IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 3, 2019, accepted August 29, 2019, date of publication September 6, 2019, date of current version September 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2939859

Cache-Aware Real-Time Virtualization for Clustered Multi-Core Platforms

YOOJIN LIM[1] AND HYOSEUNG KIM[2], (Member, IEEE)
[1]Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea
[2]Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521, USA

Corresponding author: Hyoseung Kim (hyoseung@ucr.edu)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant (No. 2017-0-00067, Development of ICT Core Technologies for Safe Unmanned Vehicles) funded by the Korea Government (MSIT).

ABSTRACT With the increasing complexity of recent autonomous platforms, there is a strong demand to better utilize system resources while satisfying stringent real-time requirements. Embedded virtualization is an appealing technology to meet this demand. It enables the consolidation of real-time systems with different criticality levels on a single hardware platform by enforcing temporal isolation. On multi-core platforms, however, shared hardware resources, such as caches and memory buses, weaken this isolation. In particular, due to the resulting cache interference, a large last-level cache in recent processors can easily jeopardize the timing predictability of real-time tasks due to cache interference. While researchers in the real-time systems community have developed solutions to tackle this problem, existing cache management schemes reveal two major limitations when used in a clustered multi-core embedded system. The first is the cache co-partitioning problem, which can lead to wrong cache allocation and cache underutilization. The second is the cache interference of inter-virtual-machine (VM) communication because prior work has considered only independent tasks. This paper presents a cluster-aware real-time cache allocation scheme to address these problems. The proposed scheme takes into account the cluster information of the system, and finds the cache allocation that satisfies the timing and memory requirements of tasks. The scheme also maximizes slack time to meet task deadline, which brings flexibility and resilience to unexpected events. Tasks using inter-VM communication are also provided with guaranteed blocking time and cache isolation. We have implemented a prototype of our scheme on an Nvidia TX2 clustered multi-core platform and evaluated the effectiveness of our scheme over cluster-unaware approaches.

INDEX TERMS Cache interference, clustered multi-core platforms, real-time systems, embedded virtualization, real-time hypervisor, partitioning hypervisor, real-time resource management.

I. INTRODUCTION

Embedded system virtualization offers an opportunity to significantly reduce space, power, and cost requirements by consolidating multiple systems into a single hardware platform. It also simplifies retrofitting legacy systems as it causes only minimal or no change in application software and avoids the need for rigorous re-certification processes. Fault isolation, intellectual property protection, and license segregation are additional benefits virtualization can bring. These features of embedded virtualization are particularly beneficial in smart and autonomous systems for automotive, avionics, robotics, and defense applications.

The associate editor coordinating the review of this manuscript and approving it for publication was Michele Magno.

For the success of embedded virtualization in safety-critical domains, ensuring predictable real-time performance is one of the key requirements. In particular, partitioning hypervisors, such as Jailhouse [1], QuestV [27], and Qplus-Hyper [25], have established a strong foundation for this purpose. They address the problems of complex hierarchical scheduling and timing analysis issues by strict partitioning of CPU and memory, and offer real-time performance close to native systems. They can also satisfy the increasing demand for mixed-criticality support, by co-hosting high-critical systems, e.g., certified real-time OS, together with low-critical systems, e.g., Linux and Android, on the same platform. However, these properties are maintained only in an ideal hardware platform. Shared hardware resources on recent multi-core platforms, such as a last-level cache (LLC) and a memory bus, can introduce a significant amount of temporal

interference among real-time workloads, e.g., up to 12× slowdown in a quad-core machine [16]. Without considering these issues, the requirement of timing predictability cannot be fully satisfied even on partitioning hypervisors.

Cache interference, which is the timing penalty caused by contention on an LLC, has been studied extensively in native multi-core real-time systems [17], [31], [38], [40]. They use an OS-level cache partitioning technique, called page coloring, to explicitly manage the cache allocation of tasks in software. Recently, the technique has been extended to the virtualization environment [21], [23] such that it is possible to allocate a portion of the LLC to individual tasks running within a virtual machine (VM). However, all these real-time cache management schemes have focused on homogeneous multi-core processors. In recent clustered multi-core embedded platforms, such as Nvidia TX2 and ODROID XU4, cluster-unaware page coloring-based techniques may cause significant cache wastage and memory requirement violation due to the *cache co-partitioning* problem, which we will discuss in Section II.

In addition, existing real-time cache management techniques assume that each task is independent of each other and involves no interaction with other tasks. However, in embedded virtualization, tasks running in different VMs often need to communicate to achieve their goals. Shared memory-based inter-VM communication [9] is widely used for this purpose due to its efficiency. To prevent race conditions and achieve bounded blocking time, one may consider adopting a virtualization-aware real-time locking mechanism [24] for inter-VM communication, but it does not solve the problem of cache interference when tasks access shared memory regions.

In this paper, we propose a *cluster-aware* cache allocation scheme to simultaneously address the problems of cache management and inter-VM data communication in a real-time partitioning hypervisor. Our scheme explicitly considers the cluster structure and per-cluster LLC information of target multi-core processors, and finds cache allocation that guarantees task schedulability, maximizes slack time, and satisfies memory requirements. In addition, tasks involved in inter-VM communication are provided with bounded blocking time and cache performance isolation for shared memory access. We have implemented a prototype of our scheme on an Nvidia TX2 embedded platform and evaluated our scheme with randomly-generated tasksets. To the best of our knowledge, this is the first work to address real-time cache management in clustered multi-core systems and the first work to ensure cache isolation in shared memory-based inter-VM communication.

The contributions of this work are built upon findings from our prior work on cluster-unaware cache management [17], [21], [23], [38] and cache-unaware critical sections [19], [24]. This paper extends these efforts by simultaneously addressing the two distinct challenges in a recent clustered multi-core system with a partitioning hypervisor. Detailed technical differences from prior work are discussed in Section VI.

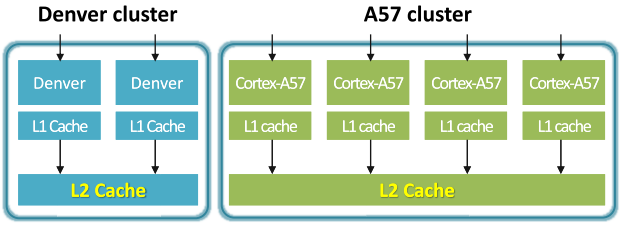

FIGURE 1. Nvidia TX2 clustered multi-core platform.

The rest of the paper is organized as follows. Section II reviews the background for our work. Section III describes our system model, notation, and assumptions. Section IV presents our cluster-aware cache allocation scheme. Section V provides detailed evaluation. Section VI reviews related work, and Section VII concludes the paper.

II. BACKGROUND

This section gives a brief description of the execution environment and the page coloring technique, and discusses the cache co-partitioning problem that occurs in a clustered multi-core system.

A. EXECUTION ENVIRONMENT

1) CLUSTERED MULTI-CORE ARCHITECTURES

This work specifically considers a clustered multi-core architecture, where each cluster has one or more CPU cores with an LLC shared among these cores. Due to the success of the ARM's big.Little architecture, this clustered multi-core architecture is prevalent in recent embedded platforms. A good example is an Nvidia TX2 processor illustrated in Figure 1. It has two clusters, Denver and Cortex A57. The Denver cluster has two CPU cores and a shared L2 cache of 2MB. The Cortex A57 cluster has four CPU cores and a shared L2 cache of 2MB.

Since each cluster has its own LLC, cache interference does happen among tasks running on different clusters. Within each cluster, cache interference can fall into two types: inter-core and intra-core cache interference [17]. Inter-core cache interference is hard to be upper-bounded because multiple tasks on different cores can simultaneously compete for the LLC at any time. In contrast, intra-core cache interference can be bounded as *cache-related preemption delay*, which accounts for the amount of cache evictions caused by preemption. Hence, we will focus on preventing inter-core interference by cache allocation and taking into account cache-related preemption delay in schedulability analysis.

