vCAT: Dynamic Cache Management using CAT Virtualization

Meng Xu Linh Thi Xuan Phan Hyon-Young Choi Insup Lee
University of Pennsylvania

***Abstract*—This paper presents vCAT, a novel design for dynamic shared cache management on multicore virtualization platforms based on Intel's Cache Allocation Technology (CAT). Our design achieves strong isolation at both task and VM levels through cache partition virtualization, which works in a similar way as memory virtualization, but has challenges that are unique to cache and CAT. To demonstrate the feasibility and benefits of our design, we provide a prototype implementation of vCAT, and we present an extensive set of microbenchmarks and performance evaluation results on the PARSEC benchmarks and synthetic workloads, for both static and dynamic allocations. The evaluation results show that (i) vCAT can be implemented with minimal overhead, (ii) it can be used to mitigate shared cache interference, which could have caused task WCET increased by up to 7.2×, (iii) static management in vCAT can increase system utilization by up to 7× compared to a system without cache management; and (iv) dynamic management substantially outperforms static management in terms of schedulable utilization (increase by up to 3× in our multi-mode example use case).**

I. Introduction

Modern real-time systems are becoming increasingly complex: they perform a variety of functions that are diverse in resource demands and often dynamic in nature. Traditionally, these systems have been implemented on dedicated hardware; however, increasingly their functionality is moving into virtual machines (VMs) that run on a shared virtualization platform, such as Xen, on powerful multicore servers. This trend towards virtualization of real-time systems has many potential benefits – including consolidation, elasticity, and better scalability. However, today's virtualization technology is not a perfect match for real-time systems, as it cannot yet deliver strong timing isolation among VMs and tasks.

An important challenge towards timing isolation in virtualization on multicore hardware is that of the shared cache interference: two concurrently running tasks (within the same VM or from different VMs) may interfere with one another by accessing memory addresses that are mapped to the same cache set, resulting in cache misses that could cause deadline misses. At first, it seems that one can always take such overhead into account in the timing analysis; however, precisely accounting for this overhead is theoretically challenging and would require fine-grained information about the layouts of the tasks in the cache and their access patterns. Further, even if precise analyses were possible, they would still not give the "cache isolation" guarantee that applications require.

Therefore, recent research has begun to develop ways to mitigate this type of interference, most notably through cache partitioning. The idea is to divide the cache into multiple non-overlapped partitions and assign them to tasks; if concurrently running tasks are assigned different disjoint sets of partitions and they use only those, then they cannot interfere with each other via concurrent cache accesses. (Note that two sequential tasks using a common partition could still interfere with each other via preemption or migration. However, there is already a rich real-time literature in handling this type of interference, which we expect can be adapted here. Therefore, we will focus only on handling the interference caused by concurrent accesses to the last level shared cache in this paper.)

Existing work in this domain relies on software-based solutions, such as page coloring [17], for cache partitioning. Most previous solutions focus on improving the average performance of the system (e.g., [10, 11, 24]) and thus are not suitable for real-time systems that require worst-case performance guarantees. Recent work from the real-time systems community has developed methods for achieving cache-level task isolation using page coloring (e.g., [15, 30]); however, it is restricted to static cache partitioning, where a fixed set of partitions is statically assigned to each task at initialization. While this approach is simple and easy to implement, it can substantially under-utilize the cache and CPU resources, and it does not work well for systems where the tasks' timing constraints and CPU/cache demands vary dynamically at run time, such as in multi-mode systems (as we shall illustrate in Section VI-D).

To bridge this gap, we present a new approach to cache management of real-time virtualization systems that can deliver strong (shared) cache isolation at both VM and task levels, and that can be configured for both static and dynamic allocations. Unlike existing work, which is software-based, our approach takes advantage of the Cache Allocation Technology (CAT), a hardware feature recently added in Intel multicore hardware for achieving core-level cache partitioning; therefore, it is much more efficient than software-based techniques. Since CAT only provides core-level cache isolation, we introduce vCAT, a novel design for CAT virtualization that can be used to achieve hypervisor- and VM-level cache allocations. Our approach to virtualizing cache partitions is analogous to memory virtualization: as the hardware provides a number of (indistinguishable) physical partitions, we can expose some number of "virtual partitions" to each VM and then transparently map them to physical partitions in the hypervisor; each VM can then allocate its virtual partitions to its tasks statically or dynamically at run time.

Virtualizing cache partitions also presents unique differences and new challenges that need to be considered in our design: First, although cache partitions can be "preempted" just like physical pages, the hypervisor needs not (and cannot) save the contents of the preempted partition; instead, it can rely on the tasks to repopulate the partitions they are being assigned. Second, the current hardware requires that partitions mapped to a core must be contiguous; this needs to be enforced when allocating partitions at the hypervisor and VM levels, and it necessitates a procedure for handling partition fragmentation.

Finally, to provide strong cache isolation among tasks, we need to guarantee that a task never accesses its old partitions that are currently allocated to another task. Therefore, the guest kernel may need to flush the partitions when it changes the partition allocations to the tasks, and we need an efficient method for flushing to minimize overhead.

To illustrate the feasibility of our approach, we provide a proof-of-concept prototype of vCAT on top of Xen and LITMUS^{RT}. Our microbenchmarks show that our vCAT prototype introduces only a small overhead. Our extensive evaluations on PARSEC benchmarks and synthetic real-time workloads show that, compared to the vanilla LITMUS^{RT}/Xen, static cache management in vCAT can reduce the task WCET in the presence of interfering tasks (by up to 7.2$\times$) and increase the system's schedulable utilization by up to 7$\times$. In addition, dynamic allocation outperforms static allocation substantially; it improves schedulable utilization by 3$\times$ in an example system with dynamic cache demands and timing requirements.

In summary, we make the following contributions:

- The design of CAT virtualization for cache management in real-time multicore virtualization systems;
- a prototype implementation of our CAT virtualization;
- extensive evaluation of the prototype's overhead; and
- extensive empirical evaluation of the WCET and performance benefits of static and dynamic managements.

The paper is organized as follows. We discuss related work in Section II and an empirical study of the CAT technology in Section II. The design and implementation of our approach are in Sections IV and V, and we present evaluation results in Section VI before concluding.

II. Related Work

There is a rich literature on accounting techniques for cache-related overhead in real-time systems. This line of work focuses on private caches [3] and on analyzing the overhead caused by preemptions or migrations [4] [6], rather than on mitigating the overhead due to shared cache interference. However, we expect that compositional approaches, such as [22, 28], can be applied on top of our design to establish cache-aware schedulability (which is an interesting research problem but out of scope of this paper).

In non-virtualization settings, cache management has been extensively studied for improving systems' real-time performance. Software-based approaches have been used to provide both static [14, 21] and dynamic [25, 29] allocations. Previous work has also explored hardware-based approaches to dynamically allocate cache partitions to tasks, e.g., using the PL310 cache controller [27] or the Intel's CAT [32]. Some techniques also combine software and hardware approaches to support finer-granularity partitions [8, 16]. However, these techniques cannot be directly applied to virtualization platforms.

In the virtualization setting, prior research focuses primarily on improving average performance [10, 11, 24], and it considers only the hypervisor-level allocation and not VM-level allocation. To improve worst-case real-time performance, recent research [15, 30] has developed software-based techniques that can provide task-level cache isolation; however, it is limited to only static management, which can substantially under-utilize cache and CPU resources, especially in cases where tasks' timing behavior can change dynamically at run time. Kim et al. [13] proposed vCache, a new hardware design for the last-level shared cache that allows a guest OS to control the cache allocation for tasks; however, vCache requires hardware modification and thus cannot be supported by current commodity hardware. In contrast, vCAT introduces a new virtualization layer for cache partitions on top of the Intel's CAT to provide support for *dynamic* cache allocation *at the task level*, which cannot be achieved by both the Intel's CAT itself and the existing cache management for virtualization settings [15, 30]. To the best of our knowledge, vCAT is the first to provide dynamic cache management for real-time virtualization systems on commodity multicore platform that can deliver strong cache isolation among tasks and VMs, and it is also the first that uses Intel's CAT in a real-time virtualization setting to achieve task-level cache isolation.

We note that the Intel's CAT has been incorporated in Xen; however, the current Xen CAT only supports cache allocation at the VM level (and not tasks), does not allow more VMs with distinct cache configurations than the number of COS registers (i.e., four on our machine), and does not guarantee cache isolation when the VM cache partitions are reconfigured. Due to these limitations, we implemented a completely different CAT management module in Xen based on our vCAT design.

