Predictable Shared Cache Management for Multi-Core Real-Time Virtualization

HYOSEUNG KIM, University of California, Riverside
RAGUNATHAN (RAJ) RAJKUMAR, Carnegie Mellon University

Real-time virtualization has gained much attention for the consolidation of multiple real-time systems onto a single hardware platform while ensuring timing predictability. However, a shared last-level cache (LLC) on modern multi-core platforms can easily hamper the timing predictability of real-time virtualization 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 article, we propose a predictable shared cache management framework for multi-core real-time 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 (i.e., 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 while satisfying timing constraints.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; **Embedded software**; **Real-time systems**; **Real-time operating systems**;

Additional Key Words and Phrases: Cache management, shared cache, multi-core architectures, virtualization, real-time systems, cyber-physical systems

ACM Reference format:
Hyoseung Kim and Ragunathan (Raj) Rajkumar. 2017. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.* 17, 1, Article 22 (December 2017), 27 pages.
https://doi.org/10.1145/3092946

1 INTRODUCTION

With the increase of core counts on modern processors, there is a high 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

Authors' addresses: H. Kim, Department of Electrical and Computer Engineering, University of California, 900 University Ave, Riverside, CA 92521, USA; R. (Raj) Rajkumar, Department of Electrical and Computer Engineering, Carnegie Mellon University 5000 Forbes Ave, Pittsburgh, PA 15213, USA.

https://doi.org/10.1145/3092946

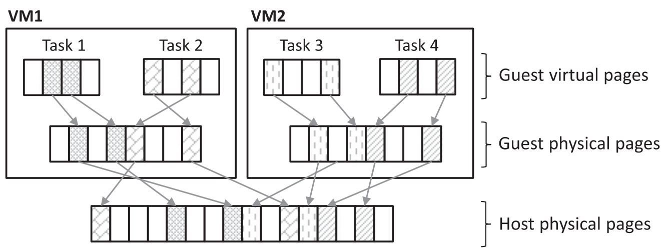


Fig. 1. Address translation layers in virtualization.

in modern virtualization platforms, such as Xen [3] and KVM [28]. 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.

The foremost requirement for real-time system virtualization is ensuring timing predictability. Hierarchical real-time scheduling theory [12, 43, 47, 48, 59] and its implementations [25, 32, 56] 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 article, we focus on the predictable management of a shared LLC in a virtualization environment.

Many previous approaches developed for nonvirtualized multi-core systems [21, 38, 52, 55] 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 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 nonvirtualized systems cannot provide an efficient solution to design a VM and to allocate the host machine's LLC to VMs to be consolidated.

In this article, we propose a predictable shared cache management framework for multi-core real-time virtualization. To address the problem of cache-to-task allocation in a VM, our framework supports two new hypervisor-level techniques, 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 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 [21, 39]) and OSs without page coloring (vanilla Linux and MS Windows Embedded).

We also 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 designs a VM in a way that the VM's resource demand is captured with respect to the number of cache colors allocated to it. 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.

This article is an extended version of our conference paper [24], with the following new contributions: (i) an expanded description of the proof-of-concept implementation, (ii) details on handling cache allocation requests under vColoring, and (iii) more extensive experimental results.

The rest of this article is organized as follows. Section 2 reviews the background. Section 3 describes the system model used. 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. Section 8 concludes the article.

2 BACKGROUND

In this section, we give a brief description on scheduling, cache interference, and address translation in a virtualization environment, and we discuss the page coloring technique. For more details, interested readers may refer to [7, 21, 33, 34, 52, 54].

2.1 Scheduling and Cache Interference in Virtualization

Virtualization generally features a two-level hierarchical scheduling structure. Each VM has one or more virtual CPUs (VCPUs), each of which is shown 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.

Hierarchical scheduling theories studied in the context of real-time systems [12, 31, 43, 47, 48] are well suited to the two-level scheduling structure of virtualization. Many researchers have recently applied these real-time hierarchical scheduling theories to open-source virtualization platforms (e.g., Xen [56, 57], KVM [11, 25], and L4/Fiasco) [60].

Cache interference among tasks in multi-core virtualization 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 in Virtualization

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 that do such translation in *full virtualization*, where unmodified guest OSs can be used.

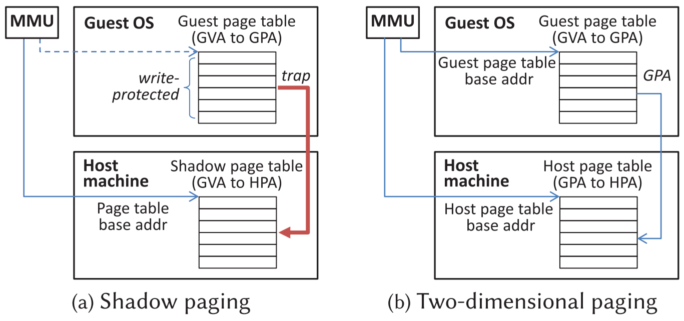

MMU
Guest OS
Guest page table (GVA to GPA)
write-protected
trap
Host machine
Shadow page table (GVA to HPA)
Page table base addr
(a) Shadow paging
MMU
Guest OS
Guest page table (GVA to GPA)
Guest page table base addr
GPA
Host machine
Host page table (GPA to HPA)
Host page table base addr
(b) Two-dimensional paging

Fig. 2. Address translation techniques in virtualization.

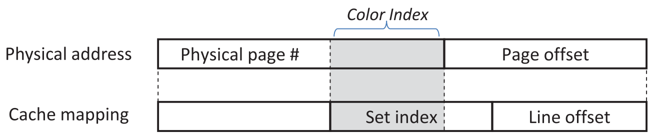


Fig. 3. Page coloring.

Shadow paging. Figure 2(a) illustrates an example of 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. Whenever a guest OS makes a change to a write-protected guest page table, a page fault exception that causes a "world switch" to the hypervisor happens. Then, the hypervisor catches the exception and updates the entry of the corresponding shadow page table, which constitutes the major overhead of shadow paging.

