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination. *ACM Transactions on Embedded Computing Systems*, 2013.

[6] CHStone. `http://www.ertl.jp/chstone/`.

[7] H. Cook, M. Moreto, S. Bird, K. N. Dao, D. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *40th ACM/IEEE International Symposium on Computer Architecture (ISCA 2013)*, 2013.

[8] Dspstone. `http://www.ice.rwth-aachen.de/`.

[9] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-hierarchy contention aware scheduling in cmps. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[10] A. Gendy and M. Pont. Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems. *IEEE Transactions on Industrial Informatics*, 2008.

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

[12] N. Guan, X. Yang, M. Lv, and W. Yi. Fifo cache analysis for wcet estimation: A quantitative approach. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization (WWC)*, 2001.

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

[15] C. Lin, H.-M. Yen, and 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.

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

[17] M. Lukasiewycz, S. Steinhorst, F. Sagstetter, W. Chang, P. Waszecki, M. Kauer, and S. Chakraborty. Cyber-physical systems design for electric vehicles. In *Proceedings of 2012 Euromicro Conference on Digital System Design (DSD)*, 2012.

[18] Malardalen real-time research center. `http://www.es.mdh.se/`.

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

[20] S. Mittal, Z. Zhang, and J. Vetter. Flexiway: A cache energy saving technique using fine-grained cache reconfiguration. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[21] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008.

[22] Altera nios datasheet. `http://www.altera.com/`.

[23] Creating multiprocessor nios systems tutorial. `http://www.altera.com`.

[24] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[25] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty. Schedule integration for time-triggered systems. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[26] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *2011 38th Annual International Symposium on*

Computer Architecture (ISCA), 2011.

[27] OpenSPARC. `http://www.opensparc.net/`.

[28] W. Wang, P. Mishra, and S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *2011 48th ACM/IEEE Design Automation Conference (DAC)*, 2011.

[29] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[30] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 2012.