An interesting use case of cache management is in defending shared-cache side channel attacks [19]. We expect that vCAT could be applied here as well, while also providing cache isolation among tasks, a property that cannot be achieved by existing CAT-based solutions such as [18].

III. Experimental Study of Intel Cache Allocation Technology (CAT)

The Intel's CAT is a new hardware feature that allows the OS or hypervisor to control the allocation of the shared last-level cache to the physical cores. In this section, we present a study of its behavior in the current hardware, and highlight its implications on the design of CAT virtualization. Our study was performed using the Intel MSR tool [2] on an Intel Xeon E5-2618L v3 processor, which has a 20MB shared cache.

A. Background on CAT

The CAT divides the shared cache into N non-overlapped equal-size cache partitions; for instance, $N = 20$ for our experimental platform. A set of such cache partitions (specified as an N-bit mask) can be allocated to a CPU (core) by programming two model-specific registers: (1) The Class of Service (COS) register, which has an N-bit Capacity Bitmask (CBM) field to specify a particular cache partition set, and (2) the CPU's IA32_PQR_ASSOC (PQR) register, which has a COS field for linking a particular COS to the CPU; when this field is set to the ID of a COS register, CAT enforces that all cache allocation requests from the CPU will only happen in the cache partitions specified by the CBM of that COS register. For example, to allocate partitions 0 to 3 to a CPU, we set 1's for the bits 0 to 3 (and zeroing the remaining) of the CBM field of the associated COS register.

We conducted a series of experiments to validate the operation of the Intel's CAT. Our experiments confirmed that the Intel's CAT specification is correct in stating the following constraints and in the way CAT works as advertised: (1) The current CAT implementations only support an allocation with at least two partitions; (2) the number of cache partitions per CPU should not exceed the number of available partitions (which varies across processors); and (3) the partition set of a CPU can only be made of contiguous cache partitions.

B. *Effects of cache partition configuration on WCET*

To validate that any combination of contiguous partitions with the same number of partitions has the same effect on the task's worst-case execution time (WCET), we constructed a task that sequentially accesses every 64 bytes in a 1MB array for 100 times, and executed the task alone on a CPU. We enumerated all possible combinations of two contiguous partitions; for each combination, we allocated the corresponding partitions to the CPU, and measured the WCET of the task across 25 runs. The results show the same WCET for the task with the same array across all combinations.

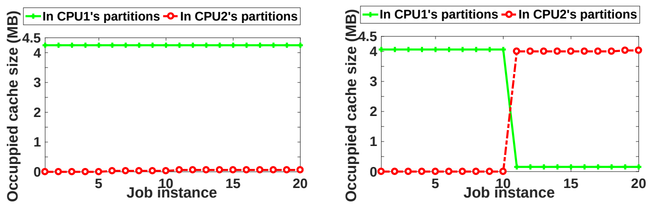
Finding 1. *Any set of contiguous partitions with the same number of partitions have the same effect on WCET.*

C. *Cache lookup control*

Under dynamic cache allocations, the partitions allocated to a task can change over time. When this happens, the task should only be allowed to access the cache lines in the newly assigned partitions and not the old ones. The CAT ensures that the task's new cache allocations (which happen in cases of cache misses) will happen in the new partitions, but the SDM does not specify the CAT behavior for cache lookup requests (which happen in cases of cache hits), which suggests that a task may still be able to read from the old partitions. If so, the task can interfere with another task that is currently using the old partitions. To examine whether CAT controls the cache lookup requests, we performed the following experiment using the Intel MSR tool on Linux 3.10.31 on our implementation platform, which has 20 cache partitions of size 1MB each.

Experiment. We reserved cache partitions 0–7 (CBM bitmask 0×000FF) to CPU1 and partitions 8–15 (CBM bitmask 0×0FF00) to CPU2. We flushed the entire cache initially, and mitigated potential interference to CPU1 and CPU2 by moving all system services to the remaining cores and assigning to them the remaining partitions (partitions 16–19). We created a periodic task that sequentially accesses a 4MB array. We executed the first 10 jobs of the task on CPU1; upon completion, we migrated it to CPU2 and continued its execution until completing the next 10 jobs. Using the Intel Cache Monitoring Technology [1], we measured the occupied cache size in each CPU's cache partition set when each job finished.

Results. As shown in Fig. 1(a), the size of the occupied cache in CPU1's partitions is always approximately the same as the array size (4MB), whereas the size of the occupied cache in CPU2's partitions is close to zero, even when the task executed on CPU2. This can be explained as follows. When the first job accessed the array, it experienced compulsory cache misses and thus was allocated cache lines in CPU1's partitions (as enforced by CAT). However, since the entire task's array (4MB) fits within CPU1's partitions (8MB), the subsequent jobs would experience cache hits and access the array directly from these partitions. Our experimental results show that this happened even when the task was already migrated to CPU2 (and should no longer use CPU1's partitions), which shows that CAT does *not* control the cache lookup.



(a) Without cache flush (b) With cache flush

Fig. 1: No cache lookup control in CAT.

Finding 2. *The CAT does not control cache lookup requests, and thus does not guarantee that cache accesses happen* only *in the currently assigned partitions.*

Challenge. Due to the lack of cache lookup control, when a partition that was owned by a task *A* is re-assigned to a task *B*, the previous cached items of *A* in this partition are simply looked up as before. As a result, if *B* (the current owner) does not happen to evict these cached items of *A* from the partition, then *A* will continue to reference its cached items in *B*'s partition.

To ensure that tasks have complete control of their partitions, in certain situations it is necessary to flush the content of a task in its old partitions when the task's partitions are changed. Our CAT virtualization uses this approach for real-time tasks, thus providing strong isolation among them. Our design also supports *shared* partitions (disjoint from those of real-time tasks) for best-effort tasks, where tasks can share the same set of partitions and no flushing is necessary.

Validation. To validate the effect of flushing, we performed the same experiment as above, except that we flushed the cache immediately after migrating the task from CPU1 to CPU2. As shown in Fig. 1(b), the size of the occupied cache in CPU1's partitions is dropped to nearly zero as soon as the task migrates to CPU2, whereas the size of occupied cache in CPU2's partitions increased to 4MB. This confirms that, with flushing, the task only accesses its newly assigned partitions.

IV. CAT Virtualization Design

In this section, we describe the design of vCAT, as well as the necessary changes to the guest kernel and the hypervisor.

A. *Overview and roadmap*

At a high level, our approach to virtualizing cache partitions is similar to classical virtual memory; however, there are also several important differences. We begin with the similarities: the hardware provides a fixed number of physical cache partitions that can be allocated to tasks, just like it provides a fixed number of physical memory pages, and—just like physical

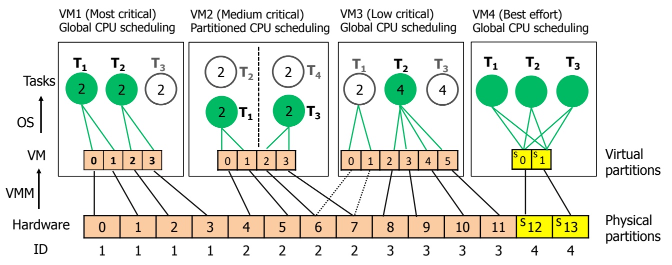


Fig. 2: Dynamic cache management with CAT virtualization. Tasks in green (white) are currently running (waiting). Partitions in orange (yellow) are isolated (shared) partitions.

memory pages—the individual partitions are indistinguishable from each other, so it should not matter to a task which specific partitions it is using. Thus, we can simply expose some number of “virtual partitions” to each VM (Section IV-B) and then transparently map them to physical partitions in the hypervisor, using a data structure that somewhat resembles a page table (Section IV-C), and each VM can then allocate its virtual partitions to tasks dynamically at run time (Section IV-D). Fig. 2 shows an example of a system with CAT virtualization.

However, there are also two key differences. First, although cache partitions can be “preempted” just like physical pages the hypervisor needs not – and, indeed, cannot – save the contents of the partition it is preempting. Instead, it can rely on the tasks to repopulate the partitions they are being assigned. Second, the CAT specification contains a requirement that allocations are contiguous. This needs to be taken into account when allocating partitions, and it requires a procedure for handling partition fragmentation (Section IV-E).

