HardHarvest: Hardware-Supported Core Harvesting for Microservices

Jovan Stojkovic
University of Illinois at Urbana-Champaign, USA
jovans2@illinois.edu

Chunao Liu
Purdue University, USA
liu2849@purdue.edu

Muhammad Shahbaz
University of Michigan, USA
msbaz@umich.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

Abstract

In microservice environments, users size their virtual machines (VMs) for peak loads, leaving cores idle much of the time. To improve core utilization and overall throughput, it is instructive to consider a recently-introduced software technique for environments with relatively long-running monolithic applications: Core Harvesting. With this technique, Harvest VMs running batch applications temporarily steal idle cores allocated by Primary VMs running latency-critical applications, and return them on demand. Unfortunately, re-assigning cores across VMs has substantial overhead, resulting from hypervisor calls, context switching, and flushing TLBs/caches. While such overhead is acceptable in monolithic application environments, it would be prohibitive in environments with sub-millisecond microservices.

To address this problem, this paper proposes, for the first time, an architecture for core harvesting *in hardware*. The architecture, called *HardHarvest*, targets microservices. It aims to: 1) maximize core utilization, 2) minimize impact on Primary VM tail latency, and 3) boost Harvest VM throughput. HardHarvest eliminates software overheads by using in-hardware request scheduling and partitioning TLBs/caches with a smart replacement algorithm. On average, compared to state-of-the-art software core harvesting, HardHarvest increases core utilization by 1.5×, increases Harvest VM throughput by 1.8×, and reduces Primary VM tail latency by 6.0×.

CCS Concepts

• **Computer systems organization → Multicore architectures**; *Cloud computing*;

Keywords

Microservices; cloud computing; core harvesting

ACM Reference Format:
Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. 2025. HardHarvest: Hardware-Supported Core Harvesting for Microservices. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731071

1 Introduction

Cloud computing is experiencing a paradigm change, as many applications shift from large monolithic deployments to compositions of lightweight, loosely-coupled *microservices* [66]. Each microservice is implemented and deployed separately, and executes a portion of the application's logic. This composable software design simplifies programming. Moreover, it allows each service to be shared among different applications, while being scaled independently of other services. As a result, microservices are popular in companies such as Amazon [62], Netflix [83], Alibaba [48], Twitter [79], Uber [10, 80, 95], Meta [32, 72], and Google [27].

Microservice *instances* are Virtual Machines (VMs) or containers that serve microservice invocations (i.e., *requests*). An instance is created with a specified number of cores and amount of memory, and serves requests for that microservice. Although requests are short-running (typically, hundreds of μseconds), instances are long-lived: they can be up and serve requests for days [32].

The frequency of request arrival for a given microservice varies substantially with time, and exhibits bursty patterns. To attain good performance even at peak loads, users typically provision instances to handle these infrequent load spikes. As a consequence, instances are typically greatly *overprovisioned*—e.g., in number of cores needed. The result is that, in microservice environments, *allocated but idle* cores are a major waste [48]. As an example, using open-source production-level microservice traces from Alibaba [48], we see that 50% and 90% of microservice instances have an average core utilization lower than 16.1% and 40.7%, respectively.

To combat resource inefficiency under *general loads*, providers have implemented various techniques in their software stacks [3, 4, 26, 54, 55]. Amazon allows a Spot VM to seize unallocated cores if needed [3]. Further, Microsoft introduced a new type of VM called *Harvest* VM [5, 22, 88, 92] that can dynamically grow by harvesting cores. In such environments, there are two types of VMs: Primary and Harvest VMs. *Primary* VMs run latency-critical applications, need predictable high performance, and are created with a specified number of cores; *Harvest* VMs run batch applications, have loose performance requirements, can tolerate resource fluctuations, and are charged at a lower cost. Harvest VMs dynamically grow by harvesting unallocated cores in the server [5] or, additionally, by taking temporarily idle cores allocated by a Primary VM [88]. When the Primary VM needs its cores, it reclaims them back.