At the platform level, there also exist other resources shared among CPU cores. Such resources often require *mutual exclusive* access to prevent data corruption and unexpected behavior. Shared memory regions for inter-VM communication belong to this category. Tasks running on different clusters may communicate each other, e.g., RTOS tasks on the Denver cluster and Linux tasks on the A57 cluster of the TX2 platform, and no more than one task should be allowed

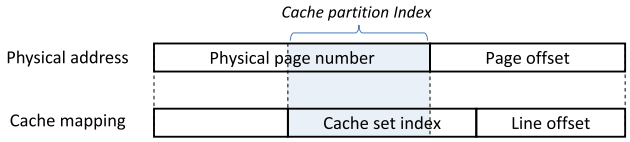

FIGURE 2. Physical address to cache mapping under page coloring.

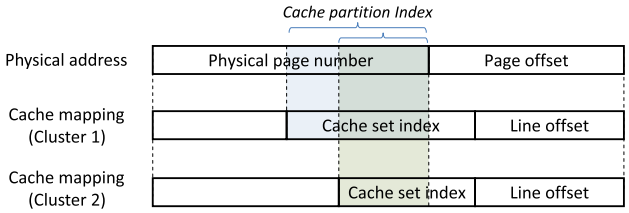

FIGURE 3. Page coloring in a clustered multi-core system.

access a shared memory region at a time. Real-time synchronization and locking protocols [24], [34]–[36] have been developed to ensure mutual exclusion and bounded blocking time in a multi-core environment.

2) PARTITIONING HYPERVISOR

Many embedded hypervisors have been developed following the real-time hierarchical scheduling structure. In this structure, each virtual CPU (VCPU) of a VM is assigned a set of timing parameters, such as CPU budget, budget replenishment period, and resource supply policy. Hence, the timeliness of tasks running in a VM is affected by the timing parameters of the corresponding VCPUs. It is also often considered to pack multiple VCPUs to a single PCPU to improve resource utilization at the cost of increased (but still bounded) response time.

On the other hand, partitioning hypervisors [1], [2], [27], [37] take a simple but strict temporal isolation approach. Instead of time-sharing a single physical CPU (PCPU) among multiple VCPUs, they assign only one VCPU to PCPU and there is a one-on-one mapping between VCPUs and PCPUs. This approach can eliminate the scheduling penalty caused by conventional hierarchical scheduling, e.g., VCPU budget depletion and preemption. If we ignore any runtime overhead caused by virtualization, they can offer the same scheduling behavior as in the native, non-virtualized environment. This is particularly appealing to safety-critical domains requiring guaranteed short latency. As the core count of embedded processors keeps increasing, partitioning hypervisors are expected to be used in a broader range of applications.

B. PAGE COLORING

Page coloring is an OS-level software-based technique that works for a physically-indexed set-associative cache. Note that LLCs in most of the recent architectures belong to this category. Page coloring uses the mapping between cache set indices and physical memory addresses. Figure 2 shows an example. Some bits of the physical page number are used to determine the cache set index. Using these bits, page coloring identifies the cache mapping of physical pages and partitions the cache into n cache partitions. The number of cache partitions, n, is determined by:

$$n = (\text{LLC size})/(\#\text{ of ways} \times \text{page frame size})$$

Page coloring simultaneously partitions the entire physical memory into n memory partitions. The allocation of a specific cache partition to a task is done by allocating physical pages in the corresponding memory partition to that task. If a task needs more than one cache partitions, physical pages from the corresponding memory partitions can be allocated in a round-robin manner to even the memory usage per partition and to achieve determinism [17]. Due to this coupled nature of cache and memory partitions, it is important to allocate enough cache partitions to tasks in order to satisfy their memory requirements.

Standard page coloring does not work within a VM due to the additional address translation layer introduced by the hypervisor. Recent work [21] proposed techniques, called vLLC and vColoring, to enable page coloring within a VM. With this technique, each task or a group of tasks in a VM can be allocated the host machine's cache partitions.

C. CACHE CO-PARTITIONING PROBLEM

In a clustered multi-core platform, page coloring partitions the LLCs of all clusters at the same time. If each cluster has a different LLC, it may result in a different number of cache partitions for each cluster. Figure 3 gives an example of a two-cluster multi-core system with page coloring. The LLC of Cluster 2 has a smaller number of cache sets, thereby yielding a smaller number of cache partitions than Cluster 1. Interestingly, a single physical page is mapped to each of the LLCs because page coloring relies on the mapping between physical addresses and cache sets. Hence, if a cache allocation scheme is unaware of cluster information, it may choose either the cache-partition counts of Cluster 1 or 2, and generate wrong cache allocation.

Even if all clusters have the same type of LLCs, the system may suffer from a cache underutilization issue. For example, consider two tasks running on two different clusters, each with 4 cache partitions ($\{1, 2, 3, 4\}$). A cluster-unaware scheme would allocate a distinct set of cache partitions to each task to avoid cache interference, e.g., $\{1, 2\}$ and $\{3, 4\}$. However, the two tasks do not cause cache interference with each other and the rest cache partitions on each cluster are unnecessarily left unused. It is desired to utilize all cache partitions on each cluster to reduce task response time and help meet the deadline. This problem cannot be easily solved by modeling each cluster as a standalone system because the physical pages corresponding to cache partitions are tied across clusters. Doing so may cause memory overbooking and fail to provide the required amount of memory to tasks.

III. SYSTEM MODEL

Our system model considers a clustered multi-core system with the last-level cache (LLC) structure described in Section II-A. The system has a set of CPU clusters, $\mathbb{L}$, and each cluster $L \in \mathbb{L}$ has one or more CPU cores. Each cluster has its own LLC that is shared among all CPUs within that cluster. In line with the design of recent clustered embedded processors, e.g., Nvidia TX2, we assume that CPUs within the same cluster are identical, but CPUs in different clusters may use different architectures or run at different clock speed. The LLC is divided into cache partitions by page coloring.

The system hosts multiple guest virtual machines (VMs), each of which has one or more virtual CPUs (VCPUs). Following the concept of partitioning hypervisors described in Section II-A, each VCPU is fully assigned one physical CPU (PCPU). Each VM is assumed to be homogeneous; thus, the VCPUs of a VM are assigned to one cluster in the system. In other words, a single VM cannot be assigned across more than one cluster, but multiple VMs can be assigned to a single cluster as long as there are enough PCPUs. VCPU-to-PCPU allocation is statically given and is unchanged at runtime in order not to cause unexpected timing delay. Guest operating systems (OSs) running in VMs use *partitioned fixed-priority preemptive scheduling* for task scheduling, which is widely used in practical real-time applications. Hence, each task is assigned to one VCPU and does not migrate to another one.

A. TASK MODEL

This paper considers tasks following the sporadic task model [32] with constrained deadlines. This means that a task releases its job in a recurrent manner and each job execution should be done by the deadline. Each task τ_i is characterized by the following parameters:

$$\tau_i := (C_i(k), T_i, D_i, M_i)$$

- $C_i(k)$: The worst-case execution time (WCET) of a single job of τ_i when k cache partitions of its cluster are assigned to it.
- T_i: the minimum time interval between any two consecutive job releases of τ_i (also called the minimum inter-arrival time).
- D_i: the relative deadline of each job τ_i ($D_i \leq T_i$).
- M_i: the maximum memory requirement of τ_i in bytes.

Note that $C_i(k)$ does not include any external cache- or scheduling-induced interference (e.g., preemption) from other tasks. The sum of the τ_i's WCET and such extrinsic interference forms the response time of τ_i, and the task τ_i is said *schedulable* if its worst-case response time is smaller than or equal to D_i. As done in the literature [21], [31], [40], $C_i(k)$ can be obtained by measurement or static analysis tools and is assumed to be a monotonically decreasing function of k. As shown in [3], monotonic over-approximation in WCET values can easily satisfy this assumption. The parameter M_i indicates the amount of physical memory required by a task τ_i. If less than M_i bytes of memory is available for τ_i, it may experience page swapping or be terminated in the middle of execution, thereby making it unschedulable. Each task τ_i has a unique priority π_i, which can be easily achieved by arbitrary tie-breaking.

B. INTER-VM COMMUNICATION AND CRITICAL SECTIONS

Some tasks may include codeblocks for inter-VM data communication using shared memory. In our system model, each shared memory region among VMs is protected by a lock for mutual exclusion, and any task that attempts to access the shared memory region must acquire the lock. With this notion, task execution time is decomposed as follows:

$$C_i := (C_{i,1}, E_{i,1}, C_{i,2}, E_{i,2}, ..., E_{i,e_i}, C_{i,e_i+1})$$

where $C_{i,j}$ and $E_{i,j}$ are the WCET of the j-th normal-execution and critical-section segments, respectively, and e_i is the number of critical sections. Each segment execution time, e.g., $C_{i,1}$ and $E_{i,1}$, is also a function of the number of cache partitions but we omitted it here for the ease of presentation. We will specifically denote that parameter when it is needed.

