ABSTRACT

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

1. INTRODUCTION

With the growth of processing core counts on recent processors, there is a strong demand for consolidating multiple real-time systems onto a single hardware platform. One of the promising solutions for such consolidation is virtualization. With virtualization, each consolidated system is contained within a virtual machine (VM), which is spatially isolated from other VMs by an additional address translation layer introduced by a hypervisor. Figure 1 illustrates the three address layers in modern virtualization platforms, such as Xen [6] and KVM [20]. Guest virtual pages for application tasks within a VM are mapped to guest physical pages by the guest OS of that VM, and those guest physical pages are mapped to host physical pages by the hypervisor. Using this approach, the hypervisor ensures that any software layout of the promising solutions for such consolidation is virtualization. With virtualization, each consolidated system is contained within a virtual machine (VM), which is spatially isolated from other VMs by an additional address translation layer introduced by a hypervisor. Figure 1 illustrates the three address layers in modern virtualization platforms, such as Xen [6] and KVM [20]. Guest virtual pages for application tasks within a VM are mapped to guest physical pages by the guest OS of that VM, and those guest physical pages are mapped to host physical pages by the hypervisor. Using this approach, the hypervisor ensures that any software...
LLC by using its own implementation of page coloring. vColoring, on the other hand, is designed for a VM that runs a guest OS having no page coloring support. vColoring allows the hypervisor to directly assign a portion of the host LLC to a task running in a VM. Hence, with vColoring, we can even control the cache allocation of tasks running on proprietary, closed-source OSs that do not support page coloring. We have implemented prototypes of vLLC and vColoring in the KVM hypervisor running on x86 and ARM multi-core platforms. Experimental results show that vLLC and vColoring are effective in controlling cache allocation to tasks and in addressing cache interference, on both an OS with page coloring (Linux/RK [16] [29]) and OSs without page coloring (vanilla Linux and MS Windows Embedded).

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

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

2. BACKGROUND

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

2.1 Scheduling and Cache Interference

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

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

2.2 Address Translation

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

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

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

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

2.3 Page Coloring

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

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

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

3. SYSTEM MODEL

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

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

Task Model: We consider periodic tasks with constrained deadlines. Task \( \tau_i \) is represented as follows:

\[
\tau_i := (C_i(k), T_i, D_i)
\]

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

\( C_i(k) \) values are assumed to be known ahead of time. They can be obtained by either measurement-based or static analysis tools [4]. We assume that \( C_i(k) \) is monotonically decreasing with \( k \). Each task \( \tau_i \) has a unique priority \( \pi_i \).

\footnote{Capturing the overhead of virtualization in task execution time is beyond the scope of this paper. However, we believe this does not limit the applicability of our work because its impact is relatively small (e.g., more than 90% of native performance can be achieved in full-virtualization mode with recent hardware virtualization techniques [8].)

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

\footnote{An arbitrary tie-breaking rule can be used to achieve this under fixed-priority scheduling.}}

VM Resource Model: Each VM is represented as follows:

\[ VM := (v_1, v_2, ..., v_{N_{vcpu}}) \]

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

\[ v_i := (C_{v_i}^i(k), T_{v_i}^i) \]

\[ C_{v_i}^i(k) \]: 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 \( v_i \)

\[ T_{v_i}^i \]: the budget replenishment period of a VCPU \( v_i \)

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

In the rest of the paper, \( C_i \) and \( C_i^j \) may be used instead of \( C_i(k) \) and \( C_i^j(k^j) \), 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, page coloring implemented in a guest OS cannot allocate cache colors to tasks in a VM due to the additional address layer in the hypervisor. vLLC overcomes this limitation. The keys to vLLC are (i) to provide a VM with “virtual LLC” information that corresponds to the cache colors assigned to the VM, and (ii) to map guest physical pages to host physical pages corresponding to the assigned cache colors. Figure 3 illustrates an example of vLLC. The virtual LLC provided to the VM is different from the actual LLC of the host machine in terms of the size of a cache and the number of cache sets, which are the main factors determining the number of cache colors. In Figure 3 since the
hypothesis assigns two colors out of four to the guest VM, the size and the number of cache sets of the virtual LLC are each half of those of the host LLC. Using this virtual LLC, the guest OS can identify that the number of available cache colors is two. The virtual LLC can be implemented by trapping page table misses and emulating cache-related operations, e.g., executions of a CPUID instruction on x86 architectures and accesses to CSSIDR and CSSERR registers on an ARM Cortex-A15 architecture.