In practice, re-assigning a core from one VM to another has high overhead. First, a scheduler must perform two hypervisor calls: one to detach the core from the first VM, and the other to attach it to the second VM. Second, an expensive cross-VM context switch is performed. In addition, the state left in the re-assigned core's private caches and TLBs is a potential source of leakage. Specifically, a malicious tenant could force conflicts in these structures and learn information from an earlier tenant such as cryptographic keys [84]. Hence, the core's private caches and TLBs are typically flushed and invalidated during the reassignment [7, 9, 24, 71, 82, 84, 85, 94]. Unfortunately, flushing and invalidating private caches and TLBs, and the resulting cold-cache and cold-TLB restart add substantial overhead. Overall, we find that the sum of all these overheads can easily exceed 5ms. These overheads are particularly insidious when a core is reclaimed by its owner Primary VM, as they directly impact the response time of latency-critical applications.

The Harvest VM concept has been applied only to setups where Primary VMs run relatively long monolithic applications [5, 22, 88, 92]. In such applications, a reassigment overhead of a few ms can be considered negligible. However, in microservice environments, it is not tolerable to suffer a few-ms reassignment overhead every time that a 100-μs microservice request is received and needs to execute. The situation is even more challenging in an aggressive environment that can re-assign a Primary VM idle core not just when the core has terminated a request execution, but also when the execution is stalled on I/O—as such events happen frequently.

To address this problem, this paper proposes the first architecture that supports core harvesting *in hardware*, called *HardHarvest*. The goal is three-fold: attain high core utilization, introduce minimal or no increase in the tail latency of Primary VM microservice requests, and deliver substantial increases in the throughput of batch workloads in Harvest VMs. To accomplish these goals, HardHarvest targets the two main overheads present in software-based core harvesting.

The first overhead is the core re-assignment. To minimize it, HardHarvest adds hardware queues of microservice requests. A microservice request arrives as a network packet that contains the name of the microservice function to invoke and the input data required by that function. The message payload is deposited into the LLC and a pointer to the payload is stored in a hardware request queue. A core is re-assigned from one VM to another by being allowed to dequeue requests from the new VM's hardware queue when the original queue is empty. There is no need for detach/attach system calls and the context switch is accelerated in hardware.

The second overhead is flushing and invalidating TLBs and private caches on core re-assignment, and the resulting cold restart. HardHarvest leverages the fact that microservices typically have small working sets, and partitions TLBs and private caches into two regions: *Harvest* and *Non-Harvest* regions. When a core executes a Primary VM, it can use both regions; when it executes a Harvest VM, it is allowed to use only the Harvest region. When a core transitions between VMs, only the Harvest region is flushed and invalidated; the Non-Harvest region preserves the Primary VM's state during harvesting. In addition, HardHarvest enhances the effectiveness of such partitioning with a smart cache/TLB replacement algorithm that retains important state in the Non-Harvest region.

We evaluate HardHarvest with full-system simulations of an 8-server cluster, where each server has a 36-core IceLake-like processor [35]. Our evaluation shows that HardHarvest is very effective. On average, compared to state-of-the-art software core harvesting, HardHarvest increases core utilization by 1.5× and Harvest VM throughput by 1.8×, while reducing Primary VM tail latency by 6.0×. Compared to a system without core harvesting, HardHarvest increases core utilization by 3.4× and Harvest VM throughput by 3.1×, without increasing the tail latency of Primary VMs.

This paper's contributions are as follows:
- A characterization of the opportunities of supporting hardware-based core harvesting in microservice environments.
- HardHarvest, the first architecture for hardware core harvesting.
- An evaluation of HardHarvest for microservice environments, comparing it to state-of-the-art core harvesting and no harvesting.

2 Background

1. Microservice Applications. Figure 1 shows an example of a microservice-based application (*ComposePost*) from the DeathStar-Bench suite [23]. Each service, e.g., *Text* or *SGraph*, performs its dedicated functionality, communicates with other services, and can scale independently of other services.

