Real-Time Cache Management for Multi-Core Virtualization

Hyoseung Kim[1,2] and Ragunathan (Raj) Rajkumar[2]
[1] University of California, Riverside
[2] Carnegie Mellon University

ABSTRACT

Real-time virtualization techniques have been investigated with the primary goal of consolidating multiple real-time systems onto a single hardware platform while ensuring timing predictability. However, a shared last-level cache (LLC) on recent multi-core platforms can easily hamper timing predictability due to the resulting temporal interference among consolidated workloads. Since such interference caused by the LLC is highly variable and may have not even existed in legacy systems to be consolidated, it poses a significant challenge for real-time virtualization. In this paper, we propose a real-time cache management framework for multi-core virtualization. Our framework introduces two hypervisor-level techniques, vLLC and vColoring, that enable the cache allocation of individual tasks running in a virtual machine (VM), which is not achievable by the current state of the art. Our framework also provides a cache management scheme that determines cache allocation to tasks, designs VMs in a cache-aware manner, and minimizes the aggregated utilization of VMs to be consolidated. As a proof of concept, we implemented vLLC and vColoring in the KVM hypervisor running on x86 and ARM multi-core platforms. Experimental results with three different guest OSs, namely Linux/RK, vanilla Linux and MS Windows Embedded, show that our techniques can effectively control the cache allocation of tasks in VMs. Our cache management scheme yields a significant utilization benefit compared to other approaches.

1. INTRODUCTION

With the growth of processing core counts on recent processors, there is a strong demand for consolidating multiple real-time systems onto a single hardware platform. One of the promising solutions for such consolidation is virtualization. With virtualization, each consolidated system is contained within a virtual machine (VM), which is spatially isolated from other VMs by an additional address translation layer introduced by a hypervisor. Figure 1 illustrates the three address layers in modern virtualization platforms, such as Xen [6] and KVM [20]. Guest virtual pages for application tasks within a VM are mapped to *guest physical pages* by the guest OS of that VM, and those guest physical pages are mapped to *host physical pages* by the hypervisor. Using this approach, the hypervisor ensures that any software failure in one VM does not propagate to other VMs.

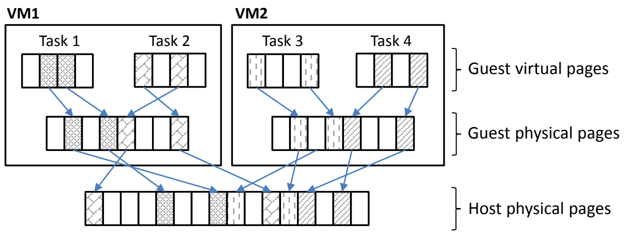


Figure 1: Address translation layers in virtualization

The foremost requirement for real-time system virtualization is ensuring timing predictability. Hierarchical real-time scheduling theory [12, 31, 35, 36, 46] and its implementations [18, 45, 22] have established a good foundation for this requirement. However, shared hardware resources on recent multi-core platforms, such as a last-level cache (LLC) and a memory bus, have not been thoroughly considered in the context of real-time virtualization. Since contention in those resources can cause significant temporal interference among consolidated workloads, the requirement of timing predictability cannot be fully satisfied without considering their effects. In this paper, we focus on the predictable management of a shared LLC in a virtualization environment.

Many previous approaches developed for non-virtualized multi-core systems [16, 28, 40, 43] use *page coloring* to address "cache interference", which is the temporal interference caused by a shared LLC. Page coloring is an OS-level technique to control the cache allocation of tasks in software, by assigning physical pages corresponding to specific cache sets to the tasks. Since page coloring does not require any hardware feature beyond that available on most of today's processors, it is considered as a practical technique. However, page coloring and cache allocation algorithms based on it cannot function properly in a VM due to the additional address layer shown in Figure 1. Although a guest OS selects guest physical pages for page coloring, those pages may be mapped to host physical pages corresponding to cache sets different from the ones intended by the guest OS, resulting in unpredictable cache allocation. Even if page coloring works in a VM, tasks running on other guest OSs that do not support page coloring will suffer from cache interference. Also, cache allocation algorithms developed for non-virtualized systems cannot provide an efficient solution to design a VM in the presence of cache interference and to allocate the host machine's LLC to VMs to be consolidated.

In this paper, we propose a real-time cache management framework for multi-core virtualization. To address the problem of cache-to-task allocation in a VM, our framework supports two new hypervisor-level techniques, named vLLC and vColoring. vLLC is designed for a VM that runs a guest OS with page coloring support. vLLC provides such a VM with a portion of the host machine's LLC in the form of a *virtual LLC*. Then, vLLC enables the guest OS to control the virtual

EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4485-2/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968478.2968480

LLC by using its own implementation of page coloring. vColoring, on the other hand, is designed for a VM that runs a guest OS having no page coloring support. vColoring allows the hypervisor to directly assign a portion of the host LLC to a task running in a VM. Hence, with vColoring, we can even control the cache allocation of tasks running on proprietary, closed-source OSs that do not support page coloring. We have implemented prototypes of vLLC and vColoring in the KVM hypervisor running on x86 and ARM multi-core platforms. Experimental results show that vLLC and vColoring are effective in controlling cache allocation to tasks and in addressing cache interference, on both an OS with page coloring (Linux/RK [16, 29]) and OSs without page coloring (vanilla Linux and MS Windows Embedded).

In addition, we propose a new cache management scheme as part of our framework. Our scheme determines a cache-to-task allocation that reduces taskset utilization while satisfying timing constraints. Our scheme also designs a VM in a way that the VM's resource requirement is captured with respect to the number of cache colors allocated. Lastly, when VMs are consolidated into the host machine, our scheme finds a cache-to-VM allocation that minimizes the total VM utilization. We use randomly-generated tasksets for the evaluation of our cache management scheme. Experimental results indicate that our scheme yields a significant benefit in VM utilization over other approaches.

The rest of this paper is organized as follows. Section 2 reviews the background for our paper. Section 3 describes our system model. Section 4 presents our vLLC and vColoring techniques. Section 5 presents our cache management scheme. Section 6 provides detailed evaluation. Section 7 reviews related work, and Section 8 concludes the paper.

2. BACKGROUND

In this section, we give a brief description on scheduling, cache interference, address translation in a virtualization environment, and discuss the page coloring technique.

2.1 Scheduling and Cache Interference

Virtualization generally features a two-level hierarchical scheduling structure. Each VM has one or more virtual CPUs (VCPUs), each of which is represented as a processing core to a guest OS. Tasks in a VM are scheduled on the VCPUs of that VM by the guest OS, and VCPUs are scheduled on physical CPUs (PCPUs) by the hypervisor. Note that a VCPU is the smallest schedulable entity in the hypervisor, analogous to a task in an OS. Hence, the hypervisor can execute only one VCPU on each PCPU at a time.

Cache interference among tasks in a multi-core virtualization environment can be categorized into two types: *inter-VCPU* and *intra-VCPU*. Inter-VCPU cache interference happens among tasks running on different VCPUs. Since those VCPUs can be scheduled on different PCPUs by the hypervisor, tasks on different VCPUs may access the LLC simultaneously. In addition, when a VCPU preempts another VCPU, the cache contents of tasks on the preempted VCPU may be evicted by tasks on the preempting VCPU. Intra-VCPU cache interference happens among tasks running on the same VCPU. Although tasks on the same VCPU cannot access the LLC simultaneously, a task preempting another task may evict the cache contents of the preempted task.

2.2 Address Translation

There are three types of addresses in a virtualized environment: guest virtual addresses (GVA), guest physical address (GPA), and host physical address (HPA). Whenever a GVA is accessed, it needs to be translated to the corresponding HPA. Shadow paging and two-dimensional paging are techniques to do such translation in *full virtualization* scenarios, where unmodified guest OSs can be used.

