USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Harvesting Spare CPU Resources in Container Systems

Adam Hall and Anirudh Sarma, *Georgia Institute of Technology;* Esha Choukse, *Microsoft Azure Research;* Umakishore Ramachandran, *Georgia Institute of Technology;* Sameh Elnikety, *Microsoft Research*

https://www.usenix.org/conference/nsdi26/presentation/hall

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Harvesting Spare CPU Resources in Container Systems

Adam Hall
Georgia Institute of Technology

Anirudh Sarma
Georgia Institute of Technology

Esha Choukse
Microsoft Azure Research

Umakishore Ramachandran
Georgia Institute of Technology

Sameh Elnikety
Microsoft Research

Abstract

Platforms like Kubernetes are widely adopted for deploying latency-sensitive cloud services in containers, and CPU resources for these containers are over-provisioned to ensure low 99th percentile tail latency under peak load. At the same time, cloud services exhibit bursty traffic patterns resulting in CPU usage variability that creates opportunity to harvest ephemerally unused CPU cores to run latency-tolerant containers. However, existing resource controls do not allow latency-sensitive containers to share unused cores without compromising their low tail latency objectives. Prior research on performance isolation is inadequate for container systems because it requires modifying applications and system software, employs offline profiling, and does not account for interference from processing container networking interrupts. We present *HarvestContainers*, a system that protects latency-sensitive containers from all sources of interference while harvesting their spare CPU cores to run latency-tolerant containers. Our solution dynamically determines the safe number of CPU cores to harvest and does not require rewriting applications or OS. We implement *HarvestContainers* integrated with Kubernetes and evaluate it experimentally. Our evaluation shows that latency-sensitive containers with microsecond-scale service level objectives can share up to 75% of their unused CPU cores while maintaining tail latency within 4% of standalone operation.

1 Introduction

Despite the widespread adoption of container-based virtualization, achieving efficient resource utilization while meeting stringent Service-Level Objectives (SLOs) in container platforms remains a challenge. Cloud platforms like Google Kubernetes Engine, Amazon Elastic Kubernetes Service, and Azure Kubernetes Service [5,9,41] allow users to deploy thousands of containers within a single cluster of servers. These platforms rely on containers to host a variety of application types, ranging from latency-sensitive (*e.g.*, web search) to latency-tolerant (*e.g.*, throughput-oriented machine learning training). Over-provisioning container resources is a common strategy to ensure latency-sensitive applications meet SLOs under peak demand [6,14,15], but leaves their compute capacity under-utilized [28,31]. Users of container platforms are faced with two unfavorable options: waste resources to protect tail-latency, or share resources and risk violating SLOs.

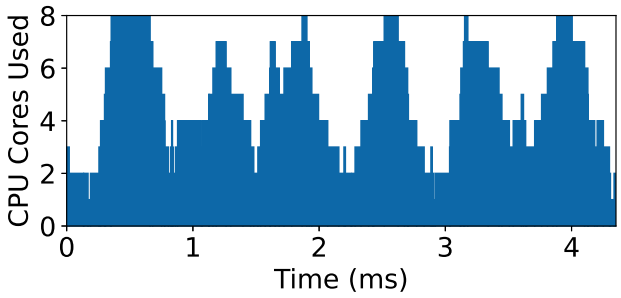

Figure 1: Number of CPU cores used by MySQL (in ■) over a 5 ms period. Ephemerally available CPU cores (in □) represent opportunity for harvest.

Latency-sensitive services typically exhibit *bursty* traffic patterns that are reflected in their CPU usage [1,2,25], and the CPU usage variability introduces opportunities for harvesting unused CPU cores to run latency-tolerant containers. For example, Figure 1 demonstrates the significant variability in CPU cores used by a latency-sensitive application. In this scenario, demand peaks at 8 cores despite using 4 cores on average. Over small periods of time (*e.g.*, on the order of a few milliseconds as depicted) core usage fluctuates, and during these brief periods unused cores are *ephemerally available* and can be harvested for use by other containers. Harvesting cores effectively is difficult because of this brief window of opportunity: cores must be reassigned quickly (*i.e.*, on the order of microseconds) to take full advantage of their availability. Furthermore, harvesting all unused cores increases the likelihood of interference, and so a subset of cores must be carefully selected with respect to the locality and burstiness of the latency-sensitive container. OS-level resource controls for containers are designed to share all unused cores and react too slowly to reassign cores within these tight constraints.

To increase resource efficiency, container platforms need a performance isolation mechanism that allows them to safely share some unused cores without compromising tail latency response times.

Existing approaches to performance isolation do not adequately address the opportunities and challenges present in container environments. First, these approaches use techniques that entail complex development and implementation overheads for containers. For example, Caladan [20] and Shenango [38] necessitate a custom thread library, requiring developers to change application source code and limiting effectiveness to highly concurrent applications that can use this library. SmartHarvest [43] supports unmodified applications in virtual machines (VMs) but instead requires modifying and recompiling the underlying hypervisor, shifting the complexity to platform operators. Second, they do not account for interference from interrupt handling. Container platforms are susceptible to interference from the large number of interrupts generated by container networking. These interrupts are often unpredictable, can be scheduled on the same cores as latency-sensitive containers, and represent an additional source of contention.

We demonstrate through proof-by-construction that container performance isolation can be achieved without changing well-established paradigms by introducing *HarvestContainers*, a system that enables latency-sensitive containers to safely share their spare CPU resources to be harvested. Our solution targets cloud container platforms running a mix of latency-sensitive workloads that occasionally need exclusive access to CPU cores and latency-tolerant workloads that can run on spare cores. It supports both bare-metal and VM deployments where the guest has full control over its cores, and is implemented as a Linux kernel module that can be loaded and unloaded on demand alongside an existing Linux kernel. This module extends the functionality of the OS without changing its source code or operations. Our design allows platform operators to easily integrate performance isolation with existing container platforms, thereby greatly increasing their CPU utilization.

The key insight to our design is that we mimic the CPU scheduling behavior when a latency-sensitive container runs standalone: when threads become ready to run, they do not wait in the dispatch queue of busy CPU cores, and are instead immediately dispatched to idle cores with warm cache. When cores are available for harvest, *HarvestContainers* holds some in reserve as a buffer that allows latency-sensitive container threads to run as soon as they become ready, mimicking standalone operation. Furthermore, it selects buffer cores that are free from interrupt interference to provide better isolation. The remainder of idle cores are shared with other containers until the buffer needs to be replenished.

Harvesting cores from individual latency-sensitive containers has a multiplicative effect when operating at scale. When *HarvestContainers* integrates with a container platform, it enables awareness of harvested resources, increasing the ability to make better use of CPU cores on each individual server. This in turn increases efficient CPU utilization across entire clusters of servers.

Our work makes the following contributions:

1. We demonstrate how performance isolation for containers can be achieved with non-invasive techniques. Our solution does not require changing OS or application source code, and supports sub-100 μs 99th percentile (P99) tail latency SLOs.
2. We identify and mitigate the impact of *interrupt request handling* on latency-sensitive containers.
3. We present a lightweight heuristic that dynamically determines (*i.e.*, without profiling) how many CPU cores can be safely harvested.
4. We integrate our solution with Kubernetes and evaluate it empirically, and harvest up to 75% of unused cores from a latency-sensitive container while keeping its tail latency within 4% of standalone operation.