The technical approach is similar to virtual memory: allocations are enforced at a per-core level, using the COS registers, just like each core has a separate page-directory base register (CR3), and the hypervisor is able to request traps on accesses to these registers to perform a “partition context switch” (Section IV-F). When the hypervisor or guest kernel changes the partition allocations to the VMs or tasks, it may need to flush the partitions if necessary (Section IV-G).

B. API changes

In order to implement a virtual CAT, we need to make four changes to the API: (1) the VMM must be able to tell the guest kernel how many partitions are available; (2) tasks must be able to request partitions from the guest kernel; (3) the guest must have a way to report the allocation, as well as any changes, to the VMM; and (4) the operator must have a way to control how partitions are divided up between the various VMs and to set/modify the mapping from virtual to physical partitions for each VM. We describe each in turn.

Since the hardware already contains a mechanism for reporting the number of available partitions (via the `cpuid` instruction), we can simply repurpose this mechanism to achieve the first goal: the hypervisor can trap on the `cpuid` instruction – which Xen already does – and change the relevant value. We do not see a good reason for reporting more partitions than are physically available, but there may be good reasons to report fewer, e.g., if the operator has divided up the available partitions between multiple VMs. If the guest kernel were to allocate more virtual partitions than the hypervisor is willing to give it, this would lead to many expensive preemptions, so it may be preferable to report the smaller number right away.

The current Linux API does not contain a system call for requesting cache partitions, so we added a call of our own that simply takes a requested number of partitions as its argument. Taking a COS-style bitmask seemed unnecessary because a task should not need to know which specific partitions it is being given – much like a task normally should not need to know which physical memory pages it is using.

We achieve the third goal by providing virtual COS registers. Thus, the guest kernel can use the same procedure to allocate partitions, whether it is running in a VM or on bare hardware. A hypercall could be added if the guest needs to communicate richer information to the VMM, e.g., to request a temporary increase in the number of partitions it can use.

To achieve the fourth goal, we added several hypercalls that can influence the partition-to-VM allocation (which we describe next), and we provided a small command-line utility for the operator to use.

C. Hypervisor-level partition allocation

When allocating partitions to VMs, the hypervisor can take three basic approaches: first, it can divide up the available physical partitions, which guarantees each VM that its partitions will not need to be preempted; second, it can allow the partitions to become oversubscribed, which can lead to preemptions; or, third, it can allow partitions to be transparently shared between VMs. The first two options are similar to physical memory, whereas the third is unique to the cache.

The first approach is clearly preferable for tasks and VMs with strict real-time requirements, since it achieves very good isolation; however, given the very small number of partitions that are available on current CPUs, it seems practical for only the most critical tasks and VMs (e.g., VM1 in Fig. 2). We expect the second approach to be the default choice (e.g., VM2 and VM3 in Fig. 2). The third approach could be used for best-effort tasks: for instance, the operator could reserve 15 of the 20 partitions for hard real-time tasks and share the remaining five among all the non-real-time tasks. This would prevent the latter from interfering with the former. In Fig. 2, this approach was used for tasks in VM4.

Internally, the hypervisor requires only two data structures to implement these policies: (1) for each VM i, a mapping from virtual partition numbers v to physical partition numbers $P_i(v)$, and (2) a flag for each physical partition to indicate whether the partition is shared. For example, the system in Fig. 2 set the shared flags (denoted as S in the figure) for partitions 12 and 13. In Section IV-F, we describe how these data structures are used during a partition context switch.

To meet the CAT specification, we enforce that the number of partitions allocated to each VM i must be at least two, and the partition numbers $P_i(v)$ must be contiguous. In our current prototype, these data structures must be configured manually by the operator. (The operator can use the provided utility to modify these data structures at run time, e.g., when a new VM is created or an existing VM is destroyed.) However, we note that there is a rich literature on working-set estimation [9, 31] and on memory management for real-time tasks [12, 20], which can be adapted for use with cache partitions.

D. Guest-level partition allocation

Just like the hypervisor, the guest kernel must allocate the available partitions to its tasks, based on the requests they have made. However, unlike the partition-to-VM allocation which does not change frequently, the partition-to-task allocation is done dynamically as tasks are scheduled. In our prototype, we simply allocate the partitions to real-time tasks based on either a first-come-first-served basis or criticality, and we share any unallocated partitions among all the best effort tasks. Since allocating zero partitions would effectively disable the cache, which would lead to an enormous slowdown, the kernel reserves a small number of partitions for these tasks and does not allow these partitions to be reserved by the real-time tasks. The kernel always allocates at least two, and always contiguous, virtual partitions to a task.

E. Partition defragmentation

Although future hardware may no longer need the contiguous partition allocations, current hardware does. This raises the possibility of "partition fragmentation": it could be that there are k total partitions available but not with contiguous partition numbers, which would prevent a request for k partitions from being satisfied at that point. This problem can appear both in the hypervisor and in the guest kernel.

However, there is an easy way to fix this problem when it appears: the kernel or hypervisor can "defragment" the partitions by preempting some allocations and by replacing them with others, so that the unallocated partition numbers are again contiguous. The caveat is that this can cause a temporary loss of performance as the tasks are repopulating their preempted partitions, which can lead to deadline misses. This can be alleviated somewhat by moving the partitions of less critical tasks first, or by carefully configuring the virtual-to-physical mappings. In our prototype, we disable automatic defragmentation in the highly critical VMs and at the hypervisor (since reallocating partitions to VMs requires flushing the addresses of some VMs); the operator can trigger defragmentation manually when she considers it to be safe.

F. Partition context switch

In order to enforce the partition allocation at the VM level, the hypervisor must update the COS registers whenever it performs a partition context switch. To this end, the hypervisor maintains, for each physical partition n, the ID $I(n)$ of the VM that is currently using that partition.

A partition context switch from a VCPU of VM i to a VCPU vcpu_j of VM j is done as follows: the hypervisor first iterates over all of the target's virtual partition numbers $n = 0 \ldots k$; if the n.th bit of vcpu_j's virtual COS is set, the hypervisor looks up the corresponding physical partition number $P_j(n)$ and checks whether (1) $I(P_j(n)) \neq j$, and (2) the partition $P_j(n)$ is not shared. If the preemption-based strategy is set and VM j has higher criticality than every VM $I(P_j(n))$ for which both conditions (1) and (2) hold, then the hypervisor preempts the VCPU that is currently using $P_j(n)$ and clearing all bits of the COS register of the core on which that VCPU is running. Next, the hypervisor updates the physical COS register of the core on which vcpu_j is scheduled (by setting only $P_j(n)$, $n = 0 \ldots k$ and clears all the other bits), setting the ID $I(P_j(n)) = j$ for all $n = 0 \ldots k$. In addition, if it did preempt a VCPU, it also then invokes a rescheduling event to the scheduler. Notice that a preemption happens only in cases where a partition is assigned to more than one VM, but is not shared. In Fig. 2, physical partitions 6 and 7 are oversubscribed by both VM2 and VM3; since VM2 has higher criticality than VM3, its VCPUs can preempt VM3's VCPUs. Here, the hypervisor preempts the VCPU currently executing T_1 of VM3, and switches the partitions' owner to the VCPU on which T_3 of VM2 will execute.

If the guest kernel is not CAT-aware, it will not modify its virtual COS from the default value (all partitions active), and the above process is sufficient. If the guest does modify the virtual COS (e.g., during a guest-level partition context switch), the kernel must intercept these accesses and modify the physical COS register and the ID of the physical partitions. Fortunately, the COS registers are machine-specific registers; they are updated with the `wrmsr` instruction, which is privileged and causes a guest exit when invoked. When the hypervisor intercepts an access, the procedure is analogous to an inter-VM partition context switch. Notice that the hypervisor cannot know whether the guest kernel is reassigning a partition from one task to another; hence, the guest must keep ownership information for the virtual partitions similar to the hypervisor's $I(n)$.