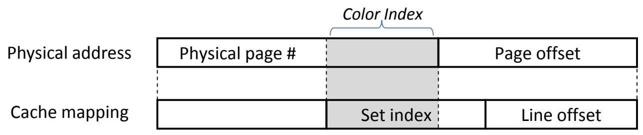


Figure 2: Page coloring

Shadow paging: Under shadow paging, the hypervisor generates shadow page tables where GVAs are directly mapped to HPAs. Although a guest OS still maintains its own page tables, the memory management unit (MMU) uses the shadow page tables for address translation so that a GVA can be directly translated to its corresponding HPA without having GVA-to-GPA translation. To maintain the validity of contents of the shadow page tables, the hypervisor has to keep track of any change in the guest page tables. A well-known approach to doing this is to write-protect the guest page tables, which triggers a page-fault exception to the hypervisor whenever any change is made to the guest page tables.

Two-dimensional paging: Two-dimensional paging refers to hardware-assisted address translation techniques introduced in recent processors, e.g., AMD Nested Page Tables (NPT), Intel Extended Page Tables (EPT), and ARM Stage-2 Page Tables. Under two-dimensional paging, the MMU can traverse both guest and host page tables. Hence, when a GVA is accessed, the MMU first translates it to a GPA by using the guest page tables and then translates that GPA to an HPA by using the host page tables. Such two-step address translation requires more memory accesses than the direct GVA-to-HPA translation of shadow paging, but it eliminates the overhead of maintaining valid shadow page tables.

Neither shadow paging nor two-dimensional paging dominates the other in terms of performance [42]. It is also currently unknown which technique is preferable for real-time virtualization. Therefore, one of our goals in this paper is to develop a cache allocation technique that is independent of a specific address translation technique used.

2.3 Page Coloring

Page coloring is a software technique to control a physically-indexed set-associative cache, which is the case for most LLCs on modern processors. On a physically-indexed cache, page coloring uses the mapping between physical addresses and cache set indices. As shown in Figure 2, there are overlapping bits between the physical page number and the cache set index. Those overlapping bits are used as a color index by page coloring. Since the OS has direct control over the mapping between physical pages and the virtual pages of an application task, it can allocate specific cache colors to a task by providing the task with physical pages corresponding to the cache colors.

The number of cache colors available in the system is calculated as follows: $n = S/(W \times P)$, where n is the number of cache colors, S is the cache size, W is the number of ways of the cache, and P is the size of a page frame and is typically 4KB. Hence, if $S = 256$KB, $W = 16$ and $P = 4$KB, the number of cache colors n is 4. One implicit assumption in page coloring is that the number of cache sets is a power of two. In some architectures like Intel Sandy Bridge and Haswell, the LLC consists of cache slices, the number of which is equal to that of physical cores [13, 21]. As shown in [16, 47], although the mapping between physical addresses and cache slices is not publicly known, page coloring on such architectures can be implemented on a per cache-slice basis. This results in the number of cache colors equal to $n = S/(W \times P \times N_P)$, where N_P is the number of physical cores.

Page coloring was originally developed for a non-virtualized system. In a virtualized system, page coloring implemented in a guest OS can no longer map a task's virtual page to a specific cache color due to the additional address translation at the hypervisor. One simple approach to consider is to implement page coloring in the hypervisor and assign cache colors to VMs, as proposed in [22, 27, 34]. However, this approach cannot allocate cache colors to individual tasks running in a VM. In other words, all tasks within the same VM share the cache colors assigned to the VM and will suffer from inter- and intra-VCPU cache interference.

3. SYSTEM MODEL

We consider a multi-core host machine, where each PCPU runs at the same fixed clock frequency and the last-level cache (LLC) is shared among all PCPUs. The host machine runs a hypervisor hosting guest VMs in full-virtualization mode. The hypervisor implements page coloring and partitions the LLC into cache colors. Each cache color is represented as a unique natural number. Guest OSs may or may not have page coloring. We assume that each VM has been allocated a sufficient number of host physical pages and that page swapping does not happen at run-time. This is a reasonable assumption in real-time virtualization scenarios because, unlike in server virtualization, memory underprovisioning is considered to be harmful to timing predictability [19]. Also, this assumption can be easily achieved by VM admission control at the hypervisor.

Scheduling: We focus on *partitioned fixed-priority preemptive scheduling* for both the hypervisor and guest OSs due to its wide usage, such as in OKL4 [3] and PikeOS [4]. Thus, each VCPU is statically assigned to a single PCPU and each task is statically assigned to a single VCPU.

Task Model: We consider periodic tasks with constrained deadlines. Task τ_i is represented as follows:

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

- $C_i(k)$: the worst-case execution time (WCET) of τ_i, when it runs alone in the system with k cache colors assigned to it
- T_i: the period of τ_i
- D_i: the relative deadline of τ_i $(D_i \le T_i)$

$C_i(k)$ values are assumed to be known ahead of time. They can be obtained by either measurement-based or static analysis tools.[1] We assume that $C_i(k)$ is monotonically decreasing with k.[2] Each task τ_i has a unique priority π_i.[3] Tasks are assumed not to share any data with others. Also, tasks do not make dynamic memory allocation, since it is typically prohibitive in many real-time systems [28]. Relaxing those assumptions is part of our future work.

[1]Capturing the overhead of virtualization in task execution time is beyond the scope of this paper. However, we believe this does not limit the applicability of our work because its impact is relatively small (e.g., more than 99% of native performance can be achieved in full-virtualization mode with recent hardware virtualization techniques [38]).

[2]This is a common assumption in the literature. The actual WCET function may not be monotonic, but this assumption can be easily satisfied by monotonic over-approximations of WCETs with insignificant pessimism [5].

[3]An arbitrary tie-breaking rule can be used to achieve this under fixed-priority scheduling.

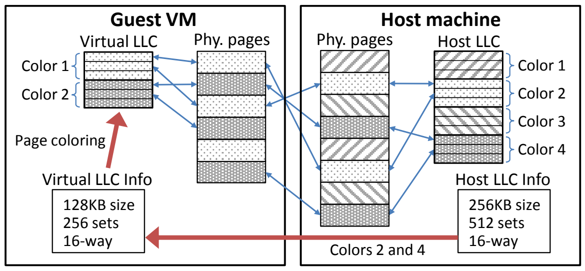


Figure 3: vLLC example

VM Resource Model: Each VM is represented as follows:

$$\text{VM} := (v_1, v_2, ..., v_{N_{vcpu}})$$

where v_i is a VCPU and N_{vcpu} is the number of VCPUs in the VM. We represent a VCPU v_i as follows:

$$v_i := (C_i^v(k), T_i^v)$$

- $C_i^v(k)$: the execution budget of a VCPU v_i, represented as a function of the total number of cache colors (k) assigned to the tasks of v_i
- T_i^v: the budget replenishment period of a VCPU v_i

Since task execution time is affected by the number of assigned cache colors, it is obvious that the required budget of a VCPU is also affected by the number of cache colors to be used by its tasks. With this model, the resource demand of each VM can be presented to the hypervisor and other VMs, without revealing its task attributes. We will show in Section 5 how to find the budget of each VCPU with respect to the number of cache colors. For the VCPU budget supply and replenishment policies, we consider *periodic server* [33], *sporadic server* [37], and *deferrable server* [39] variants, because they have been widely used in real-time virtualization [45, 22, 18, 17].

In the rest of the paper, C_i and C_j^v may be used instead of $C_i(k)$ and $C_j^v(k')$, respectively, when each task and VCPU is assumed to have been assigned its cache colors.

4. CACHE CONTROL IN VIRTUALIZATION

In this section, we present our vLLC and vColoring techniques. Both techniques provide a way to allocate cache colors to individual tasks running in a VM. They do not rely on the page-fault exception of shadow paging or the hardware support of two-dimensional paging. Our techniques differ in their target guest OSs: vLLC is for guest OSs with page coloring (coloring-aware OSs) and vColoring is for guest OSs without page coloring (coloring-unaware OSs).