2 Limitations of Existing Mechanisms

2.1 OS Resource Controls

Linux cgroups. Container CPU resource controls are provided by the cgroups *cpu* subsystem [21] under Linux. These controls are not comprehensive enough to share CPU and also protect latency-sensitive containers from interference. They partition CPU by time, whereas latency-sensitive containers need exclusive use of entire cores during periods of peak demand. Cgroups expresses CPU resources as *shares* (proportion of CPU time relative to other containers) and *quota* (limit on maximum amount of execution time during a given period). Neither setting restricts the specific cores where a container may be scheduled, only how long it can execute. Furthermore, these constraints are only enforced during periods of resource contention. This enforcement occurs on the order of milliseconds, which is too slow to meet the needs of applications with microsecond tail-latency SLOs. To achieve consistently low tail-latency, a container must have resources available immediately upon request. Any delay dispatching its ready threads that handle incoming requests greatly affects its tail latency.

Figure 2 demonstrates the limitations of cgroups. A latency-sensitive container running MySQL at 5k Requests Per Second (RPS) is scheduled alongside a throughput-oriented container that keeps cores fully utilized when executing. The MySQL container is provisioned with 8 cores needed to handle peak demand. During periods of lower demand, idle cores can be shared with the throughput-oriented container. We attempt resource sharing via cgroups shares and quota settings. The first setting restricts the throughput-oriented container to 1/9 shares of CPU time and gives 8/9 shares to the MySQL container. The result is all cores are utilized but MySQL tail latency (P99 response time) is 250% over standalone op-

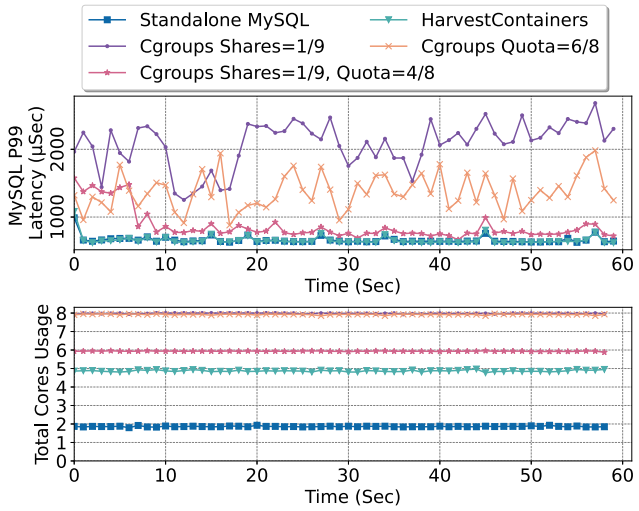

Figure 2: Latency-sensitive MySQL container tail latency (top) and CPU core utilization (bottom) when running standalone and co-located with a throughput-oriented container limited by cgroup resource controls. Our solution "HarvestContainers" achieves the same latency as Standalone MySQL while harvesting three CPU cores.

eration. If we restrict the throughput-oriented container to a quota of 6/8 cores, MySQL latency improves, but is still 225% higher than standalone. Restricting the throughput-oriented container to both 1/9 shares and 4/8 quota further improves MySQL latency, but it is still 19% higher than standalone. We provide head-to-head evaluations of cgroups vs. *HarvestContainers* in Section 5.7. In our experience, no combination of shares/quota can both improve CPU utilization and also protect tail latency SLOs.

Realtime CPU Schedulers. Linux offers a *realtime scheduler class (RT)* with two fixed priority scheduling polices named *SCHED_FIFO* and *SCHED_RR*, and a deadline-based policy named *SCHED_DEADLINE*. These policies are useful in embedded systems scenarios, but have major drawbacks for the cloud systems targeted by *HarvestContainers*. Threads scheduled with SCHED_FIFO do not execute within a time slice and are only preemptible by other threads with higher priority. Applications scheduled this way must be designed to gracefully yield when not doing useful work since they can dominate CPU resources indefinitely. Container orchestrators manage server clusters with generic workloads, and operators cannot rely on deployed containers to be realtime-aware. Making this assumption leads to platform instability [35]. SCHED_RR works in a round-robin fashion and has the same limitation because its round-robin scheduling only applies to tasks with the same priority. SCHED_DEADLINE can provide tasks with guaranteed timeslices for low response times under predictable workloads, but the kernel forbids those tasks from forking so as not to jeopardize admission control. This limitation prevents it from working with containers since the fork/exec paradigm is a cornerstone of most applications [7].

2.2 Orchestration Platforms

Container orchestration platforms like Kubernetes rely on OS resource controls like cgroups, so the limitations of these controls extend up the stack. For example, in Kubernetes CPU resource requirements are expressed as *requests* and *limits*, which map to cgroups *shares* and *quota*, respectively. They are used to categorize containers into Quality of Service (QoS) classes [4]. QoS classes can guarantee exclusive cores for a latency-sensitive container, but prevent those cores from being shared when unused. Since requests and limits are enforced by cgroups, they cannot protect the tail latency of latency-sensitive containers while harvesting spare CPU cores to run latency-tolerant containers.

2.3 Interrupt Request Handling

Latency-sensitive containers cannot maintain low tail latency with interrupt request (IRQ) activity on their cores. Table 1 demonstrates the impact of interrupt interference on latency-sensitive containers, which inflates the P99 latency by up to 33% over standalone. Interference from IRQ handling is a new problem for performance isolation and remains unaddressed in related work. Handling interrupts for containers is difficult because container networking generates up to 3x more interrupts than bare metal or virtual machines. This interrupt activity necessitates reserving an even larger buffer to reduce the chance of disrupting the container, diminishing the potential for harvest. We discuss this problem further in Section 3.5.

App	RPS	Standalone P99	P99 with IRQ Interference
Memcached	50k	49.1 μs	65.5 μs (+33%)
MySQL	4k	411 μs	501 μs (+22%)
Xapian	2.5k	2.8 ms	3.4 ms (+21%)

Table 1: Impact of interrupt interference on latency-sensitive applications.

Existing solutions can steer interrupt handling to different cores but lack semantic awareness of latency-sensitive containers, and are unable to prevent interrupt handling interference. The Linux kernel mechanisms Receive-Side Scaling (RSS) and Receive Packet Steering (RPS) [19] target improving network throughput by spreading interrupt handling across multiple cores. The *irqbalance* [10] userspace tool distributes interrupts across different cores to spread the burden of interrupt handling. And Linux Receive Flow Steering (RFS) [19] and proprietary hardware-specific tools like Intel *FlowDirector* [23] place network traffic interrupts on the cores of applications that generated the traffic. These tools work well for processes on bare metal machines, but cannot track flows across virtual Ethernet interfaces in container systems.

3 HarvestContainers: Approach and Design

In this section, we present our approach to container performance isolation. To aid in discussion, we refer to containers hosting latency-sensitive applications as *Primary* and containers hosting latency-tolerant applications as *Secondary*. We discuss scenarios involving a single Primary and single Secondary, but note that *HarvestContainers* supports running multiple Primaries and Secondaries simultaneously.

3.1 Design Objectives

A container-focused performance isolation system should achieve the following objectives:

1. It must recognize the immediate state of a Primary's CPU cores. As the Primary runs, its cores change between Active (currently executing tasks) and Idle (not currently executing any task) states.

2. It must draw on CPU state information to determine the resource needs of the Primary container. Workload changes create a surplus or deficit of unused cores, which determines whether resource sharing can occur.

3. It must modify the Secondary's core assignments in response to a surplus or deficit of cores. If there is a surplus, some cores should be shared with the Secondary. If there is a deficit, cores should be unshared and made exclusive to the Primary.

4. It must communicate with the container orchestrator to detect new container deployments to manage their resources accordingly.