We assume that locks and critical sections are managed by vMPCP [24], which is the virtualization-aware version of the well-known Multiprocessor Priority Ceiling Protocol (MPCP) [35], [36]. In the partitioning hypervisor environment where the entire budget of one PCPU is dedicated to a single VCPU, vMPCP and MPCP behave the same except that vMPCP provides system primitives for inter-VM locking. Under MPCP and vMPCP, a task waiting for a lock is suspended and inserted to the waiting queue of that lock. If the lock is released, the highest-priority task in the lock's waiting queue acquires the lock and enters the corresponding critical section. The priority of the lock-holding task is immediately boosted to $\pi_B + \pi_k$, where π_B is the base priority level greater the normal priority of any task in th system and π_k is the highest normal priority of any task using that lock. The task recovers its normal priority as soon as it releases the lock.

IV. CLUSTER-AWARE CACHE ALLOCATION

This section presents our proposed cluster-aware cache allocation scheme. We first assume that tasks do not have any inter-VM data communication, i.e., no critical section, and give the details of the allocation algorithms. We then relax this assumption by extending our scheme.

A. TASK SCHEDULABILITY ANALYSIS

Before introducing our cache allocation scheme, we will review the task schedulability analysis that our algorithms use. As mentioned, we consider tasks with no critical section first. Hence, $e_i = 0$ for all task τ_i.

We check the schedulability of a task τ_i based on the recursion-based response-time test given in [21]:

$$R_i^{n+1} = C_i(k) + \sum_{\tau_h \in V(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{R_i^n}{T_h} \right\rceil (C_h(k) + \gamma) \quad (1)$$

where R_i^n is the worst-case response time of τ_i at the n^{th} iteration, $V(\tau_i)$ is the VCPU of τ_i, π_i is the priority of τ_i, and γ

is the cache-related preemption delay (CRPD). The test starts with $R_i^0 = C_i$ and terminates when $R_i^{n+1} = R_i^n$ or $R_i^{n+1} > D_i$. The task τ_i is schedulable if $R_i^{n+1} \leq D_i$. For simplicity, we assume that all tasks on the same VCPU share all k cache partitions assigned to that VCPU. Thus, the same k parameter is used for C_i and C_h. The CRPD is given by $\gamma = k \cdot \Delta$, where Δ is the maximum time to refill one cache partition with data from memory. If the LLC uses a write-back policy, Δ should include the extra memory access time to handle dirty cache lines [21]. To reduce the amount of CRPD, one may consider per-task cache allocation and CRPD analysis, e.g., [10], [17], which can also be used with our algorithms presented in the next subsection.

B. CACHE ALLOCATION

In many practical applications, it is good to have sufficient slack time so that even under unexpected interference or workload surges, tasks have higher chances to satisfy their deadlines. Such slack time can also be used for executing non-real-time tasks or for cooling down the processor. To better represent the relative importance of each task's slack, we define the *weighed worst-case slack time* of a task τ_i as follows:

$$S_i = \frac{D_i - R_i}{T_i} \cdot \frac{\pi_i}{n} \tag{2}$$

where n is the total number of tasks in the system and π_i is the priority level of τ_i. Eq (2) uses task period and relative priority order as normalized weights. Recall that each task is assumed to have a unique priority. Thus, there are n distinct priority levels and the second term of the equation gives the normalized priority order of τ_i. In the first term, the numerator is the worst-case slack of τ_i. It is then divided by the task period T_i to obtain the normalized slack of τ_i. With this, we can compare each task's slack time in a quantitative manner. The weighted slack of a VCPU v_j is given by:

$$S_j^v = \sum_{\tau_i \in v_j} S_i \tag{3}$$

C. GOAL

The goal of our cache allocation scheme is to maximize slack time (and thus minimize response time) as long as the system resources permit.

1) WEIGHTED SLACK OF VCPU

To do so, we first present Algorithm 1 that derives each VCPU's weighted slack with respect to the number of cache partitions. This algorithm takes the target VCPU v_i and the number of cache partitions available to that VCPU. If v_i has no cache partition, tasks on v_i cannot execute and the resulting slack is set to $-\infty$. The algorithm then iterates from one to the maximum number of cache partitions. For k cache partitions, the algorithm checks if all tasks of v_i are schedulable by using Eq (1) (line 3). If so, the slack time of v_i with k cache partitions, $S_i^v(k)$, is obtained. It also checks the memory usage

Algorithm 1 VcpuWeightedSlack(v_i, N_{cache})

Input: v_i: VCPU, N_{cache}: the number of available cache partitions
Output: S_i^v and MP_i^v

1: $S_i^v(0) \leftarrow -\infty$
2: **for** $k \leftarrow 1$ **to** N_{cache} **do**
3: **if** *schedulable*(v_i, k) **then**
4: $S_i^v(k) \leftarrow \sum_{\tau_j \in v_i} S_j(k)$
5: $MP_i^v(k) \leftarrow \lceil \frac{\sum_{\tau_j \in v_i} M_j}{k} \rceil$
6: **else**
7: $S_i^v(k) \leftarrow -\infty$
8: **end if**
9: **if** $S_i^v(k) < S_i^v(k-1)$ **then**
10: $S_i^v(k) \leftarrow S_i^v(k-1)$
11: $MP_i^v(k) \leftarrow MP_i^v(k-1)$
12: **end if**
13: **end for**
14: **return** $S_i^v(0...N_{cache})$, $MP_i^v(0...N_{cache})$

of v_i per memory partition, MP_i^v. At line 5, the numerator within the ceiling function gives the total memory usage of all tasks and it is divided by the number of cache (thus memory) partitions. This memory usage per partition which will be used later by another algorithm. From lines 9 to 12, the algorithm ensures $S_i^v(k)$ to be a monotonic increasing function. Finally, S_i^v and MP_i^v are returned. The time complexity of Algorithm 1 is bounded by $O(N_{cache} \cdot |\Gamma^v|)$, where $|\Gamma^v|$ denotes the number of tasks per VCPU.

2) CACHE ALLOCATION TO VCPUS

Once the set of weighted slack values is obtained for each VCPU, we can now determine the number of cache partitions for each VCPU. Algorithm 2 presents our cluster-aware cache allocation algorithm that maximizes the overall slack time while satisfying the memory requirements of tasks. For each cluster L, the algorithm first determines M_L, which is the maximum amount of physical memory available to be used by L. It is a fraction of the total memory availability M_{total}, and is computed as the proportion of the cumulative memory requirements of VCPUs in L to those in the entire system. Note that $MP_i^v(1)$ is precomputed by Algorithm 1 for all VCPUs. The number of cache partitions in L is N_{cache}. Other clusters may have a different number of cache partitions depending on their LLC structures. The algorithm then finds k_i^{min} for each VCPU v_i, where k_i^{min} is the minimum number of cache partitions required for non-negative valid slack time, i.e., $S_i^v(k_i^{min}) \geq 0$. The sum of all k_i^{min}, z, gives the minimum number of cache partitions needed to schedule all VCPUs in the cluster L. (line 5). If there are not enough cache partitions in L, then the algorithm returns fail.

