Holistic multi-resource allocation for multicore real-time virtualization

Meng Xu
FoundationDB

Robert Gifford
University of Pennsylvania

Linh Thi Xuan Phan
University of Pennsylvania

Abstract

This paper presents vC^{2}M, a holistic multi-resource allocation framework for real-time multicore virtualization. vC^{2}M integrates shared cache allocation with memory bandwidth regulation to mitigate interferences among concurrent tasks, thus providing better timing isolation among tasks and VMs. It reduces the abstraction overhead through task and VCPU release synchronization and through VCPU execution regulation, and it further introduces novel resource allocation algorithms that consider CPU, cache, and memory bandwidth altogether to optimize resources. Evaluations on our prototype show that vC^{2}M can be implemented with minimal overhead, and that it substantially improves schedulability over existing solutions.

1 Introduction

Arguably the de-facto standard for cloud computing, virtualization has recently gained substantial traction in real-time embedded systems, due to the rapid increase in complexity and the prevalence of powerful multicore processors [12]. In the automotive domain, for instance, virtualization provides a natural means to consolidate features onto fewer processors, thus enabling a seamless integration of closely related functionalities while reducing size, weight, and power. However, virtualization also introduces a new challenge: how to provide *timing guarantees* for tasks and virtual machines (VMs).

Compositional analysis provides a step towards addressing this challenge [9, 13]. In this approach, tasks within a VM are scheduled by the OS on one or more virtual processors (VCPUs), which in turn are scheduled as conventional periodic tasks on the physical cores by the hypervisor. To achieve timing guarantees, compositional analysis first abstracts each VM into an *interface*, which describes the minimum resources needed to ensure that all tasks within the VM are schedulable. The VMs' interfaces are then transformed into a set of (abstract) explicit-deadline periodic tasks whose periods and worst-case execution times (WCETs) represent the periods and budgets of the VCPUs, such that if the VCPUs are schedulable then all VMs (and thus the entire system) will also be schedulable.

Despite substantial results on compositional analysis, prior work has two important limitations: First, it ignores the potential *interferences* among tasks on different cores that concurrently access shared resources, such as the shared cache and memory bus. Such interferences introduce extra latency to tasks' execution times, which is not considered by existing analysis; hence, tasks may be unschedulable even if the analysis proves otherwise. Second, existing work suffers from high *abstraction overhead* – that is, the bandwidth of a VCPU (defined as the ratio of its CPU budget to its period) can be substantially larger than the total utilization of the tasks scheduled on it. For example, consider a taskset with a single task $(10,1)$ with period/deadline of 10ms, WCET of 1ms, and utilization of $1/10 = 0.1$, the minimum VCPU's budget computed by existing work [13] is 5.5, which is 55$\times$ the utilization of the taskset itself!

In this paper, we present vC^{2}M, a holistic solution towards timing guarantees in multicore virtualization systems. vC^{2}M addresses the above challenges using two key insights: First, it integrates cache partitioning and memory bandwidth regulation in the hypervisor to provide better isolation among concurrent tasks. By dividing the shared cache into multiple non-overlapped partitions, and by assigning disjoint sets of partitions to different cores, we eliminate the cache interferences among concurrent tasks since they can no longer access the same cache set. Further, by regulating the amount of memory bandwidth that tasks from each core can access, each task is guaranteed to receive the allocated memory bandwidth (BW).

Second, by mapping each task to a separate VCPU and by synchronizing their releases, vC^{2}M completely removes the abstraction overhead. This approach works in most practical systems, where the number of tasks in each VM does not exceed the supported maximum number of VCPUs per VM. (For example, the Micrium II RTOS supports 256 tasks, whereas a Xen-based VM can have up to 512 VCPUs.) However, in rare cases where this condition does not hold, we additionally present a solution for harmonic tasks that removes the abstraction overhead by regulating the VCPU execution, which can be supported by existing hypervisors (e.g., Xen).

Based on these capabilities, vC^{2}M further provides an efficient resource allocation algorithm that—given a set of VMs with real-time tasks on a multicore platform—computes a set of VCPUs for each VM, an assignment of tasks to VCPUs and an assignment of VCPUs to cores, and the amounts of cache and BW resources for each core, so as to guarantee schedulability and maximize utilization.

In summary, we make the following contributions:

- vC^{2}M, a framework for CPU, cache and memory bandwidth resource allocation for multicore virtualization systems;
- a concrete design of vC^{2}M and a low-overhead prototype implementation on top of Xen;
- novel approaches to removing the abstraction overhead; and
- efficient and effective multi-resource allocation algorithms.

Evaluations using PARSEC benchmarks on our prototype show that vC^{2}M can be implemented with minimal overhead, and that it can effectively reduce the interferences among tasks and VMs. Further, vC^{2}M substantially outperforms existing solutions in terms of schedulability performance, while being highly efficient.