We distill these objectives into four types of operations. First, we *Monitor* the system to track the immediate state of Primary CPU cores. Monitoring must happen as quickly as possible to recognize ephemerally available unused cores (a surplus) or reclaim cores the Primary needs (a deficit). It must also be comprehensive enough to include CPU usage from sources both visible and invisible to the scheduler. Second, we *Balance* cores shared between Primary and Secondary. Balancing must take into account the historic resource needs of a Primary to determine how many cores can be safely shared. To absorb sudden bursts of activity, a Primary should always hold some spare cores in reserve as a buffer. When deciding which cores to keep exclusive and which to share we take into account the locality of Primary threads to preserve warm caches. After the *Balance* operation has identified cores to share, we perform an *Actuate* operation to update assignments for the Secondary. Actuate never modifies the Primary's assignments, so the Primary can behave as if it is running standalone. Instead, we only grow the Secondary's core allocation when there is a surplus and shrink it when there is a deficit. Finally, we *Listen* for messages from the container orchestrator to know when a new container is deployed. The *Listen* operation works with the orchestrator control plane to determine the type of the container being deployed (Primary or Secondary) and what its resource needs are. In the remainder of this section, we discuss how each of these four operations fits into the design.

3.2 Monitoring the System

The *Monitor* operation continually records the instantaneous state of CPU cores. This state data is also accumulated to include how often each core has been used by a Primary or Secondary container, and how long the Primary demonstrated a deficit or surplus of idle cores over a system-defined period. Because the CPU scheduler is unaware of interrupt activity, *Monitor* also gathers information on how often each core handles interrupts. This state data is recorded historically and used to determine the resource needs of the Primary during the *Balance* operation. Continuous monitoring is necessary to quickly identify when and how unused cores should be shared, increasing resource utilization and decreasing the potential for disrupting the Primary. Many performance isolation techniques achieve monitoring by modifying the application [8, 20] or OS [43]. We can avoid this overhead in a container environment since all containers share the same kernel. Instead, we perform monitoring from within kernel space via a kernel module that allows us to recognize CPU state changes on the order of nanoseconds.

3.3 Sizing Buffer Cores

When a Primary's workload increases, its threads wake up and need to be immediately scheduled on unused cores. To ensure this condition, we reserve some unused cores as **Buffer** to absorb sudden bursts in Primary workload. Using a buffer allows the Primary to operate as if it is the only container on the system by preventing delays in scheduling threads to idle cores. The subset of unused cores not reserved as Buffer is safely shared with the Secondary. We refer to these cores as **Harvest** cores. They represent the set difference between unused and Buffer cores.

Sizing buffer cores is critical to allowing the Primary to behave as if it is the only container on the system. We dynamically determine this sizing via a heuristics-based approach. Existing approaches require OS/application changes (*e.g.*, through the use of green threads [20] and instrumentation [8]) or result in sub-optimal decision making (*e.g.*, over/underestimation via machine learning prediction [43]). Our evaluations show that employing a lightweight heuristic can achieve the same tail-latency guarantees as existing approaches while minimizing overhead.

The *Balance* operation employs our heuristic and decides which cores to use for Buffer and Harvest. The number of cores that can be harvested depends on the amount of buffer a Primary needs. The need for buffer changes as a Primary's workload changes. During periods of higher demand, more buffer is required to absorb sudden bursts in activity. During periods of lower demand, less buffer is required. The number of idle cores available relative to the amount of buffer needed

creates a surplus or deficit of cores. Cores that have been historically used by Primary threads are preferred as Buffer to preserve cache locality. Doing so improves the performance of the Primary, especially when operating at microsecond-scale latency. Cores that show heavy interrupt activity are preferred as Harvest cores. Biasing these cores toward the Secondary helps to shield the Primary from disruption.

How much Buffer a Primary needs depends on its *burstiness*, which is the number of its threads that become active over a short period of time and must be dispatched on idle cores. Burstiness can be characterized by how often the Primary needs to use all of its available cores. We quantify this property online using a system-defined sampling time (*e.g.*, 1 second), where we record the fraction of the sampling time that *all* of a Primary's provisioned cores, *as stipulated by the developer as the container size*, are utilized. We term this value the *All-Cores Busy Fraction* (ACBF) since it is indicative of a condition where the Primary's utilization is at 100%. ACBF is specific to a given Primary for a specific workload. To confirm this intuition, we plot the ACBF values for different Primaries as a function of request rate for different workloads in Figure 3. The ACBF for the given Primaries is small due to their bursty nature that results in very brief but unpredictable periods where all threads wake up simultaneously and require the use of all cores. The ACBF for the peak workload of a Primary represents the most difficult scenario for harvest.

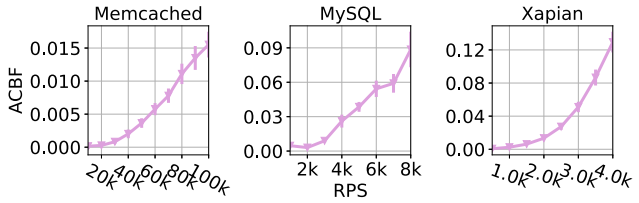

Figure 3: All-Cores Busy Fraction (ACBF) values for different Primaries at different loads. Note that ACBF increases with increase in RPS.

As the workload decreases, the ACBF also decreases since fewer resources are needed. For example, at 4k RPS Xapian's ACBF is 0.13 whereas at 2k RPS the ACBF is 0.01. Figure 3 shows this trend is consistent across all the Primaries we tested (Memcached, MySQL, and Xapian). We refer to these increases and decreases in ACBF as the *waxing* and *waning* of the Primary, respectively. *HarvestContainers* reacts to waxing and waning conditions through the use of an *Observation Phase* and a *Harvest Phase*.

Observation Phase. Each workload a Primary handles has a different waxing and waning pattern that varies the ACBF over time. We employ an Observation Phase to capture these patterns and use them to decide when to increase or decrease buffer size. A Primary begins waxing when its workload continually increases, thereby increasing CPU utilization. It begins waning when its workload continually decreases, in turn decreasing CPU utilization. During the Observation Phase of a Primary, harvesting is disabled (*i.e.*, no cores are shared with the Secondary) and its ACBF values are recorded at regular intervals (*e.g.*, 1 second). The values collected during the Observation Phase form the *ACBF curve*. The duration of the Observation Phase is system-defined, and is set to 60 seconds in our implementation. If the ACBF curve is bounded by its $ACBF_{High}$ and $ACBF_{Low}$ values, a window is formed that characterizes the CPU utilization of the Primary when it is operating at a specific workload. We term this window the *Current Harvest Window* (CHW). When a Primary is deployed, it always begins in an Observation Phase.

Harvest Phase. Once an Observation Phase completes, the system enters a Harvest Phase during which it dynamically adjusts the Primary's buffer size while allowing spare cores to be harvested for the Secondary. These adjustments are guided by computing $ACBF_{Current}$ and comparing this value with $ACBF_{High}$ and $ACBF_{Low}$ to determine if the Primary is waxing or waning.

Waxing: $ACBF_{Current} > ACBF_{High}$ indicates that the ACBF has moved above the CHW and serves as a strong signal that the Primary's workload is waxing. Therefore, we increase the Primary's buffer size by 1 to protect tail latency while maximizing harvested cores. A more conservative approach would increase the buffer by multiple cores at a time, but in our experience this is unnecessary due to our agility in recognizing and reacting to changes on the fly.

Within the CHW: If a sampled $ACBF_{Current}$ value is within the CHW, we continue to gather more samples over a system-defined sampling period (*e.g.*, 10 seconds) to build confidence that the Primary's workload is stable. This design protects the Primary's tail latency by reducing the risk associated with shrinking the buffer core size.

Waning: The inequality $ACBF_{Current} < ACBF_{Low}$ is a good indicator that the Primary is waning. In this case, we immediately reduce the buffer size since there is a high degree of confidence that fewer buffer cores are needed.