G. Flushing

At first glance, it may seem that, when a cache partition is reassigned from one VM or task to another, updating the COS register is all that is required. However, as discussed in Section III-C, if the new owner does not happen to evict all cached content of the previous owner from the partition, the previous owner will continue to reference its cached items and prevent the new owner from gaining full control over the partition. To reliably avoid this, it is necessary to flush the previous owner's content from the cache when it is assigned a new set of partitions that is *not* a superset of its previous partitions, if the previously-assigned partitions are not shared.[1]

For this purpose, we maintain for each task τ_i its currently assigned set of virtual partition numbers S_i. A flushing is initiated when τ_i is scheduled to run and if it is assigned a new set of partition numbers S'_i such that $S'_i \not\supseteq S_i$ and there exists $v \in S_i - (S_i \cap S'_i)$ where the shared flag of v is 0. Consider VM1 in Fig. 2, for instance, which has two VCPUs. Suppose T_3 was previously assigned partitions $S_3 = \{0,1\}$, but it is preempted by T_1. Suppose later, T_2 finishes, then the kernel will assign partitions $S'_3 = \{2,3\} \not\supseteq S_3$ to T_3. Since T_3 may still access its old partitions 0 and 1 via cache hits, which are now owned by T_1, we need to flush the content of T_3 in these partitions.

Notice that partitions are only re-allocated at the hypervisor level when a mapping of virtual-to-physical partition numbers

[1]If the previous owner's new partitions include all of its old partitions, it experiences cache hits only in its own (old/new) partitions, and thus cannot access the partitions currently assigned to another running task.

changes; therefore, flushing at the hypervisor level happens only very infrequently (i.e., during defragmentation or triggered by the operator when a VM joins or leaves the system, or when some VMs request more partitions).

Ideally, we would like to simply flush the specific partition whose ownership is changing (e.g., partitions 0 and 1 in the above example). However, the current CAT does not provide a way to do this, so our only option is to flush the cache contents of the *entire* VM or task that is being replaced. The Intel CPUs offer two ways to do this: the `clflush` instruction, which flushes the cache line that contains a specific linear address, and the `wbinvd` instruction, which writes back any modified data in the cache and then invalidates the *entire* shared cache. (A third option, the `invd` instruction, would simply discard modified cache lines, so it is not an option here.) When the `clflush` instruction is used to flush the cache content of a task, it is issued on all valid linear addresses (not the entire virtual address space) of the task, with a step of a cache line (i.e., 64B). The linear addresses of a task can be found in the task's control block.

Neither option is strictly better than the other: `clflush` can avoid side effects on other tasks by flushing specific content, and it is potentially faster than `wbinvd` if the previous owner's working set is small; however, it can also be slower if there are a lot of addresses to be flushed. For simplicity, our implementation uses `wbinvd` for the hypervisor-level flushing. At the guest level, it uses a simple heuristic to choose the option to use: if the previous owner's working set is smaller than a threshold Thresh, it uses `clflush`, otherwise `wbinvd`. In Section V, we will discuss in more detail how this threshold can be chosen.

V. Implementation

Next, we describe a prototype of vCAT that we have built for our experiments. Our prototype extends the Xen hypervisor (version 4.6) and LITMUS^{RT} 2015.1 guest kernel, running on top of the Intel Xeon CPU E5-2618L v3 processor.

A. Extended data structures and API

We extended the task structure to include a field for specifying the number of partitions a task requests, a `execOnFewer` flag that is set when the task can execute even if it receives fewer (non-zero) partitions than the requested number, and a set of currently allocated partition numbers. By default, a real-time task can only execute if it is allocated partitions, and concurrently running real-time tasks do not share partitions to ensure isolation. We added a system call that allows a task (or the operator) to request a different number of partitions from the guest kernel at run time.

A VCPU's virtual COS register in a VM has the same format (bit mask) and operation as that of a physical COS register, except that it specifies the virtual partitions allocated to the VCPU's currently running task. Like physical partitions, each virtual partition has a shared flag that is set if the partition can be shared among concurrent tasks; this is useful for allocating a shared set of virtual partitions to concurrent tasks (e.g., the standard global EDF scheduling without cache allocation within a VM).

B. Partition allocation and partition context switch

Hypervisor-level allocation: We implemented a command-line utility for the operator to configure the virtual-to-physical mappings P_i and the shared flags of the physical/virtual partitions.[2] For simplicity, we require the operator to configure these data structures when a new VM is created; she can also modify them at run time if desired. To fully utilize the cache, our prototype allows the physical partitions oversubscribed by VMs (and performs a VM partition context switch, if needed).

We also implemented a hypercall that allows a guest to release some unused partitions or request more partitions at run time. In our prototype, the hypervisor simply puts the released partitions in an unused pool and later allocates them to any VM that requests additional partitions. Internally, whenever there is a change in the virtual-to-physical mappings, the hypervisor invokes the mapping procedure, which updates the mapping P_i for each (relevant) VM and the physical COS registers, as well as performs a VM partition context switch and/or flushes the cache, if necessary (c.f. Sections IV-F and IV-G, respectively).

Guest-level allocation: The kernel allocates the VM's available virtual partitions to tasks based on their requests. It reserves a small (configurable) number of partitions to be shared among all best-effort tasks, and uses all the rest for real-time tasks. We implemented two strategies for allocation: the first allocates partitions to tasks in a first-come-first-served basis, as they are scheduled on the VCPUs; and the second gives priority to a more critical task, i.e., allows it to preempt lower-criticality tasks to acquire sufficient partitions, similar to the approach used in [27]. In both cases, if the task's `execOnFewer` flag is set, and if the VM has some but fewer than the requested number, the kernel simply allocates the available partitions to the task (and let it execute) to maximize core utilization and to minimize preemption overhead.

Whenever the kernel of VM i (re-)allocates partitions to a task, it would update the relevant data structures (the task's assigned partition set, the ID $I(v)$ of each allocated virtual partition v the task is assigned), and flush the task's content in the old partition sets if required (c.f. Section IV-G). If necessary, it would also modify COS registers of its VCPU and the VCPUs of the preempted tasks (if any) by executing the `wrmsr` instruction. We extended the hypervisor to trap on this instruction and modify the physical COS registers of these VCPUs' cores (based on the mapping P_i). Notice that, when the physical partitions are oversubscribed, it is possible that VM i might set a virtual COS bit representing a virtual partition that is mapped to a physical partition currently used by another VM j. If the partition is not shared, the hypervisor simply returns failure to VM i by default, thus allocating the oversubscribed partitions in a first-come-first-served basis. However, we also implemented a preemption-based mechanism, where the hypervisor preempts VM j and reassigns the partition to VM i, if VM i has higher priority than VM j, according to some algorithm. Our prototype uses static priority when this choice is configured, but it can easily be extended to include other algorithms for deciding the priority.

[2]Determining the best number of partitions to reserve for each VM is an interesting but orthogonal research question; one promising direction here is to extend the cache-aware compositional analysis in [27].

When a preemption occurs, the hypervisor will perform a VM-level partition context switch (as described in Section IV-F).

C. Flushing heuristics

For simplicity, our prototype always uses the `wbinvd` instruction for the hypervisor-level flushing, since this operation often involves flushing the working sets of several tasks in one or more VMs and thus `clflush` can take a long time. At the guest level, we implemented a simple heuristics that uses `clflush` if the working set size (WSS) of the task is smaller than a threshold Thresh, and uses `wbinvd` otherwise. Intuitively, Thresh is the smallest WSS for which the overhead when using `clflush` is larger than that when using `wbinvd`. (If the strong isolation requirement flag is set, we always use `clflush` at the guest level. Otherwise, we use the heuristic.)

At a high level, the overhead of each approach includes (1) the latency of the cache flush operations, and (2) the extra latency when tasks access the content that was but is no longer in the cache because of flushing. For `clflush`, our empirical evaluation shows that the overhead of cache flush operation is approximately linear to the task's WSS, and the cache reload overhead is linear to the WSS but converges to $\mathsf{D}_{\mathsf{reloadLLC}}$ (the overhead of reloading the entire cache) once the WSS exceeds the cache size. Thus, the estimated overhead is

$$\begin{aligned}\mathsf{Overhead(clflush)} &= \mathsf{D}_{\mathsf{clflush}} + \mathsf{D}_{\mathsf{reloadLLC}} \\ &\approx \mathsf{k}_1 \cdot \mathsf{WSS} + \min\{\mathsf{k}_2 \cdot \mathsf{WSS},\ \mathsf{D}_{\mathsf{loadLLC}}\}.\end{aligned}$$

where $\mathsf{k}_1 \approx 1.58$ (ms/MB), $\mathsf{k}_2 = 1.65$ (ms/MB), and $\mathsf{D}_{\mathsf{loadLLC}} = 26.63$ (ms) on our platform.