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, as shown in Figure 2(b). 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 [54]. It is also currently unknown which technique is preferable for real-time virtualization. Therefore, one of our goals in this article 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 3, 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 [14, 30]. As shown in Kim et al. 2012 and Ye et al. 2014 [21, 61], 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 nonvirtualized 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 Li et al., Ma et al., and Shi et al. [32, 37, 46]. 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 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 integer. Guest OSs may or may not support page coloring. We assume that each VM has been allocated a sufficient number of host physical pages and that page swapping does not occur at runtime. This is a reasonable assumption in real-time virtualization because, unlike in server virtualization, memory underprovisioning is considered to be harmful to timing predictability [27]. 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 [13] and QNX Hypervisor [41]. Thus, each VCPU is statically assigned to a single PCPU and each task is statically assigned to a single VCPU. Any fixed-priority assignment (e.g., Rate Monotonic [35]) can be used for both VCPUs and tasks.

Task model. We consider periodic tasks with constrained deadlines. A task τ_i is characterized by the following parameters:

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

- $C_i(k)$: the t-case execution time (WCET) of a task 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 ($D_i \leq T_i$)

$C_i(k)$ values are assumed to be known ahead of time. They can be obtained by either measurement-based or static analysis tools. Capturing the overhead of virtualization in task execution time is beyond the scope of this article. 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 [50]). We assume that $C_i(k)$ is monotonically decreasing with k, which our algorithm in Section 5.3 needs to reduce the search space. Although the actual WCET function may not be monotonic, this assumption can be easily satisfied in practice by monotonic overapproximations of WCETs with insignificant pessimism, as discussed in Altmeyer et al. [1].

Each task τ_i has a unique priority π_i. An arbitrary tie-breaking rule can be used to achieve this under fixed-priority scheduling. 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 [38]. Relaxing those assumptions is part of our future work.

VM resource model. Each VM is represented as follows:

$$\mathrm{VM} := (v_1, v_2, \ldots, v_{N_{vcpu}})$$

where v_i is a VCPU and N_{vcpu} is the number of VCPUs in the VM. A VCPU v_i is characterized by the following parameters:

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

- $C_i^v(k)$: 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 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* [45], *sporadic server* [49], and *deferrable server* [51] variants because they have been widely used in real-time virtualization [25, 26, 32, 56].

In the remainder of this article, 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 running 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 4 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 available cache colors. In Figure 4, since the 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

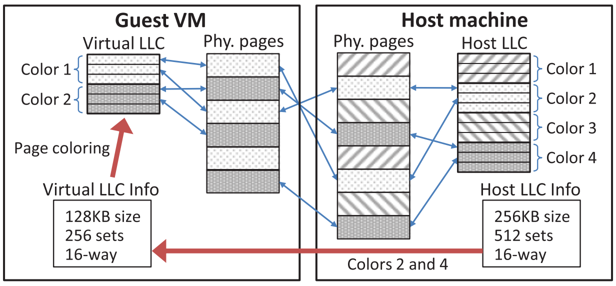


Fig. 4. vLLC example.

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 [16] and accesses to `CCSIDR` and `CSSERR` registers on an ARM Cortex-A15 architecture [2]).

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 done 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 4, 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.

4.2.1 Remapping GPPs to New HPPs. Figure 5 shows the detailed steps for remapping 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, independently of a specific address translation technique used by the hypervisor. The second step is to find out *present* and *user-level accessible* GPPs in the task's page tables. This

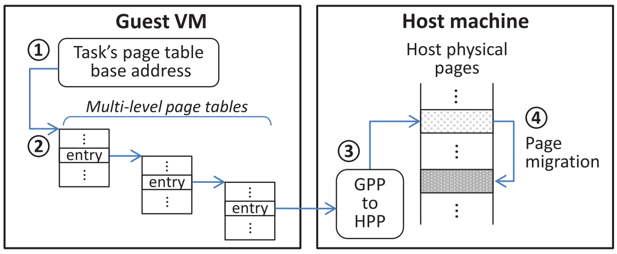


Fig. 5. Steps for remapping GPPs to new HPPs under vColoring.

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; under shadow paging, shadow page tables are updated; under two-dimensional paging, host page tables are updated. 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 remap GPPs present at that time, we must 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. Also, it is not recommended to make cache allocation requests dynamically in the runtime phase because the remapping overhead may introduce a nontrivial amount of delay in task execution. We will examine the remapping overhead in Section 6.1.

The hypervisor is unaware of the task and memory data structures of a guest OS, so it cannot directly read an arbitrary task's PTBA value. However, on many architectures, the currently executing task's PTBA is stored in a specific register to facilitate address translation (e.g., a CR3 register on x86 and a Translation Table Base register on ARM). We will refer to such a register as a page table base register (PTBR). Under shadow paging, the hypervisor traps on write accesses to the PTBR and stores the base address of the corresponding shadow page table into the PTBR. The real PTBA value trapped by the hypervisor is stored in the hypervisor's memory space and used to synchronize the shadow page table with the guest page table. Under two-dimensional paging, the MMU has two PTBRs, 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.

4.2.2 Making Cache Allocation Requests. We now explain how a task can make a cache allocation request to the hypervisor. Basically, a task needs to have a communication channel to the hypervisor in order to deliver its cache allocation request. On x86 architectures, we propose the use of a "hypercall" instruction (either `vmcall` or `vmmcall`) for this purpose. It can be executed by a user-level task running in a VM and results in a world switch to the hypervisor, regardless of whether the hardware-assisted virtualization is enabled or not [16]. The hypervisor running on x86 architectures typically ignores a hypercall if it has been executed by a user-level task (e.g., both KVM and Xen do so). vColoring registers a new *hypercall identifier* and its *handler* to the hypervisor, and it lets the hypervisor handle a hypercall with the new identifier even if it has been executed by a user-level task. Parameters for the hypercall are (i) our new hypercall identifier and (ii) cache colors in a bit field (e.g., 0011b for Colors 1 and 2). Once a task executes a hypercall with those parameters, the registered handler is executed by the hypervisor. The handler can easily get the task's PTBA because that task is the currently executing one. Recall that only the cache colors in the extra color set of the task's VM are valid ones. If the requested colors are valid, the handler allocates these colors to the task by following the remapping steps explained earlier.

When a task that has been allocated cache colors is terminating, it needs to explicitly free the allocated colors. Otherwise, the GPPs of the task will remain mapped to the HPPs corresponding to the colors used by the task and will possibly be reused by other tasks, resulting in unintended cache allocation to those tasks. The task can free its colors by making a cache allocation request with no color information (e.g., 0000b in a color bit field). Then, vColoring remaps the GPPs of the task to the HPPs corresponding to the colors in the default cache set.