2 Related work

Compositional analysis provides a natural way for computing the VCPU parameters and for analyzing schedulability in a virtualization

This research was supported in part by ONR N00014-16-1-2195, NSF CNS 1703936, CNS 1563873 and CNS 1750158, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317840

system [9–11, 13]. However, prior work focuses primarily on CPU resource and ignores the interferences via the shared cache and memory bandwidth resources. vC^{2}M provides a first step towards bridging this gap by considering multiple resource types.

The abstraction overhead in compositional analysis is a known issue and has been addressed in existing work, e.g., using resource-efficient interface representations [4] or bandwidth-optimal interface composition methods [3]. Our work offers an entirely different approach catered to virtualization environments that is both simple and effective; by directly mapping each task to a VCPU and synchronizing their releases, or by regulating the VCPU execution, we can completely remove the abstraction overhead at the VM level.

Memory bandwidth regulation has been extensively studied in the *non-virtualization* setting (see, e.g., [6, 18, 19]). Our bandwidth regulation is similar in concept to MemGuard [18] (which cannot be directly applied to the virtualization setting) but differs in several aspects: First, it is a built-in feature of the hypervisor instead of a loadable module (which is not always supported by a hypervisor such as Xen). Second, it works directly with the low-level hardware, which is more efficient than relying on Linux's `perf` tool. Finally, our approach keeps the cores idle when they exceed their bandwidth allocation (rather than keeping them busy as MemGuard does), which is more energy efficient. Note also that, unlike ours, none of the existing work in this line considers the shared cache.

Prior work has also developed cache management techniques for virtualization systems to improve real-time performance (see e.g., [5, 7, 8, 16]). vC^{2}M leverages the existing vCAT system [16] for the shared cache allocation, but it also integrates cache management with BW regulation, which these techniques do not support.

3 Approach and design

System architecture. The system consists of multiple VMs running on a multicore platform with a shared cache and a shared memory bus. Tasks within each VM are scheduled on a set of VCPUs by the VM's OS, and the VCPUs are scheduled on the cores by the hypervisor. We assume both the hypervisor and OS use the partitioned Earliest Deadline First (EDF) scheduling algorithm, which are supported by open-source hypervisors (e.g., Xen) and real-time OS (e.g., LITMUS^{RT} [2]). Our goal is to develop resource allocation techniques to provide schedulability for the system.

3.1 Approach

Our approach is based on three key insights. First, vC^{2}M mitigates the potential interferences among concurrently running tasks by implementing shared cache partitioning and BW regulation in the virtualization setting. Specifically, it divides the shared cache into multiple equal partitions and allocates a disjoint subset of partitions to each core to ensure shared cache isolation. vC^{2}M incorporates this with a new BW regulation mechanism that enforces a given bandwidth budget to a core to provide bandwidth isolation among cores; this is done by ensuring that the number of memory requests from a core at run-time does not exceed the core's allocated budget.

Second, vC^{2}M extends existing compositional analysis to consider not only CPU but also cache and BW resources. For this, it introduces two novel solutions to removing the abstraction overhead in the analysis: The first directly maps each task to a separate VCPU and synchronizes their releases, thus effectively flattening the scheduling hierarchy. This strategy assumes that the number of tasks per VM is no more than the supported maximum number of VCPUs per VM, which applies to most practical systems. The second removes the abstraction overhead by regulating the VCPU execution, such that its execution pattern repeats in each period; this strategy relaxes the above assumption and thus applies to all general systems.

Finally, vC^{2}M provides heuristic resource allocation algorithms that leverage the overhead-free analysis and consider the interdepence between a task's WCET and its allocated cache and BW, so as to effectively assign tasks to VCPUs, VCPUs to cores, and resources to cores. We next discuss a concrete system design for vC^{2}M.[1]

3.2 Design

Cache and memory bandwidth allocation. Our shared cache allocation approach can be done by simply leveraging vCAT [16], an existing cache allocation system based on Intel's Cache Allocation Technology. For the BW regulation, we rely on hardware performance counters (PCs) to monitor the number of last-level cache misses, which can be treated as the number of memory requests [18], from each core in each regulation period (a small configurable interval, e.g., 1ms). Whenever a core exceeds a configured number of memory requests (i.e., its bandwidth budget), we throttle the core by notifying the hypervisor to *leave the core idle* for the rest of the regulation period. When a new period begins, we un-throttle the core by triggering the hypervisor to execute a VCPU on the core. With this mechanism, each core is guaranteed to receive its configured budget in each period, and it never is allowed to use more BW than is allocated. Fig. 1 shows a high-level architecture of our BW regulator. It has two core components:

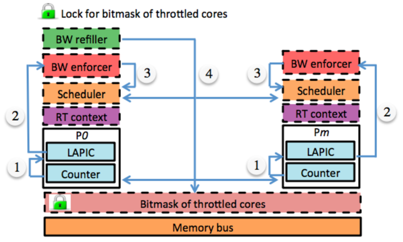


Figure 1: Memory bandwidth regulation in vC^{2}M**.**

Setup. This component sets up the system upon initialization, including (i) configuring an unused PC on each core to monitor the number of memory requests from the core, and preseting the PC value so that it will overflow when the core runs out of its bandwidth budget; (ii) configuring each core's Local Advanced Programmable Interrupt Controller (LAPIC) to deliver the PC overflow interrupt to the core when its PC overflows; (iii) creating a periodic timer to periodically replenish each core's bandwidth budget; and (iv) clearing the overflow status register that indicates which PCs overflow.

Regulation. Once the regulator has been initialized and enabled, the PCs will begin counting the number of memory requests from the cores. When a core's PC overflows, the LAPIC delivers the PC overflow interrupt to the *BW enforcer* handler running on the core (Steps ① and ② in Fig. 1). Upon receiving the interrupt, the *BW enforcer* handler invokes the hypervisor's scheduler to de-schedule its currently running VCPU (Step ③). (The hypervisor's scheduler has been modified to be aware of the throttled cores, to avoid scheduling a VCPU on them.) In addition, the bandwidth replenishment handler

[1]Our design targets x86 platforms; however, our approach is general and can be implemented on other platforms, e.g., on ARM hardware, where similar features exist.

(*BW refiller*) periodically replenishes the budget for each core and invokes the scheduler on each throttled core to schedule a VCPU onto the core at the beginning of each regulation period (Step ④).

Synchronization of task and VCPU releases. Using compositional analysis, the CPU budget and period of a VCPU can be computed based on the worst-case demands of its tasks. When the releases of a task and of its VCPU are not synchronized, the task has a larger worst-case demand as it needs to wait longer to be scheduled. For instance, if the task is released when its VCPU has just completed, it has to wait until the VCPU's budget is replenished and the VCPU is scheduled again. Consequently, the VCPU's bandwidth can be substantially larger than the total utilization of its tasks.

vC^{2}M eliminates the above overhead by providing a mechanism for synchronizing the releases of a task and its VCPU, and by mapping each task to a separate VCPU. Our insight is that, if a task is scheduled alone on a VCPU and its release time is synchronized with its VCPU's, then by assigning the budget and period of the VCPU to be that of the task, respectively, the task is guaranteed to be schedulable if the VCPU is schedulable.

Our release synchronization works by adjusting the VCPU's release time to be in sync with the task's. As the VM's and the hypervisor's clock may not be synchronized, instead of simply passing the task's release time to the hypervisor's scheduler directly, we use the offset between the task's initialization and its first release for our synchronization. When a task is initialized at vt_0 in VM time, (t_0 in *wall time*), the VM's scheduler calculates the delay to the task's first release, $L = vt_2 - vt_0$, and then issues a customized hypercall to pass L to the hypervisor. When the hypercall is executed at xt_0 in hypervisor time (t'_0 in *wall time*), the hypervisor's scheduler computes a new release time for the VCPU, $xt_2 = xt_0 + L$, and modifies the VCPU's first release time from xt_1 to xt_2. (Due to the hypercall delay, which is $t'_0 - t_0$, the VCPU will be released slightly after the task; as this delay is negligible, we ignore it in the analysis.)

Regulation of VCPU execution. The above solution assumes each VM can have at least as many VCPUs as it does tasks. When this assumption does not hold (e.g., systems with very large-size tasksets), vC^{2}M can remove the abstraction overhead for harmonic tasksets by regulating the resource supply patterns of the VCPUs. (Details and proofs are available in [15].) Intuitively, by making the execution pattern of each VCPU in each period the same, we can schedule its tasks with a VCPU bandwidth identical to the tasks' total utilization.

vC^{2}M uses a combination of mechanisms to achieve this regulated execution pattern: (i) using periodic servers to manage the VCPUs' budgets; (ii)) assigning to the VCPUs harmonic periods and the same release offset; and (iii) using a deterministic tie-breaking rule (for EDF) when scheduling VCPUs with the same absolute deadline: first by the VCPU period where smaller period has higher priority, and then by the VCPU index where smaller index has higher priority.

3.3 Implementation and empirical evaluation

Prototype. We implemented a prototype of our design to be used for our experiments. Our prototype extends the Xen hypervisor (version 4.8.0) with vCAT features built in and LITMUS^{RT} 2015.1 guest OS, which run on an Intel Xeon CPU E5-2618L v3 processor. The Xen hypervisor has a real-time scheduler called RTDS, which we modified to incorporate our design. We leveraged the various hardware components (PC counters, LAPIC controllers, overflow status register, and overflow control register) for the memory request monitoring and notification, and implemented two interrupt handlers for the *BW enforcer handler* and *BW refiller handler*.