The Harvest Phase performs two operations. First, it determines the ideal buffer size based on the Primary's ACBF curve. It grows or shrinks the buffer size as the curve moves above, within, or below the window. Second, it maintains this buffer size by continually increasing or decreasing the number of cores harvested. When the Primary uses one or more of its buffer cores, the buffer is replenished by reclaiming harvest cores from the Secondary.

When the Harvest Phase begins, the Primary's buffer is set to its maximum size and then gradually adjusted. Growing and shrinking the buffer happens less frequently (on the order of seconds) since the ideal buffer size for a Primary's workload does not change often. Conversely, increasing and decreasing the number of harvested cores happens very frequently (on the order of microseconds) since the Primary's buffer must be replenished as soon as it has been used.

Phase Transitions. *HarvestContainers* automatically tran-

sitions between Observation and Harvest phases. When the system is in a Harvest Phase, if $ACBF_{Current}$ moves above the CHW it will continually increase the Primary's buffer size by 1 until either the ACBF moves back within the CHW or the Primary is using its maximum number of cores. When the buffer size cannot be increased further, harvesting is disabled and an Observation Phase is started to re-establish the *new* CHW. After an Observation Phase, the system sets the CHW bounds and transitions back into a Harvest Phase. Note that re-establishing a new CHW is only required during waxing. It is expected that the system will remain in Harvest Phase most of the time and sporadically switch to an Observation Phase as warranted by the ACBF sampling.

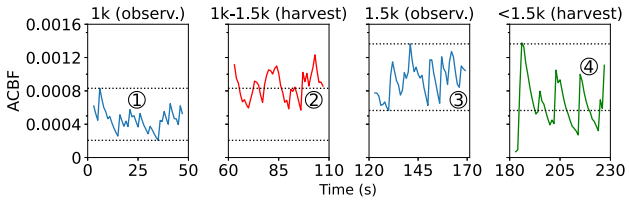

Figure 4: The ACBF characterizes a Primary's burstiness at a given workload. HarvestContainers shifts between observation and harvest phases to establish a Current Harvest Window (CHW) based on a Primary's current workload.

Figure 4 demonstrates how CHW works in practice. Here we see a Primary running at 1k RPS in the Observation Phase (①). During this phase the CHW is established based on the activity of the Primary (blue line). After the Observation Phase, the system transitions into a Harvest Phase where the CHW is used to estimate the ideal buffer size for the Primary running under a 1k RPS workload (②). While in the Harvest Phase, the Primary's workload increases from 1k to 1.5k RPS (red line in ②). *HarvestContainers* notices this increase due to $ACBF_{Current}$ going above the CHW (②). *HarvestContainers* responds to this change by increasing the size of the Primary's buffer in an attempt to size it for the new workload. After the buffer has been increased to its maximum size, the ACBF still remains above the window (red line) and it becomes necessary to trigger a new Observation Phase. This new Observation Phase (③) establishes a new CHW that matches the increase in Primary workload. After the new Observation Phase completes, the system returns to a Harvest Phase. During this Harvest Phase (④, green line), the system operates at or below 1.5k RPS and the ACBF remains within or below the window. As a result, *HarvestContainers* is able to safely decrease the buffer size to match the workload. This continual adjustment of the buffer size and CHW allows *HarvestContainers* to preserve the shape of scheduling, dynamically adapting to changes that arise due to a Primary, a Secondary, or underlying systems effects.

3.4 Managing CPU Core Assignments

When a Primary has a deficit or surplus of cores, the system must act immediately to either take advantage of the surplus or correct the deficit. Delays in correcting a deficit increase the potential to disrupt the Primary since it may not have enough exclusive cores to handle sudden bursts in workload. Delays in taking advantage of a surplus lead to underutilization of harvest cores that are ephemerally available. The *Actuate* operation applies decisions made by the *Balance* operation in response to system state changes. A deficit is corrected by immediately evicting Secondary threads from harvested cores and making those cores exclusive to the Primary, thereby replenishing its buffer. A surplus is taken advantage of by moving Secondary threads to harvested cores as soon as they are available. We refer to these operations as shrinking and growing the Secondary's core allocation, and they occur continually throughout the lifecycle of a Primary and Secondary that share resources.

3.5 Managing Interrupt Activity

Interrupt activity is a significant source of disruption to latency-sensitive containers for three reasons:

1. Container networking generates significantly more interrupt activity than bare metal or virtual machines: each container requires a pair of virtual Ethernet interfaces (one inside the container and one on the host) and a bridge connecting them to the outside world. Each traversal of a virtual interface raises a software interrupt, so there are up to 3 times as many interrupts for a container vs. a hardware NIC [27, 29].

2. Interrupt processing is invisible to the CPU scheduler: from the CPU scheduler point of view, a CPU core would appear idle even when busy processing an interrupt.

3. Existing system support to steer interrupt handling to different cores makes no distinction between latency-sensitive and latency-tolerant container locations. In effect they cannot protect latency-sensitive containers from interrupt handling interference.

The Primary should be shielded from cores that handle interrupts as much as possible. We address this requirement in three ways. First, we directly manage where interrupt handling occurs by affinitizing interrupts to a system-defined subset of cores. Affinitization is not a new technique, but unlike other tools our design guarantees some cores are always available for interrupt handling while ensuring those cores do not overlap with the Primary's allocation. In our experience, affinitizing interrupts to a subset of cores is not disruptive to applications running in containers. Second, we dynamically adjust the affinity of interrupts to use harvested cores. This adjustment works similarly to how spare cores are shared with Secondary containers. Third, we prioritize using cores

with high interrupt activity as Harvest cores to bias them away from the Primary.

3.6 Interfacing with the Cluster Orchestrator

Large scale container clusters rely on an orchestrator like Kubernetes that deploys containers to servers. We integrate with the orchestrator through the *Listen* operation. The *Listen* operation runs on each server in the cluster. It informs the orchestrator of the number of exclusive cores that are available for Primary containers and whether there are enough spare resources for Secondary containers to use. The orchestrator uses this information when deciding whether to place a container on a particular server. When the orchestrator makes a decision, it informs the *Listen* operation of its intent to deploy a Primary or Secondary container so that it can be managed by *HarvestContainers*.

3.7 Tradeoffs & Limitations

Although our design requires no underlying system modification, there are tradeoffs to this black-box approach. Existing approaches that require application instrumentation have a unique insight into workload dynamics, and can adapt to changes based on near perfect knowledge. Our heuristics-driven approach achieves strong performance, but requires switching between Observation and Harvest Phases if the Primary's CPU activity changes significantly. This switching can temporarily reduce harvest, but the cost is amortized over the lifetime of the Primary. Approaches that modify the OS also achieve near perfect knowledge, and place threads on CPU cores by sidestepping the OS scheduler. Our approach has to work alongside the scheduler, and accommodate its mechanisms. Scheduler redesigns are rare, but if they occur then our design would need to adapt in kind. Modifying the OS/application also allows for better estimation and handling of contention for resources like memory and cache. We do not manage these resources directly, but since our design is non-invasive it can work seamlessly alongside mechanisms that do (*e.g.*, Intel CAT [24]). Approaches that rely on syscalls do not modify the OS or application, and have similar flexibility to our design. However, their performance is constrained due to syscall overheads, and thus they can only support millisecond-scale applications with limited harvest. We consider our design to be a firm middle ground that performs on par with contemporary solutions while avoiding the need for modifications that disrupt the underlying ecosystem.

4 Implementation

HarvestContainers is implemented as a Linux kernel module and a complementary daemon that runs in userspace. Both components are written in C and consist of 3,004 lines of code in total. Our implementation does not require changing the source code of the underlying OS or applications and supports running multiple Primary and multiple Secondary containers simultaneously. To reduce communications overhead, these components read/write data and control signals via a shared memory region. In this section, we describe the implementation. Although our discussion provides Linux-specific details, the design principles apply universally. Increasing harvested cores while protecting tail latency requires significant agility in monitoring and assigning Buffer and Harvest cores, which comes from working in tandem with OS kernel mechanisms.