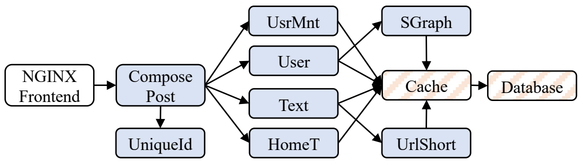

Figure 1: *ComposePost* microservice-based application. Blue boxes are microservices. White and patterned boxes are frontend and backend helper applications, respectively.

Microservices make software development and application scaling easier, but introduce numerous overheads. Thus, a large body of work has optimized different aspects of microservice environments [11, 33, 38, 58, 59, 63, 73, 76]. Importantly, the application decomposition places tight latency objectives on individual services [73], making the *tail latency* a key performance metric [16]. Any overheads, such as request queuing, that are considered negligible for large monolithic applications, can significantly degrade the tail latency of microservices.

Requests for a given microservice are served by one or more *Instances* present as separate VMs or containers. We focus on VMs due to their current prevalence in the cloud. When created, a VM is assigned a certain number of cores and amount of memory that it can use. Cloud providers use these resource limits to pack the VMs on servers. To accommodate the peak load, users typically overprovision VMs, leaving resources such as cores underutilized throughout the majority of the VM lifetime. Even when there is substantial load for the microservice, requests may not fully utilize cores, as cores often stall on synchronous RPCs to read/write to/from remote storage, or to invoke other microservices.

2. Resource Harvesting in the Cloud. Traditionally, a good way to improve server utilization in datacenters has been to co-locate batch workloads with user-facing applications [47, 50]. Batch workloads, such as machine learning training [86] or in-background data processing [92], can then use resources left idle by user-facing applications to improve their throughput. Recently, the same approach has been used for VMs in the cloud, where software support allows VMs running batch applications to harvest cores [5, 88, 92], memory [22], and storage [93].

Core harvesting has received the most attention, and has been explored through a new type of VM called *Harvest VM* [5, 88, 92]. Harvest VMs dynamically change their size by temporarily stealing idle cores. In the initial designs, Harvest VMs could steal only cores that were not allocated to any latency-critical VM (i.e., *Primary* VM), and had to return the cores once a new Primary VM was created [5]. State-of-the-art harvesting schemes, such as SmartHarvest [88], further improve core utilization by temporarily stealing idle cores allocated to Primary VMs. The system monitors the core utilization of all Primary VMs in the server. Then, based on the predictions of core utilization in the near future, the system may decide to reassign some cores to the Harvest VM. When a Primary VM needs the cores back, SmartHarvest returns the borrowed cores. Since current approaches to re-assign a core from one VM to another involve substantial software overheads, SmartHarvest keeps a few idle cores on stand-by in an emergency buffer. If needed, these cores can be reclaimed by Primary VMs.

The Harvest VM concept has been applied only to environments with long-running monolithic applications in Primary VMs [5, 22, 88, 92]. For these applications, the overhead of core re-assignment may be tolerable. For microservices, however, which typically run for hundreds of μseconds, we will see that the core re-assignment overhead is too high. The situation is even more challenging in a microservice environment where idle Primary VM cores can be stolen not just after a core has terminated the execution of a microservice request, but also when a core has stalled execution due to I/O—which is frequent.

3. Microarchitecture Structure Flush and Invalidation on Core Re-Assignment. In multi-tenant clouds, when a core is reassigned from one VM to another, the state left in the core's private microarchitectural structures such as caches and TLBs is a potential source of information leakage. The new VM being scheduled could observe private state that the old VM being preempted has left in these structures. Consequently, structures such as private caches and TLBs are flushed and invalidated when a core switches from one VM to another. This is both described in research papers (e.g., [7, 9, 24, 71, 82, 94]) and documented in literature from cloud providers. For example, a 2024 blog from Microsoft production [85] and a 2024 paper from Microsoft Research [84] say that they use microarchitecture state flushing and scrubbing in their production systems when a core moves from one VM to another.