To synchronize task and VCPU releases, we implemented a customized system call in LITMUS^{RT} to compute the release delay in the kernel space, and a customized hypercall to pass the release delay L and the VCPU index to Xen's scheduler. The hypercall invokes the modified RTDS scheduler to update the VCPU's next release time.

Run-time overhead. We ran a series of experiments on our prototype to measure its run-time overhead, using the approach in [14].

Table 1: Memory bandwidth regulator's overhead (μs).

Throttle			Memory BW budget replenishment		
min	avg	max	min	avg	max
0.33	0.37	1.15	8.81	52.22	108.65

Table 2: Scheduler's overhead (μs).

	24 VCPUs			96 VCPUs		
	min	avg	max	min	avg	max
CPU budget replenish.	0.29	0.74	2.95	0.34	1.26	3.73
Scheduling	0.13	0.57	1.73	0.13	0.55	2.03
Context switching	0.04	0.23	32.07	0.04	0.27	24.67

Tables 1 and 2 show the overhead for BW regulation and the scheduling overhead of the extended RTDS scheduler. We observe that BW regulation in vC^{2}M introduces only a very small overhead; in addition, the maximum scheduling-related overhead is minimal, and it increases slowly as the number of VCPUs increases. This illustrates that our design is practical for real systems.

Impact of resource isolation on WCET. To explore the impact of cache and BW isolation that is enabled by vC^{2}M, we further performed a series of experiments using PARSEC benchmarks [1] on our prototype, and measured their WCETs with and without cache and BW isolation. The results show that vC^{2}M can effectively mitigate the interferences caused by concurrent accesses to the shared cache and memory bus, and it can reduce a task's WCET as a result. Our results also show that a task's WCET depends on the allocated cache and BW, but the exact relationship varies across benchmarks.

4 Analysis and resource allocation

The vC^{2}M system presented so far provides mechanisms for shared resource allocation in virtualization systems. We now present an analysis and resource allocation algorithms on top of vC^{2}M to ensure timing guarantees for such systems. We first introduce a new cache- and memory bandwidth-aware model for tasks and VCPUs.

4.1 Theoretical modeling

The system consists of multiple VMs running on a multicore platform with M identical cores, a shared cache with C equal-size cache partitions, and a shared memory bus whose bandwidth is divided into B equal-size partitions (as done in [18]). Each BW partition is a unit of BW budget allocation (which is set equal to a configurable number of bytes/second). To accommodate hardware constraints, we denote by $C_{\min}$ and $B_{\min}$ the minimum numbers of cache and BW partitions that a core can be allocated, respectively.

Each VM executes a set of independent implicit-deadline periodic tasks $\tau_i = \langle p_i, \{e_i(c,b) \mid C_{\min} \leq c \leq C \wedge B_{\min} \leq b \leq B\}\rangle$, where p_i is the period, and $e_i(c,b)$ is the WCET when the task executes alone on a core that is allocated c cache partitions and b BW partitions. (The WCET values can be obtained, e.g., by measurement on vC^{2}M). We call $e_i^* = e_i(C,B)$ the reference WCET of τ_i, and $s_i = [e_i(c,b)/e_i^*]$ ($C_{\min} \le c \le C; B_{\min} \le b \le B$) the slowdown vector of τ_i. Intuitively, s_i captures how sensitive τ_i is to cache and BW resources.

Tasks within a VM are partitioned and scheduled on a set of VCPUs, which further are partitioned and scheduled on the cores, both under EDF. Each VCPU j is modeled by $V_j = \langle \Pi_j, \{\Theta_j(c,b) \mid C_{\min} \le c \le C \wedge B_{\min} \le b \le B\}\rangle$, where Π_j is the VCPU's period and $\Theta_j(c,b)$ is the VCPU's execution time budget when it is given c cache partitions and b BW partitions. As for tasks, we call $\Theta_j^* = \Theta_j(C,B)$ the reference budget, and $S_j = [\Theta_j(c,b)/\Theta_j^*]$ ($C_{\min} \le c \le C; B_{\min} \le b \le B$) the slowdown vector of V_j. The CPU-bandwidth of V_j is defined as Θ_j/Π_j.

Objective. Our goal is to compute (1) the tasks-to-VCPUs mapping and the corresponding VCPUs' parameters, and (2) the VCPUs-to-cores mapping and the numbers of cache and BW partitions for each core, such that all tasks meet their deadlines. Our solution integrates a VM-level allocation method for solving (1) and a hypervisor-level allocation method for solving (2), which we discuss next.