Monitor. *Monitor* operations record the immediate state of a Primary's CPU cores. We recognize CPU core state changes on the order of nanoseconds by running *Monitor* operations inside of kernelspace via a kernel module. Running inside the kernel is necessary because accessing CPU statistics from userspace is too slow. Linux provides these statistics via its */proc* virtual filesystem [18]. Reading a virtual file requires system call context switching, and when that call is made the data exported to */proc* is generated on demand. Each query to */proc* is on the order of milliseconds, exceeding the small window of opportunity to act on state changes. Instead, we query the *idle_cpu()* kernel function from within our kernel module to determine the immediate state of each Primary core. This function is also used by the OS scheduler, and provides a strong indicator of CPU core state. Because interrupt handling is a significant source of interference to containers, it is also crucial to recognize when Primary cores spend time handling interrupts. However, this interrupt activity is invisible to the scheduler, and cannot be gathered through *idle_cpu()* queries. Instead, we gather interrupt handling statistics from the *kernel_cpustat* kernel data structure to infer how much time each core spends handling interrupts.

Fast monitoring both increases protection for the Primary and improves resource utilization. For optimal performance, we run the kernel module pinned to a dedicated core, allowing us to detect core state changes in under 60 ns. This implementation decision is most effective for several reasons. First, recognizing when a Primary begins using its buffer cores allows us to immediately begin replenishing that buffer. Second, the small window of opportunity to harvest cores means they must be used as soon as they are available to increase resource utilization. Third, interrupt handling activity needs to be identified early prevent it from disrupting the Primary. Dedicating a core to monitoring is a reasonable tradeoff since it results in harvesting a significant number of otherwise unused cores.

Balance & Actuate. The *Balance* operation is responsible for the logic that determines Buffer and Harvest core assignments. It is implemented in userspace to reduce the footprint of code running inside a privileged kernel context. The Balancer uses information from the Monitor to answer two questions: (1) how many unused cores the Primary has and (2) which of those cores to keep as Buffer and which to share as Harvest. The buffer is sized using the heuristic described in Section 3.3. Cores recently used by the Primary are prioritized to become Buffer cores, followed by cores with the least amount of interrupt activity. Cores with the most interrupt activity are prioritized as Harvest cores.

The *Actuate* operation runs in kernelspace and applies the core assignments made by the Balancer. Assignments are modified by changing the CPU core affinity of a container. When a container's affinity changes, the Linux scheduler rebalances its threads across the new set of cores. The scheduler's rebalance algorithm chooses good thread-to-core placements, but takes several milliseconds to complete a single rebalance. This delay makes existing kernel affinity mechanisms too slow to use for *Actuate*. The limitation arises from the *sched_setaffinity()* kernel function used for affinity changes. When the affinity of a container is updated to include new cores, *sched_setaffinity()* does not immediately migrate threads to those cores. Instead, it waits until the periodic rebalance algorithm is triggered. For example, if a container has four threads on core1 and its affinity is updated to core1-4, all threads will remain on core1, even though that core is busy and idle cores are available. The threads are only migrated when the scheduler performs a rebalance, which occurs on the order of milliseconds. Since harvest cores are ephemerally available, by the time rebalance occurs the window of opportunity to utilize them would have passed.

We introduce a novel method called *Two-Phase Affinity* that immediately places work on idle cores, allowing us to complete core assignments in 30 μs. *Two-Phase Affinity* is complementary to *sched_setaffinity()*. It works by splitting Secondary container core reassignment into two phases. During Phase 1, we affinitize individual threads from Secondary containers to the individual cores that have just been made available for harvest. By setting the affinity directly to a single core, we ensure a thread is immediately moved from its current core to an idle core. For example, if a Secondary has two threads running on core4 and the harvest grows to core4-5, we remove core4 from one thread's affinity and add core5. The thread gets migrated to core5 immediately since core4 no longer exists in the affinity. Forcing migrations in this way avoids the need to wait for the Linux scheduler to rebalance. Phase 1 ends after threads have been assigned to new harvest cores. During Phase 2, the "true" affinity of each thread is set. In our previous example, Phase 2 would adjust the affinity of each thread to include core4-5. This final phase is important, since it allows the Linux scheduler to eventually achieve a proper balance of threads over all harvest cores.

We balance interrupts similar to how we balance Secondary containers across harvest cores. If Secondary containers have their own NIC, we affinitize their interrupts to a system-defined set of non-Primary cores using the *irq_set_affinity()* kernel function. To ensure there is sufficient resource for interrupt processing, interrupts can always run on these cores. When the number of harvest cores changes, interrupt affinity is increased or decreased depending on whether there is a surplus or deficit, allowing Secondary interrupts to take advantage of unused resources while protecting the Primary from interference. If Primary and Secondary share a single NIC, protection from interrupt interference is still provided by prioritizing cores with heavy interrupt activity to act as Harvest cores, biasing them away from the Primary.

Listen. We implement the *Listen* operation as a local userspace agent that binds to a system-defined TCP port and listens for incoming connections from the container orchestrator. For our evaluations, we implemented a custom Kubernetes scheduler with a QoS class that we denote as *Harvest*. The *Harvest* class allows the system to dynamically switch between treating core allocations as "exclusive" when needed and as "shared" otherwise. Our custom scheduler works as a complementary component to the Kubernetes global scheduler. When a developer creates a container's configuration in Kubernetes, they specify the container type (Primary or Secondary), the total number of cores it needs, and the maximum number of cores that can be shared for harvest. Our scheduler decides where to place a container by comparing its core allocation requests with the metadata from all servers in the Kubernetes cluster. After a server has been chosen, the scheduler contacts its agent and sends the container identifier and configuration. Note that *Listen* is not specific to Kubernetes. Any orchestrator can interact with *HarvestContainers* by requesting and sending the metadata necessary to make scheduling decisions.

5 Evaluation

In this section, we perform a systematic evaluation to answer key questions on *HarvestContainers*' performance:

1. How much can it harvest from a Primary while still meeting SLOs under different workloads? (Section 5.3)
2. Why is it effective at harvesting significant unused resources while also protecting the Primary? (Section 5.4)
3. How effective is it at shielding a Primary from interrupt interference? (Section 5.5)
4. Does it remain effective in a realistic scenario with multiple Primary and Secondary containers running side-by-side? (Section 5.6)
5. How does it compare to existing resource controls (cgroups, SmartHarvest)? (Sections 5.7 and 5.8)

5.1 Methodology

We evaluate *HarvestContainers* using a mix of latency-sensitive Primary containers and throughput-oriented Secondary containers. Our Primary containers run applications that were chosen for their low, medium, and high CPU utilization and SLO tail latency response times at the P99. They are listed in Table 2. Our Secondary containers run throughput-oriented applications that stress the CPU. They are listed in Table 3. CPUBully allows us to create predictable synthetic workloads that demonstrate a worst-case scenario for managing resource sharing between Primary and Secondary. The other Secondary applications are used to gauge the performance of *HarvestContainers* under realistic conditions.