For `wbinvd`, the cache flush operation overhead depends on the status of the cache when the instruction is invoked, and our evaluation shows that it is upper bounded by $\mathsf{D}_{\mathsf{wbFflush}} = 0.7$ (ms). Since `wbinvd` flushes the entire cache, and without knowledge of which data need to be reloaded, we assume the worst-case scenario where we need to reload the entire cache; thus, the overhead is at most $\mathsf{D}_{\mathsf{loadLLC}}$. In other words, the overhead when using `wbinvd` is approximately $\mathsf{Overhead(wbinvd)} \approx \mathsf{D}_{\mathsf{wbFlush}} + \mathsf{D}_{\mathsf{loadLLC}}$.

Based on the above analysis, we can derive Thresh as the smallest WSS such that $\mathsf{Overhead(clflush)} > \mathsf{Overhead(wbinvd)}$, i.e.,

$$\mathsf{Thresh} \approx \max\{(\mathsf{D}_{\mathsf{wbFlush}} + \mathsf{D}_{\mathsf{loadLLC}})/(\mathsf{k1}+\mathsf{k2}),\ \mathsf{D}_{\mathsf{wbFlush}}/\mathsf{k1}\}.$$

On our experimental platform, $\mathsf{Thresh} \approx 8.46$ (MB).

D. Overhead introduced by CAT virtualization

We ran a series of micro benchmarks to evaluate the extra overhead introduced by CAT virtualization based on our prototype. The results show that our design introduces only minimal overhead in terms of partition context switch and partition allocations (within a few microseconds), and the overhead caused by flushing and defragmentation in general depends on the tasks' WSS but is always less than 27.35ms on our experimental platform (which has a 20MB shared cache). Due to space constraints, we include the details in the appendix.

VI. Performance Evaluation

To illustrate the applicability and benefits of CAT virtualization, we conducted an extensive set of experiments on our prototype using the PARSEC benchmarks [5] and synthetic workloads. Our goal is to evaluate (i) how well task-level cache isolation using CAT virtualization can protect a task's WCET from other concurrently running tasks, and (ii) how much CAT virtualization can improve the system's real-time performance in two use cases (static and dynamic cache allocations).

A. Experimental setup

Hardware. Our prototype ran on a CAT-capable Intel Xeon CPU E5-2618L v3 processor, which has a 20MB 20-way set-associative L3 shared cache (divided into 20 partitions of 1MB each) and 32GB main memory, and with four cores enabled. Like in most existing real-time research [15], we disabled hyper-threading, SpeedStep, and hardware cache prefetcher features to avoid non-deterministic timing behavior. To minimize interference with the experimental workload, we shut down all non-essential system services during our experiments.

System configuration. We booted the hypervisor with the RTDS scheduler and the VMs with LITMUS^{RT} as the guest kernel, which uses the *PSN-EDF* scheduler. We created two user VMs, $\mathsf{benchVM}$ and $\mathsf{polluteVM}$, which execute the tasks under evaluation and the interfering tasks, respectively. $\mathsf{benchVM}$'s tasks are statically partitioned into two full-capacity VCPUs, each of which is pinned to a dedicated core. Similarly, $\mathsf{polluteVM}$'s are also statically assigned into two full-capacity VCPUs, which are pinned to two remaining cores. To minimize interference to $\mathsf{benchVM}$, we allocated two VCPUs to the high-privilege VM (Domain 0) and directly pinned them to the two cores used by $\mathsf{polluteVM}$. (Further necessary details will be described in the relevant evaluation.)

Workload. We considered two types of workload: the PARSEC benchmark suite [5] and synthetic workload. For the PARSEC benchmarks, we used *simsmall* as the default input for our WCET-related evaluation. For our real-time performance evaluation, for each benchmark, we first explored the influence of different input sets provided by the benchmark suite (i.e., test, simdev, simsmall, simmedium, and simlarge) on the WCET performance, and then selected the one that most influences the WCET performance for using in the schedulability evaluation.

The synthetic workload consists of two types of programs (similar to the ones used in [23]): (1) *cache-bench*, which uses a linked list to sequentially access every 64 bytes (i.e., cache line size) of an 8MB array for 50 times; and (2) *cache-bomb*, which uses the array index to sequentially access every 64 bytes of a 40MB array for 240 times.

To evaluate the relationship between the number of partitions and WCET, we measured the WCET of each workload program across 25 runs when the number of partitions it is allocated varies. As expected, as the number of allocated partitions increases, task's WCET also tends to decrease. (Details are available in the appendix.) This observed relationship provides useful information for selecting the number of partitions for a task to optimize schedulability.

B. Benefits of task-level cache isolation on WCET

Experiment. This experiment aims to evaluate how well task-level cache isolation with CAT virtualization can protect a task's WCET from being affected by concurrent accesses to the cache by other co-running tasks. For this, we executed the task-under-test (a PARSEC benchmark or a *cache-bench* task) alone on one VCPU of benchVM, and we executed a *cache-bomb* task in the second VCPU of benchVM and in each of polluteVM's VCPUs. (Recall that these four VCPUs are pinned to four different cores.) We configured the cache allocation data structures in our prototype to statically allocate 14 exclusive partitions for the task-under-test and 2 exclusive partitions for each of the three *cache-bomb* tasks. We then measured the WCET of the task-under-test across 25 runs, which we refer to as WCET under the PolluteCAT setting.

For comparison, we conducted the same experiment for (i) the Alone setting, where we disabled all three *cache-bomb* tasks; and (ii) the Pollute, where we ran the tasks in vanilla LITMUS^{RT}/Xen, which does not support cache allocation and thus all tasks share the entire cache.

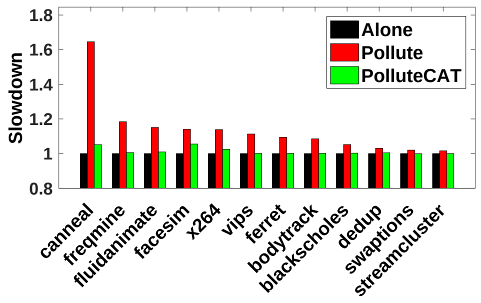


Fig. 3: Measured WCETs of PARSEC benchmarks.

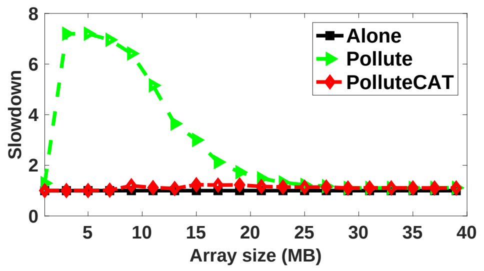


Fig. 4: Measured WCETs of the *cache-bench* workload.

Results for PARSEC benchmarks. Fig. 3 shows the slowdown factor of each PARSEC benchmark task for the three settings, where the slowdown factor for a setting is the ratio of the task's WCET obtained in that setting to that was obtained in the *Alone* setting. The results show that the WCET of the benchmark task can increase substantially (up to 1.65×) in the *Pollute* setting; this is because there is no cache management in this setting and thus other co-running tasks may interfere with the benchmark task by accessing the cache. It is also worth noting that the obtained slowdown factor is with respect to a default input and not the worst-case slowdown. In contrast, the benchmark task has approximately the same or only slightly increased WCET in the PolluteCAT setting as in the Alone setting across most benchmarks. (One reason for the slight increase in WCET could be because we did not isolate the main memory in our experiments and thus, tasks may still interfere with one another due to memory bus or bank contention.) In summary, the results demonstrate that cache isolation with CAT virtualization can effectively avoid the WCET slowdown caused by cache interference.

Results for the synthetic workload. Fig. 4 shows the WCET slowdown of the *cache-bench* task under each setting when we varied the task's array size from 1MB to 40MB. The results further confirm that, without cache management, the shared cache interference can increase the task's WCET by a significant factor, e.g., up to 7.2× (when the array size is between 3MB and 5MB). On the contrary, CAT virtualization can effectively mitigate this problem, as evident by the slowdown factor of close to 1. Notice that when the array size is larger than the cache size (20MB), the task begins to experience cache misses even when it executes alone; as a result, the WCET in the Alone setting begins to increase, leading to a decrease in the slowdown in the Pollute setting.