Remarks. With shared cache and BW isolation provided by vC^{2}M, tasks running concurrently on different cores can no longer interfere with one another through cache line conflicts and memory bus contention. (Detailed evaluations of the isolation enabled by vC^{2}M are available in [15].) However, tasks and VCPUs running on the *same* core may experience cache-related overhead due to task preemption, VCPU preemption and completion events. Such intra-core overhead can be accounted for by inflating the WCETs of tasks with the overhead caused by task preemptions, and by inflating the (computed) execution budgets of VCPUs with the overhead caused by VCPU preemptions and completions, using the same technique as in [17]. The task and VCPU inflations are performed before applying our VM-level and hypervisor-level allocation methods, respectively. Note that, due to space constraints, we omit all proofs and pseudo-code.

4.2 VM-level resource allocation

Flattening. To remove the abstraction overhead, our strategy always maps a task to a dedicated VCPU and synchronizes their releases, thus effectively flattening the scheduling hierarchy. This strategy provides not only a tasks-to-VCPUs mapping but also a straightforward computation of the VCPUs' parameters.

THEOREM 1. *Any task $\tau_i = \langle p_i, \{e_i(c,b) \mid C_{\min} \le c \le C \wedge B_{\min} \le b \le B\}\rangle$ is schedulable on a VCPU V_i with period $\Pi_i = p_i$ and budget $\Theta_i(c,b) = e_i(c,b)$, for all $C_{\min} \le c \le C$ and $B_{\min} \le b \le B$, if τ_i is the only task executing on V_i and their releases are synchronized.*

Since V_i is released whenever τ_i is released and τ_i executes alone on V_i, τ_i is executed iff the VCPU is executed. Hence, the theorem.

This direct mapping method assumes that each VM can have at least many VCPUs as it does tasks. We next discuss a general strategy for systems in which this condition may not hold.

Overhead-free analysis for the general case. To remove the abstraction overhead, our strategy ensures all VCPUs are well-regulated, i.e., their execution patterns repeat in each of their periods. Formally, a well-regulated VCPU V_j is executed at time t iff it is executed at time $t + k \cdot \Pi_j$, where Π_j is V_j's period and $k \in \mathbb{N}$. As a well-regulated VCPU provides CPU time to its tasks in a regular fashion, it reduces tasks' waiting time and makes them easier to schedule. Further, if tasks are harmonic (which is common in embedded systems) – i.e., for every two tasks τ_i and τ_j, either p_i divides p_j or p_j divides p_i – then we can feasibly schedule a taskset on a well-regulated VCPU with CPU-bandwidth equal to the taskset's utilization. Theorem 2 establishes this condition; its proof can be found in [15].

THEOREM 2. *A harmonic taskset $\tau = \{\tau_1, \ldots, \tau_n\}$, where $\tau_i = \langle p_i, \{e_i(c,b) \mid C_{\min} \le c \le C \wedge B_{\min} \le b \le B\}\rangle$ for all $1 \le i \le n$, is guaranteed to be schedulable under EDF on a well-regulated VCPU with period $\Pi_j = \min_{1 \le i \le n} p_i$ and execution budget $\Theta_j(c,b) = \Pi_j \cdot \left(\sum_{i=1}^{n} \frac{e_i(c,b)}{p_i}\right)$, for all $C_{\min} \le c \le C$ and $B_{\min} \le b \le B$.*

Based on the above overhead-free analysis, our allocation algorithm maps tasks in a VM onto m VCPUs, where m is the minimum of the number of tasks and the number of cores, using a clustering-based heuristics. At the high-level, the algorithm works as follows: To fully utilize the cache and BW allocated to a VCPU, it first groups tasks with similar slowdown vectors (i.e., similar sensitivity to cache and BW resources) into the same cluster using the KMeans algorithm. It then packs tasks in each cluster onto a VCPU, in decreasing order of task's reference utilization (i.e., ratio of reference WCET to period), such that the total reference utilization of tasks on each VCPU is similar to each other; this is to balance the loads across all VCPUs. Finally, for each VCPU, it computes the VCPU's parameters as defined in Theorem 2.