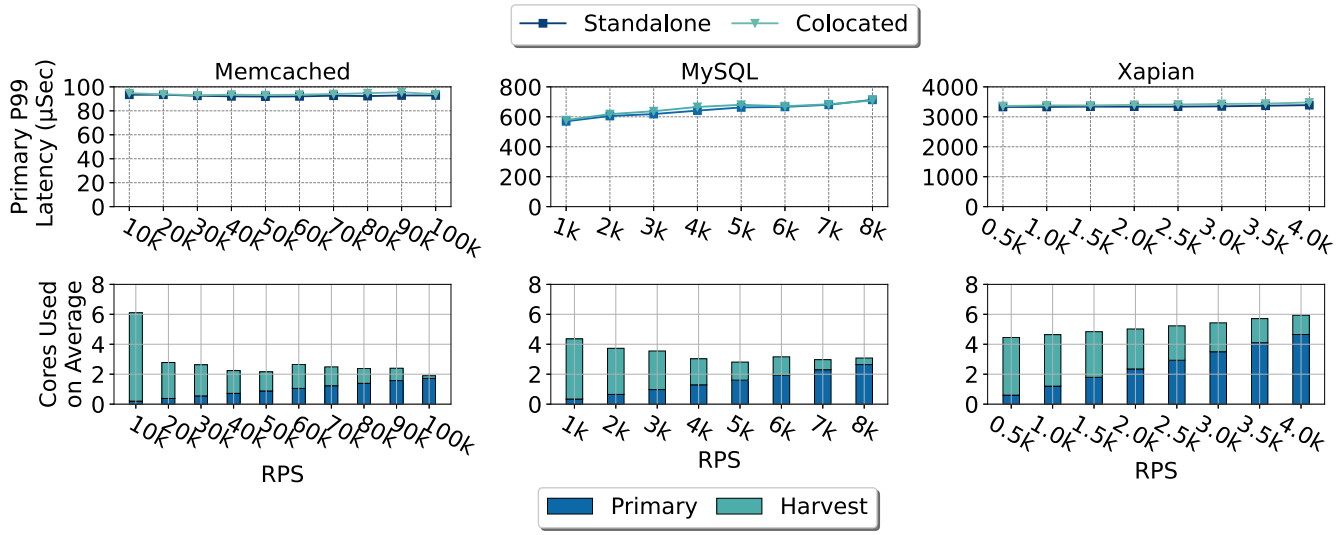

Figure 5: HarvestContainers performance while dynamically sharing cores from each of the three Primary containers with a CPUBully container. As RPS decreases, buffer is decreased and more cores are harvested (bottom graphs) while maintaining P99 close to standalone operation (top graphs).

Primary Application	Low RPS	Mid RPS	High RPS	Baseline P99
Memcached	10k	50k	100k	96 μs
MySQL	1k	4k	8k	715 μs
Xapian	0.5k	2.5k	4k	2.9 ms

Table 2: Primary applications.

Secondary Application	Progress Metric	Baseline (1 core, 60 s)
CPUBully	Computations completed	64
x264	Frames encoded	364
Dedup	Size of file deduped in MB	1086

Table 3: Secondary applications.

5.2 Experiment Setup

System. All experiments are run on servers from Cloudlab [16] configured with dual 16-core Intel Xeon Gold 6142 CPUs and 384 GB of RAM. Servers are interconnected by 10 GbE NICs in a Kubernetes cluster. To achieve stable, repeatable test results we disable c-states, p-states, turbo boost, and SMT on all nodes and limit test containers to run on cores within the same NUMA node where possible. CPU frequencies are set to their maximum of 2.6 GHz.

Each Primary container is provisioned with 8 cores, such that all Primary containers can run side-by-side on a single server in the cluster. A Primary may only run on the 8 cores it has been provisioned. Workloads are sized so that low-latency SLOs can be met using 8 cores under peak demand. Reducing the number of cores would result in P99 violations due to application threads sharing core runqueues during periods of bursty activity. This determination was made and verified empirically. Each Secondary container is provisioned with 1 exclusive core and otherwise may only use harvest cores as they become available. *HarvestContainers* components are restricted to run on a single core.

Workloads. For our Primary Memcached workload, we use the mutilate load generator [31] to perform queries with real-world trace distributions published by Facebook [3]. For our Primary MySQL workload we use YCSB [12] to perform a Zipfian distribution of read-only scan queries, with each query retrieving a range of 15 records, each 250 bytes. And for our Primary Xapian workload we use TailBench [26] to perform queries against a 15 GB corpus of entries from Wikipedia, with request inter-arrivals sampled from an exponential distribution and search queries uniformly sampled from an input dataset.

For Secondary workloads, we configure CPUBully to perform enough work to fully utilize all available cores for the entire duration of each benchmark; dedup to repeatedly compress a 20 GB data file with enough threads to occupy all available cores; and x264 to continually encode a 1.5 GB video file with enough threads to occupy all available cores. The efficacy of our system is determined by its ability to allow Primary containers to maintain P99 tail latency response times close to standalone operation and Secondary containers to make progress relative to running standalone.

5.3 Performance of *HarvestContainers*

We first show that *HarvestContainers* can dynamically determine the amount of buffer needed and harvest cores by running each Primary alongside a CPUBully Secondary. This setup represents harvesting in the most difficult scenario where a Secondary always uses 100% of any core it is given. Figure 5 shows the results of our evaluations. Note that the figure depicts *average* number of cores used. Recall that Primary workloads are bursty, resulting in a low average core utilization but requiring all 8 cores when workload activity bursts. This variability necessitates reserving a large

number of buffer cores at higher RPS, and conversely also creates greater opportunities for harvest at lower RPS. Dynamic buffer sizing capitalizes on this opportunity to significantly improve harvest. The upper charts show the P99 response times for Primary containers running different workloads. The response times are within 4% of standalone while cores are being harvested. Latencies remain stable during harvest because we dynamically resize the buffer to match the Primary's needs. As the workload of each Primary decreases, the necessary buffer size also decreases and more cores become available for harvest. The "Harvest" bars in the lower charts show the number of cores (on average) that CPUBully gets to use at different workloads. CPUBully's progress is captured by the number of computations it makes over the duration of the experiment, and translates to between 0.2 to 6 cores harvested. Without our solution, cores shown by the "Harvest" bars would have otherwise remained unused.

5.4 Performance Analysis

There are two aspects to strong harvest: right-sizing the Buffer and acting quickly to take advantage of Harvest cores. *HarvestContainers* is effective at reclaiming unused resources due to its two novel techniques:

Dynamic buffer core sizing allows us to quickly and accurately determine the ideal amount of Buffer and Harvest cores for a Primary under different workloads. To understand why this is important, we re-ran our evaluations from Section 5.3 using static buffer settings. The buffer for each Primary was set to a fixed size that matches its needs under peak demand. Figure 6 compares the results of this experiment to the dynamic approach. When a fixed buffer size is used, an unnecessary amount of cores are kept in reserve as the workload decreases. As a result, significantly fewer cores can be reclaimed for harvest. Our novel heuristic presented in Section 3.3 allows us to size the Buffer in a way that closely reflects the needs of a Primary at any given time, increasing harvest by up to 260% in our evaluations.

Two-Phase Affinity places Secondary threads directly on Harvest cores, allowing them to begin executing as soon as those cores are available. *HarvestContainers* shows strong resource utilization because it does not rely on OS mechanisms like *cpuset* that are slow in migrating threads to cores. Section 4 discusses why this slowdown occurs. To demonstrate the impact of slow migration, we repeat our Memcached benchmark from Section 5.3 while using *cpuset* and Two-Phase Affinity *(2PA)* to harvest cores. Figure 7 shows the results of this experiment when Memcached is running at 80k RPS. Both methods protect the P99 of the Primary, but *cpuset* reclaims just 0.1 cores because it takes several milliseconds to move Secondary threads to harvest cores. The window of opportunity to use those cores often passes before the migration is even completed. Conversely, 2PA is able to harvest 2.4 cores because it allow Secondary threads to begin work immediately instead of waiting for a costly rebalance operation.