4.1 vLLC for Coloring-aware Guest OSs

As discussed in Section 2.3, page coloring implemented in a guest OS cannot allocate cache colors to tasks in a VM due to the additional address layer in the hypervisor. vLLC overcomes this limitation. The keys to vLLC are (i) to provide a VM with "virtual LLC" information that corresponds to the cache colors assigned to the VM, and (ii) to map guest physical pages to host physical pages corresponding to the assigned cache colors. Figure 3 illustrates an example of vLLC. The virtual LLC provided to the VM is different from the actual LLC of the host machine in terms of the size of a cache and the number of cache sets, which are the main factors determining the number of cache colors. In Figure 3, since the

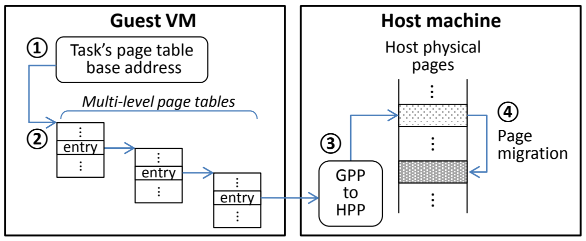


Figure 4: Steps for re-mapping GPPs to new HPPs

hypervisor assigns two colors out of four to the guest VM, the size and the number of cache sets of the virtual LLC are each half of those of the host LLC. Using this virtual LLC, the guest OS can identify that the number of available cache colors is two. The virtual LLC can be implemented by trapping and emulating cache-related operations, e.g., executions of a `CPUID` instruction on x86 architectures [2] and accesses to `CCSIDR` and `CSSERR` registers on an ARM Cortex-A15 architecture [1].

In addition to the virtual LLC information, vLLC maps guest physical pages (GPPs) to host physical pages (HPPs) such that guest colors are mapped to their corresponding host colors. This can be easily done by the hypervisor because the hypervisor has both the virtual LLC information and the control of the GPP-to-HPP mapping. When a GPP needs to be mapped to an HPP, vLLC in the hypervisor checks the guest color of the GPP, finds out the corresponding host color, and maps the GPP to an HPP with that host color. For instance, in Figure 3, Colors 2 and 4 of the host machine are represented as Colors 1 and 2 in the guest VM, respectively, and GPPs with Colors 1 and 2 are mapped to HPPs with Colors 2 and 4, respectively. With this approach, a guest OS can allocate cache colors to tasks. It is worth noting that the GPP-to-HPP mapping happens only once per GPP during the lifetime of a VM. Therefore, once all GPPs used by a task have been populated, vLLC does not cause any runtime overhead to that task.

There are two constraints in vLLC. First, virtual LLC information should be in accordance with the assumption of page coloring, where the number of cache sets is a power of two. This means that, with vLLC, the number of cache colors that can be assigned to a VM is restricted to a power of two. Second, it cannot support a guest OS where page coloring is hard-coded (e.g., using fixed cache parameters, instead of checking them when the system boots). If these constraints become a problem, one can disable the page coloring feature of the guest OS and use our vColoring technique.

4.2 vColoring for Coloring-unaware Guest OSs

With vColoring, a VM is assigned two sets of cache colors, *default* and *extra*. The default color set is used whenever a GPP needs to be mapped to an HPP. The hypervisor maps a GPP to an HPP corresponding to one of the colors in the default color set. Hence, by default, all tasks are constrained to use only the default cache colors. The extra color set is used for explicit color allocation requests. When a task running in a VM makes such a request, the hypervisor remaps all GPPs used by that task to HPPs corresponding to the requested colors in the extra color set.

Re-mapping GPPs to new HPPs: Figure 4 shows the detailed steps for re-mapping all the GPPs of a task from the currently-used HPPs to new HPPs for the requested colors. The first step is to obtain the task's page table base address (PTBA), which we will explain in detail later. Once the PTBA is obtained, the hypervisor can traverse the task's page tables that are maintained by the guest OS. The second step is to find out *present* and *user-level accessible* GPPs in the task's page tables. This can be done by checking the information bits of page table entries (PTEs). The third step is to find an HPP mapped to each of the GPPs found in the second step. The fourth step is to migrate each HPP obtained in the third step to a new HPP that corresponds to one of the requested colors. As part of page migration, references to the previous HPP are also updated to the new one. During all these steps, guest page tables are not changed at all. Therefore, the task can be assigned its requested colors transparent to the guest OS. Note that, since the above steps re-map GPPs present at that time, it is desirable to make a cache allocation request at the end of the initialization phase, where a real-time task typically initializes and places all the required data into memory.

PTBA identification: On most processors, the currently-executing task's PTBA is stored in a specific register to facilitate address translation, e.g., a CR3 register in x86 architectures and a Translation Table Base register in ARM architectures. We will refer to such a register as a PTB (Page Table Base) register. Under shadow paging, the hypervisor traps on write accesses to the PTB register and stores the base address of the corresponding shadow page table into the PTB register. The real PTBA value trapped by the hypervisor is stored in the hypervisor's memory space and used for synchronizing the shadow page table with the guest page table. Under two-dimensional paging, the MMU has two PTB registers, one for a guest PTBA and the other for a host PTBA, and the hypervisor has access to both registers. Therefore, under both address translation techniques, the current task's PTBA can be obtained by the hypervisor.

Cache allocation request: To make a cache allocation request to the hypervisor, on x86 architectures, a task can use a "hypercall" instruction. It can be executed by any user-level task in a VM and results in a world switch to the hypervisor [2]. Then, the hypervisor can easily get the task's PTBA because that task is the currently executing one, and the hypervisor can allocate requested colors to the task by following the re-mapping steps explained before. On other architectures, a user-level task is *not* allowed to execute a hypercall. Hence, we propose the inclusion of a simple driver that provides a user-level task with an interface to issue a hypercall. Then, the task can make a cache allocation request through the driver interface. Since many recent real-time OSs such as VxWorks [44] support implementing device drivers as loadable kernel modules, this approach can be easily used for such OSs without rebuilding the entire kernel image.

5. CACHE MANAGEMENT SCHEME

In this section, we present our cache management scheme which (i) allocates cache colors to tasks within a VM while satisfying timing constraints, (ii) designs a VM in a cache-aware manner so that the VM's resource requirement is specified w.r.t. the number of cache colors allocated, and (iii) determines the allocation of cache colors to a set of VMs to be consolidated.

Recall that, in multi-core virtualization, there are two types of cache interference: inter- and intra-VCPU cache interference. To avoid both types of interference, a simple approach would be assigning each task a dedicated set of cache colors for its own exclusive use. Hence, tasks do not share their assigned cache colors with others, resulting in no conflicts in the LLC. We will refer to this approach as *complete cache* *partitioning* (CCP). However, due to the availability of a limited number of cache colors, CCP may result in performance degradation. Many prior studies in non-virtualized environments [11, 10, 16, 5] have shown that sharing of cache colors among tasks on the same core yields better task schedulability than CCP, and the resulting cache interference can be safely upper-bounded by the notion of *cache-related preemption delay* (CRPD). Therefore, our scheme builds on this idea in that (i) cache colors are not shared among tasks on different VCPUs to prevent inter-VCPU cache interference, and (ii) cache colors can be shared among tasks on the same core with the cost of intra-VCPU cache interference.

5.1 Schedulability Analysis

Before presenting our scheme, we first review VCPU and task schedulability analyses. The schedulability of a VCPU v_i can be determined by the following recurrence equation [14]:

$$W_i^{v,n+1} = C_i^v + \sum_{v_h \in \mathbb{P}(v_i) \wedge \pi_h^v > \pi_i^v} \left\lceil \frac{W_i^{v,n} + J_h^v}{T_h^v} \right\rceil C_h^v \qquad (1)$$

where $W_i^{v,n}$ is the worst-case response time (WCRT) of a VCPU v_i at the n^{th} iteration ($W_i^{v,0} = C_i^v$), π_i^v is the priority of a VCPU v_i, $\mathbb{P}(v_i)$ is the PCPU of v_i, and J_h^v is a release jitter ($J_h^v = T_h^v - C_h^v$ for the deferrable server policy and $J_h^v = 0$ for the periodic and sporadic server policies [8]). It terminates when $W_i^{v,n+1} = W_i^{v,n}$, and the VCPU v_i is schedulable if its WCRT does not exceed its period, i.e., $W_i^{v,n} <= T_i^v$.