We now consider other architectures, such as ARM [2], where a user-level task is *not* allowed to execute a hypercall instruction. For a guest OS that allows the addition of a new device driver, 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 with the same set of hypercall parameters as in the x86 case. Since many recent real-time OSs such as VxWorks [42] support implementing device drivers as loadable kernel modules, this approach can be easily used for such OSs without rebuilding the entire kernel image.

It is possible that a guest OS may not allow the addition of any new device driver. If a network device driver is embedded in such an OS image, we can use a virtual networking interface emulated by the hypervisor as a communication channel between a task and the hypervisor. One thing to consider is that, when the hypervisor receives a task's cache allocation request, the corresponding task should be executing at that time. This is because the hypervisor can only get the PTBA of the currently executing task. Considering this constraint, we propose the use of the following steps:

(1) Set up a virtual network connection between a task τ_i and the hypervisor. Then, suspend all other tasks, except indispensable system services including networking, to minimize temporal interference from them.
(2) Send a message from τ_i to the hypervisor with a cache color allocation request. Then, let τ_i busy-wait on incoming messages for $T + \epsilon$ time units, where T is the duration for the hypervisor to check the PTBA and ε is an upper-bound on exchanging a message over the virtual network interface.
(3) Once the hypervisor receives a message from τ_i, find a busy VCPU among all the VCPUs of the τ_i's VM. Then, check the current PTBA value of the busy VCPU every T' during T, where $T' \ll T$. The shorter T', the more samples can be obtained.
(4) Find the most-frequently-observed PTBA value and use it for the cache allocation to τ_i, as it is likely to be the one for τ_i that is busy-waiting. If the PTBA is found, the hypervisor sends a message back to τ_i. Otherwise, it does not send a message.
(5) If τ_i has received a message from the hypervisor during $T + \epsilon$, disconnect the connection and continue its execution. If not, redo these steps from Step 2.

Note that the above steps are not guaranteed to converge. If this happens, one may try raising the priority of a task τ_i to further reduce interference from other tasks. The priority of τ_i must be reverted afterward.

4.3 Implementation in KVM/QEMU

As a proof of concept, we implemented vLLC and vColoring on the KVM hypervisor included in the Linux 3.10.39 kernel. Note that, at the time of writing, this was the latest version supported by our ARM platform. We chose KVM for its convenience, such as supporting various architectures and providing both shadow paging and two-dimensional paging modes. However, it is worth noting that our techniques, vLLC and vColoring, can also be implemented in other hypervisors. Our

```
qemu-system-arm -cpu cortex-a15 -smp 2 -m 512
    -virtrk cache,colors=0-7,ext_mem=256,ext_colors=8-15
    -virtrk vcpu,id=vcpu0,vcpu=0,pcpu=0,c=5,t=10
    -virtrk vcpu,id=vcpu1,vcpu=1,pcpu=1,c=5,t=10
```

Fig. 6. Parameters of our modified QEMU for ARM architectures.

implementation supports a 64-bit x86 architecture as well as a 32-bit ARM Cortex-A15 architecture. Along with KVM, QEMU [5] version 2.2.0 is used for device emulation.

Our techniques need page coloring implemented at the hypervisor level. For this purpose, we used the memory reservation mechanism of Linux/RK [22, 23]. Linux/RK with version 1.5 or higher can read the system's LLC information and discover the number of cache colors available in the system. Using this color information, Linux/RK categorizes unallocated physical pages according to their corresponding cache colors and allows each application to reserve a portion of physical pages with specific cache colors. We modified QEMU to use Linux/RK such that QEMU gets a cache color request when starting a guest VM and creates a memory reservation with the requested cache colors for that VM. Therefore, during runtime, the VM uses only the cache colors associated with physical pages in its memory reservation. We also modified QEMU to use the CPU reservation mechanism of Linux/RK for VCPUs. Figure 6 shows the parameters of our modified QEMU. The guest VM created with these parameters is assigned a memory reservation of 512MB (`-m 512`) with 8 host cache colors (`colors=0-7`). The number of VCPUs of the VM is two (`-smp 2`). Each VCPU is assigned 5 ms of budget and 10 ms of budget replenishment period (`c=5,t=10`), resulting in 50% of PCPU time. In addition, each VCPU is statically allocated to one PCPU (`id=vcpu0,vcpu=0,pcpu=0` and `id=vcpu1,vcpu=1,pcpu=1`).

On the x86 architecture, LLC information can be retrieved by executing a `CPUID` instruction. As part of our vLLC implementation, we modified the `CPUID` trap-and-emulate handler of QEMU so that a guest VM can see a virtual LLC corresponding to the cache colors assigned to the VM. On the ARM Cortex-A15 architecture, LLC information can be read from the `CSSERR` and `CCSIDR` registers, which can be trapped and emulated in KVM itself. Hence, we modified KVM to emulate a virtual LLC on ARM. The GPP-to-HPP mapping of vLLC is implemented in the page allocation of the Linux/RK memory reservation mechanism.

For the implementation of vColoring, we allow each VM to have two memory reservations so that one is used for a default color set and the other is used for an extra color set. Similar to the default memory reservation, the size and cache colors of the extra memory reservation can also be specified in the parameters of our modified QEMU. For instance, in Figure 6, the guest VM is assigned an extra memory reservation of 256MB (`ext_mem=256`) with 8 host colors (`ext_colors=8-15`). The GPP remapping of vColoring is implemented as part of the Linux/RK memory reservation mechanism. To handle cache allocation requests from tasks in guest VMs, we registered a new hypercall handler in KVM. On x86, by modifying the Current Privilege Level (CPL) check condition, we could make KVM handle our hypercall even if it has been executed by a user-level task running in a VM. On ARM, we implemented a loadable device driver for tasks running in Linux guests to make cache allocation requests. The source code of our implementation is available at http://rtml.ece.cmu.edu/redmine/projects/rk/.

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 with regard to 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 nonvirtualized environments [1, 9, 10, 21] have shown that sharing of cache colors among tasks on the same core yields better task scheduling than doesCCP, 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 [18]:

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

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 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 [6]). It terminates when $W_i^{v,n+1} = W_i^{v,n}$, and 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:

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

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 Equation (2) is based on the task schedulability test under hierarchical scheduling given in Saewong et al. [43] but extended with CRPD [10, 21] to bound intra-VCPU cache interference. $\gamma_{j,i}$ is given by:

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

where $\mathbb{S}_j$ is the set of cache colors assigned to τ_j, and Δ is the maximum time to reload one cache color. In case of a write-back cache, Δ should consider the effect of a *dirty* cache line that requires two memory accesses to fetch a new cache line [44].