At line 9 of Algorithm 2, $k_{i,z}$ denotes the number of cache partitions assigned to v_i when z cache partitions are given. The algorithm assigns k_i^{min} to $k_{i,z}$ because this is the only schedulable allocation with z partitions. Using these, the total

Algorithm 2 ClusterAwareCacheAlloc($\mathbb{L}$, M_{total})

Input: $\mathbb{L}$: a set of clusters, M_{total}: the amount of system memory available for real-time tasks
Output: Success or Fail

1: **for all** $L \in \mathbb{L}$ **do**
2: $M_L \leftarrow M_{total} \cdot \frac{\sum_{v_i \in L} MP_i^v(1)}{\sum_{v_j \in \mathbb{L}} MP_j^v(1)}$ /* max cluster memory */
3: $N_{cache} \leftarrow$ # of caches in the cluster L
4: Find k_i^{min} for each VCPU $v_i \in L$
5: $z \leftarrow \sum_{v_i \in L} k_i^{min}$
6: **if** $N_{cache} < z$ **then**
7: **return** Fail
8: **end if**
9: $\forall v_i \in L : k_{i,z} \leftarrow k_i^{min}$
10: $S^L(z) \leftarrow \sum_{v_i \in L} S_i^v(k_{i,z})$ /* total weighted slack */
11: /* maximize S^L with remaining cache partitions */
12: **for** $p \leftarrow z+1$ **to** N_{cache} **do**
13: Find $S^L(p)$ using Eq. (5). and Alg. 3
14: $\forall v_i \in L : k_{i,p} \leftarrow$ # of partitions assigned to v_i for $S^L(p)$
15: **end for**
16: /* final check for memory size validity */
17: **if** Alg. 3 is Invalid for $\{k_{i,N_{cache}}\}$ **then**
18: **return** Fail
19: **end if**
20: **end for**
21: **return** Success

weighted slack of the cluster L for z cache partitions, $S^L(z)$, is obtained (line 10).

Starting from line 12, the algorithm maximizes the total weighted slack, S^L, by utilizing remaining cache partitions in the cluster L. Suppose that, for q given cache partitions, $k_{i,q}$ cache partitions have been assigned to a VCPU v_i. Since the slack of v_i, $S_i^v(k)$, is monotonically increasing with k, any additional cache partition to v_i will give either zero or positive gain in the slack. Let $ES_i^v(\alpha, q)$ denote the amount of *extra* slack time obtained by assigning α additional cache partitions to v_i:

$$ES_i^v(\alpha, q) = S_i^v(k_{i,q} + \alpha) - S_i^v(k_{i,q}) \tag{4}$$

Using these properties, we can find $S^L(p = z+1)$ by

$$S^L(z) + \max_{v_i \in L} ES_i^v(1, z)$$

and $S^L(p = z+2)$ by

$$\max(S^L(z) + \max_{v_i \in L} ES_i^v(2, z), \quad S^L(z+1) + \max_{v_i \in L} ES_i^v(1, z+1)).$$

This can be generalized to any $p > z$ as follows:

$$S^L(p) = \max_{z \le x < p} \left(S^L(x) + \max_{v_i \in L} ES_i^v(p - x, x) \right) \tag{5}$$

This recurrence can be solved by dynamic programming. The algorithm uses it at line 13 to find the maximum $S^L(p)$.

Algorithm 3 MemoryUsageCheck(L, $\{k'_{i,p}\}$, M_L)

Input: L: a cluster, $\{k'_{i,p}\}$: a set of the number of cache partitions of each VCPU v_i when a total of p cache partitions of L are used, M_L: the maximum memory size available for tasks in L
Output: Valid or Invalid

1: $MP_{max}^v \leftarrow \max_{v_i \in L} MP_i^v(k'_{i,p})$
2: $M_{used} \leftarrow MP_{max}^v \times p$ /* total memory used by VCPUs */
3: **if** $M_{used} > M_L$ **then**
4: **return** Invalid
5: **end if**
6: **return** Valid

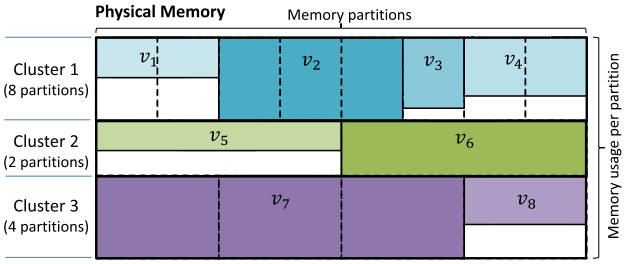

FIGURE 4. Physical memory usage of VCPUs on three clusters.

In addition, it is important to check if the memory requirements of tasks are satisfied while determining cache allocation to VCPUs. Algorithm 3 is the function performing this check for a given cluster L with cache allocation $k'_{i,p}$ to each VCPU v_i. It first finds MP_{max}^v, the maximum per-partition memory usage among all VCPUs of L. It is multiplied by p, which is the total number of cache partitions used, to upper bound the total memory usage of the cluster L (M_{used}). If M_{used} exceeds the maximum available memory of the cluster L (M_L), then the function returns invalid. The reasoning behind pessimistically upper-bounding the cluster memory usage, i.e., $M_{used} = MP_{max}^v \times p$, is to facilitate the memory usage analysis of clusters with different cache-partition counts. Figure 4 gives the example of three CPU clusters, each with a different number of cache partitions. Since page coloring divides physical memory into the same number of memory partitions, the size of each memory partition is different in all three clusters, which makes it complex to precisely compare the memory usage of individual VCPUs located in different clusters. Hence, we take this upper-bounding approach to simplify the memory check procedure.

For each cache configuration considered by Eq.(5), we check the validity of the resulting memory usage by using Algorithm 3, and discard any configuration that is invalid with respect to the memory requirements. Once the remaining cache allocation is done, Algorithm 2 checks the memory size validity of the final cache allocation ($\{k_{i,N_{cache}}\}$ in line 17) and updates M_{used}.

The time complexity of Algorithm 2 is bounded by $O(|\mathbb{L}| \cdot (N_{cache})^2 \cdot |L|^2)$, where $|\mathbb{L}|$ is the number of clusters and $|L|$ is the number of VCPUs per cluster.

D. CACHE ISOLATION FOR INTER-VM DATA COMMUNICATION

We now relax our assumption that tasks do not have any critical section for shared memory-based inter-VM data communication. As discussed in Section II, accessing shared memory can cause cache interference to other tasks. In order to prevent such interference and achieve cache performance isolation, we propose to assign a separate set of cache partitions to each shared memory region. This can be done by creating a shared memory region using the physical pages corresponding to the cache partitions separately assigned. These cache partitions are not used by any normal execution segments of tasks, and are not shared among different shared memory regions. Therefore, this approach ensures that while a task on one VCPU is accessing a critical section, tasks on other VCPUs do not experience any cache interference.

The analysis given in Eq. 1 should be extended to consider critical sections. Based on the MPCP and vMPCP analysis [24], [26], we present an extended schedulability test that considers both cache-related preemption delay and blocking time caused by critical sections:

$$R_i^{n+1} = C_i(k) + B_i^l + B_i^r + \sum_{\tau_h \in V(\tau_i)} \left\lceil \frac{R_i^n + J_h}{T_h} \right\rceil (C_h(k) + \gamma) \quad (6)$$

where B_i^l and B_i^r are local and remote blocking time, respectively, and J_h is the term to capture the dynamic self-suspending behavior of high-priority tasks accessing critical sections [6]. J_h is given by:

$$J_h = \begin{cases} W_h - C_h(k) & : \text{if } e_h > 0 \\ 0 & : \text{otherwise} \end{cases}$$

where e_h is the number of critical sections of τ_h. The terms B_i^l and B_i^r can be obtained directly by the analysis in [24], [26]. B_i^l is given by:

$$B_i^l = (e_i + 1) \cdot \sum_{\tau_l \in V(\tau_i) \wedge \pi_l < \pi_i} \max_{1 \le u \le e_l} E_{l,u}(k')$$

where k' is the number of cache partitions corresponding to the critical section $E_{l,u}$. B_i^r is given by:

$$B_i^r = \sum_{1 \le j \le e_i} B_{i,j}^r$$

$$B_{i,j}^r = \max_{\substack{\pi_l < \pi_i \wedge \\ R(\tau_{l,u}) = R(\tau_{i,j})}} W_{l,u} + \sum_{\substack{\pi_h > \pi_i \wedge \\ R(\tau_{h,u}) = R(\tau_{i,j})}} (\lceil \frac{B_{i,j}^r}{T_h} \rceil + 1) W_{h,u}$$

$$W_{l,u} = E_{l,u}(k') + \sum_{\tau_x \in V(\tau_l)} \max_{1 \le y \le e_x \wedge \pi_{x,y} > \pi_{l,u}} E_{x,y}(k'')$$

where $R(\tau_{l,u})$ is the index of the shared memory region accessed by the u-th critical section of τ_l, and $\pi_{l,u}$ is the priority ceiling of the u-th critical section of τ_l.