The schedulability of task τ_j running on a VCPU v_i can be determined by:

$$W_j^{n+1} = C_j + \sum_{\substack{\tau_h \in \mathbb{V}(\tau_j) \\ \wedge \pi_h > \pi_j}} \left\lceil \frac{W_j^n + J_h}{T_h} \right\rceil (C_h + \gamma_{h,j}) + \left\lceil \frac{W_j^n + C_i^v}{T_i^v} \right\rceil (T_i^v - C_i^v) \qquad (2)$$

where W_j^n is the WCRT of task τ_j at the n^{th} iteration ($W_j^0 = C_j$), π_j is the priority of τ_j, $\mathbb{V}(\tau_j)$ is the VCPU of τ_j, J_h is the release jitters of a task τ_h ($J_h = T_i^v - C_i^v$), and $\gamma_{h,j}$ is the cache-related preemption delay (CRPD) caused by τ_h and imposed on τ_j. Task τ_j is schedulable if its WCRT does not exceed its deadline, i.e., $W_j^n <= D_j$. Note that Eq. (2) is based on the task schedulability test under hierarchical scheduling given in [31] but extended with CRPD [11, 16] to bound intra-VCPU cache interference. $\gamma_{j,i}$ is given by:

$$\gamma_{j,i} = \left| \mathbb{S}_j \cap \bigcup_{\tau_k \in \mathbb{V}(\tau_i) \wedge \pi_k < \pi_j \wedge \pi_k \geq \pi_i} \mathbb{S}_k \right| \cdot \Delta \qquad (3)$$

where $\mathbb{S}_j$ is the set of cache colors assigned to τ_j, and Δ is the maximum time needed to reload data in one cache color.[4]

In the presence of intra-cache VCPU interference, the utilization of a taskset Γ allocated to the same VCPU is calculated as follows [7, 16]:

$$util(\Gamma) = \sum_{\tau_i \in \Gamma} \left(\frac{C_i}{T_i} + \frac{\gamma_{i,n}}{T_i} \right) \qquad (4)$$

where n is the index of the lowest-priority task in Γ.

5.2 Allocating Cache Colors to Tasks

Suppose that we have a set of tasks running on the same VCPU and a set of cache colors is to be allocated to the tasks. Our goal is to find a cache-to-task allocation that minimizes taskset utilization while satisfying taskset schedulability. When cache sharing is allowed, the problem of cache-to-task allocation is known to be NP-hard [10]. Hence, we present in Alg. 1 a heuristic to solve this problem. It first checks if N_{cache} is non-zero because page coloring requires tasks to be assigned at least one cache color [16]. Then, for each task τ_i, it finds the number of cache colors, S_i, that minimizes the sum of the utilization of and CRPD caused by τ_i (line 6). Since cache allocation is not done yet, we approximate $\gamma_{i,n}$ by assuming that all other tasks have been allocated all N_{cache} colors. Once the number of cache colors for τ_i is found, our heuristic finds cache color indices to be allocated (line 9). It records the index of the next cache color to be allocated in *cache_idx* and begins the allocation starting from *cache_idx*, with an increment of 1 and a modulo of N_{cache}. This approach ensures that the difference in the number of tasks sharing each color does not exceed 1.

Algorithm 1 CacheToTaskAlloc(Γ, N_{cache})

```
Input: $\Gamma$: taskset, $N_{cache}$: the number of cache colors
Output: Utilization of $\Gamma$ if schedulable, and $\infty$ otherwise
1: if $N_{cache} = 0$ then
2:     return $\infty$
3: $cache\_idx \leftarrow 1$
4: for all $\tau_i \in \Gamma$ do
5:     /* Find the number of cache colors for $\tau_i$ */
6:     $S_i \leftarrow \mathrm{argmin}_{1 \leq k \leq N_{cache}} (\frac{C_i(k)}{T_i} + \frac{\gamma_{i,n}}{T_i})$
7:     /* Find cache-color indices for $\tau_i$ */
8:     $\mathbb{S}_i \leftarrow \emptyset$
9:     for $k \leftarrow 1$ to $S_i$ do
10:        $\mathbb{S}_i \leftarrow \mathbb{S}_i \cup \{cache\_idx\}$
11:        $cache\_idx \leftarrow (cache\_idx + 1) \mod N_{cache}$
12: if $schedulable(\Gamma)$ then
13:     return $util(\Gamma)$
14: else
15:     return $\infty$
```

5.3 Designing a Cache-Aware VM

The resource requirement of a VM is the aggregate of the resource requirements of all VCPUs in that VM, and it is affected by the allocation of tasks to VCPUs. Especially, when cache-sensitive tasks are allocated together to the same VCPU, the benefit of cache sharing increases, thereby reducing the resource requirement. Hence, we propose a cache-aware VM designing algorithm (CAVM) that (i) allocates tasks to VCPUs in a way so as to increase the benefit of cache sharing, and (ii) derives each VCPU's resource requirement w.r.t. the number of cache colors allocated to its taskset. Our algorithm can be used for designing a new VM as well as calculating the resource requirement of an existing VM.

Alg. 2 presents the pseudo-code of CAVM. It takes four input parameters: Γ is a taskset to be allocated, N_{vcpu} is the number of VCPUs in the VM, N_{cache} is the number of available cache colors, and T^v is the VCPU period that will be assigned to all VCPUs in the VM.[5] CAVM initializes the budget of each VCPU v_i to be full, i.e., $C_i^v = T^v$, and the number of cache colors for v_i (S_i^v) to zero (line 2).

CAVM consists of two phases. The first phase is allocating tasks to VCPUs. Our allocation strategy is to group cache-sensitive tasks into a "bundle" and allocate as many tasks in the bundle as possible onto the same VCPU. To do so, CAVM first groups all tasks in Γ into a single bundle φ. Then, it checks the utilization of φ, assuming each

[4]In case of a write-back cache, Δ should take into account the effect of a *dirty* cache line that requires two memory accesses to fetch a new cache line [32].

[5]There are many ways to choose T^v. For example, system designers may use a hyperperiod to improve VCPU schedulability, or utilize the findings in [36] to reduce the overhead of hierarchical scheduling.

Algorithm 2 CacheAwareVM($\Gamma, N_{vcpu}, N_{cache}, T^v$)

```
Input: Γ: taskset, $N_{vcpu}$: the number of VCPUs, $N_{cache}$: the
    number of cache colors, $T^v$: VCPU period
Output: Success or Fail
 1: $\mathcal{V} \leftarrow \{v_1, v_2, ..., v_{N_vcpu}\}$
 2: $\forall v_i \in \mathcal{V} : T_i^v \leftarrow T^v, C_i^v(1, ..., N_{cache}) \leftarrow T^v, S_i^v \leftarrow 0$
 3: $N_{rem} \leftarrow N_{cache}$ /* Remaining cache colors */
 4: /* Phase 1: Allocate task bundles to VCPUs */
 5: $\varphi \leftarrow \Gamma; \Phi \leftarrow \emptyset$
 6: while $util(\varphi) > 1$ do
 7:     $(\varphi', \varphi'') \leftarrow$ BreakBundle($\varphi, 1, N_{cache}$)
 8:     $\Phi \leftarrow \Phi \cup \{\varphi'\}; \varphi \leftarrow \varphi'$
 9: $\Phi \leftarrow \Phi \cup \{\varphi\}$
10: while $\Phi \neq \emptyset$ do
11:     /* Allocate bundles */
12:     $\Phi_{rest} \leftarrow \emptyset$
13:     for all $\varphi_i \in \Phi$ in dec. order of average utilization do
14:         $(v_{BF}, k) \leftarrow$ BestFitWithCache($\varphi_i, \mathcal{V}, N_{rem}$)
15:         if $v_{BF} \neq invalid$ then
16:             $\Gamma_{BF} \leftarrow \Gamma_{BF} \cup \varphi_i; S_{BF}^v \leftarrow S_{BF}^v + k; N_{rem} \leftarrow N_{rem} - k$
17:         else
18:             $\Phi_{rest} \leftarrow \Phi_{rest} \cup \{\varphi_i\}$
19:     /* Break unallocated bundles */
20:     $\Phi \leftarrow \emptyset; singletons \leftarrow true$
21:     for all $\varphi_i \in \Phi_{rest}$ do
22:         if $|\varphi_i| > 1$ then
23:             $singletons \leftarrow false$; $size \leftarrow 1 - \min_{v_j \in \mathcal{V}} util(\Gamma_j)$
24:             $(\varphi', \varphi'') \leftarrow$ BreakBundle($\varphi_i, size, N_{cache}$)
25:             $\Phi \leftarrow \Phi \cup \{\varphi', \varphi''\}$
26:         else
27:             $\Phi \leftarrow \Phi \cup \{\varphi_i\}$
28:     if $singletons = true$ then
29:         return Fail
30: /* Phase 2: Determine VCPU budget */
31: for all $v_i \in \mathcal{V}$ do
32:     $C_i^v(0) \leftarrow invalid$
33:     for $k \leftarrow 1$ to $N_{cache}$ do
34:         if CacheToTaskAlloc($\Gamma_i, k$) $\leq 1$ then
35:             $S_i^v \leftarrow k$
36:             Binary search to find the minimum budget $x$
37:             $C_i^v(k) \leftarrow x$
38:         else
39:             $C_i^v(k) \leftarrow invalid$
40:         if $C_i(k-1) \neq invalid \wedge (C_i^v(k-1) < C_i^v(k) \vee C_i^v(k) = invalid)$ then
41:             $C_i^v(k) \leftarrow C_i^v(k-1)$
42: return Success
```