Following these ideas, in this paper: 1) we partition the shared last-level cache into one partition per VM and 2) we require that, on a core context switch from one VM to another, all the levels of private caches and TLBs in the core are flushed and invalidated.

3 Motivation for In-Hardware Core Harvesting

To understand the opportunities of hardware-supported core harvesting for microservices, we analyze a large set of applications: production-level traces of Alibaba's microservices [48], and DeathStarBench [23] microservices (acting as latency-sensitive workloads running in Primary VMs). The traces provide a time series of average, maximum, and minimum core utilization of microservice instances. We run the microservices on an Intel IceLake server with 36 2.4GHz cores, 256GB of DRAM, and an LLC with 1.5MB per core. The server runs Ubuntu 22.04 and the KVM hypervisor. To generalize the insights, we also run the microservices as Docker containers and observe similar results. We identify several opportunities for hardware-based core harvesting.

Opportunity: **Cores allocated to microservice instances in the cloud are heavily underutilized.** When a user deploys a microservice instance in the cloud, they specify the resources needed for the instance, including the number of cores [2, 25, 56]. The instance is typically sized to guarantee it can handle the peak load, leading to low average utilization [12, 48, 87, 88]. Figure 2 shows the distribution of the average and maximum core utilization of Alibaba's microservice instances. The utilization is low: half of the instances have an average core utilization lower than 16.1%, and 90% of instances have a maximum core utilization lower than 40.7%. These numbers represent great opportunities for harvesting.

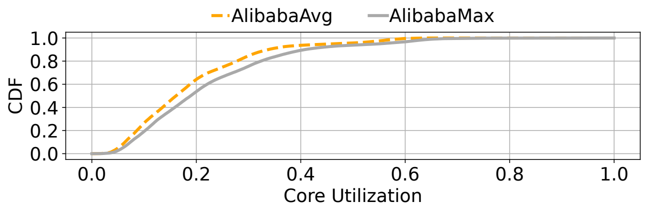

Figure 2: Core utilization of Alibaba's microservice instances.

Opportunity: **If the overhead of hardware core harvesting is low, significant performance gains can be attained.**

To understand this opportunity, we make three observations.

• **There are large fluctuations in a VM's core utilization or load over time.** We analyze Alibaba's traces, which provide a granularity of 30-second measurements. Figure 3 shows the core utilization of a representative Alibaba VM over time.

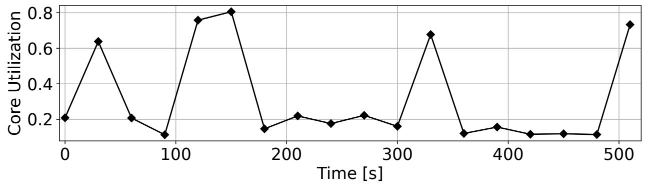

Figure 3: Core utilization of an Alibaba microservice.

While core utilization is often low, we observe that it can increase due to bursts of requests. Let us assume that these VMs are Primary ones. During the low-load periods, some of their cores can be harvested by Harvest VMs. Then, when the load spikes, the harvested cores must be quickly returned to Primary VMs to prevent increases in the tail latency of Primary VM requests.

In current systems, there are two main sources of overhead to move a core between VMs: 1) software overhead of invoking the hypervisor to reassign the core from one VM to another, and 2) flushing and invalidating the caches of the reassigned core (so that no cache state leaks across a VM transition) and subsequent cold-cache re-start of the execution.

• **Core re-assignment via hypervisor is costly.** In state-of-the-art software-based harvesting [88], a user-space agent monitors the utilization of all the cores in Primary VMs and, based on the utilization, may decide to migrate a core to a Harvest VM. To do so, it performs a hypervisor call to detach the core from the first VM and another to attach it to the second VM, using cgroup tools. With the KVM hypervisor, moving a core across VMs takes ≈5ms. Half of this time is spent on detaching/attaching the core, and half on loading the new VM's context. The state-of-the-art SmartHarvest design [88] reduces the cost of detaching/attaching the core to 100s of μs (which is about the execution time of a microservice).