There may exist many approaches to determine the number of cache partitions for each shared memory region. Here we present two simple approaches that can be used together with the cache allocation scheme presented in the previous subsection. The first approach is to assign the minimum cache partitions which are just enough to satisfy the size requirement of a shared memory region. If the memory region is shared among only the VCPUs within the same cluster, the number of cache partitions can be easily computed by $k = \lceil M_r/(M_L/p) \rceil$, where M_r is the memory requirement of the shared region r, M_L is the total amount of memory available for the cluster, and p is the total number of cache partitions of the cluster. If the memory region is also shared with the VCPUs of other clusters, the actual number of cache partitions for the shared region in each cluster may differ because the clusters may have different partition counts. Hence, the lowest cache partitioning granularity (smallest partition counts) among the clusters should be chosen and this granularity needs to be used for allocation. The second approach is to assign more cache partitions to those accessed by cache-sensitive critical sections. This approach can begin with finding out the minimum number of cache partitions (based on the first approach) and then increase cache partitions one at a time until the system becomes schedulable. These two approaches for cache allocation to shared memory regions can be done before executing Algorithm 2. We will explore in the evaluation section the impact of critical sections and their allocated cache size on task schedulability.

V. EVALUATION

This section first presents the prototype implementation of our scheme on an Nvidia TX2 platform and our measurements of cache interference. It then shows our experimental results with randomly-generated tasksets.

A. IMPACT OF CACHE INTERFERENCE

1) PROTOTYPE IMPLEMENTATION

We have used Virt/RK [22] as a partitioning hypervisor in order to construct an open-source based evaluation environment. Virt/RK is a real-time multi-core virtualization framework, originally developed as an extension to the Linux KVM hypervisor for x86 and 32-bit ARMv7 machines. It supports virtualization-aware multiprocessor synchronization and page coloring techniques, vMPCP [24] and vLLC [21], respectively. While Virt/RK supports real-time hierarchical scheduling interfaces, we configured it to follow the scheduling structure of the partitioning hypervisor discussed in Section II. We also used the vLLC implementation of Virt/RK for cache allocation to tasks running in VMs.

Our target hardware, Nvidia TX2, has two clusters of 64-bit ARMv8 Cortex A57 and Denver cores. We used Tegra L4T R28.2.1, Linux kernel v4.4.38, and QEMU v2.12.1 (latest as of Aug 8, 2018) as the baseline for our implementation. Since Virt/RK was only available for old Linux kernels (v3.8),[1] we have ported the Linux kernel and KVM part of Virt/RK to

[1]The source code of the original Virt/RK implementation is available at http://rtml.ece.cmu.edu/redmine/projects/rk/.

our baseline version. The major change conducted was for supporting the 64-bit ARMv8 architecture. Since both Cortex A57 and Denver cores of TX2 are in ARMv8, this was a mandatory step for our work. QEMU performs device emulation for KVM-based virtualization which Virt/RK is built upon. Hence, we have also modified the QEMU emulation code of AAach64 system registers required for vLLC and made miscellaneous changes to QEMU for Virt/RK resource configurations.

2) EXPERIMENTAL SETUP

On Nvidia TX2, each cluster has a shared L2 of 2 MB with 16-way associativity and 2048 cache sets. This gives 32 cache partitions per cluster. During all experiments, we disabled the dynamic clock frequency scaling of the processor to reduce measurement inaccuracies. Processor clock frequencies are fixed to either 1.4 GHz or 2.0 GHz depending on experimental settings.

We used a total of two VMs, each assigned to a different cluster. The first VM has four VCPUs. In accordance with a partitioning hypervisor, each VCPU of the first VM is allocated to exactly one Cortex A57 CPU core with 100% of budget. The second VM has two VCPUs and each allocated to one Denver core. Each VM is assigned all the 32 cache partitions of the corresponding cluster. On the host side, the QEMU process and VCPU threads are assigned real-time priorities, which prevents unexpected delays from indispensable system services that could not be disabled. For the guest OS, we have used Linux/RK [18] based on the kernel v4.4.38.

3) RESULTS

In a multi-core environment, the major amount of cache interference happens among tasks running on different physical cores. Hence, we measure the impact of cache interference on TX2. For the Cortex-A57 VM, we execute four instances (tasks) of the `latency` program [44] on four different VCPUs simultaneously. `latency` traverses a randomly-ordered linked list and its execution time is highly dependent on memory access time. The working set size of `latency` was configured to 1MB to make it cache sensitive. For the Denver VM, two `latency` tasks are run on two different VCPUs. Then, we measure the execution time of the task running on the first VCPU (v_1) to capture cache interference caused by other tasks. When cache partitioning and allocation are not used (Baseline), all the four tasks share the 32 cache partitions of the corresponding cluster. Next, to check the effect of cache isolation, we manually assigned 31 private cache partitions to the task on v_1, and let the three other tasks share the remaining 1 partition.

The execution time results of the task on v_1 are shown in Figure 5. The results are normalized to the case when it runs in isolation with all 32 cache partitions. As can be seen, the amount of cache interference is significant. We observed almost 5× increase in task execution time on the Cortex A57 cluster running at 2 GHz and 3× increase on the Denver cluster at 2 GHz. In other words, when the system is suffering from cache interference due to the lack of cache isolation, it may be better to use just a single CPU core rather than turning on other cores! With cache partitioning and allocation, the execution time is almost unaffected; only 1% increase on A57 and 9% on Denver. These results of this experiment clearly show that cache interference can be very significant in clustered multi-core platforms like Nvidia TX2, and cache partitioning and allocation are effective in preventing cache interference in real-time virtualization.

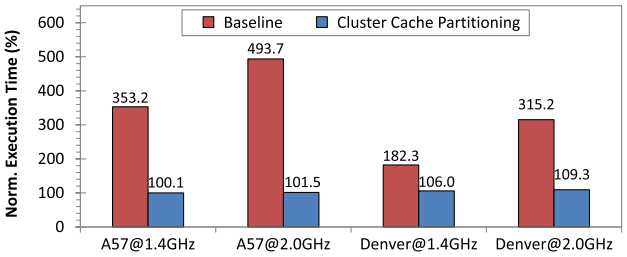

FIGURE 5. Execution time of a cache-sensitive task on Nvidia TX2 while other interfering tasks are executing on different cores simultaneously.

TABLE 1. Parameters for taskset generation.

Type	Parameters	Values
System	Number of clusters	2
	Number of VCPUs per cluster	4
	LLC size per cluster	2 MB
	Total memory size (M_{total})	4096 MB
	Cache partitions per cluster	32
	Cache partition reload time (Δ)	207 μsec
Taskset	Total number of tasks	[20, 30]
	Taskset utilization	7.0
	Inter-VM crit.-sec. length (E_i/C_i)	[1, 10] %
WCET	Memory accesses per job	[100k, 1m]
	Neighborhood size	[16, 64]
	Locality	[1.5, 3.0]
	Task memory usage	[8, 40] MB
	*Resulting WCET	[8.47, 202.02] msec

B. BENEFIT OF PROPOSED ALGORITHM

1) TASKSET GENERATION

We used randomly-generated tasksets to evaluate the benefit of our cache allocation scheme. Table 1 shows the base parameters used in the experiment. These parameters are based on those used in other prior work [17], [21], [34]. To generate a WCET function ($C_i(k)$) for each task τ_i, we use the method given in [8]. This method first calculates a cache miss rate for given cache size, neighborhood size, locality, and task memory usage, by using the analytical cache behavior model proposed in [39]. It then generates an execution time with the calculated cache miss rate, the timing delay of a cache miss, and the number of memory accesses. With this method, we were able to generate WCETs with different cache sensitivities. Then, the total taskset utilization is split into n random-sized pieces, where n is the total number of tasks. The size of each piece represents the utilization of the corresponding task when one cache color is assigned to it. The period of a task τ_i is calculated by dividing $C_i(1)$ by its utilization. Once a taskset is generated, they are evenly

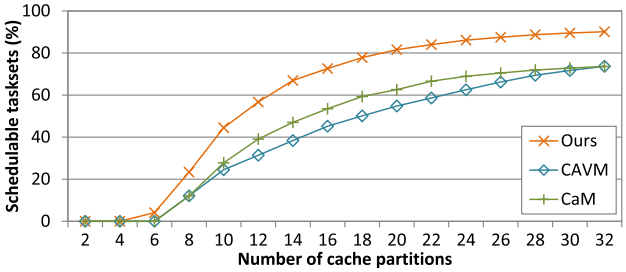

FIGURE 6. Percentage of schedulable tasksets as the number of cache partitions increases.