task in φ uses one dedicated cache color (line 6). If it is greater than 1, φ is broken into two sub-bundles by `BreakBundle()` such that the size of the first sub-bundle does not exceed 1. The pseudo-code of `BreakBundle()` is given in Alg. 4. To keep as many cache-sensitive tasks as possible in the first sub-bundle, `BreakBundle()` removes tasks from the first sub-bundle in increasing order of cache sensitivity, which is calculated by $(C_i(1) - C_i(N_{cache}))/T_i$, until the size of the first sub-bundle becomes not to exceed the given size constraint. When `BreakBundle()` returns, CAVM puts the first sub-bundle into Φ that is the set of bundles to be allocated (line 8 of Alg. 2), and continues to check the second bundle if it needs to be broken. As a result, each bundle in Φ has a utilization not exceeding 1 and is ready to be allocated.

CAVM allocates bundles in Φ to VCPUs based on the best-fit decreasing (BFD) heuristic (from line 13 to line 18). Here, we define the average utilization of a bundle φ_i as $\sum_{\tau_j \in \varphi_i} \sum_{k=1}^{N_{cache}} \{(C_j(k)/T_j)/N_{cache}\}$. Bundles are sorted in descending order of average utilization and CAVM tries to allocate each bundle to a VCPU by using `BestFitWithCache()` given in Alg. 3. This function finds the best-fit VCPU that can schedule a given bundle with k additional cache colors

Algorithm 3 BestFitWithCache($\varphi, \mathcal{V}, N_{rem}$)

```
Input: $\varphi$: a bundle of tasks to be allocated, $\mathcal{V}$: a set of VCPUs,
    $N_{rem}$: the number of cache colors
Output: $(v_i, k)$: a tuple of the best-fit VCPU and the number of
    additional cache colors needed
1: for $k \leftarrow 0$ to $N_{rem}$ do
2:     for all $v_i \in \mathcal{V}$ in decreasing order of $util(\Gamma_i)$ do
3:         if CacheToTaskAlloc($\Gamma_i \cup \varphi, S_i^v + k$) $\leq 1$ then
4:             return $(v_i, k)$
5: return $(invalid, -1)$
```

Algorithm 4 BreakBundle($\varphi, size, N_{cache}$)

```
Input: $\varphi$: a bundle to be broken, $size$: the size constraint for the
    first sub-bundle, $N_{cache}$: the number of colors
Output: $(\varphi', \varphi'')$: a tuple of sub-bundles
1: $\varphi' \leftarrow \varphi; \varphi'' \leftarrow \emptyset$
2: for all $\tau_i \in \varphi$ in increasing order of cache sensitivity do
3:     $\varphi' \leftarrow \varphi' \setminus \tau_i$; $\varphi'' \leftarrow \varphi'' \cup \tau_i$
4:     /* Get $util(\varphi')$ assuming each task uses one color */
5:     if $util(\varphi') \leq size$ then
6:         break
7: return $(\varphi', \varphi'')$
```

assigned to it, where k starts from 0 to the number of remaining cache colors (N_{rem}). If a best-fit VCPU is found (line 15 of Alg. 2), the bundle is allocated to that VCPU, and the number of cache colors of that VCPU (S_{BF}^v) and the number of remaining cache colors are updated. Otherwise, the bundle is put into Φ_{rest} (line 18).

Then, CAVM attempts to break all unallocated bundles in Φ_{rest}. If a bundle in Φ_{rest} has more than one task (line 22), it is broken into two sub-bundles by `BreakBundle()` such that the size of the first sub-bundle does not exceed the remaining capacity of a VCPU having the minimum taskset utilization. The resulting two sub-bundles are put into Φ so that they can be allocated in the next iteration. If all unallocated bundles are singletons (line 28 of Alg. 2), CAVM returns *fail* because none of these bundles can be broken into sub-bundles.

After finishing the first phase of task allocation, each VCPU v_i is allocated its own taskset Γ_i. The second phase of CAVM determines the budget $C_i^v(k)$ of a VCPU v_i for all possible k values ($1 \leq k \leq N_{cache}$). If Γ_i with k colors is schedulable (line 34), CAVM finds the minimum possible budget of v_i by using a binary search between 0 and T_v, and sets $C_i^v(k)$ to x. Otherwise, $C_i^v(k)$ is marked as invalid. Here, it may happen that, due to CRPD, $C_i^v(k-1)$ is smaller than $C_i^v(k)$ or is valid while $C_i^v(k)$ is invalid. In such cases (line 40), CAVM sets $C_i^v(k)$ to $C_i^v(k-1)$ and lets v_i use only $k-1$ colors if k colors are given. With this, CAVM can find $C_i^v(k)$ values that are monotonically decreasing with k.

5.4 Allocating Host Cache Colors to VMs

We now present our cache-to-VM allocation algorithm that determines the number of cache colors for each VCPU of the VMs to be consolidated, while minimizing the total utilization of those VMs. Once cache colors are allocated, conventional bin-packing heuristics such as BFD can be used to allocate the VCPUs of those VMs to PCPUs.

Let $\sigma_{i,k}$ denote the number of cache colors assigned to v_i when a total of k colors is provided in the host machine, and let $\mathcal{V}$ denote a set of VCPUs of all VMs to be consolidated. Then, the total utilization of VMs with k cache colors is given by:

$$\sum_{v_i \in \mathcal{V}} \frac{C_i^v(\sigma_{i,k})}{T_i} \tag{5}$$

Algorithm 5 CacheToVMAlloc($\mathcal{V}, N_{cache}$)

```
Input: $\mathcal{V}$: a set of VCPUs of all VMs to be consolidated, $N_{cache}$: the number of available cache colors
Output: Success or Fail
1: Find $x_i$ for each VCPU $v_i \in \mathcal{V}$
2: $z \leftarrow \sum_{v_i \in \mathcal{V}} x_i$
3: if $N_{cache} < z$ then
4:     return Fail
5: $\forall v_i \in \mathcal{V} : \sigma_{i,x} \leftarrow x_i$
6: $U(z) \leftarrow \sum_{v_i \in \mathcal{V}} \frac{C_i^v(\sigma_{i,z})}{T_i^v}$ /* $U(z)$: total utilization */
7: for $k \leftarrow z+1$ to $N_{cache}$ do
8:     $U(k) \leftarrow \min_{z \le k' < k} \left( U(k') - \max_{v_i \in \mathcal{V}} \frac{C_i^v(\sigma_{i,k'}) - C_i^v(\sigma_{i,k'} + (k-k'))}{T_i^v} \right)$
9:     $\forall v_i \in \mathcal{V}: \sigma_{i,k} \leftarrow$ # of colors of $v_i$ contributing to $U(k)$
10: $\forall v_i \in \mathcal{V} : S_i^v \leftarrow \sigma_{i,N_{cache}}$
11: return Success
```