In addition to the virtual LLC information, vLLC maps guest physical pages (GPPs) to host physical pages (HPPs) such that guest colors are mapped to their corresponding host colors. This can be easily done by the hypervisor because the hypervisor has both the virtual LLC information and the control of the GPP-to-HPP mapping. When a GPP needs to be mapped to an HPP, vLLC in the hypervisor checks the guest color of the GPP, finds out the corresponding host color, and maps the GPP to an HPP with that host color. For instance, in Figure 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 re-maps all GPPs used by that task to HPPs corresponding to the requested colors in the extra color set.

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

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

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

5. CACHE MANAGEMENT SCHEME

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

Recall that, in multi-core virtualization, there are two types of cache interference: inter- and intra-VCPU cache interference. To avoid both types of interference, a simple approach would be assigning each task a dedicated set of cache colors for its own exclusive use. Hence, tasks do not share their assigned cache colors with others, resulting in no conflicts in the LLC. We will refer to this approach as complete cache
partitioning (CCP). However, due to the availability of a limited number of cache colors, CCP may result in performance degradation. Many prior studies in non-virtualized environments [1, 10, 16] have shown that sharing of cache colors among tasks on the same core yields better task schedulability than CCP, and the resulting cache interference can be safely upper-bounded by the notion of cache-related preemption delay (CRPD). Therefore, our scheme builds on this idea in that (i) cache colors are not shared among tasks on different VCPUs to prevent inter-VCPU cache interference, and (ii) cache colors can be shared among tasks on the same core with the cost of intra-VCPU cache interference.

5.1 Schedulability Analysis

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

$$W_{i}^{v,n+1} = C_i + \sum_{v_k \in \mathcal{P}(v_i) \cap \gamma_{i} > \gamma_{j}} W_{i}^{v,n} + J_{i} \left( \frac{T_h}{T_h} \right) C_h$$

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

The schedulability of task $\tau_j$ running on a VCPU $v_i$ can be determined by:

$$W_{j}^{\tau,n+1} = C_j + \sum_{\tau_k \in \mathcal{V}(\tau_j) \cap \gamma_{i} > \gamma_{j}} W_{j}^{\tau,n} + J_{i} \left( \frac{T_h}{T_h} \right) C_h$$

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

$$\gamma_{j,i} = S_j \cap \bigcup_{\tau_k \in \mathcal{V}(\tau_j) \cap \gamma_{i} > \gamma_{j}} S_k$$

where $S_j$ is the set of cache colors assigned to $\tau_j$, and $\Delta$ is the maximum time needed to reload data in one cache color.

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

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

where $n$ is the index of the lowest-priority task in $\Gamma$.

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 scheduling.

### Algorithm 1 CacheToTaskAlloc($\Gamma$, $N_{cache}$)

**Input:** $\Gamma$: taskset, $N_{cache}$: the number of cache colors

**Output:** Utilization of $\Gamma$ if schedulable, and $\infty$ otherwise

1. if $N_{cache} = 0$ then
2. return $\infty$
3. $\text{cache_idx} \leftarrow 1$
4. for all $\tau_i \in \Gamma$ do
5. /* Find the number of cache colors for $\tau_i$ */
6. $S_i \leftarrow \arg\min_{1 \leq k \leq N_{cache}} \left( \frac{C_i}{T_i} \gamma_{i,n} \right) \frac{C_i}{T_i}$
7. /* Find cache-color indices for $\tau_i$ */
8. $S_i \leftarrow \emptyset$
9. for $k = 1$ to $S_i$ do
10. $S_i \leftarrow S_i \cup \{ \text{cache_idx} \}$
11. $\text{cache_idx} \leftarrow (\text{cache_idx} + 1) \mod N_{cache}$
12. if schedulable($\Gamma$) then
13. return $util(\Gamma)$
14. else
15. return $\infty$

5.3 Designing a Cache-Aware VM

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

Alg. 3 presents the pseudo-code of CAVM. It takes four input parameters: $\Gamma$ 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 $T$ is the VCPU period that will be assigned to all VCPUs in the VM. CAVM initializes the budget of each VCPU $v_i$ to be full, i.e., $C_i = T$ and the number of cache colors for $v_i$ ($S_i$) to zero (line 2).