4.3 Hypervisor-level resource allocation

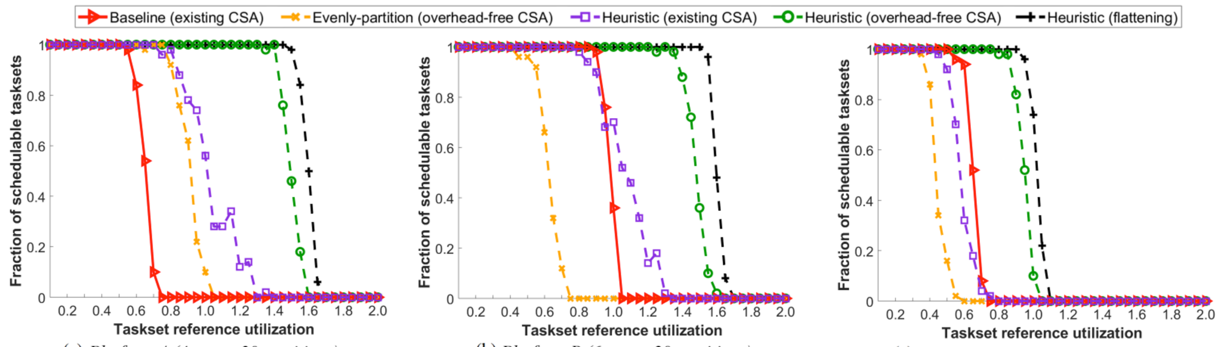
Next, we discuss a heuristics for mapping the computed VCPUs to cores and for allocating cache and BW partitions to each core, so that all VCPUs are schedulable. Each VCPU's budget is first inflated to account for the intra-core overhead caused by VCPU preemption and completion events, using the technique in [17].

Our algorithm computes the mapping and resource allocation assuming a fixed number of m cores. We begin with $m = 1$ and increase m until either the algorithm reports success or $m > M$ (in which case, the system is not schedulable). At the high level, the algorithm works as follows: To enable the VCPU to effectively utilize the cache and BW allocated to its core, we first group VCPUs with similar slowdown vectors into the same cluster. We then repeat the following three phases, until either the system is schedulable or a user-defined maximum number of iterations has been reached.

Phase 1 (Packing): We randomly pick a permutation of the VCPU clusters and, following the permutation order, pack VCPUs in each cluster to cores in decreasing order of VCPU's reference utilization, such that all cores have similar total reference utilizations.

Phase 2 (Resource allocation): We assign cache and BW resources to cores, with $C_{\min}$ cache partitions and $B_{\min}$ bandwidth partitions per core initially. We then check for the schedulability of each core (i.e., whether its total utilization under the allocated cache and BW partitions is at most 1). If some cores are unschedulable, we incrementally add additional cache and BW partitions to each unschedulable core, until it is schedulable or there is no impact on its utilization; here, we prioritize the unschedulable core that has the highest reduction in utilization if being given the additional partitions. If the system is schedulable, the algorithm terminates and outputs the current mapping, along with the numbers of cache and BW partitions for each core; otherwise, it moves to the next phase.

Phase 3 (Load balancing): We balance the utilizations across cores by migrating VCPUs from unschedulable cores to schedulable cores. For each unschedulable core, we keep moving a VCPU to



(a) *Platform A (4 cores, 20 partitions)* (b) *Platform B (6 cores, 20 partitions)* (c) *Platform C (4 cores, 12 partitions)*

Figure 2: Performance for different platforms (uniform taskset utilization distribution).

the schedulable core that will have the smallest utilization after the migration, until the unschedulable core becomes schedulable.

Since the cores that are previously schedulable may become unschedulable after this balancing step, we re-allocate resources to cores and check whether the system becomes schedulable (Phase 2). We keep performing the load balancing and resource allocation steps until the system becomes schedulable or there is no benefit in balancing. In the latter case, the algorithm repeats from Phase 1, where it performs packing using a new order of the VCPU clusters.

5 Evaluation

To evaluate the performance of vC^{2}M, we conducted a series of experiments using real-time workloads that were randomly generated based on PARSEC benchmarks. We considered five solutions:

- *Heuristic (flattening)*: vC^{2}M with the flattening strategy.
- *Heuristic (overhead-free CSA)*: vC^{2}M with the overhead-free analysis based on well-regulated VCPUs.
- *Heuristic (existing CSA)*: a variance of vC^{2}M that uses our heuristic allocation algorithms but relies on the existing compositional analysis [13] for computing the VCPU parameters.
- *Evenly-partition (overhead-free CSA)*: another variance of vC^{2}M that relies on our overhead-free analysis, but it evenly distributes cache and BW among cores, and it uses the best-fit bin-packing technique to pack tasks to VCPUs and VCPUs to cores.
- *Baseline (existing CSA)*: a baseline solution that uses the existing compositional analysis [13] for computing VCPU parameters, assuming that tasks have the worst-case BW and no cache allocated, and it uses best-fit bin-packing for packing tasks and VCPUs.