To find the minimum total utilization of VMs with k cache colors, $U(k)$, we use a dynamic programming approach. Let x_i denote the smallest number of cache colors that gives a valid budget for v_i, i.e., $C_i^v(x_i) \neq$ *invalid* and $C_i^v(x_i - 1) =$ *invalid*, and let z denote the minimum number of cache colors needed to schedule all VCPUs in $\mathcal{V}$. Then, z is calculated by $z = \sum_{v_i \in \mathcal{V}} x_i$, and $\sigma_{i,z}$ is equal to x_i because there is only one valid cache allocation to v_i when z colors are provided. For $k < z$, we represent $U(k)$ as ∞ because there is no valid allocation. For $k = z$, $U(k)$ can be computed by Eq. (5) because $\sigma_{i,k} = x_i$. For $k = z+1$, $U(k)$ cannot be computed by Eq. (5) because $\sigma_{i,k}$ is unknown. Instead, we can compute $U(k)$ from $U(z)$. Recall that our CAVM algorithm given in Section 5.3 ensures that $C_i^v(k)$ is monotonically decreasing with k. Hence, if any additional cache color is assigned to v_i, a non-negative utilization gain is obtainable. Based on this observation, we can compute $U(k = z+1)$ by $U(z) - \max \frac{C_i^v(\sigma_{i,z}) - C_i^v(\sigma_{i,z}+1)}{T_i^v}$, which subtracts the maximum utilization gain made by one additional color from $U(z)$. We can also find $\sigma_{i,z+1}$ by recording the number of colors of v_i that leads to $U(z+1)$. For $k = z+2$, $U(k)$ can be calculated by the minimum between $U(z) - \max \frac{C_i^v(\sigma_{i,z}) - C_i^v(\sigma_{i,z}+2)}{T_i^v}$, which subtracts the maximum gain by two additional colors from $U(z)$, and $U(z+1) - \max \frac{C_i^v(\sigma_{i,z+1}) - C_i^v(\sigma_{i,z+1}+1)}{T_i^v}$, which subtracts the maximum gain by one additional color from $U(z+1)$. This approach can be extended to all $k > z$, and $U(k)$ is given by the following recurrence:

$$U(k) = \begin{cases} \infty \text{ (unschedulable)} & : k < z \\ \sum_{v_i \in \mathcal{V}} \frac{C_i^v(\sigma_{i,k})}{T_i} & : k = z \\ \min_{z \le k' < k} \left(U(k') - \max_{v_i \in \mathcal{V}} \frac{C_i^v(\sigma_{i,k'}) - C_i^v(\sigma_{i,k'} + (k-k'))}{T_i^v} \right) & : k > z \end{cases} \quad (6)$$

Alg. 5 shows our cache-to-VM allocation algorithm based on the recurrence in Eq. (6). Our algorithm first finds z, and if a given number of cache colors (N_{cache}) is smaller than z, it returns fail (line 4). Otherwise, it computes $U(k)$ iteratively (line 8) and saves $\sigma_{i,k}$ that leads to $U(k)$ (line 9). Once the iteration completes, our algorithm sets the number of cache colors for each VCPU to $\sigma_{i,N_{cache}}$ and returns success. The time complexity of our algorithm is $O((N_{cache})^2 \cdot |\mathcal{V}|)$.

6. EVALUATION

This section presents our experimental results on our vLLC, vColoring, and cache management scheme.

Table 1: Implementation cost of vLLC and vColoring

Name	Items	Cost (nsec)	
		x86	ARM
vLLC	Virtual LLC emulation	787	12212
	Color check in GPP-to-HPP mapping	34	921
vColoring	Page migration for GPP re-mapping	2359	31864

6.1 vLLC and vColoring

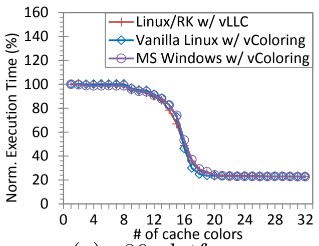
Experimental Setup: We have implemented vLLC and vColoring on the KVM hypervisor included in the Linux 3.10.39 kernel. We chose KVM for its convenience, such as supporting various architectures and providing both shadow paging and two-dimensional paging. However, it is worth noting that our techniques, vLLC and vColoring, can also be implemented in other hypervisors. In our experiments, we use two-dimensional paging because it is the default address translation technique of KVM and shadow paging is not yet supported by KVM for ARM.

We use x86 and ARM platforms as host machines for our experiments. The x86 platform is equipped with an Intel i7-2600 3.4GHz quad-core processor and 16GB of DDR3 1666MHz memory. The Intel processor has a unified 8MB shared LLC that consists of four 2MB cache slices, providing 32 cache colors. We disabled hardware prefetcher, simultaneous multithreading, and dynamic clock frequency scaling to reduce measurement inaccuracies. The ARM platform used is an ODROID-XU4 board. It has 2GB of LPDDR3 933MHz memory and a Samsung Exynos 5422 SoC that combines a cluster of four ARM Cortex-A15 cores with a cluster of four Cortex-A7 cores. However, we only use the cluster of Cortex-A15 cores because the performance of the other cluster seems inadequate for our experiments. The LLC shared among four Cortex-A15 cores is 2MB, providing 32 cache colors. We disabled dynamic clock frequency scaling and configured each core to run at its maximum speed, 2GHz.

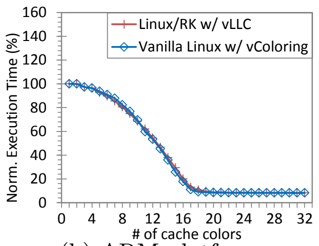
Since our focus is on cache interference imposed on tasks in a VM, each platform hosts one VM that has four VCPUs (VCPUs 1-4). Each VCPU is allocated to a different PCPU with 100% of budget. Hence, there is only one VCPU per PCPU on both the x86 and ARM platforms. The VM is assigned all the 32 cache colors of the host machine. On the host side, VCPU threads are assigned real-time priorities, which prevents unexpected delays from indispensable system services that could not be disabled.

Three different guest OSs are used in our experiments: Linux/RK and the vanilla Linux kernel 3.10.39 for x86 and ARM, and MS Windows Embedded 8.1 Industry for x86. Linux/RK is used as a guest OS to evaluate vLLC because it supports page coloring. The vanilla Linux and MS Windows Embedded OSs are used to evaluate vColoring because they both do not support page coloring. Specifically, MS Windows Embedded is chosen to verify that vColoring can be used for proprietary, closed-source guest OSs.

Implementation Overhead: Table 1 shows the computational overhead of vLLC and vColoring, measured with hardware performance counters on the x86 and ARM platforms. vLLC performs the virtual LLC emulation when a guest OS reads the VM's LLC information, which is typically done during the system initialization phase. The GPP-to-HPP mapping occurs only once per GPP, as described in Section 4, and the overhead added by the color check of vLLC in the GPP-to-HPP mapping is less than 5% of the original mapping time on both platforms. Hence, we consider that the overhead of vLLC is acceptably small. vColoring re-maps GPPs when cache colors are assigned to a task. Since the



(a) x86 platform



(b) ARM platform

Figure 5: Execution times of the *latency* task

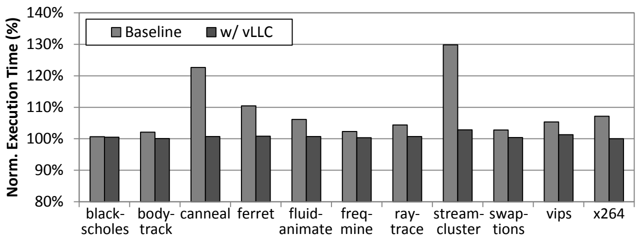


Figure 6: Execution times of the PARSEC benchmarks when synthetic tasks run on different VCPUs in parallel

major overhead of this re-mapping is caused by page migration, we present per-page migration time in Table 1.

Results with a Synthetic Task: As the first step of our experiments, we check if vLLC and vColoring can correctly assign cache colors to a task running in a VM. We use the *latency* task [48] which traverses a randomly-ordered linked list. The execution time of the *latency* task highly depends on the memory access time, due to the data dependency of pointer-chasing operations in linked-list traversals. To make the *latency* task cache-sensitive, we configured the working set size of the *latency* task to be half of the LLC of each platform, i.e., 4MB on x86 and 1MB on ARM. We compiled this task for both Linux and MS Windows guests on x86.