CAVM consists of two phases. The first phase is allocating tasks to VCPUs. Our allocation strategy is to group cache-sensitive tasks into a “bundle” and allocate as many cache-sensitive tasks into a “bundle” and allocate as many tasks in the bundle as possible onto the same VCPU. To increase the benefit of cache sharing, and (ii) derives each VCPU’s resource requirement.
Algorithm 2 CacheAwareVM($\Gamma, N_{vcpu}, N_{cache}, T^v$)

**Input:** $\Gamma$: taskset, $N_{vcpu}$: the number of VCPUs, $N_{cache}$: the number of cache colors, $T^v$: VCPU period

**Output:** Success or Fail

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

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

**Input:** $\phi$: a bundle of tasks to be allocated, $V$: a set of VCPUs, $N_{rem}$: the number of cache colors

**Output:** $(v_i, k)$: a tuple of the best-fit VCPU and the number of additional cache colors needed

1: for $k \leftarrow 0$ to $N_{rem}$ do
2: \quad for all $v_i \in V$ in decreasing order of $\text{util}(\Gamma_i)$ do
3: \quad \quad if $\text{CacheToTaskAlloc}(\Gamma_i, \phi, S_i^v + k) \leq 1$ then
4: \quad \quad \quad return $(v_i, k)$
5: \quad return $(\text{invalid}, -1)$

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

**Input:** $\phi$: a bundle to be broken, size: the size constraint for the first sub-bundle, $N_{cache}$: the number of colors

**Output:** $(\phi', \phi'')$: a tuple of sub-bundles

1: $\phi' \leftarrow \phi' \leftarrow \emptyset$
2: for all $\tau_i \in \phi$ in increasing order of cache sensitivity do
3: $\phi' \leftarrow \phi' \leftarrow \phi''\lor \tau_i$
4: /* Get $\text{util}(\phi')$ assuming each task uses one color */
5: if $\text{util}(\phi') \leq \text{size}$ then
6: break
7: return $(\phi', \phi''\prime)$

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 $V$ denote a set of VCPUs of all VMs to be consolidated. Then, the total utilization of VMs with $k$ cache colors is given by:

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

This section presents our experimental results on our v LLC, v Coloring, and cache management scheme.

6.1 v LLC and v Coloring

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

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

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

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

Implementation Overhead: Table 1 shows the computational overhead of v LLC and v Coloring, measured with hardware performance counters on the x86 and ARM platforms. v LLC 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 5.3 and the overhead added by the color check of v LLC 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 v LLC is acceptably small. v Coloring re-maps GPPs when cache colors are assigned to a task. Since the
In this subsection, we evaluate our real-time cache management scheme for multi-core virtualization. To do this, we use randomly-generated tasksets and capture the total utilization of VMs as the metric.

**Experimental Setup:** We generated 10,000 tasksets with the parameters in Table 2. Cache hit/miss delay and cache color reload time (∆) were obtained by measurement on our ARM platform. To generate a WCET function (\(C_{i}(k)\)) for each task \(i\), we use the method described in [10]. This method first calculates a cache miss rate for given cache size, neighborhood size, locality, and task memory usage, by using the analytical cache behavior model proposed in [41]. It

---

**ARM platform w/ vLLC**

**Figure 5:** Execution times of the PARSEC benchmarks when synthetic tasks are scheduled on the same VCPU

---

**ARM platform**

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

---

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

---

**Figure 10:** Cache management scheme for multi-core virtualization.
### Table 2: Parameters for taskset generation

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Number of PCPUs</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Number of VMs</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Number of VCPUs per VM</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>VCPU replenishment period</td>
<td>10 msec</td>
</tr>
<tr>
<td></td>
<td>Cache (LLC) size</td>
<td>2048 KB</td>
</tr>
<tr>
<td></td>
<td># of cache colors (N_{cache})</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Cache hit delay</td>
<td>26 nsec</td>
</tr>
<tr>
<td></td>
<td>Cache miss delay</td>
<td>202 nsec</td>
</tr>
<tr>
<td></td>
<td>Cache color reload time ((\Delta))</td>
<td>207 µsec</td>
</tr>
<tr>
<td>Taskset</td>
<td>Total number of tasks</td>
<td>[10, 15]</td>
</tr>
<tr>
<td></td>
<td>Total utilization ((U_{taskset}))</td>
<td>3.0</td>
</tr>
<tr>
<td>WCET</td>
<td>Memory accesses per job</td>
<td>[100000, 100000]</td>
</tr>
<tr>
<td></td>
<td>Neighborhood size</td>
<td>[16, 64]</td>
</tr>
<tr>
<td></td>
<td>Locality</td>
<td>[1.5, 3.0]</td>
</tr>
<tr>
<td></td>
<td>Task memory usage</td>
<td>[8, 40] MB</td>
</tr>
<tr>
<td></td>
<td>*Resulting working-set size</td>
<td>[64 KB, 40 MB]</td>
</tr>
<tr>
<td></td>
<td>*Resulting WCET</td>
<td>[8.47, 202.02] msec</td>
</tr>
</tbody>
</table>