We quantify the impact of core reassignment on the tail latency of microservices. We run DeathStarBench microservices [23] on our server with 4-core VMs, inducing the same core utilization as the one in the Alibaba traces. We detach an idle core from a Primary VM and attach it to a Harvest VM. Later, when the Primary VM receives a request, we move the core back. In our experiments, the Harvest VM is always idle. Hence, the caches of the reassigned core are not flushed/invalidated. Figure 4 shows the tail latency of microservices in Primary VMs without (*No-Move*) and with the overhead of core reassignment. We execute the system with the open-source KVM hypervisor [40] and move a core from the Primary VM either on termination of a request invocation only (*KVM-Term*), or on both termination and at every blocking I/O call in the invocation (*KVM-Block*). In either case, we move the core from a Primary to the Harvest VM *only* if the Primary VM has no other requests ready to run. We observe an average of 11 and 36 core reassignments per second with *KVM-Term* and *KVM-Block*, respectively. We also emulate the optimized reassignment latencies reported in SmartHarvest [88] (*Opt-Term* and *Opt-Block*).

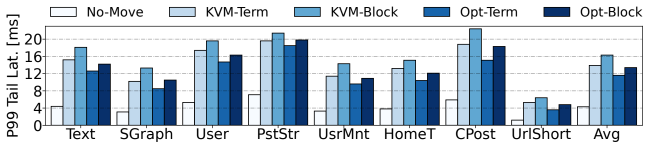

Figure 4: P99 tail latency of microservices in Primary VMs with the hypervisor overheads of core reassignment.

We see that core reassignment with *KVM-Term*, *KVM-Block*, *Opt-Term*, and *Opt-Block* increase the P99 tail latency substantially: by 3.2×, 3.8×, 2.7×, and 3.1×, respectively, on average. It can be shown that the overheads increase even further with higher numbers of cores and VMs, and with higher loads. Since these overheads are high, SmartHarvest keeps some idle cores on stand-by in an "emergency" buffer, resulting in even lower core utilization. These cores are reclaimed by Primary VMs when they receive new work.

• **Cache flush and invalidation on core re-assignment and subsequent cold-cache restart are expensive.** As discussed in Section 2.3, we flush and invalidate a core's private caches and TLBs when the core is reassigned from one VM to another. Unfortunately, current processors do not have an efficient way to do it, resulting in high overheads. For example, Intel's wbinvd [20] flushes and invalidates the *whole* cache hierarchy of a given core, and takes 300–500μs. Further, Intel's clflush flushes only one cache line, so one needs to execute it many times. In addition, as an invocation starts on a re-assigned core, it finds cold caches and TLBs.

We again consider two cases: a conservative design where an idle core is taken from a Primary VM only when it has finished executing a request, and an aggressive design where an idle core is also taken when it is blocked on I/O. We note that starting (or restarting) a request on a cold cache and TLB is costly. Even threads that handle different invocations of the same microservice share a large fraction of their memory footprint.

Figure 5 quantifies the impact of flushing and invalidating caches and TLBs (via the wbinvd instruction) and restarting with cold caches and TLBs. The figure shows the tail latency of microservices in Primary VMs without flushing, with flushing in the conservative design (*Flush-Term*), and with flushing in the aggressive design (*Flush-Block*). With wbinvd, the processor does not wait for the external caches to complete their write-back and invalidation operations. It only waits for internal caches (L1/L2) to be written-back and invalidated. As we use the existing wbinvd instruction, these experiments do not include the time it takes to write back and invalidate external caches. Hence, this implementation is not safe. In our evaluation (Section 6), when we simulate this architecture, we place a fence after the wbinvd instruction. The figure has two extra bars (*Harvest-Term* and *Harvest-Block*) which, additionally, add the overhead of core re-assignment using the optimized hypervisor software shown in Figure 4. These two extra bars represent the current true cost of core re-assignment [88].