Figure 5 compares the maximum observed execution times of the *latency* task when it runs alone in each VM with different numbers of cache colors assigned to it. The x-axis of each graph denotes the number of cache colors assigned to the task. The y-axis shows the execution time normalized to the case where the task runs with one cache color. On both x86 and ARM platforms, the execution time of the task begins to plateau after more than 16 cache colors are assigned to it. This is because the entire working set of the task can fit into the LLC after that point. On each platform, a very similar execution-time pattern is observed although different guest OSs are used. This shows that both vLLC and vColoring work as expected.

Results with PARSEC Benchmarks: We use the PARSEC benchmarks [9], which are closer to the memory access patterns of real applications compared to the synthetic task, *latency*. A total of eleven PARSEC benchmarks is used. We have excluded two PARSEC benchmarks, *dedup* and *facesim*, due to their excessive disk accesses for data files. Since we have shown in the previous subsection that vLLC and vColoring are equivalent in preventing cache interference on x86 and ARM platforms, we use only vLLC on x86 for simplicity.

We first identify the impact of inter-VCPU cache interference on the PARSEC benchmarks. Each benchmark is assigned to VCPU 1 and the three instances of the *latency* task are assigned to the other VCPUs to generate interfering cache requests. When vLLC is not used, the benchmark and the three instances share all 32 cache colors. When vLLC is used, our objective here is to protect the cache behavior of

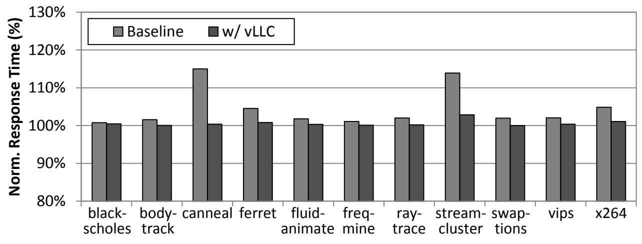


Figure 7: Response times of the PARSEC benchmarks when synthetic tasks are scheduled on the same VCPU

the benchmark from the three instances of *latency*. Hence, with vLLC, each benchmark is assigned 31 private cache colors and the three instances share the remaining 1 color.

Figure 6 compares the execution time of each PARSEC benchmark with and without vLLC. The x-axis denotes the benchmark names, and the y-axis shows the execution time of each benchmark normalized to the case when it runs alone in the VM with 32 cache colors. When vLLC is not used (Baseline), there is up to 30% of execution time increase. When vLLC is used, only *streamcluster* has an execution time increase of 2% and the other benchmarks have no noticeable difference in their execution times. The reason for the increase in *streamcluster*'s execution time is due to the fact that it is assigned a smaller number of cache colors when vLLC is used, compared to when vLLC is not used.

Next, we explore the impact of intra-VCPU cache interference on the PARSEC benchmarks. Each benchmark and the three instances of *latency* are assigned to the same VCPU, and the SCHED_RR policy with a time quantum of 10 msec is used to time-share that VCPU. When vLLC is used, the benchmark is assigned 31 private cache colors and the three instances share 1 remaining cache color, just like the inter-VCPU interference experiment.

Figure 7 shows the response time of each benchmark when the three instances of *latency* are scheduled on the same VCPU. The response time of a benchmark is normalized to the case when it is scheduled on the same VCPU with three instances of a *busyloop* task. *busyloop* runs an empty infinite while loop, thereby causing no cache interference. When vLLC is not used, the response time increases by up to 15%. When vLLC is used, all the benchmarks except *streamcluster* have no noticeable difference in their response times. The increase in *streamcluster*'s execution time is again because a smaller number of cache colors is assigned to the benchmark when vLLC is used. To summarize, the results with the PARSEC benchmarks show that both inter- and intra-VCPU cache interference can significantly degrade task performance, and our techniques are effective in allocating cache colors to tasks running in a VM.

6.2 Cache Management Scheme

In this subsection, we evaluate our real-time cache management scheme for multi-core virtualization. To do this, we use randomly-generated tasksets and capture the total utilization of VMs as the metric.

Experimental Setup: We generated 10,000 tasksets with the parameters in Table 2. Cache hit/miss delay and cache color reload time (Δ) were obtained by measurement on our ARM platform. To generate a WCET function ($C_i(k)$) for each task τ_i, we use the method described in [10]. 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 [41]. It

Table 2: Parameters for taskset generation

Type	Parameters	Values
System	Number of PCPUs	4
	Number of VMs	2
	Number of VCPUs per VM	4
	VCPU replenishment period	10 msec
	Cache (LLC) size	2048 KB
	# of cache colors (N_{cache})	32
	Cache hit delay	26 nsec
	Cache miss delay	202 nsec
	Cache color reload time (Δ)	207 μsec
Taskset	Total number of tasks	[10, 15]
	Taskset utilization ($U_{taskset}$)	3.0
WCET	Memory accesses per job	[100000, 1000000]
	Neighborhood size	[16, 64]
	Locality	[1.5, 3.0]
	Task memory usage	[8, 40] MB
	*Resulting working-set size	[64 KB, 40 MB]
	*Resulting WCET	[8.47, 202.02] msec

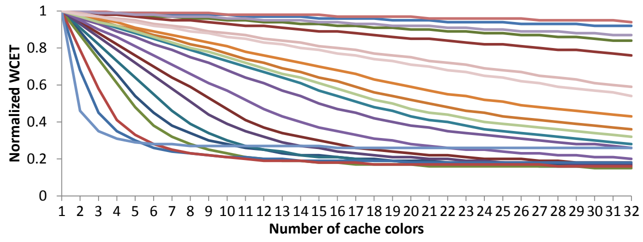


Figure 8: Some of WCETs generated for our experiments

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, as shown in Figure 8. Then, the total taskset utilization ($U_{taskset}$) 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 randomly distributed to two VMs, each of which has four VCPUs. Within each VM, the priorities of tasks are assigned by the Rate-Monotonic Scheduling (RMS) policy [25]. The priorities of VCPUs are arbitrarily assigned since they use the same period. The sporadic server policy is used for VCPU budget replenishment.

Results: For comparison with our scheme, we consider variants of the best-fit decreasing (BFD), worst-fit decreasing (WFD), and first-fit decreasing (FFD) heuristics. Each heuristic is used for task-to-VCPU allocation within a VM and combined with two different cache-to-task allocation policies: complete cache partitioning (CCP) and complete cache sharing (CCS). CCP allocates private cache colors to tasks in proportion to their working-set sizes. On the other hand, CCS lets tasks on the same VCPU share all their cache colors. Hence, we compare our scheme against a total of six approaches: BFD+CCP, WFD+CCP, FFD+CCP, BFD+CCS, WFD+CCS, and FFD+CCS. For each approach, k cache colors, where $1 \leq k \leq N_{cache}$, are evenly distributed to all VCPUs of the two VMs such that the difference in the number of cache colors of each VCPU does not exceed 1. Tasks are sorted in decreasing order of utilization w.r.t. the number of cache colors per VCPU. Once task-to-VCPU allocation is done, we determine the budget of each VCPU by the binary search approach used in the Phase 2 of our CAVM algorithm given in Alg. 2. Finally, we calculate the total utilization of VMs by summing up the utilization of all VCPUs.

Figure 9 shows the total VM utilization as the number of cache colors increases. Since CCP cannot find a schedulable allocation if the number of colors is smaller than that of tasks, we compare only the cases where the number of cache colors is greater than 15. Our scheme outperforms all other approaches, yielding 1.18× to 1.54× lower utilization. This is because our scheme allocates cache-sensitive tasks together to the same VCPU to increase the benefit of cache sharing and finds the minimum total VM utilization for a given number of cache colors. The heuristics with CCS perform better than the ones with CCP. This is because Δ obtained from our ARM platform is relatively small so that the reduction in task execution time from cache sharing is larger than the resulting CRPD in our experiments.