C. Real-time performance: static cache management

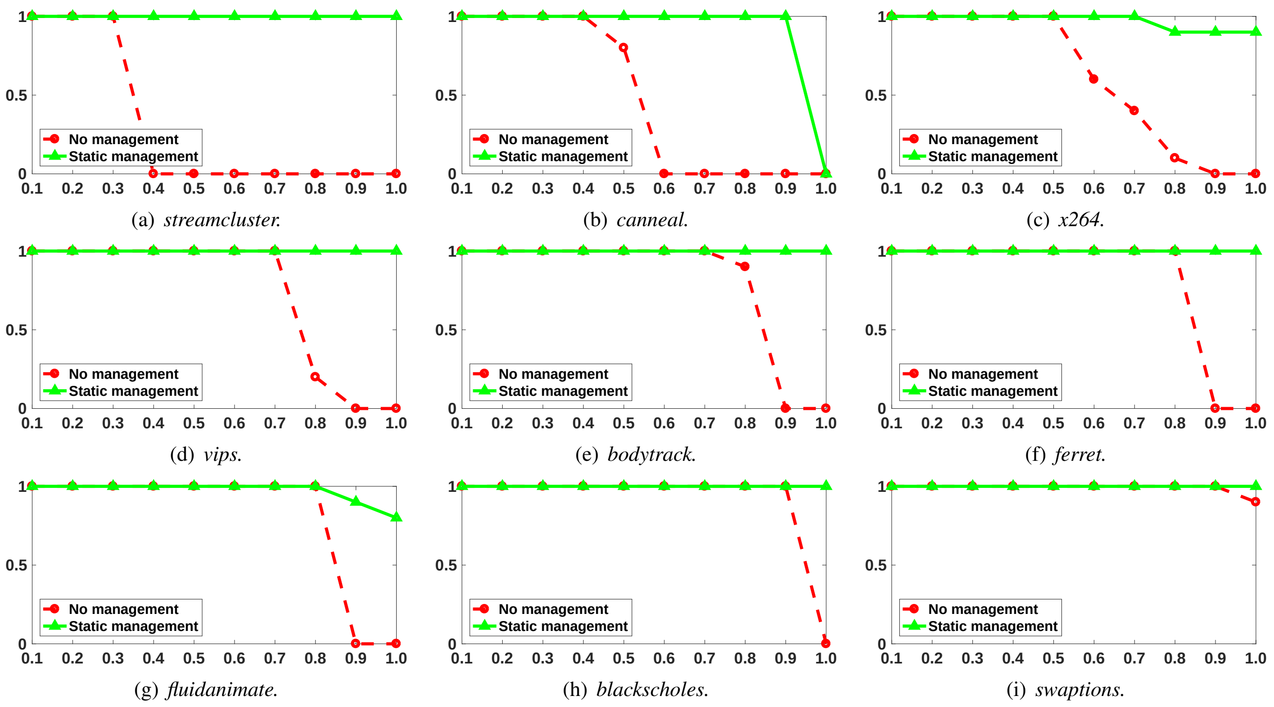
Next, we evaluate how much CAT virtualization can help improve the system's schedulability compared to the cache-agnostic vanilla LITMUS^{RT}/Xen system. To this end, we consider two use cases of CAT virtualization: one for static cache management, and the other for dynamic cache management. We focus on the former in this section.

Allocation configuration. Our experiments used task sets that each consist of two workload types: (1) either the PARSEC benchmark or the *cache-bench* program, and (2) the *cache-bomb* program. We used the same configuration as in the preceding experiment: the benchmark (*cache-bench*) tasks are scheduled on one VCPU (pinned to a dedicated core) with 14 partitions; the *cache-bomb* tasks are statically partitioned into three VCPUs, each of which is allocated 2 partitions. We measured the WCET of each PARSEC benchmark, *cache-bench*, or *cache-bomb* task under this cache allocation.

Task set creation. We first converted the PARSEC benchmarks into LITMUS^{RT}-compatible real-time tasks. While doing so, we found that three benchmarks (*facesim*, *vips* and *freqmine*) contained memory leak bugs; unfortunately, we could not fix the bug in the *freqmine* benchmark and thus could not use it for our schedulability experiments. In addition, the *facesim* benchmark took too long to complete; we omitted it due to time constraints. We conducted the schedulability experiments for all the remaining ten PARSEC benchmarks.

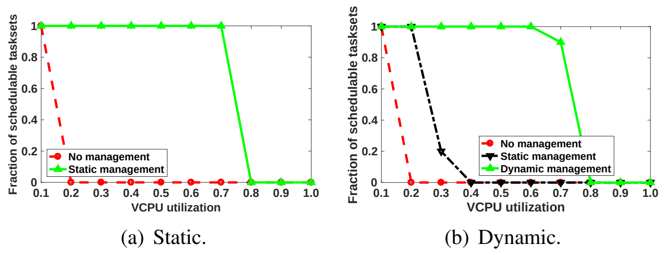
To generate a real-time task τ_i, we first randomly generated a harmonic period p_i, and then computed the task's utilization u_i based on both p_i and its WCET (determined above).

A task set for the benchmark VCPU was created based on a chosen target VCPU utilization U_{vcpu}. Specifically, we randomly generated real-time tasks for the benchmark VCPU until the total utilization of the generated tasks reaches U_{vcpu}. We repeated this generation 10 times to create 10 task sets per U_{vcpu}, where U_{vcpu} ranges from 0.1 to 1.0, with a step of 0.1; this led to a total of 10 × 10 = 100 task sets. For each task set, we executed it for two minutes both on the vanilla LITMUS^{RT}/Xen and on our prototype, and we measured the schedulability of the task set in each setting.



(a) *streamcluster.* (b) *canneal.* (c) *x264.* (d) *vips.* (e) *bodytrack.* (f) *ferret.* (g) *fluidanimate.* (h) *blackscholes.* (i) *swaptions.*

Fig. 5: Schedulability of PARSEC benchmarks. The x-axis shows the VCPU utilization, and the y-axis shows the fraction of schedulable tasksets.



(a) Static. (b) Dynamic.

Fig. 6: Schedulability of synthetic benchmarks.

Benchmark results. Fig. 5 shows the fraction of schedulable task sets of the PARSEC benchmarks when varying the target VCPU utilization.[3] The results across all benchmarks show that our vCAT cache management can substantially improve the system's schedulability. It can also be observed from Fig. 5(a) that, for the *streamcluster* benchmark, on the vanilla LITMUS^{RT}/Xen, the fraction of schedulable task sets begins to decrease quickly once the target VCPU utilization U_{vcpu} is more than 0.3, and all task sets become unschedulable when $U_{vcpu} \geq 0.4$. In contrast, with static cache allocation, all task sets remain schedulable even when each VCPU's utilization is at 1.0. The static management in vCAT can increase system utilization by up to $\frac{1.0}{0.3} = 3.3\times$.

Synthetic results. Fig. 6(a) shows the schedulability results for task sets with the *cache-bench* workload, which further highlights the performance benefits of and the needs for static cache allocation. Without cache management, tasks begin to miss deadlines as soon as $U_{vcpu} > 0.1$, whereas a task is only become unschedulable when $U_{vcpu} > 0.7$ under cache allocation. In other words, the static cache allocation enabled by our CAT virtualization can help increase schedulable utilization by up to 7 times.

[3]Due to space constraint, we omit the results of the *dedup* benchmark, since all task sets are schedulable across all utilizations for both techniques.

D. Real-time performance: dynamic cache management

In the previous use case, cache allocation is performed statically, where each task is always allocated a fixed number and a fixed set of partitions (and thus has a fixed WCET). While this approach is a preferred and more efficient choice in systems with relatively static timing behavior, it may substantially underutilize the cache when the task's timing constraints (such as deadline, period, and cache demand) vary dynamically at run time. In this section, we investigate the performance benefits of our dynamic cache management enabled by CAT virtualization, using a multi-mode system use case.

Dual-mode task sets. We constructed multi-mode tasks based on unimodal *cache-bench* tasks as follows. We first generated a unimodal *cache-bench* task as in the static use case, and then created two dual-mode versions: the *cache-bench-mm1* version uses the same unimodal task parameters for both modes, except that the task period (deadline) in Mode 2 is K times the unimodal period; and the *cache-bench-mm2* version also uses the unimodal parameters for both modes, except that the period in Mode 1 is K times the unimodal period. Intuitively, K captures the degree of dynamism in the task's WCET when varying the number of allocated cache partitions. We set K to be the ratio of the WCET of *cache-bench* when requesting two (the minimum possible) partitions to its WCET when requesting 20 (the maximum possible) partitions; in our experiments, $K = \frac{707}{114} \approx 6.202$. In addition to multi-mode *cache-bench* tasks, we also used the unimodal *cache-bomb* tasks generated as in the static use case.