The utilization of a taskset Γ allocated to the same VCPU is calculated as follows [4, 21]:

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

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

ALGORITHM 1: CacheToTaskAlloc(Γ, N_{cache})

```
Input: Γ: taskset, $N_{cache}$: # of cache colors
Output: Utilization of Γ if schedulable, and ∞ otherwise
 1: if $N_{cache} = 0$ then
 2:     return ∞
 3: $cache\_idx \leftarrow 1$
 4: for all $\tau_i \in \Gamma$ do
 5:     /* Find the number of cache colors for $\tau_i$ */
 6:     $S_i \leftarrow \text{argmin}_{1 \le k \le N_{cache}} (\frac{C_i(k)}{T_i} + \frac{\gamma_{i,n}}{T_i})$
 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 ∞
```

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 [9], and finding a computationally-efficient solution has been left as future work [21]. Hence, we present in Algorithm 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 [21]. Then, for each task τ_i, it finds the number of cache colors, S_i, that minimizes the sum of the utilization 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 colors for τ_i is found, our heuristic finds 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.

5.3 Designing a Cache-Aware VM

The resource demand of a VM is the aggregate resource demands of all VCPUs in that VM, and it is affected by the allocation of tasks to VCPUs. Specifically, when cache-sensitive tasks are allocated together to the same VCPU, the benefit of cache sharing increases, thereby reducing the resource demand. Hence, we propose a cache-aware VM designing algorithm (CAVM) that (i) allocates tasks to VCPUs in a way to increase the benefit of cache sharing and (ii) derives each VCPU's resource demand with regard to the number of cache colors allocated to its taskset. CAVM can be used for designing a new VM as well as calculating the resource demand of an existing VM.

Algorithm 2 shows the pseudocode 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 the VCPUs.[1] 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).

[1]There are many ways to choose T^v. For example, system designers may use a hyperperiod or utilize the findings in Shin and Lee [48] to reduce the overhead of hierarchical scheduling.

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

Input: Γ: taskset, N_{vcpu}: # of VCPUs, N_{cache}: # of cache colors, T^v: VCPU replenishment period
Output: Success or Fail

```
1: $\mathcal{V} \leftarrow \{v_1, v_2, \ldots, v_{N_vcpu}\}$
2: $\forall v_i \in \mathcal{V} : T_i^v \leftarrow T^v, C_i^v(1, \ldots, N_{cache}) \leftarrow T^v, S_i^v \leftarrow 0$
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 decreasing 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** $C_i(k-1) \neq invalid \wedge ((C_i^v(k) = valid \wedge C_i^v(k-1) < C_i^v(k)) \vee C_i^v(k) = invalid)$ **then**
41:             $C_i^v(k) \leftarrow C_i^v(k-1)$
42: **return** Success
```

ALGORITHM 3: BreakBundle($\varphi, size, N_{cache}$)

```
Input: $\varphi$: a bundle to be broken, $size$: the size limit of the first sub-bundle, $N_{cache}$: # 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 $(\varphi', \varphi'')$
```

ALGORITHM 4: BestFitWithCache($\varphi, \mathcal{V}, N_{rem}$)

```
Input: $\varphi$: a bundle of tasks to be allocated, $\mathcal{V}$: a set of VCPUs, $N_{rem}$: # of cache colors
Output: $(v_i, k)$: a tuple of the best-fit VCPU and # of 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 $(invalid, -1)$
```

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. This is the key difference from prior work, [21] where tasks are allocated individually. To do so, CAVM first groups all tasks in Γ into a single bundle φ. Then, it checks the utilization of φ, assuming each 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 Algorithm 3. 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 does not 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 Algorithm 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 (Lines 13–18). 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 Algorithm 4. This function finds the best-fit VCPU that can schedule a given bundle with k additional cache colors 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), 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

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

```
Input: $\mathcal{V}$: a set of VCPUs of all VMs to be consolidated, $N_{cache}$: # 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
```

the next iteration. If all unallocated bundles are singletons (Line 28), CAVM returns *fail* because none of these bundles can be broken into sub-bundles.

After finishing the first phase of 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 ($1 \le k \le N_{cache}$). If Γ_i with k colors is schedulable (Line 34), CAVM finds the minimum possible budget x 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, due to CRPD, $C_i^v(k-1)$ can be smaller than $C_i^v(k)$ or be 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.

Note that the idea of bundling tasks for task allocation has been used in prior work to address various issues (e.g., mitigating timing penalties caused by shared resources [29] and reducing memory interference delay [20]). To extend CAVM for such issues, one may change `BreakBundle()` so that it considers multiple constraints when breaking a bundle. This remains as future work.

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:

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

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., $C_i^v(x_i) \ne invalid$ 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 Equation (5) because $\sigma_{i,k} = x_i$. For $k = z + 1$, $U(k)$ cannot be computed by Equation (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:

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

Algorithm 5 shows our cache-to-VM allocation algorithm based on the recurrence in Equation (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 $O((N_{cache})^2 \cdot |\mathcal{V}|)$.

Our algorithms proposed in this section assume partitioned fixed-priority scheduling for tasks and VCPUs, as described in Section 3. We conclude with a short discussion on their extensibility and applicability to other scheduling policies. First, CAVM performs task-to-VCPU allocation by checking task schedulability. If a cache-aware schedulability test for dynamic-priority task scheduling in a VM environment is given, we believe that it can be easily plugged into CAVM with minor changes. If global scheduling is used, the first phase of CAVM can be removed and only the second phase can be used to calculate VCPU budget. Second, the cache-to-VM allocation algorithm determines only the number of host cache colors for VCPUs, with no assumption on specific VCPU-to-PCPU allocation used. Hence, it can also be used for dynamic-priority and global VCPU scheduling policies.

6 EVALUATION

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

6.1 vLLC and vColoring

6.1.1 Experimental Setup. We used x86 and ARM platforms as host machines in 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, 16-way set-associative cache slices. As described in Section 2.3, page coloring can be applied to the LLC on a per cache-slice basis and yields 32 cache colors on this processor. During all experiments, we disabled the hardware prefetcher, simultaneous multithreading, and dynamic clock frequency scaling of the processor to reduce measurement inaccuracies.