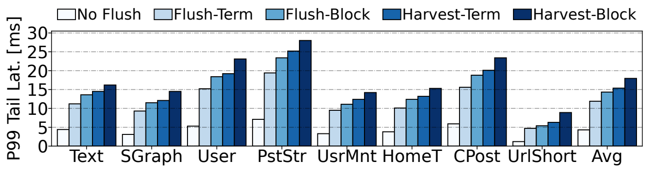

Figure 5: P99 tail latency of microservices in Primary VMs with cache/TLB flushing and, for the last two bars, both cache/TLB flushing and hypervisor core reassignment.

On average, cache/TLB flushing increases the P99 tail latency by 2.7× (*Flush-Term*) and 3.3× (*Flush-Block*). Further, if we add the hypervisor reassignment overhead, the tail latencies of *Harvest-Term* and *Harvest-Block* are 3.6× and 4.2× higher, respectively, than the no-flush design. These are the substantial costs that hardware-supported harvesting may help minimize.

Putting it all together, Figure 6 shows the execution time of a single service request in steady state without and with core harvesting. The core harvesting environment includes the optimized hypervisor core reassignment design in [88] and cache/TLB flushing and invalidation. The figure shows two bars per service: one with no harvesting (left bar) and one with harvesting (right bar). The latter is broken down into three components: hypervisor reassignment of cores (*Core Reassign*), flush/invalidation of the caches and TLBs (*Flush/Inval*), and execution of the request (*Execution*). We see that, on average, a request takes 1.9× longer with core harvesting. In addition to the core reassignment and flush/invalidation overheads, the execution time itself under core harvesting takes 1.2× longer than before due to using cold microarchitectural structures.

Opportunity: **Microservice invocations have relatively small working sets.** In an environment with frequent cache and TLB

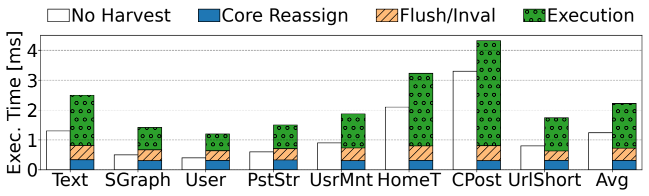

Figure 6: Execution time of a single service request in steady state without core harvesting (left bar) and with core harvesting (right bar). The latter is broken down into its components.

flushing and invalidation, it may be reasonable to reserve a section of these structures for the Primary VM, and not flush this section. But, this approach is only plausible if microservice invocations have small working sets. In practice, this is the case. We assess the working set sizes of DeathStarBench microservices in two ways. First, we configure our server with a smaller LLC and do not observe any performance change. Specifically, our IceLake server has a 54MB LLC organized in 12 ways. We use Intel CAT [34] to partition the LLC and allow a microservice to use either the full LLC, 3/4, 1/2, or 1/4 of the LLC. We observe that, even with a 1/4 of the LLC, the tail latency of microservices does not degrade more than 1%.

Second, we simulate a server where *all* caches and TLBs are smaller: we model the server with the full caches and TLBs, and then we reduce the number of ways in all structures to 75%, 50%, and 25%, while keeping the number of sets constant. We also model the performance of a simulated environment with infinite caches and TLBs. We use the SST simulator [67] with QEMU [70], and validate the simulation accuracy by calibrating the results with the real system, with and without LLC partitioning. Figure 7 shows the tail latency of microservices when running with different sizes of all caches and TLBs. All microservices see a very small impact even when operating with 1/2 of the whole cache/TLB hierarchy.