5.1 Experimental setup

Workload. Each taskset contained a number of implicit-deadline periodic tasks, whose periods were harmonic and uniformly distributed in [100, 1100] (as in [9]). The tasks' utilizations followed one of four distributions: a uniform distribution within the range $[0.1, 0.4]$ and three bimodal distributions, where the utilizations were distributed uniformly over either $[0.1, 0.4]$ or $[0.5, 0.9]$, with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). The maximum WCET $e_i^{\max}$ of a task τ_i is the product of its utilization and its period. The tasks' WCET values were generated based on the PARSEC benchmarks, as follows:

For each benchmark k, we used our prototype and experimental machine (c.f. Section 3.3) to profile its execution time (with simlarge input) when it executes on a dedicated VCPU on a dedicated core, under each allocation (c, b) of cache and BW, where $c = 2, \ldots, 20$ and $b = 1, \ldots, 20$. We obtained k's slowdown vector s_k from the measured execution time across 25 runs. We next estimated the benchmark's maximum WCET $e_k^{\max}$ as the execution time under the worst-case bandwidth and with the cache disabled, and then calculated the benchmark's (maximum) slowdown factor $s_k^{\max}$ as the ratio of its maximum WCET to its measured reference WCET.

To generate the WCET values $e_i(c, b)$ for a task τ_i under each different configuration (c, b) of cache and BW allocation, we randomly picked a PARSEC benchmark k to be used as the task's workload. We first computed the task's reference WCET, e_i^*, as the ratio of its maximum WCET $e_i^{\max}$ to the benchmark's slowdown factor $s_k^{\max}$. We then scaled the task's reference WCET by the slowdown vector of the benchmark to obtain its WCET values, i.e., $e_i(c, b) = e_i^* \cdot s_k(c, b)$; this is to preserve the sensitivity of the benchmark to cache and BW resources. We kept generating a new task for the taskset in this manner until the total reference utilization of the tasks (i.e., the sum of e_i^*/p_i for all τ_i) reached the target reference utilization of the taskset.

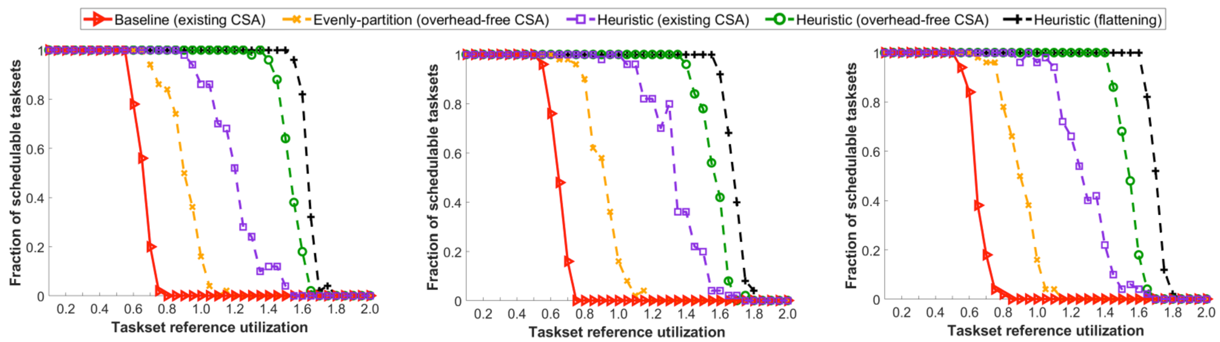
Platforms. We considered three platform configurations (based on the Intel Xeon 2618v3, Intel Xeon D-1528, and Intel Xeon D-1518 processors, respectively): Platform A has 4 cores and 20 cache partitions; Platform B has 6 cores and 20 cache partitions; and Platform C has 4 cores and 12 cache partitions. The maximum number of BW partitions is the same as the maximum number of cache partitions (i.e., $C = B$) on each platform. Our analyses were performed on an Intel Xeon E5-2683 v4 processor, which has 32 cores (with hyper threading enabled) operating at 2.10GHz.

5.2 Schedulability performance

We generated tasksets with reference utilization ranging from 0.1 to 2, with a step of 0.05. For each taskset reference utilization, we generated 50 independent tasksets (i.e., 1950 tasksets in total), with tasks' utilizations uniformly distributed in $[0.1, 0.4]$. We analyzed the tasksets for Platform A using each of the five solutions described above. Fig. 2(a) shows the results.

The results show that the fractions of schedulable tasksets under vC^{2}M– i.e., *Heuristic (flattening)* and *Heuristic (overhead-free CSA)* – are substantially higher than that of all other three solutions. The tasksets' reference utilization after which tasksets start to become unschedulable is 0.5 under the baseline solution, while it is at least 1.3 under ours. Thus, by removing the abstraction overhead and by effectively allocating resources, vC^{2}M increased the system's workload by $1.3/0.5 = 2.6\times$ without sacrificing schedulability.

Further, the overhead-free analysis on well-regulated VCPUs performs very closely to the flattening approach: only 100 out of 1950 generated tasksets (5%) are schedulable under the latter but not



(a) *Bimodal light* (b) *Bimodal medium* (c) *Bimodal heavy*

Figure 3: Performance for different taskset utilization distributions on Platform A.

under the former. Thus, the general solution is an effective alternative for systems where direct task-to-VCPU mapping is not possible.