Table 1. Implementation Cost of vLLC and vColoring

Name	Items	Cost (ns)	
		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

The ARM platform used is an ODROID-XU4 board manufactured by Hardkernel [15]. This board has 2GB of LPDDR3 933MHz memory and is equipped with a Samsung Exynos 5422 SoC that combines a cluster of four ARM Cortex-A15 cores with a cluster of four Cortex-A7 cores. Each cluster has an LLC shared among cores within the same cluster. We used only the cluster of Cortex-A15 cores because the performance of the other cluster seemed inadequate to perform our experiments. The LLC in the cluster of four Cortex-A15 cores is 2MB, 16-way set-associative and gives 32 cache colors with page coloring. We disabled the dynamic clock frequency scaling and configured each core to run at its maximum speed, 2GHz.

Both the x86 and ARM platforms run our modified KVM hypervisor described in Section 4.3, with the manufacturer's patch for ARM. We primarily use two-dimensional paging because (i) it is the default setting of KVM, and (ii) shadow paging is not yet supported by KVM for ARM. At the end of this subsection, we will compare the results with two-dimensional paging and shadow paging on the x86 platform.

Since our focus is on cache interference among tasks running within 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, the QEMU process and VCPU threads are assigned real-time priorities, which prevents unexpected delays from indispensable system services that could not be disabled.

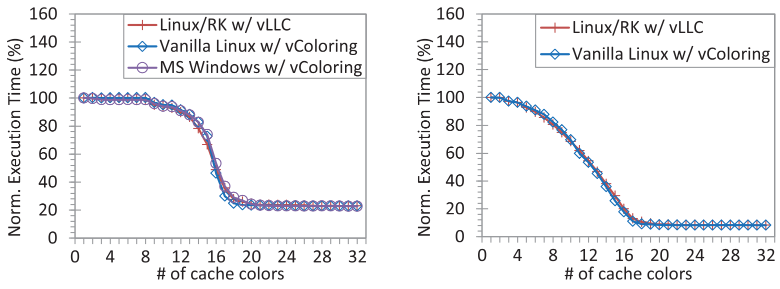
Three different guest OSs are used in our experiments: Linux/RK and the vanilla Linux kernel 3.10.39 for x86 and ARM, and MS Windows Embedded 8.1 Industry for x86. Linux/RK supports page coloring and is used as a guest OS to test vLLC. The vanilla Linux and MS Windows Embedded OSs do not support page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

6.1.2 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 4.1, 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 remaps GPPs when cache colors are assigned to a task. Since the major overhead of this remapping is caused by page migration, we present per-page migration time in Table 1.

There are other factors contributing to the GPP remapping time of vColoring, such as page-table traversal. Hence, in Table 2, we present the GPP remapping time as the number of GPPs changes. As can be seen, the remapping time increases linearly with the number of GPPs to be

Table 2. vColoring GPP Remapping Time

# of GPPs	Size (MB)	GPP re-mapping time (ms)	
		x86	ARM
256	1	0.89	10.68
512	2	1.45	19.47
1024	4	2.92	36.69
2048	8	5.38	70.65
4096	16	10.48	140.92
8192	32	21.06	274.92



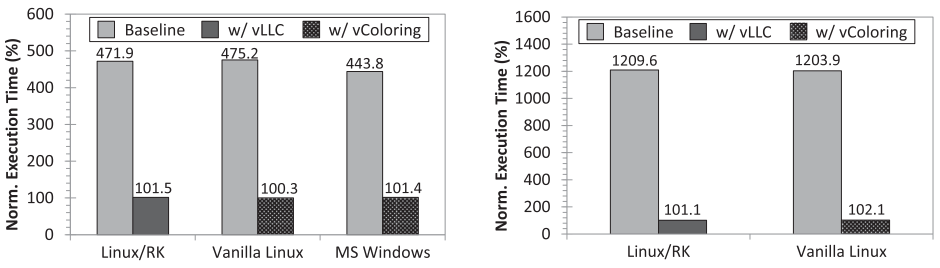
(a) x86 platform (b) ARM platform

Fig. 7. Execution time of the *latency* task.

mapped. Since the GPP remapping can suspend the corresponding VCPU, the dynamic allocation of cache colors after the initialization phase needs a careful attention, just like dynamic memory allocation in real-time systems [23, 38]. If vColoring needs to be used for a task that requests cache allocation after the initialization phase, one may take into account the GPP remapping time as a blocking term in the schedulability analysis for hierarchical real-time scheduling, such as in Kim et al. and Shin and Lee [25, 48].

6.1.3 Results with Synthetic Tasks. 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 [62] 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, and for Linux on ARM.

Figure 7 compares the maximum observed execution time 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.



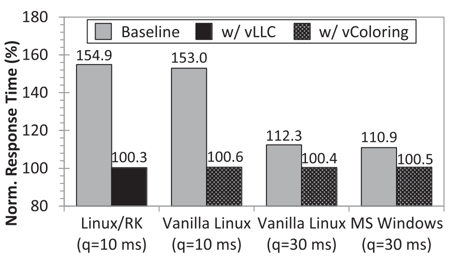
(a) x86 platform (b) ARM platform

Fig. 8. Execution time of the *latency* task when other task instances run on different VCPUs simultaneously.

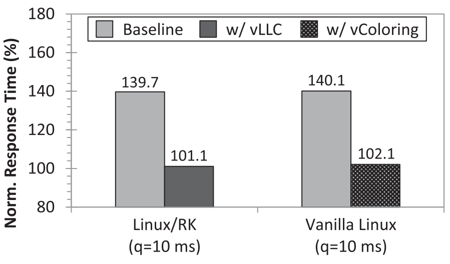
Recall that, in virtualization, cache interference can be categorized into inter-VCPU and intra-VCPU cache interference. We first evaluate if vLLC and vColoring can be used to protect a task from inter-VCPU cache interference. We execute four instances of the *latency* task on four different VCPUs simultaneously. Then we measure the execution time of the instance running on VCPU 1 to identify the impact of inter-VCPU interference caused by other instances. When our techniques are not used, all the four instances share the 32 cache colors of the VM. We will refer to this case as "Baseline." It is worth noting that previous work focusing on software-based cache management in a virtualization environment [17, 32, 37] falls into this case because the previous work cannot assign different cache colors to individual tasks running in a VM. When our techniques are used, we dedicate 31 colors to the instance running on VCPU 1 for its private use and let the other instances share the remaining 1 color. Note that this is an extreme cache configuration because the other three tasks will suffer severely from fewer shared cache colors. The reasons for doing so are (i) to segregate the cache behavior of the instance on VCPU 1 from the others, (ii) to make the instance on VCPU 1 have enough cache space to keep its working set within the cache, and (iii) to observe how much cache isolation can be achieved in such an extreme configuration.