Figure 8: Some of WCETs generated for our experiments then generates an execution time with the calculated cache miss rate, the timing delay of a cache miss, and the number of memory accesses. With this method, we were able to generate WCETs with different cache sensitivities, as shown in Figure 8. Then, the total taskset utilization \((U_{taskset})\) is split into \(n\) random-sized pieces, where \(n\) is the total number of tasks. The size of each piece represents the utilization of the corresponding task when one cache color is assigned to it. The period of a task \(\tau_i\) is calculated by dividing \(C_i(1)\) by its utilization. Once a taskset is generated, they are randomly distributed to two VMs, each of which has four VCPUs. Within each VM, the priorities of tasks are assigned by the Rate-Monotonic Scheduling (RMS) policy \[25\]. The priorities of VCPUs are arbitrarily assigned since they use the same period. The sporadic server policy is used for VCPU budget replenishment.

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

Figure 9 shows the total VM utilization as the number of cache colors increases. Since CCP cannot find a scheduleable allocation if the number of colors is smaller than that of tasks, we compare only the cases where the number of cache colors is greater than 15. Our scheme outperforms all other approaches, yielding 1.18× to 1.54× lower utilization. This is because our scheme allocates cache-sensitive tasks together to the same VCPU to increase the benefit of cache sharing and finds the minimum total VM utilization for a given number of cache colors. The heuristics with CCS perform better than the ones with CCP. This is because \(\Delta\) 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 10 shows the total VM utilization when \(\Delta = 10\) msec. This experiment is to evaluate our scheme when CRPD is extremely high. Overall, the benefit of using more cache colors is smaller compared with the previous experiment. Our scheme outperforms other approaches because it can balance between the utilization gain and CRPD from cache sharing. The heuristics with CCS perform worse than the ones with CCP due to the high CRPD. In case of BFD+CCS and FFD+CCS, the utilization even increases as more cache colors are provided. WFD+CCS is affected less by the high CRPD compared with BFD+CCS and FFD+CCS, because WFD results in a fewer number of tasks per VCPU. Based on these results, we conclude that our scheme allocates cache colors efficiently in a virtualization environment and yields a significant utilization benefit.

7. RELATED WORK

Cache management schemes have been extensively studied in the context of non-virtualized systems. Liedtke et al. \[23\] proposed to use page coloring to prevent cache interference from other tasks in a single-core platform. Lin et al. \[24\] conducted a comparative study on various multi-core cache partitioning schemes by implementing them with page coloring. Mancuso et al. \[28\] proposed the Colored Lockdown technique that combines page coloring and cache lockdown to better keep the frequently accessed pages of real-time tasks in a cache. Ye et al. \[44\] developed COLORIS that supports both static and dynamic cache partitioning based on page coloring. Ward et al. \[43\] focused on cache management issues in multi-core mixed-criticality systems and proposed cache locking and scheduling techniques that use page coloring. Bui et al. \[10\] developed a genetic algorithm to find a near optimal solution for cache partition allocation on a...
single-core platform. Paoliieri [30] proposed IA3, which is a heuristic algorithm for allocating cache partitions to cores in a multi-core real-time system. All these schemes, however, cannot be directly applied to a virtualized system.

There also exist many research efforts on taking into account cache interference delay in the schedulability analyses [22, 27, 34]. Specifically, Altmeyer and Davis [5] compared the performance of cache partitioning and cache-related pre-emption delay (CRPD) analysis on a single-core platform. Xu et al. [26] extended multi-core compositional analysis to incorporate cache interference delay, assuming that there is no shared cache. Lunniss et al. [25] extended CRPD analysis to a single-core hierarchical scheduling environment. However, none of these approaches focuses on a shared cache in a multi-core platform.

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

8. CONCLUSIONS

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

As future work, we plan to address temporal interference from main memory in a virtualization environment.

9. REFERENCES

[21] O. Lempel. 2nd generation Intel Core processor family: Intel Core i7, i5 and i3. In Hot Chips (HotChips), 2011.