All *cache-bench-mm1* tasks and all *cache-bench-mm2* tasks are executed on the first VCPUs of benchVM and polluteVM, respectively. We statically partitioned the *cache-bomb* tasks into the second VCPUs of both VMs. We generated 10 task sets for each target VCPU utilization, using the same procedure as in the static use case.

Experiment. We ran each task set for two minutes on our prototype with dynamic cache allocation and measured its schedulability. The cache allocation was configured dynamically as follows. Each VCPU running *cache-bomb* tasks is always allocated two partitions. We configured each multi-mode task to execute in Mode 1 during the first minute, but in Mode 2 during the second minute. The VCPU running *cache-bench-mm1* tasks is allocated 14 partitions in Mode 1 and 2 partitions in Mode 2, whereas the VCPU running *cache-bench-mm2* tasks is allocated 2 partitions in Mode 1 and 14 partitions in Mode 2. This configuration was chosen to balance the VCPU utilization across the two modes.

For comparison, we also ran each task set on Vanilla LITMUS^{RT}/Xen and on our prototype with static allocation, where we statically allocated 8 partitions to each VCPU running multi-mode tasks, and 2 partitions to each VCPU running *cache-bomb* tasks.

Results. Fig. 6(b) shows the fraction of schedulable task sets per VCPU utilization for each of the three settings. As expected, both static and dynamic cache management can help improve the schedulability of the task sets substantially. The results also show that dynamic cache management outperforms static management by a substantial factor in terms of improving schedulable utilization (3×), which is expected since it is much more effective in handling workloads with dynamic timing constraints.

VII. Conclusion

We have presented a novel approach to shared cache management in multicore virtualization systems, through an integration of Intel CAT and cache partition virtualization. Our CAT virtualization design is highly general: it can be configured to provide strong isolation among tasks and/or VMs, to support both real-time tasks—potentially with different criticality levels – and best-effort tasks, and to achieve both static and dynamic cache allocations. We implemented a prototype of the design on top of Xen and LITMUS^{RT}. Experimental results using both PARSEC benchmarks and synthetic workloads show that our prototype introduces only a small overhead while improving both the WCET and the schedulability of the system significantly. The results also show that dynamic allocation is much more effective in improving schedulability than static allocation, especially under dynamic task sets. In future work, we plan to apply our design to several other settings, as well as develop new compositional analysis techniques for cache-aware schedulability test and VM interfaces' computation.

References

[1] Cache monitoring technology and cache allocation technology. https://github.com/01org/intel-cmt-cat. Accessed: 2015-11-19.

[2] MSR-tools. https://01.org/msr-tools. Accessed: 2016-02-01.

[3] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, 2009.

[4] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *OSPERT 2010*, Brussels, Belgium, 2010.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[6] B. B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[7] B. B. Brandenburg and J. H. Anderson. Feather-trace: A light-weight event tracing toolkit, 2007.

[8] M. Caccamo, M. Cesati, R. Pellizzoni, E. Betti, R. Dudko, and R. Mancuso. Real-time cache management framework for multi-core architectures. In *RTAS*, 2013.

[9] A. M. Dani, B. Amrutur, and Y. N. Srikant. Toward a scalable working set size estimation method and its application for chip multiprocessors. *IEEE Transactions on Computers*, 63(6):1567–1579, June 2014.

[10] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven llc allocation. In *USENIX ATC*, 2016.

[11] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li. A simple cache partitioning approach in a virtualized environment. In *ISPA*, 2009.

[12] S. Kato, Y. Ishikawa, and R. R. Rajkumar. CPU scheduling and memory management for interactive real-time applications. *Real-Time Syst.*, 47(5):454–488, Sept. 2011.

[13] D. Kim, H. Kim, N. S. Kim, and J. Huh. vCache: Architectural support for transparent and isolated virtual LLCs in virtualized environments. In *MICRO*, 2015.

[14] H. Kim, A. Kandhalu, and R. R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.

[15] H. Kim and R. R. Rajkumar. Real-time cache management for multi-core virtualization. In *EMSOFT*, 2016.

[16] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS*, 2016.

[17] J. Liedtke, H. Haertig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *RTAS*, 1997.

[18] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[19] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[20] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *ECRTS*, 2007.

[21] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES*, 1995.

[22] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *RTAS*, 2013.

[23] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[24] X. Wang, X. Wen, Y. Li, Z. Wang, Y. Luo, and X. Li. Dynamic cache partitioning based on hot page migration. *Frontiers of Computer Science*, 6(4):363–372, 2012.

[25] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.

[26] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *EMSOFT*, 2014.

[27] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS*, 2016.

[28] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *RTSS*, 2013.

[29] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *PACT*, 2014.

[30] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *RTSS*, 2016.

[31] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *USENIX ATC*, 2011.

[32] H. Zhu and M. Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *ASPLOS*, 2016.

ACKNOWLEDGEMENT

We would like to thank the RTAS reviewers and our shepherd for their constructive feedback and helpful suggestions, and Chao Peng for his clarification of the Xen CAT. This work was supported in part by NSF CNS-1505799, CNS 1563873 and CNS 1329984, ONR N00014-16-1-2195, Global Research Laboratory Program (2013K1A1A2A02078326) through NRF, the DGIST Research and Development Program (CPS Global Center), the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056.

APPENDIX A: OVERHEAD OF CAT VIRTUALIZATION

In this appendix, we present the benchmarking results for our overhead evaluation of the vCAT. We consider five different (but intertwined) types of overhead that our design introduces: cache flush, cache reload, context switch, partition allocations, and defragmentation. We describe each in turn.

A. Cache flush operation latency

Recall that Intel CPUs offer two ways for cache flushing: the `clflush` instruction, which flushes the cache line that contains a specific linear address; and the `wbinvd` instruction, which writes back any modified data in the cache and then invalidates the entire shared cache. We measured the latency for each operation, as follows.

Latency of the `clfush` approach. We created a synthetic task that sequentially accessed (i.e., either read or write) an array. We varied the task's array size from 1MB to 40MB with a step of 1MB. The task first accesses its array, and then the system flushes the task out of the cache by using the `clflush` instruction. We achieved this by enumerating all linear addresses of the task, and invoked the `clflush` instruction on all these addresses. We measured the latency of the cache flush operation when the task either reads from or writes to its array. The result is shown in Fig. 7.

The measured result shows that the latency of the cache flush operation with the clflush approach, denoted as $\mathsf{D}_{\mathsf{clflush}}$, is proportional to the task's working set size (WSS), i.e.,

$$\mathsf{D}_{\mathsf{clflush}} = \mathsf{k}_1 \cdot \mathsf{WSS}$$

where $\mathsf{k}_1 = \mathsf{D}_{\mathsf{clflush}}/\mathsf{WSS} \leq \mathsf{D}_{\mathsf{clflush}}/\mathsf{array\_size\_i} \leq 62.89\mathsf{ms}/40\mathsf{MB} \leq 1.58\mathsf{ms}/\mathsf{MB}$.

Latency of the `wbinvd` approach. We repeated the same experiment as above, but we used wbinvd (instead of clflush) to flush the task. The results show that the latency of the cache flush operation with the wbinvd approach, denoted as $\mathsf{D}_{\mathsf{wbFflush}}$, is not affected by the task's WSS, and $\mathsf{D}_{\mathsf{wbFflush}} \leq 0.7ms$.

B. Cache reload latency

The cache reload latency is determined by the size of the content that was but is no longer in the cache because of flushing. The size of the content to reload is upper bounded by the shared cache size.

We created a synthetic task that uses a *linked list* to access (i.e., read or write) every 64 bytes in an array for three times. The task does the following steps sequentially: (1) access the entire array for two consecutive times; (2) flush the task's content out of the cache; and (3) access the entire array for the third time. We measured the latency of accessing the array at the second time (i.e., when the array is already cached) and at the third time (i.e., when the array is not cached). Then, the time difference between the two measured latencies is the cache reload latency because of flushing. We varied the size of the array from 1MB to 20MB (i.e., the shared cache size) with a step of 1MB, and we measured the cache reload latency under each array size. The result is shown in the Fig. 8.

We also repeated the same experiment but changed the synthetic task to use *array index* to iterate the same array. The result is illustrated in Fig. 9.