Figure 8 illustrates the execution time of the instance assigned to VCPU 1 while the other three instances run on different VCPUs in parallel. The x-axis shows the guest OS used, and the y-axis shows the execution time normalized to the case when the *latency* task runs alone in the VM with 32 cache colors. When our techniques are not used (Baseline), all the instances compete for the LLC, resulting in significant increase in execution time. For example, we observed more than 4× increase in task execution time on x86 and 12× increase on ARM, which are very close to the case where the *latency* task is assigned only 1 cache color. When our techniques are used, the execution time is almost unaffected. The results of this experiment clearly show the benefit of vLLC and vColoring in preventing inter-VCPU cache interference.

We now evaluate the effects of our proposed techniques on intra-VCPU cache interference by executing multiple task instances on the same VCPU. In Linux guests, the SCHED_RR policy with time quanta of 10 ms and 30 ms is used to time-share the single VCPU. Hence, any task instance is preempted by another instance every time quantum. In a guest using MS Windows, we use the default time-sharing scheduling policy that has 30 ms of time quantum. We first execute one instance of the *latency* task along with three instances of the *busyloop* task on the same VCPU and measure the response time of the *latency* task instance. The *busyloop* task continuously consumes CPU time with no memory access. Therefore, it does not cause any intra-VCPU cache interference to the instance of *latency* although preemption happens. We will refer to the response time of the *latency* task measured in this case as the ideal response time because it is unaffected by cache interference. Then, we execute four instances of the *latency* task on the same VCPU and measure the



(a) x86 platform



(b) ARM platform

Fig. 9. Response times of the *latency* task when other task instances are scheduled on the same VCPU.

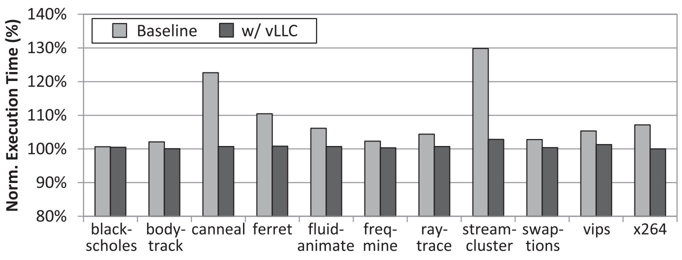


Fig. 10. Execution time of the PARSEC benchmarks when synthetic tasks run on different VCPUs in parallel.

response time of one instance with and without our techniques. As in the inter-VCPU experiment, when our techniques are used, the instance to be measured is assigned 31 private cache colors and the other three instances are assigned 1 shared cache color.

Figure 9 shows the response times of the *latency* task instance when other instances are scheduled on the same VCPU. The y-axis denotes the response time normalized to the ideal response time described earlier. When our techniques are not used, we observed up to 1.5× of response time increase on the x86 platform and 1.4× increase on the ARM platform. When our techniques are used, the observed response time is close to the ideal response time. Therefore, we conclude that both vLLC and vColoring are effective in preventing intra-VCPU cache interference as well as inter-VCPU cache interference in a virtualization environment.

6.1.4 Results with PARSEC Benchmarks. We use the PARSEC benchmarks [8], which are closer to the memory access patterns of real applications compared to the synthetic task, *latency*. A total of eleven PARSEC benchmarks are 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 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 10 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

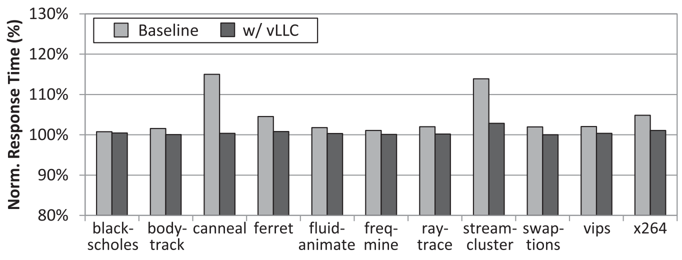


Fig. 11. Response times of the PARSEC benchmarks when synthetic tasks are scheduled on the same VCPU.

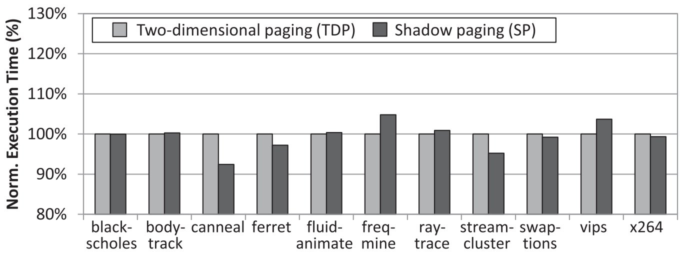


Fig. 12. Execution times of the PARSEC benchmarks under TDP and SP.

(Baseline), there is up to a 30% increase in execution time. When vLLC is used, only *streamcluster* has an execution time increase of 2%, and the other benchmarks have no noticeable difference. 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 the *latency* task are assigned to the same VCPU, and the `SCHED_RR` policy with a time quantum of 10 ms 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, as in the inter-VCPU interference experiment.

Figure 11 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.