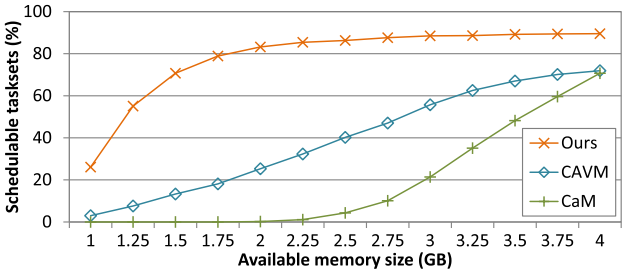

FIGURE 7. Percentage of schedulable tasksets with respect to the size of available system memory.

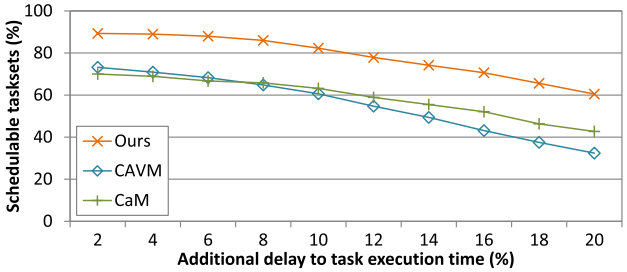

FIGURE 8. Percentage of schedulable tasksets in the presence of additional delay in task execution time.

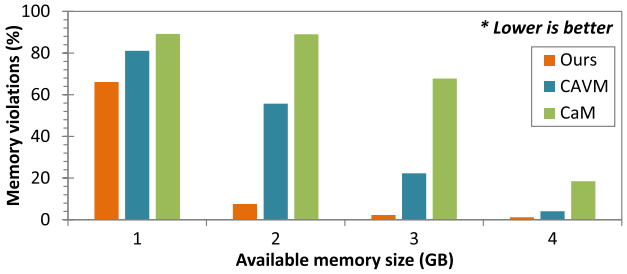

FIGURE 9. Percentage of tasksets with memory violations.

distributed to two clusters, each of which has four VCPUs. The priorities of tasks are assigned by the Rate-Monotonic (RM) [29] scheduler. For tasks using inter-VM data communication, we considered one critical section per task and assumed a single shared memory region for all such tasks to impose higher blocking delay.

2) RESULTS

We compared our cluster-aware scheme with two recent cache allocation schemes, CAVM [21] and CaM [51], which are both cluster-unaware. It is worth noting that the two existing schemes also have other features, e.g., VM parameter design in CAVM and bandwidth allocation in CaM, but we limit our focus to their cache allocation part. While implementing CAVM and CaM, we made three changes to enable comparison with our work. First, since CAVM and CaM do not consider task memory usage, we implemented them such that they check the memory requirements of tasks by using Algorithm 3 after completing the entire allocation process. Second, schedulability during cache allocation is checked by Eq. (1). Third, tasks are pre-allocated to VCPUs based on the worst-fit decreasing (WFD) heuristic, and in accordance with our system model, they cannot be moved to other VCPUs during cache allocation. Besides, since CAVM and CaM do not consider inter-VM communication and critical sections, we used only the tasks with no critical section here. Experimental results with tasks using inter-VM communication will be shown later.

Figures 6, 7, and 8 illustrate the comparative results of the cluster-aware (Ours) and the two cluster-unaware schemes (CAVM and CaM) under different settings. In each graph, the y-axis indicates the percentage of schedulable tasksets[2] (higher is better), and the results are obtained using the schedulability analysis given by Eg. (1). As can be seen, our cluster-aware scheme significantly outperforms the cluster-unaware approach in all three cases. The main reason for such a large difference is that our scheme utilizes the LLCs of the given clusters efficiently by using the dynamic programming approach given in Eq. (5). The improvement is particularly high in Figure 7, with upto 84% point and 60% point higher schedulability than CaM and CAVM, respectively. This is due to that memory budget is tight in this figure, but our scheme keeps track of memory usage during cache allocation and find an allocation that does not lead to memory violation. The issue of memory violation will be presented later with another experiment. Figure 8 considers the additional delay to task execution time, which may happen due to various reasons such as unexpected overload and imprecise WCET estimation. In this figure, the amount of delay on the x-axis is applied to the highest-priority task in each taskset. As the amount of delay increases, the difference in schedulability between ours and the other schemes becomes larger. This is because the slack time made by our scheme adds tolerance to such additional delay. We will assess the slack time achieved by our work in later experiments.

The proposed cluster-aware scheme is designed to tackle the cache co-partitioning problem that causes cache underutilization and memory violation issues. In order to evaluate the effectiveness of the proposed scheme, the two issues need to be checked. We first show in Figure 9 the percentage of tasksets that experience memory violations (lower is better). Since memory violation is one of the two reasons to make a taskset unschedulable (the other is deadline viola-

[2]A schedulable taskset means all tasks in this taskset are guaranteed to meet their deadlines.

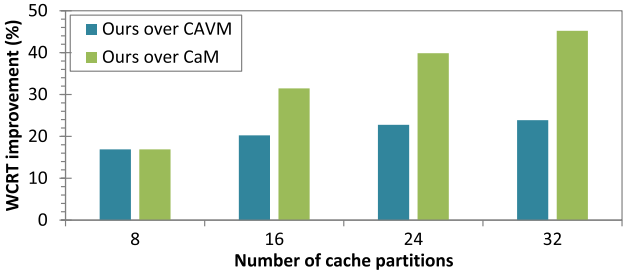

FIGURE 10. Reduction in the worst-case response time by our scheme.

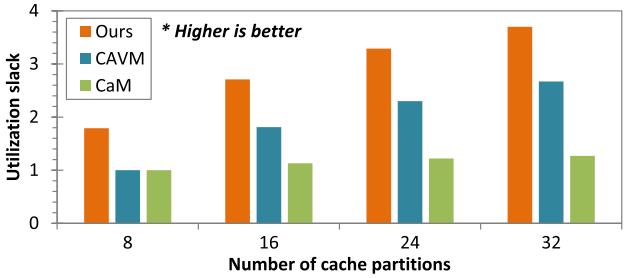

FIGURE 11. Utilization slack with respect to the number of cache partitions.

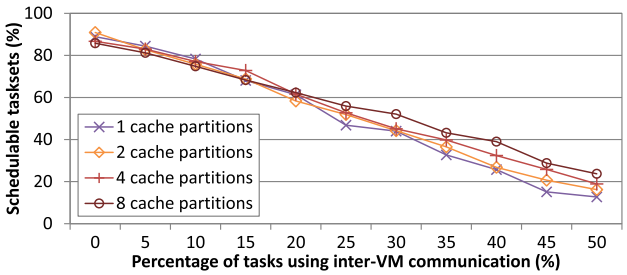

FIGURE 12. Percentage of schedulable tasksets where tasks use inter-VM data communication ($N_{cache} = 32$).

tion), the results of this figure can be compared with those in Figure 7 to understand the negative impact of memory violation. As can be observed from the two figures, the majority of unschedulable tasksets in Figure 7 are due to memory violations and our scheme achieves higher schedulability by significantly reducing the occurrence of memory violations.

Unlike memory violation, the exact degree of cache utilization is difficult to measure. If we simply compare the number of cache partitions used by tasks, the proposed scheme can be reported to outperform CAVM and CaM by L times, where L is the number of clusters. This is because CAVM and CaM do not distinguish cache partitions with the same partition index from different clusters, and treat them as the same cache partition. Even if the same number of cache partitions are used, the resulting effect may vary depending on how the cache partitions are assigned to tasks.