We observed that the cache reload latency is proportional to the size of the content to reload. When a task is flushed, the cache reload latency for the task, denoted as $\mathsf{D}_{\mathsf{reloadLLC}}$, is upper bounded by

$$\mathsf{D}_{\mathsf{reloadLLC}} \leq \min\{k_2 \cdot WSS,\ D_{loadLLC}\}$$

Where $k_2 = \mathsf{D}_{\mathsf{reloadLLC}}/\mathsf{WSS} \leq \mathsf{D}_{\mathsf{reloadLLC}}/\mathsf{array\_size\_i} \leq 24.64\mathsf{ms}/15\mathsf{MB} \leq 1.65\mathsf{ms}/\mathsf{MB}$, and $\mathsf{D}_{\mathsf{loadLLC}} = 26.63\mathsf{ms}$ is the maximum latency of reloading the entire LLC.

We also observe, by comparing Fig. 8 and Fig. 9, that the cache reload overhead drops by 89.67% (from 26.63 ms to 2.75 ms) when the task changed the way of accessing its array from *linked list* to *array index*. This is because changing from *linked list* to *array index* for the task eliminates the data dependence in accessing each element of the task's array. Therefore, the task can benefit from the Memory Level Parallelism (MLP) in accessing or reloading its array.

C. Partition context switch overhead

We measured the partition context switch overhead both in the vCAT and in the vanilla LITMUS^{RT}/Xen system. The overhead difference is the extra context switch overhead the vCAT introduces in managing the partition context.

We boot 4 guests, each with 4 full-capacity VCPUs. We randomly generated periodic task sets whose size is 50 or 450 tasks, for each domain. We generated 10 task sets per task set size. Under each environment, we used the feather-trace tool [7] to measure the context switch overhead in a VM (running LITMUS^{RT}), as in earlier LITMUS^{RT}-based studies [15] [4]. We used the Xentrace tool to measure the context switch overhead in the VMM (i.e., Xen), as in earlier RT-Xen study [26]. The result is shown in Table 1.

We observe the extra context switch overhead incurred by our vCAT prototype is very small (upper bounded by $0.25\mu s$).

TABLE 1: Partition context switch overhead (μs).

	Taskset size: 50			Taskset size: 450		
	Vanilla	vCAT	Overhead	Vanilla	vCAT	Overhead
VM	5.49	5.74	0.25	5.02	5.17	0.15
VMM	0.8	0.81	0.01	0.7	0.73	0.03

For completeness, we also measured other types of scheduling-related overhead in VM and VMM. Table 2 shows the task release overhead (REL) and the scheduling overhead (SCH1) within a VM, as well as the scheduling overhead (SCH2) in the VMM. The results show that vCAT incurs negligible extra overhead for all these three types.

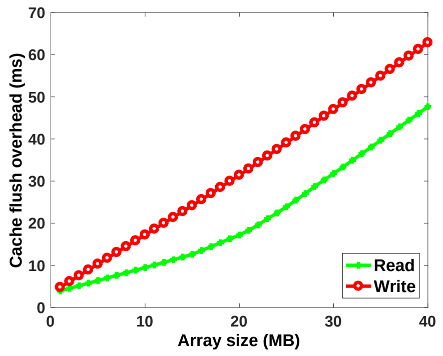


Fig. 7: Cache flush overhead.

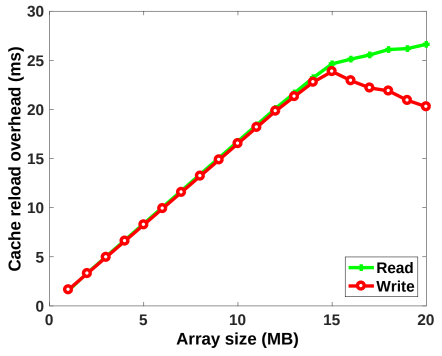


Fig. 8: Cache reload overhead without MLP (Access via linked list).

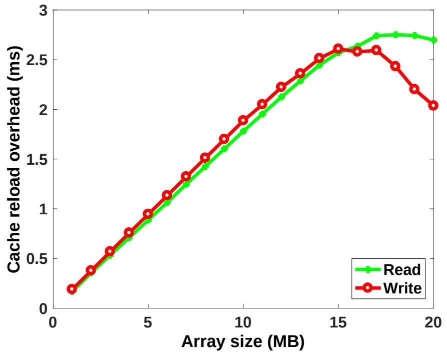
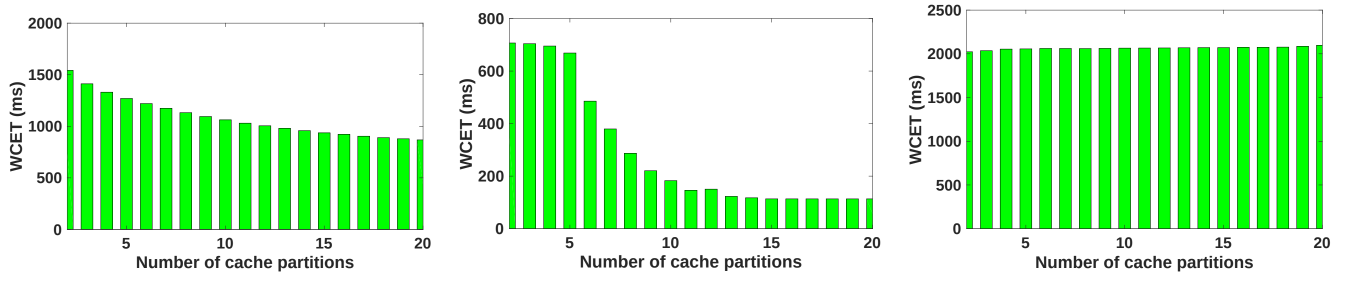


Fig. 9: Cache reload overhead with MLP (Access via array index)



(a) *canneal* benchmark. (b) *cache-bench* program. (c) *cache-bomb* program.

Fig. 10: WCET vs. Number of allocated cache partitions.

TABLE 2: Average scheduling-related overhead (μs).

	Taskset size: 50			Taskset size: 450		
	Vanilla	vCAT	Overhead	Vanilla	vCAT	Overhead
REL	1.77	1.96	0.19	1.13	1.31	0.18
SCH1	2.74	2.80	0.06	3.23	3.33	0.10
SCH2	0.37	0.39	0.02	0.32	0.33	0.01

D. Partition allocation and deallocation overhead

In order to measure the partition allocation and deallocation overhead, we extended the feather-trace tool by adding these two overhead events in LITMUS^{RT}.

We booted one VM with 4 full-capacity VCPUs pinned to 4 cores. We randomly generated periodic task sets whose size is 50 or 450 tasks. We generated 10 task sets per task set size. We measured the average and the maximum latency the vCAT takes to allocate or deallocate cache partitions for tasks. The result is shown in Table. 3.

We observed that the partition allocation and deallocation overheads are negligible. The cache allocation and deallocation overheads are respectively upper bounded by $550ns$ and $318ns$.

TABLE 3: Partition allocation and deallocation overhead (ns).

	Taskset size: 50		Taskset size: 450	
	Average	Maximum	Average	Maximum
Allocation	175	550	178	463
Deallocation	102	318	96	301

E. Defragmentation overhead

When the defragmentation procedure happens, it involves two operations: (1) Reallocating partitions for tasks, which involves deallocating old partitions and then allocating new partitions for tasks; this overhead is upper bounded by the sum of the maximum allocation and deallocation overheads, i.e., $550ns + 318ns = 868ns$. (2) Flushing the entire cache, which has an overhead of at most $\mathsf{D}_{\mathsf{wbFflush}} \leq 0.7ms$.

The defragmentation overhead is the sum of the overhead of reallocating partitions for tasks and the overhead of flushing the entire cache. Therefore, it is upper bounded by $868ns + 0.7ms \leq 0.701ms$.

Appendix B: WCET with respect to the number of allocated partitions

Since the WCET of a task when it executes alone in the system can highly depend on how much cache it is allocated, we first measured the WCET of each workload program (*canneal*, *cache-bench* and *cache-bomb*) across 25 runs when the number of partitions it is allocated varies. The results are shown in Fig. 10.

As expected, as the number of allocated partitions increases, task's WCET also tends to decrease, which is the case for the *canneal* benchmark and the *cache-bench* program. Note, however, that the WCET of the *cache-bomb* program is relatively stable regardless of the number of partitions; this is because its array size is twice the entire cache's size, and thus all accesses to its array elements are cache misses even if it is allocated the entire cache. This observed relationship between WCET and the allocated number of partitions provides useful information for selecting, or dynamically modifying, the number of partitions allocated to each task to optimize the overall system's performance (e.g., schedulability).