The results of the three methods *Heuristic (existing CSA)*, *Evenly-partition (overhead-free CSA)*, and *Heuristic (overhead-free CSA)* also show that when using only the heuristic resource allocation algorithms or only the overhead-free analysis, the fraction of schedulable tasksets is substantially smaller than when combining them. This illustrates that a combination of removing the abstraction overhead *and* effectively allocating resources is critical to schedulability.

5.3 Impact of platforms and task parameters

We investigated the impact of the platform configurations and the tasks' parameters on schedulability for all five solutions. For this, we repeated the above experiment on the remaining two platforms (Platforms B and C), as well as using tasksets with the bimodal-light, bimodal-medium and bimodal-heavy utilization distributions.

The results for Platforms B and C (using tasksets with uniform utilization distribution) are shown in Figs. 2(b) and 2(c), respectively. We observe that vC^{2}M continues to substantially outperform the existing solutions as before. We also observe that the more powerful (e.g., with more cores) the platform is, the more performance benefit vC^{2}M provides over solutions based on existing techniques.

Fig. 3 shows the results for tasksets with the three bimodal distributions on Platform A. Again, we observe that the superior performance of vC^{2}M is consistent across all taskset parameters.

5.4 Running time complexity

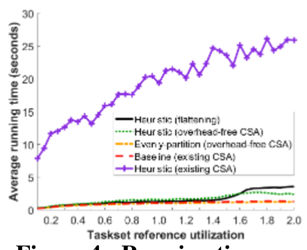


Figure 4: Running time.

Fig. 4 shows the measured running time of the five solutions in the evaluation in Section 5.2. We observe that our overhead-free analysis and heuristic allocation algorithms are highly efficient; e.g., the average running time of *Heuristic (overhead-free CSA)* is always less than 3 seconds. In contrast, the running time for *Heuristic (existing CSA)* is much higher (e.g., up to 25 seconds), and it increases quickly as the taskset utilization increases. This demonstrates that removing the abstraction overhead substantially improves not only schedulability but also analysis efficiency.

6 Conclusion

We have presented a framework called vC^{2}M for CPU, cache and memory bandwidth resource allocation on multicore virtualization systems, along with its design and implementation. Our framework integrates cache partitioning and BW regulation to provide strong cache and BW isolation among concurrently running tasks in a virtualized environment. Based on this capability, it introduces a novel resource allocation technique that can substantially improves schedulability by removing the abstraction overhead in the analysis and by allocating all three types of resources in a holistic manner. We have shown through extensive evaluations that vC^{2}M can effectively reduce interferences, and that it offers substantial performance benefits over existing solutions while being highly efficient.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*.
[2] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. 2006. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*.
[3] S. Chen, L. T. X. Phan, J. Lee, I. Lee, and O. Sokolsky. 2011. Removing Abstraction Overhead in the Composition of Hierarchical Real-Time Systems. In *RTAS*.
[4] A. Easwaran, M. Anand, and I. Lee. 2007. Compositional Analysis Framework Using EDP Resource Models. In *RTSS*.
[5] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. 2016. Ginseng: Market-Driven LLC Allocation. In *USENIX ATC*.
[6] M. Hassan, H. Patel, and R. Pellizzoni. 2015. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS*.
[7] D. Kim, H. Kim, N. S. Kim, and J. Huh. 2015. vCache: Architectural Support for Transparent and Isolated Virtual LLCs in Virtualized Environments. In *MICRO*.
[8] H. Kim and R. Rajkumar. 2016. Real-Time Cache Management for Multi-Core Virtualization. In *EMSOFT*.
[9] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. 2012. Realizing Compositional Scheduling through Virtualization. In *RTAS*.
[10] H. Leontyev and J. H. Anderson. 2008. A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees. In *ECRTS*.
[11] G. Lipari and E. Bini. 2010. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In *RTSS*.
[12] N. Navet, B. Delord, and M. Baumeister. 2010. Virtualization in Automotive Embedded Systems: An Outlook. In *RTS Embedded Systems*.
[13] I. Shin and I. Lee. 2003. Periodic Resource Model for Compositional Real-Time Guarantees. In *RTSS*.
[14] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. 2014. Real-time Multi-core Virtual Machine Scheduling in Xen. In *EMSOFT*.
[15] M. Xu, R. Gifford, and L. T. X. Phan. 2019. Holistic multi-resource allocation for multicore real-time virtualization systems. In *Technical Report*. http://www.cis.upenn.edu/~linhphan/papers/dac19-vC2M-techreport.pdf.
[16] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. 2017. vCAT: Dynamic Cache Management using CAT Virtualization. In *RTAS*.
[17] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. 2013. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *RTSS*.
[18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*.
[19] Y. Zhou and D. Wentzlaff. 2016. MITTS: Memory Inter-arrival Time Traffic Shaping. In *ISCA*.