6.1.5 Two-Dimensional Paging vs. Shadow Paging. Our experiments so far have been conducted with two-dimensional paging (TDP). Since our techniques can also be used with shadow paging (SP), we now compare the execution times of the PARSEC benchmarks under TDP and SP. Each PARSEC benchmark is executed alone in a VM with all the 32 cache colors assigned to it. Hence, there is no inter- and intra-VCPU cache interference in this experiment. Figure 12 shows the results. The y-axis denotes the execution time normalized to the TDP case. As can be seen, neither TDP nor SP dominates the other in terms of execution time. We have also measured with different

Table 3. Parameters for Taskset Generation

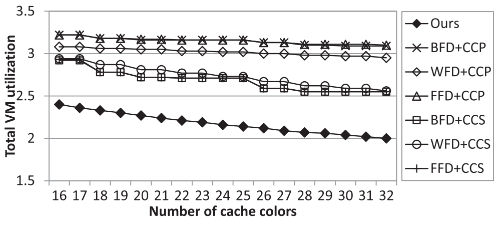
Type	Parameters	Values	Type	Parameters	Values
System	Number of PCPUs	4	**Taskset**	Total number of tasks	[10, 15]
	Number of VMs	2		Taskset util. ($U_{taskset}$)	3.0
	Number of VCPUs per VM	4	**WCET**	Memory accesses per job	[10^5, 10^6]
	VCPU replenishment period	[1, 10] ms		Neighborhood size	[16, 64]
	Cache (LLC) size	2048 KB		Locality	[1.5, 3.0]
	# of cache colors (N_{cache})	32		Task memory usage	[8, 40] MB
	Cache hit / miss delay	26 / 202 ns		*Resulting working-set size	[64 KB, 40 MB]
	Cache color reload time (Δ)	207 μs		*Resulting WCET	[8.47, 202.02] ms

numbers of cache colors on each benchmark, but could not find any case where one technique dominates the other. Although our techniques are independent of a specific address translation technique used, identifying the performance characteristics of TDP and SP in the context of real-time virtualization is an interesting topic that remains as future work.

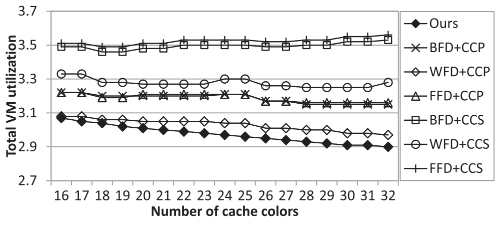
6.2 Cache Management Scheme

6.2.1 Experimental Setup. We randomly generated 10,000 tasksets with the parameters in Table 3. 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 Bach et al. [9]. 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 Thiebault et al. [53]. 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 ($U_{taskset}$) 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. The priorities of tasks and VCPUs are assigned by the Rate-Monotonic Scheduling (RMS) policy [35], with an arbitrary tie-breaking. The sporadic server policy is used for VCPU budget replenishment.

6.2.2 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



(a) VM utilization ($\Delta = 207\ \mu s$)



(b) VM utilization ($\Delta = 10$ ms)

Fig. 13. VM utilization with regard to the number of cache colors.

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 with regard to 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 Algorithm 2. Finally, we calculate the total utilization of VMs by summing up the utilization of all VCPUs.

Figure 13(a) 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 1.19× to 1.55× 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 13(b) shows the total VM utilization when Δ is 10 ms. 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 virtualization and yields a significant utilization benefit.

7 RELATED WORK

Predictable cache management schemes have been extensively studied in the context of non-virtualized systems. Page coloring [33] was first used to prevent cache interference from other tasks on a single-core platform. For multi-core platforms, page coloring was used to implement various cache partitioning schemes [34]. Specifically, Mancuso et al. [38] developed Colored Lockdown, which combines page coloring and cache lockdown to keep frequently accessed pages in a cache. Ward et al. [55] focused on cache management in mixed-criticality systems and proposed cache locking and scheduling techniques with page coloring. Ye et al. [61] developed COLORIS, which implements dynamic cache partitioning with page coloring. Bui et al. [9] developed a genetic algorithm to find a near-optimal cache allocation, and Paolieri [40] proposed IA^3 to partition a shared cache among CPU cores in a heuristic manner. With our proposed techniques, those prior schemes could possibly be applied to virtualization on a per-VCPU basis. For instance, once prior work determines cache allocation for VCPUs, just like for tasks in a nonvirtualized environment, vLLC and vColoring can realize the allocation by assigning cache colors to the tasks of each VCPU. Our task schedulability analysis given in Section 5.1 can be used to bound intra-VCPU cache interference. However, prior work cannot determine the budget of VCPUs and the number of cache colors for VMs, which our proposed algorithms do.

There also exist many research efforts on taking into account cache interference delay in the schedulability analyses. Altmeyer et al. [1] compared the performance of cache partitioning and cache-related preemption delay (CRPD) analysis on a single-core platform. Xu et al. [59] extended multi-core compositional analysis to incorporate cache interference delay, assuming that there is no shared cache. Lunniss et al. [36] extended CRPD analysis to single-core hierarchical scheduling. However, these approaches have not focused on the problem of a shared cache in multi-core virtualization, which our work addresses.

Prior work on software-based cache management for virtualization [32, 37, 46] implements page coloring in the hypervisor and allocates cache colors to VMs. This approach, however, cannot address cache interference among tasks running in the same VM, as we discussed in Section 2.3. Kim et al. [19] proposed a hardware-based solution to enable page coloring implemented in a guest OS to work. Recently, Xu et al. [58] proposed a cache management approach using Intel's CAT virtualization. Those hardware-based approaches are alternatives to vLLC and vColoring. One advantage of vColoring over them is that it can be used for unmodified guest OSs. Our proposed cache management algorithms can be used together with those hardware-based approaches.

8 CONCLUSION

In this article, we present our proposed predictable shared cache management framework for multi-core real-time virtualization. Our framework provides vLLC and vColoring, which are hypervisor-level techniques to enable the cache allocation of individual tasks running in a VM. We 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 explore other heuristics to improve cache management and address interference caused by contention in main memory in the context of real-time virtualization.

REFERENCES

[1] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. 2014. Evaluation of cache partitioning for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[2] ARM. 2016. ARM Cortex-A15 Reference Manual. (2016). Retrieved Dec. 14, 2016 from http://arm.com.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM Operating Systems Review* 37, 5 (2003), 164–177.

[4] Swagato Basumallick and Kelvin Nilsen. 1994. Cache issues in real-time systems. In *ACM Workshop on Language, Compiler, and Tools for Real-Time Systems*.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (USENIX ATC)*.

[6] Guillem Bernat and Alan Burns. 1999. New results on fixed priority aperiodic servers. In *IEEE Real-Time Systems Symposium (RTSS)*.