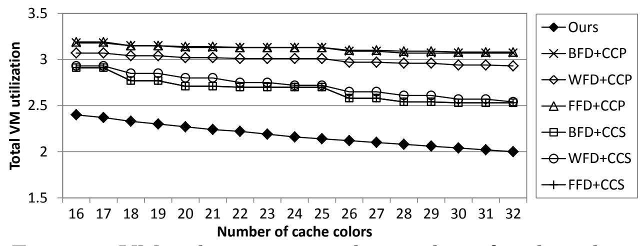


Figure 9: VM utilization w.r.t the number of cache colors

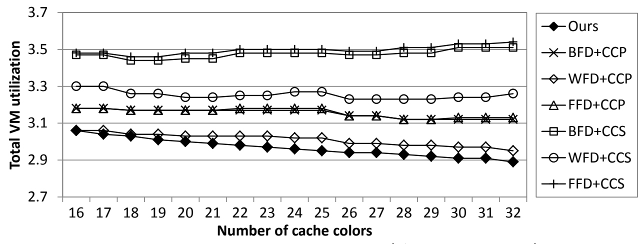


Figure 10: VM utilization ($\Delta = 10$ msec)

Figure 10 shows the total VM utilization when Δ is 10 msec. This experiment is to evaluate our scheme when CRPD is extremely high. Overall, the benefit of using more cache colors is smaller compared with the previous experiment. Our scheme outperforms other approaches because it can balance between the utilization gain and CRPD from cache sharing. The heuristics with CCS perform worse than the ones with CCP due to the high CRPD. In case of BFD+CCS and FFD+CCS, the utilization even increases as more cache colors are provided. WFD+CCS is affected less by the high CRPD compared with BFD+CCS and FFD+CCS, because WFD results in a fewer number of tasks per VCPU. Based on these results, we conclude that our scheme allocates cache colors efficiently in a virtualization environment and yields a significant utilization benefit.

7. RELATED WORK

Cache management schemes have been extensively studied in the context of non-virtualized systems. Liedtke et al. [23] proposed to use page coloring to prevent cache interference from other tasks in a single-core platform. Lin et al. [24] conducted a comparative study on various multi-core cache partitioning schemes by implementing them with page coloring. Mancuso et al. [28] proposed the Colored Lockdown technique that combines page coloring and cache lockdown to better keep the frequently accessed pages of real-time tasks in a cache. Ye et al. [47] developed COLORIS that supports both static and dynamic cache partitioning based on page coloring. Ward et al. [43] focused on cache management issues in multi-core mixed-criticality systems and proposed cache locking and scheduling techniques that use page coloring. Bui et al. [10] developed a genetic algorithm to find a near optimal solution for cache partition allocation on a single-core platform. Paolieri [30] proposed IA^3, which is a heuristic algorithm for allocating cache partitions to cores in a multi-core real-time system. All these schemes, however, cannot be directly applied to a virtualized system.

There also exist many research efforts on taking into account cache interference delay in the schedulability analyses [5, 26, 46]. Specifically, Altmeyer and Davis [5] compared the performance of cache partitioning and cache-related preemption delay (CRPD) analysis on a single-core platform. Xu et al. [46] extended multi-core compositional analysis to incorporate cache interference delay, assuming that there is no shared cache. Lunniss et al. [26] extended CRPD analysis to a single-core hierarchical scheduling environment. However, none of these approaches focuses on a shared cache in a multi-core platform.

Previous work on software-based cache management in a virtualization environment [22, 27, 34] proposed to implement page coloring in the hypervisor and to allocate cache colors to VMs. This approach, however, cannot be used to address cache interference among tasks running in the same VM, as we discussed in Section 2.3. Kim et al. [15] proposed a hardware-based solution to enable page coloring implemented in a guest OS to work. However, hardware modification required by this approach does not allow the use of commodity multi-core processors. In addition, if a guest OS does not have page coloring support, tasks running on that guest OS cannot get any benefit. In this paper, we have addressed these limitations.

8. CONCLUSIONS

In this paper, we present our proposed real-time cache management framework for multi-core virtualization. Our framework has vLLC and vColoring, hypervisor-level techniques to enable the cache allocation of individual tasks running in a VM. We have implemented vLLC and vColoring on the KVM hypervisor running on x86 and ARM platforms. Experimental results with three different guest OSs show that both vLLC and vColoring can effectively control the cache allocation of tasks in a VM. Our framework also supports a cache management scheme that determines cache to task allocation, designs a VM in the presence of cache interference, and minimizes the total utilization of VMs to be consolidated into the host machine. Experimental results with randomly-generated tasksets show that our scheme yields a significant utilization benefit compared to other approaches. As future work, we plan to address temporal interference from main memory in a virtualization environment.

9. REFERENCES

[1] ARM Cortex-A15 technical reference manual. http://arm.com.
[2] Intel 64 & IA-32 software developer's manual. http://intel.com.
[3] OKL4 Microvisor. http://www.ok-labs.com.
[4] SYSGO PikeOS Embedded Virtualization. http://sysgo.com.
[5] S. Altmeyer et al. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, 2014.
[6] P. Barham et al. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
[7] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM Workshop on Language, Compiler, and Tools for Real-Time Systems*, 1994.
[8] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *RTSS*, 1999.
[9] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
[10] B. D. Bui et al. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*, 2008.
[11] J. Busquets-Mataix et al. Hybrid instruction cache partitioning for preemptive real-time systems. In *ECRTS*, 1997.
[12] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, 2005.
[13] P. Hammarlund et al. 4th generation Intel Core processor, codenamed Haswell. In *Hot Chips (HC25)*, 2013.
[14] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.
[15] D. Kim et al. vCache: Providing a transparent view of the LLC in virtualized environments. *Computer Architecture Letters*, 13(2):109–112, 2014.
[16] H. Kim et al. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.
[17] H. Kim et al. Responsive and enforced interrupt handling for real-time system virtualization. In *RTCSA*, 2015.
[18] H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
[19] J. Kiszka. Towards Linux as a real-time hypervisor. In *RTLWS*, 2009.
[20] A. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.
[21] O. Lempel. 2nd generation Intel Core processor family: Intel Core i7, i5 and i3. In *Hot Chips (HC23)*, 2011.
[22] Y. Li et al. A virtualized separation kernel for mixed criticality systems. In *VEE*, 2014.
[23] J. Liedtke et al. OS-controlled cache predictability for real-time systems. In *RTAS*, 1997.
[24] J. Lin et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.
[25] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
[26] W. Lunniss et al. Accounting for cache related pre-emption delays in hierarchical scheduling. In *RTNS*, 2014.
[27] R. Ma et al. Cache isolation for virtualization of mixed general-purpose and real-time systems. *Journal of Systems Architecture*, 59(10):1405–1413, 2013.
[28] R. Mancuso et al. Real-time cache management framework for multi-core architectures. In *RTAS*, 2013.
[29] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *RTSS Work-In-Progress*, 1998.
[30] M. Paolieri et al. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS*, 2011.
[31] S. Saewong et al. Analysis of hierarhical fixed-priority scheduling. In *ECRTS*, 2002.
[32] F. Sebek. Cache memories and real-time systems. Technical report, Mälardalen University, 2001.
[33] L. Sha et al. Solutions for some practical problems in prioritized preemptive scheduling. In *RTSS*, 1986.
[34] J. Shi et al. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*, 2011.
[35] I. Shin et al. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.
[36] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM TECS*, 7(3):30, 2008.
[37] B. Sprunt et al. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
[38] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys*, 2010.
[39] J. K. Strosnider et al. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. on Computers*, 44(1):73–91, 1995.
[40] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.
[41] D. Thiebaut et al. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. on Computers*, 41(4), 1992.
[42] X. Wang et al. Selective hardware/software memory virtualization. *ACM SIGPLAN Notices*, 46(7):217–226, 2011.
[43] B. C. Ward et al. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.
[44] Windriver VxWorks. http://www.windriver.com.
[45] S. Xi et al. RT-Xen: towards real-time hypervisor scheduling in Xen. In *EMSOFT*, 2011.
[46] M. Xu et al. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *RTSS*, 2013.
[47] Y. Ye et al. COLORIS: a dynamic cache partitioning system using page coloring. In *PACT*, 2014.
[48] H. Yun et al. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, 2014.