We therefore introduce two metrics that can indirectly check the effectiveness of our work in cache utilization. The first metric is the improvement of our scheme in the worst-case response time (WCRT) over CAVM and CaM. This can be checked by comparing the WCRT of each task under all three schemes and averaging the WCRT reduction of our scheme. The second is the amount of utilization slack, which can be obtained by subtracting the final taskset utilization from the given processor utilization. Note that both metrics should be measured only from the tasksets that are schedulable by all the schemes because it is not possible to compare correct WCRT and utilization values for unschedulable tasksets. The results for these two metrics are shown in Figures 10 and 11 (higher is better in both). The WCRT improvement and utilization slack of our scheme become larger as more cache partitions are available. This means that our scheme utilizes cache partitions more effectively than the other schemes, thereby contributing to solving the cache underutilization issue. Based the results, we conclude that our scheme yields substantial benefits in mitigating the cache co-partitioning problem in clustered multi-core systems.

Next, we have examined the impact of tasks using inter-VM communication under our scheme. The results are obtained using Eq. (6) and shown in Figure 12. Four different numbers of cache partitions are used for the inter-VM shared memory region and the corresponding critical sections. Assigning more cache partitions to inter-VM communication means that the execution time of critical sections can be reduced but fewer cache partitions are available for the normal execution segments of tasks. Hence, for example, with 8 cache partitions for inter-VM communication, the system yields the lowest schedulability among all the four cases when the percentage of inter-VM tasks is low, but the highest when the percentage of such tasks exceeds 25%.

The proposed scheme is developed primarily for use in static cache allocation. The allocation algorithm can run offline or during the initialization phase of the system, and the running time of the algorithm does not affect the runtime behavior of real-time tasks. Nonetheless, it is interesting to check the running time of the algorithm, which is reported in Figure 13. It was implemented in C++ with a single thread and the running time was measured on the Cortex A57 core of Nvidia TX2 running at 1.4GHz. As can be seen, while the running time increases with the number of tasks, it is acceptably small even in an embedded platform. Thus, it can be potentially applicable to runtime systems.

VI. RELATED WORK

A. CACHE MANAGEMENT

With cache partitioning, system performance is largely affected by how cache partitions are allocated to tasks. This has motivated developing cache allocation algorithms for various objectives, such as multi-core scheduling [43], [50], [51], non-preemptive tasks [15], [33], low-power management [13], and mixed-criticality systems [40], [45], [46]. Specifically, [51] is the latest work on cache allocation for real-time tasks scheduled in multi-core systems. It presents a variant of the bin-packing heuristics to allocate tasks and memory bandwidth to cores, in addition to cache allocation. While it uses the same scheduling policy

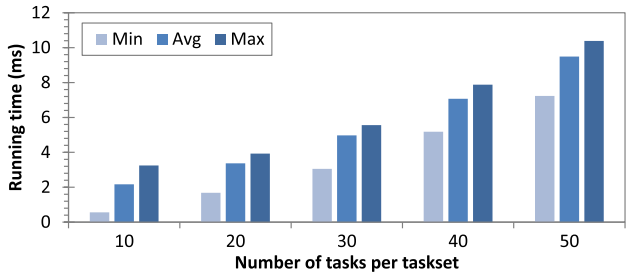

FIGURE 13. Running time of the proposed scheme on TX2 (A57@1.4GHz).

(partitioned scheduling) as in our work, it does not aim to minimize response time and to tackle the memory requirements of tasks. Hence, a system can be fragile to additional delays and suffer from memory violations, as shown in our experimental results. More importantly, it does not consider the cache co-partitioning problem in clustered multi-core systems.

Many researchers have studied system software issues in cache management. Wolfe [41] and Liedtke *et al.* [28] used page coloring to for uniprocessor real-time systems. Bui *et al.* [8] focused on improving task schedulability when page coloring is used for uniprocessor systems. Mancuso *et al.* [31] proposed the Colored Lockdown technique that combines page coloring and cache lockdown to better keep the frequently accessed pages of tasks in a cache. Page coloring has been also used for general-purpose multi-core systems [11], [42]. Valsan *et al.* [49] studied additional delay caused by miss status holding registers in non-blocking caches. Kim *et al.* [48] considered cache interference caused by shared libraries and proposed to replicate some of the libraries which are read only. However, due to the nature of replication, it is not applicable to shared data regions in our work.

While all the aforementioned approaches (both allocation algorithms and systems support) have been developed for a non-virtualized environment, there also exist some studies for a virtualized environment. Researchers [27], [30], [47] have developed software-based cache management in a hypervisor to allocate cache partitions to virtual machines, but their approaches cannot perform task-level cache allocation. Motivated by this, Kim and Rajkumar [21] proposed vLLC and vColoring, which enable task-level cache allocation in a virtualization environment. All of these approaches, however, are cluster-unaware approaches, meaning that they are subject to the cache co-partitioning problem. Xu *et al.* [52] developed cache-aware compositional analysis for virtualized components with hybrid EDF scheduling. It assumes each core has a private cache; hence, it can potentially benefit from the cache allocation of our work. vCAT [53] is a dynamic cache allocation technique for virtualization based on Intel's Cache Allocation Technology (CAT). Our work is different in that it is independent of specific hardware features and applicable to most embedded processors like ARM.

This paper addresses the above limitations by extending our own prior efforts [17], [21], [23], [38]. We clarify the technical differences between this paper and our earlier work. Reference [17] focuses on cache partitioning in a non-clustered multi-core system with no virtualization. It presents a bin-packing heuristic to allocate caches to tasks and tasks to cores. On the other hand, this work presents a cache allocation algorithm based on dynamic programming and does not rely on the bin-packing heuristic. Reference [21] proposes hypervisor-level cache allocation techniques, and [23] is an extended journal version with additional discussions on implementations and experiments. However, they do not consider the cache co-partitioning problem of a clustered multi-core system and the cache interference issue of shared-memory-based inter-VM communication, which are the main concerns of this paper. Reference [38] presents techniques to partition both shared cache and DRAM banks in a non-clustered, no-hypervisor environment. For cache allocation, it uses complete partitioning, meaning that even the tasks on the same core do not share any cache partition and suffer from low cache utilization. The disadvantage of the complete cache partitioning has been reported in [21], [46], and this work allows cache sharing among tasks on the same core at the cost of cache-related preemption delay.

B. LOCKING AND TASK SYNCHRONIZATION

Real-time locking and task synchronization have been extensively studied in the literature. The Multiprocessor Priority Ceiling Protocol (MPCP) [26], [35], [36] is a well-known technique to offer bounded blocking time on accessing shared resources in partitioned fixed-priority multi-core systems. MPCP has been extended to virtualization [24] and access control for shared hardware accelerators, such as graphics processing units (GPUs) [7], [20], [34]. The Multiprocessor Stack-based Resource Policy (MSRP) [14] is an extension of the uniprocessor SRP [5] for resource sharing under partitioned EDF scheduling. The Flexible Multiprocessor Locking Protocol (FMLP) [7] supports both partitioned and global EDF scheduling. There also exist prior studies conducted for a hierarchical scheduling environment. The Hierarchical Stack Resource Policy (HSRP) [12] is developed for uniprocessor systems and uses budget overrun and payback mechanisms to limit priority inversion. The Rollback Resource Policy (RRP) [4] uses a rollback mechanism to avoid a lock-holding task to be blocked while holding a lock. In the context of virtualization, vMPCP [24] is the first technique developed for real-time tasks running in multi-core virtual machines. However, all these techniques have assumed no cache interference, which is addressed in our work with the consideration of clustered architectures. Specifically, our work extends the schedulability analysis of [24], [26] to capture cache interference in critical sections and evaluates the impact of cache allocation for shared-memory-based inter-VM communication.

VII. CONCLUSION

In this paper, we proposed a real-time cache management scheme for partitioning hypervisors in clustered multi-core

systems. Our work is motivated by the two critical issues: cache underutilization and memory violation caused by the co-partitioning of last-level caches in clustered architectures, and cache interference caused by shared memory regions in inter-VM communication. Our scheme simultaneously addresses these two issues, and provides bounded response time and minimized slack time. We have examined the impact of cache interference on a recent Nvidia TX2 platform and evaluated the effect of our scheme.

Intelligent autonomous platforms, such as unmanned aerial vehicles (UAVs), are expected to be more prevalent in future safety-critical application domains. Embedded hypervisor technologies coupled with predictable real-time resource management can be effectively used for developing complex embedded systems with safety and reliability guarantees. We are currently developing a new type-1 partitioning hypervisor, called EARTH, with a particular focus on its use in safety-critical UAV platforms. The proposed cluster-aware cache management techniques will be integrated into this hypervisor and the practical aspects of the hypervisor will be investigated in the context of UAVs with self-adjusting maneuvers. There also exists plenty of interesting research topics in this area, such as GPU, power and thermal management, and we plan to tackle these issues in the future.