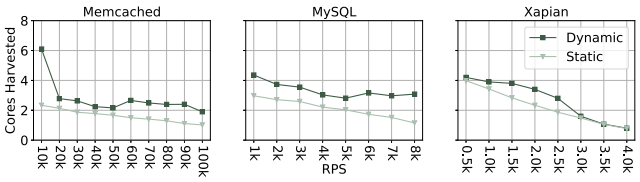

Figure 6: Cores harvested with Dynamic vs. Static Buffer.

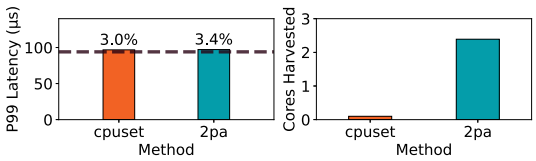

Figure 7: P99 and harvested cores when using cpuset and Two-Phase Affinity (2PA) to harvest.

5.5 Handling Interrupt Interference

HarvestContainers shields Primary containers from interrupt interference. Figure 8 demonstrates the impact of this interference with and without our solution. In this experiment, we run each of our Primary containers at low, medium, and high workloads (Table 2) alongside NetworkBully: a version of CPUBully that both consumes CPU and generates network traffic to create IRQ activity. IRQ activity causes P99 to inflate by up to 41% over baseline if unaccounted for. When we shield the Primary from IRQ interference, the number of cores harvested is approximately the same while the P99 stays within 4% of the baseline. These results indicate our method introduces minimal overhead but provides significant benefit.

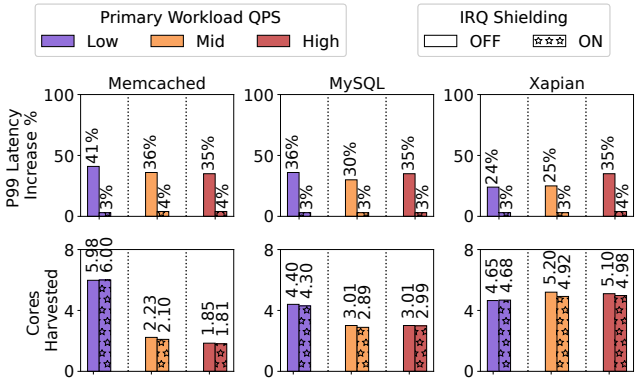

Figure 8: Impact of IRQ interference on latency-sensitive containers. When HarvestContainers IRQ shielding is enabled, tail latencies remain within tolerance and harvesting remains high.

5.6 Multiple Primaries and Secondaries

We now evaluate the efficacy of *HarvestContainers* when multiple Primary and Secondary containers run simultane-

ously. To simulate the bursty and unpredictable behavior of real world applications, we select x264 and Dedup as our Secondary workloads. The baseline progress of both Secondaries is measured as the amount of work they can perform using a single core over the test duration (Table 3).

We deploy two Primary containers (MySQL and Xapian) alongside the Secondary containers running simultaneously on the same server. Each Primary is assigned 8 exclusive cores. Each Secondary is assigned 1 exclusive core and is eligible to use any cores harvested. The Primaries run under low, medium and high workloads (Table 2). Figure 9 demonstrates the results of this experiment. The latency of each Primary is within tolerance of its baseline SLO, while Secondaries make progress under low, medium, and high workloads due to *HarvestContainers* reclaiming up to 50% of unused cores.

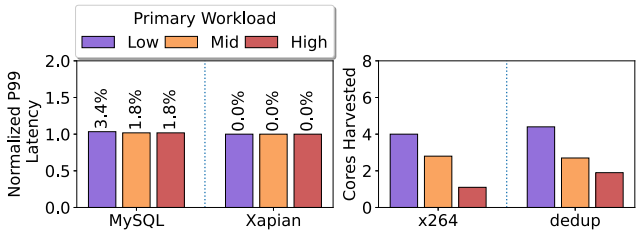

Figure 9: HarvestContainers supports running multiple Primary and Secondary containers side-by-side.

5.7 Comparison to Cgroups

Existing resource controls for Linux cannot provide adequate performance isolation since they focus on partitioning CPU resources in terms of time instead of whole cores. To demonstrate this limitation, we evaluate cgroups with a combination of *shares* and *quota* settings that most closely match the buffer core settings for workloads used to evaluate *HarvestContainers*.

Figure 10 demonstrates the results of our evaluations. Cgroups fails to keep Primary tail latency within an acceptable threshold at high workloads. Under less demand, P99 is stable but harvest is negligible because of the need to place hard limits on Secondary execution time via quotas, as discussed in Section 2.1. Comparatively, *HarvestContainers* is able to reclaim an acceptable number of cores for the Secondary while ensuring increases to Primary tail latency response times remain close to standalone operation.

5.8 Comparison to SmartHarvest

SmartHarvest diverges from our work in four respects:

1. Its main core reassignment method takes 10 ms, diminishing the opportunity to harvest and necessitating a large buffer to protect the Primary. We complete core reassignments in 30 μs.

2. It has a faster alternative core reassignment method, but requires modification of the Windows hypervisor. We do not require any modification to existing systems.

3. It uses an online learner that over-predicts the number of buffer cores needed, significantly diminishing harvest. Our agility allows us to use a heuristics-based method to more accurately recognize the actual number of buffer cores at any given time.

4. It does not recognize and handle interrupt interference, a major source of disruption for containers.

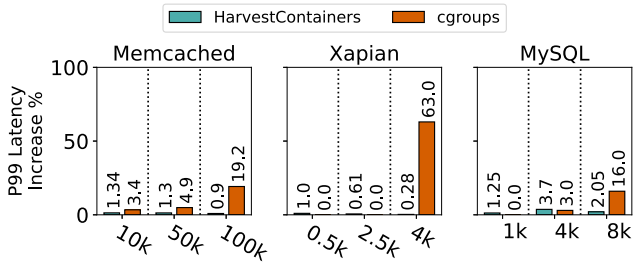

Figure 10: Comparison of P99 of Primary containers when sharing CPU resources via HarvestContainers and cgroups.

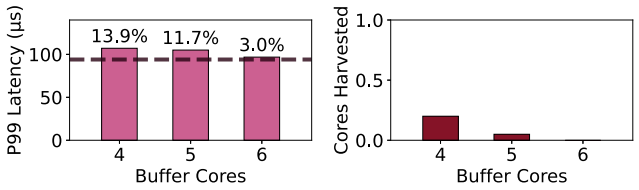

Figure 11: P99 and harvested cores when sharing resources between Memcached and CPUBully via a simulated version of SmartHarvest.

Harvest Speed. SmartHarvest is closed-source, so we cannot evaluate it directly. Instead, we simulate SmartHarvest by modifying *HarvestContainers* to use *cpuset*, the Linux equivalent to the *cpugroups* method SmartHarvest uses. Both methods have similar overhead that necessitates the need for our novel Two-Phase Affinity solution (Section 5.4). We repeat an experiment from the SmartHarvest paper that harvests from Memcached under an 80k RPS workload. Figure 11 shows the results of repeating this experiment with different buffer core sizes. When using 4 or 5 buffer cores, simulated SmartHarvest is unable to protect tail latency, causing it to spike up to 14% over baseline. At the same time, a maximum of just 0.2 cores are harvested. SmartHarvest needs a large buffer size due to its slow harvest speed, and as a result the Primary is only protected when the buffer is sized to 6 out of 8 cores. With a buffer of this size and a reaction time in milliseconds, no cores can be effectively harvested.