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. *ACM Operating Systems Review* 42, 2 (2008), 26–35.
[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[9] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. 2008. Impact of cache partitioning on multi-tasking real time embedded systems. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.
[10] José V. Busquets-Mataix, Juan José Serrano, and Andy Wellings. 1997. Hybrid instruction cache partitioning for preemptive real-time systems. In *Euromicro Workshop on Real-Time Systems (ECRTS)*.
[11] Tommaso Cucinotta, Gaetano Anastasi, and Luca Abeni. 2009. Respecting temporal constraints in virtualised services. In *IEEE International Computer Software and Applications Conference (COMPSAC)*.
[12] Robert I. Davis and Alan Burns. 2005. Hierarchical fixed priority pre-emptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*.
[13] General Dynamics. 2016. OKL4 Microvisor. (2016). Retrieved Dec. 14, 2016 from https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/.
[14] Per Hammarlund. 2013. 4th generation Intel core processor, codenamed haswell. In *Hot Chips: A Symposium on High Performance Chips (HC25)*.
[15] Hardkernel. 2016. ODROID. (2016). Retrieved Dec. 19, 2016 from http://www.hardkernel.com.
[16] Intel. 2016. Intel 64 and IA-32 Developer's Manual. (2016). Retrieved Dec. 14, 2016 from http://intel.com.
[17] Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li. 2009. A simple cache partitioning approach in a virtualized environment. In *IEEE Symposium on Parallel and Distributed Processing with Applications*.
[18] Mathai Joseph and Paritosh K. Pandya. 1986. Finding response times in a real-time system. *Computer Journal* 29, 5 (1986), 390–395.
[19] Daehoon Kim, Hwanju Kim, and Jaehyuk Huh. 2014. vCache: Providing a transparent view of the LLC in virtualized environments. *Computer Architecture Letters* 13, 2 (2014), 109–112.
[20] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2016. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems* 52, 3 (2016), 356–395.
[21] Hyoseung Kim, Arvind Kandhalu, and Ragunathan (Raj) Rajkumar. 2013. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.
[22] Hyoseung Kim and Ragunathan Rajkumar. 2012. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.
[23] Hyoseung Kim and Ragunathan Rajkumar. 2014. Memory reservation and shared page management for real-time systems. *Journal of Systems Architecture (JSA)* 60, 2 (2014), 165–178.
[24] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *ACM International Conference on Embedded Software (EMSOFT)*.
[25] Hyoseung Kim, Shige Wang, and Ragunathan Rajkumar. 2014. vMPCP: A synchronization framework for multi-core virtual machines. In *IEEE Real-Time Systems Symposium (RTSS)*.
[26] Hyoseung Kim, Shige Wang, and Ragunathan Rajkumar. 2015. Responsive and enforced interrupt handling for real-time system virtualization. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.
[27] Jan Kiszka. 2009. Towards Linux as a real-time hypervisor. In *Real-Time Linux Workshop (RTLWS)*.
[28] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: The Linux virtual machine monitor. In *Linux Symposium*, Vol. 1. 225–230.
[29] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. 2009. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*.
[30] Oded Lempel. 2011. 2nd generation Intel core processor family: Intel core i7, i5 and i3. In *Hot Chips: A Symposium on High Performance Chips (HC23)*.
[31] Hennadiy Leontyev and James H. Anderson. 2009. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems* 43, 1 (2009), 60–92.
[32] Ye Li, Richard West, and Eric Missimer. 2014. A virtualized separation kernel for mixed criticality systems. In *ACM Conference on Virtual Execution Environments (VEE)*.
[33] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. 1997. OS-Controlled cache predictability for real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*.

[34] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*.

[35] C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20, 1 (1973), 46–61.

[36] Will Lunniss, Sebastian Altmeyer, Giuseppe Lipari, and Robert I. Davis. 2014. Accounting for cache related pre-emption delays in hierarchical scheduling. In *International Conference on Real-Time and Network Systems (RTNS)*.

[37] Ruhui Ma, Wei Ye, Alei Liang, Haibing Guan, and Jian Li. 2013. Cache isolation for virtualization of mixed general-purpose and real-time systems. *Journal of Systems Architecture* 59, 10 (2013), 1405–1413.

[38] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*.

[39] Shuichi Oikawa and Ragunathan Rajkumar. 1998. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium (RTSS) Work-In-Progress*.

[40] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Robert I. Davis, and Mateo Valero. 2011. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*.

[41] QNX. 2017. QNX Hypervisor. (2017). Retrieved Apr. 14, 2017 from http://www.qnx.com.

[42] Wind River. 2016. VxWorks. (2016). Retrieved Dec. 14, 2016 from http://www.windriver.com.

[43] Saowanee Saewong, Mark H. Klein, Ragunathan Rajkumar, and John P. Lehoczky. 2002. Analysis of hierarchical fixed-priority scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[44] Filip Sebek. 2001. *Cache memories and real-time systems*. Technical Report. Mälardalen University. Retrieved Dec. 14, 2016 from http://www.es.mdh.se/publications/290-.

[45] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. 1986. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*.

[46] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W)*.

[47] Insik Shin, Arvind Easwaran, and Insup Lee. 2008. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[48] Insik Shin and Insup Lee. 2008. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 30.

[49] Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989), 27–60.

[50] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *ACM European Conference on Computer Systems (EuroSys)*.

[51] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computing* 44, 1 (1995), 73–91.

[52] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Embedded Software and Systems (ICESS)*.

[53] Dominique Thiebaut, Joel L. Wolf, and Harold S. Stone. 1992. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers* 41, 4 (1992).

[54] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective hardware/software memory virtualization. *ACM SIGPLAN Notices* 46, 7 (2011), 217–226.

[55] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*.

[56] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *ACM International Conference on Embedded Software (EMSOFT)*.

[57] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in xen. In *ACM International Conference on Embedded Software (EMSOFT)*.

[58] Meng Xu, Linh TX Phan, Hyon-Young Choi, and Insup Lee. 2017. vCAT: Dynamic cache management using CAT virtualization. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*.

[59] Meng Xu, Linh TX Phan, Insup Lee, Oleg Sokolsky, Sisu Xi, Chenyang Lu, and Christopher Gill. 2013. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *IEEE Real-Time Systems Symposium (RTSS)*.

[60] Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong, and Insik Shin. 2011. Implementation of compositional scheduling framework on virtualization. *ACM SIGBED Review* 8, 1 (2011), 30–37.

[61] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *International Conference on Parallel Architectures and Compilation Techniques*.

[62] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*.

Received December 2016; revised April 2017; accepted May 2017