REFERENCES

[1] *Jailhouse: Linux-Based Partitioning Hypervisor*. Accessed: Aug. 1, 2019. [Online]. Available: https://github.com/siemens/jailhouse

[2] *Mentor Embedded Hypervisor*. Accessed: Aug. 1, 2019. [Online]. Available: http://www.mentor.com/embedded-software/hypervisor/

[3] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Evaluation of cache partitioning for hard real-time systems," in *Proc. 26th Euromicro Conf. Real-Time Syst.*, Jul. 2014, pp. 15–26.

[4] M. Åsberg, T. Nolte, and M. Behnam, "Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2013, pp. 129–140.

[5] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[6] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions," Polytech. Inst. Porto, Porto, Portugal, Tech. Rep. CISTER-TR-150713, 2015.

[7] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proc. IEEE Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2007, pp. 47–56.

[8] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *Proc. 14th IEEE Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2008, pp. 101–110.

[9] A. Burtsev, K. Srinivasan, P. Radhakrishnan, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2009, pp. 1–14.

[10] J. V. Busquets-Mataix, J. J. Serrano, and A. Wellings, "Hybrid instruction cache partitioning for preemptive real-time systems," in *Proc. 9th Euromicro Workshop Real-Time Syst. (ECRTS)*, Jun. 1997, pp. 56–63.

[11] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2006, pp. 455–468.

[12] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2006, pp. 257–270.

[13] X. Fu, K. Kabir, and X. Wang, "Cache-aware utilization control for energy efficiency in multi-core real-time systems," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, Jul. 2011, pp. 102–111.

[14] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, May 2003, pp. 189–198.

[15] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. Int. Conf. Embedded Softw. (EMSOFT)*, 2009, pp. 245–254.

[16] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2014, pp. 145–154.

[17] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, Jul. 2013, pp. 80–89.

[18] H. Kim, J. Kim, and R. Rajkumar, "A profiling framework in Linux/RK and its application," in *Proc. Open Demo Session IEEE Real-Time Syst. Symp. (RTSSWork)*, 2012, p. 1.

[19] H. Kim, P. Patel, S. Wang, and R. Rajkumar, "A server-based approach for predictable GPU access control," in *Proc. 23rd Int. IEEE Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2017, pp. 1–10.

[20] H. Kim, P. Patel, S. Wang, and R. Rajkumar, "A server-based approach for predictable GPU access with improved analysis," *J. Syst. Archit.*, vol. 88, pp. 97–109, Aug. 2018.

[21] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, Oct. 2016, pp. 1–10.

[22] H. Kim and R. Rajkumar, "Virt/RK: A real-time virtualization framework for multi-core platforms," in *Proc. Open Demo Session IEEE Real-Time Syst. Symp. (RTSS Work)*, Dec. 2015, p. 1.

[23] H. Kim and R. Rajkumar, "Predictable shared cache management for multi-core real-time virtualization," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, 2018, Art. no. 22.

[24] H. Kim, S. Wang, and R. Rajkumar, "vMPCP: A synchronization framework for multi-core virtual machines," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2014, pp. 86–95.

[25] T. Kim, D. Kang, S. Kim, J. Shin, D. Lim, and V. Dupre, "Qplus-hyper: A hypervisor for safety-critical systems," in *Proc. 9th Int. Symp. Embedded Technol. (ISET)*, 2014, pp. 102–103.

[26] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2009, pp. 469–478.

[27] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proc. ACM Conf. Virtual Execution Environ. (VEE)*, 2014, pp. 201–212.

[28] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Jun. 1997, pp. 213–224.

[29] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[30] R. Ma, W. Ye, A. Liang, H. Guan, and J. Li, "Cache isolation for virtualization of mixed general-purpose and real-time systems," *J. Syst. Archit.*, vol. 59, no. 10, pp. 1405–1413, 2013.

[31] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Proc. 19th IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2013, pp. 45–54.

[32] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 1983.

[33] M. Paolieri, E. Quiñones, F. J. Cazorla, R. I. Davis, and M. Valero, "IA^3: An interference aware allocation algorithm for multicore hard real-time systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2011, pp. 280–290.

[34] P. Patel, I. Baek, H. Kim, and R. Rajkumar, "Analytical enhancements and practical insights for MPCP with self-suspensions," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2018, pp. 177–189.

[35] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. 10th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, May/Jun. 1990, pp. 116–123.

[36] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 1988, pp. 259–269.
[37] C. Shin, C. Lim, J. Kim, H. Roh, and W. Lee, "A software-based monitoring framework for time-space partitioned avionics systems," *IEEE Access*, vol. 5, pp. 19132–19143, 2017.
[38] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *Proc. IEEE 16th Int. Conf. Comput. Sci. Eng.*, Dec. 2013, pp. 685–692.
[39] D. Thiebaut, J. L. Wolf, and H. S. Stone, "Synthetic traces for trace-driven simulation of cache memories," *IEEE Trans. Comput.*, vol. 41, no. 4, pp. 388–410, Apr. 1992.
[40] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Proc. 25th Euromicro Conf. Real-Time Syst. (ECRTS)*, 2013, pp. 157–167.
[41] A. Wolfe, "Software-based cache partitioning for real-time applications," *J. Comput. Softw. Eng.*, vol. 2, no. 3, pp. 315–327, 1994.
[42] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Aug. 2014, pp. 381–392.
[43] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems," in *Proc. 32nd IEEE Real-Time Syst. Symp. (RTSS)*, Nov./Dec. 2011, pp. 227–238.
[44] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2014, pp. 155–166.
[45] M. A. Awan, K. Bletsas, P. F. Souto, B. Akesson, and E. Tovar, "Mixed-criticality scheduling with dynamic redistribution of shared cache," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, 2017, pp. 1–26.
[46] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson, "Cache sharing and isolation tradeoffs in multicore mixed-criticality systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2015, pp. 305–316.
[47] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, "A simple cache partitioning approach in a virtualized environment," in *Proc. IEEE Symp. Parallel Distrib. Process. Appl. (ISPA)*, 2009, pp. 519–524.
[48] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith, "Allowing shared libraries while supporting hardware isolation in multicore real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2017, pp. 223–234.
[49] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, Apr. 2016, pp. 1–12.
[50] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee, "Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2016, pp. 1–12.
[51] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2019, pp. 345–356.
[52] M. Xu, L. T. X. Phan, O. Sokolsky, S. Xi, C. Lu, C. Gill, and I. Lee, "Cache-aware compositional analysis of real-time multicore virtualization platforms," *Real-Time Syst.*, vol. 51, no. 6, pp. 675–723, 2015.
[53] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vCAT: Dynamic cache management using CAT virtualization," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2017, pp. 211–222.

YOOJIN LIM received the B.S. degree in computer science and computer engineering and the M.S. degree in information technology from Handong Global University, South Korea, in 2004, and the Ph.D. degree in business information technology from Kookmin University, South Korea, in 2016. From 2004 to 2008, he was a Senior Software Engineer with LG Electronics, South Korea. From 2008 to 2016, he was a Research Fellow, a Lecturer, and a Research Assistant with Kookmin University. Since 2017, he has been a Postdoctoral Researcher with the SW Contents Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, South Korea. His current research interests include embedded real-time software, hypervisor, cyber-physical systems, and big data systems.

HYOSEUNG KIM received the B.S. and M.S. degrees in computer science from Yonsei University, South Korea, in 2005 and 2007, respectively, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, in 2016. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of California at Riverside. His research interests include the design, implementation, and evaluation of real-time embedded and cyber-physical systems that offer robustness, predictability, and reliability guarantees in uncertain environments. His research contributions have been recognized with Best Paper Awards at the IEEE Real-Time Embedded Technology and Applications Symposium (RTAS), in 2014, and the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), in 2017, and Best Paper Nominations at the ACM SIGBED International Conference on Embedded Software (EMSOFT), in 2016, and the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS), in 2013. More details can be found at http://www.ece.ucr.edu/hyoseung/.