Prediction vs. Heuristics. Because it is slow to reassign cores, SmartHarvest needs an online learner to predict demand ahead of time. This prediction overestimates the number of cores needed, resulting in little to no harvest. To understand why, we repeat the last experiment but estimate buffer

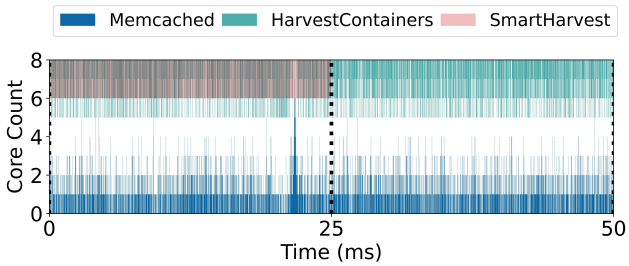

Figure 12: SmartHarvest cannot harvest in the 2nd window because it overestimates demand in the 1st window due to a transient burst in Primary demand. HarvestContainers harvests more CPU in both windows because it reacts to current demand instead of relying on estimates.

size with the *Vowpal Wabbit* [11] predictor SmartHarvest uses. Figure 12 demonstrates the results of this experiment. SmartHarvest predicts core demand over 25 ms windows. During the first window, it correctly estimates a demand of 6 cores. Toward the end of this window, a transient spike in CPU utilization causes the predictor to overestimate a demand of 8 cores, and in the next window, SmartHarvest is unable to harvest. The behavior was consistent when tested across other workloads. Conversely, *HarvestContainers* leverages heuristics that recognizes demand as it occurs, and harvests 30% more cores over the same period (■ lines) while providing the same performance isolation guarantees.

6 Related Work

Performance Isolation. There are many existing approaches to performance isolation. PerfIso [25] targets bare metal machines and relies on Windows kernel mechanisms to monitor core utilization and assign harvest cores. They access these mechanisms in userspace via system calls, which introduces overhead that limits them to millisecond-scale operation. Further, they use a static set of buffer cores sized for peak demand that is determined by offline profiling, limiting the amount of harvest and the agility of the solution in adapting to changing workloads. SmartHarvest [43] improves on PerfIso by bringing performance isolation to VMs and using machine learning to predict CPU demand. However, it still relies on slow hypervisor mechanisms that limit its potential for harvest and prevent it from supporting ultra-low-latency workloads. Techniques requiring detailed application metrics [8, 17, 22, 32, 33, 37] approach performance isolation from the perspective of the application instead of the OS. This approach provides strong insight into application dynamics, but requires extensive modification to the application. Other approaches focus on reclaiming CPU cycles (as seen with MSH [34]) or enabling hardware support for harvesting (as seen with HardHarvest [42]), but also modify the underlying system. We avoid offline profiling and deliver strong performance isolation for co-located and unmodified applications.

Resource Efficiency. There is significant interest in improving the efficiency of myriad datacenter resources for cost savings. *Iron* [27] and *Falcon* [30] focus on improving network throughput by mitigating deficiencies in the network stack and are complementary to *HarvestContainers*. Similarly, work that targets isolating other resources such as memory, storage, GPU, and networking [36, 44–47] cover aspects of isolation that *HarvestContainers* does not and are also complementary to the functionality it provides.

Scheduling Systems. Many scheduling systems [22, 38, 39] rely on custom thread libraries to control the scheduling of applications. Although performant, their design decisions come at additional cost, requiring invasive instrumentation of Primary and Secondary applications and context switching between user- and kernel- spaces to manage their threads. Caladan [20] iterates on this method by using a kernel module to more rapidly manage thread-to-core allocations, but still requires a custom application thread library. We also leverage a kernel module for agility, but require no modification to any part of the system to manage containers from within that module. Scheduling systems that focus on container scaling and cluster-level orchestration, such as AutoPilot [40] and Escra [13], attempt to right-size container resources instead of safely sharing them, and are complementary to our work.

7 Conclusion

We present *HarvestContainers*, a system that provides performance isolation to latency-sensitive containers while harvesting their spare CPU resources to run latency-tolerant containers. When a latency-sensitive container has spare cores, we hold some in reserve as a buffer to absorb sudden bursts of activity and harvest the rest. This buffer ensures latency-sensitive threads can be dispatched as soon as they become ready. Our system dynamically determines the ideal buffer size as the latency-sensitive container's workload increases and decreases, protecting tail latency response times while increasing the number of cores that can be harvested. It also protects latency-sensitive containers from being disrupted by interrupt handling, a significant source of interference unique to container systems. *HarvestContainers* exists as a complementary solution to the Linux scheduler, and does not require changing the source code of the OS, kernel, or applications that it manages. We implement and integrate it with Kubernetes and verify its performance empirically. Our evaluations show that *HarvestContainers* is able to harvest up to 75% of a latency-sensitive container's unused cores while keeping its P99 tail latency responses within 4% of standalone operation.

8 Acknowledgements

We thank our anonymous reviewers, our shepherd Dr. Raja Sambasivan, members of the Embedded Pervasive Lab at Georgia Tech, and our collaborators at Microsoft for their thoughtful feedback. This work was funded in part by an NSF grant (CNS-2423711) and a gift from Microsoft Corp.

References

[1] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjöstedt-de Luna, Johan Tordsson, and Erik Elmroth. Measuring cloud workload burstiness. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 566–572. IEEE, 2014.

[2] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, 2018.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] The Kubernetes Authors. Pod Quality of Service Classes. https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/, 2023.

[5] Microsoft Azure. Managed Kubernetes Service (AKS). https://azure.microsoft.com/en-us/products/kubernetes-service, 2025.

[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Springer, 2013.

[7] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork () in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 14–22, 2019.

[8] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Google Cloud. Google Kubernetes Engine (GKE). https://cloud.google.com/kubernetes-engine, 2025.

[10] Irqbalance Contributors. Irqbalance. https://irqbalance.github.io/irqbalance/, 2024.

[11] Vowpal Wabbit Contributors. Vowpal Wabbit. https://github.com/VowpalWabbit/vowpal_wabbit, 2024.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[13] Greg Cusack and Maziyar Nazari. Escra: Event-driven, sub-second container resource allocation. In *IEEE International Conference on Distributed Computing Systems*, 2022.

[14] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[15] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[17] Xenofon Foukas and Bozidar Radunovic. Concordia: Teaching the 5g vran to share compute. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 580–596, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Linux Foundation. The /proc Filesystem. https://docs.kernel.org/filesystems/proc.html, 2023.

[19] Linux Foundation. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2024.

[20] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[21] Red Hat. 3.2 CPU. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu, 2021.

[22] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. Ghost: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Intel. How to Set Up Intel Ethernet Flow Director. https://www.intel.com/content/www/us/en/developer/articles/training/setting-up-intel-ethernet-flow-director.html, 2024.

[24] Intel. Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2025.

[25] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[26] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[27] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, April 2018. USENIX Association.

[28] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 7–16, 2015.

[29] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. Parallelizing packet processing in container overlay networks. *EuroSys 2021*, 2021.

[30] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. Parallelizing packet processing in container overlay networks. In *Proceedings of the 2021 EuroSys Conference*, pages 261–276, 2021.

[31] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2), may 2016.

[34] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting memory-bound {CPU} stall cycles in software with {MSH}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 57–75, 2024.

[35] Michael M Madden. Challenges using linux as a real-time operating system. In *AIAA Scitech 2019 Forum*, page 0502, 2019.

[36] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[37] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 409–420. IEEE, 2017.

[38] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[39] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[40] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling

at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[41] Amazon Web Services. Amazon Elastic Kubernetes Service (EKS). https://aws.amazon.com/eks/, 2025.

[42] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. Hardharvest: Hardware-supported core harvesting for microservices. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 708–722, 2025.

[43] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 1–16, 2021.

[44] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 607–618, New York, NY, USA, 2013. Association for Computing Machinery.

[46] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the i/o when allocating your llc. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[47] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. {PilotFish}: Harvesting free cycles of cloud gaming with deep learning training. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 217–232, 2022.