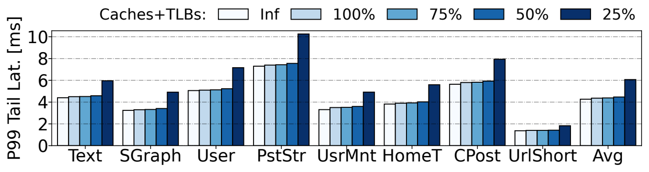

Figure 7: Tail latency of microservice invocations on a system with a fraction of the whole cache and TLB hierarchy.

4 HardHarvest: Core Harvesting in Hardware

Based on the previous observations, we propose *HardHarvest*, the first processor micro-architecture that delivers high-performance core harvesting by *supporting it in hardware*. HardHarvest allows Harvest VMs to steal idle cores from Primary VMs with very low overhead, and return them on demand with minimal impact on the tail latency of Primary VMs. In this way, HardHarvest simultaneously enables high utilization of cores, high throughput for Harvest VMs, and minimal impact on Primary VMs. HardHarvest targets the two main sources of overhead in core harvesting: (i) core reassignment and (ii) cache and TLB flush/invalidation and cold restart. We present how we address each source in turn.

4.1 Minimizing Core Reassignment Overhead

4.1.1 Main Idea of the Proposed Solution

To minimize the latency of core reassignment, we propose to support part of its operation in hardware. Specifically, we design hardware schedulers that schedule requests for VMs. The schedulers optimize core migration between VMs. Further, cores dequeue requests from their own VM's queue by using a low-overhead *dequeue* instruction. When a core cannot find work, it is automatically reassigned to a Harvest VM by simply allowing it to dequeue a job from that VM's queue. There is no need to issue (de)attach system calls as scheduling is done in hardware.

A conventional *detach* operation involves: 1) issuing a hypervisor call (switching from user-space to privileged mode) [44], 2) acquiring a lock [42], and 3) sending an interrupt to the affected core [43]. An *attach* operation follows the same steps. In each case, HardHarvest's hardware avoids the first two software overheads. First, it bypasses the hypervisor and directly re-assigns the cores across VMs in hardware. Second, as hardware schedulers work in a decentralized manner, there is no need to acquire the global lock.

In addition, when a Primary VM receives a request and all its cores are busy, if any of its cores is executing a request for a Harvest VM, an interrupt is sent to one of such cores. That core then saves the state of its Harvest VM's request, performs a context switch, and dequeues the new request from the Primary VM's queue without issuing any hypervisor (de)attach system calls.

With this hardware, we estimate that a core re-assignment from Harvest to Primary VM takes a few μs. If, in addition, we speed-up context switching by adding hardware support for saving and restoring the process state, we estimate that a core re-assignment takes *a few 10s of ns*. Recall that a software approach to reassign a core across VMs takes from a few hundreds of μs to a few *ms*.

4.1.2 Detailed Hardware Design

In HardHarvest, a processor has a hardware controller with request queues. Figure 9 shows the design. There is a single hardware request queue (RQ) that is dynamically divided into different (logical) subqueues, one for each running VM. To ensure isolation, VMs cannot access each other's subqueues. In addition, there are a number of hardware *Queue Managers* (each of which can control a request subqueue) and a number of *VM State Register Sets* (each set can store a VM state shared by all the threads of a VM). Such state includes registers such as the VMCS pointer, CR0, CR3, CR4, GDTR, LDTR, and IDTR. Each subqueue is given a Queue Manager and a VM State Register Set.

When a user allocates a VM, they specify the number of cores to use. Those many cores are then logically bound to the VM. This is done by setting a core register (*MyManager*) with the ID of the Queue Manager in charge of the VM. The relative number of cores bound to each VM determines the relative fraction of the RQ entries assigned to each VM's subqueue. Hence, the sizes of the individual subqueues may dynamically change as new VMs arrive to the server and old VMs are removed.

To allow such flexibility, the physical RQ is broken into chunks, and each VM's subqueue is composed of one or more chunks. When a new VM is spawned on a server, it gets a few chunks from the currently-active VMs. A VM donates one or more chunks from the tail of its subqueue. If some of the entries in those chunk(s) are

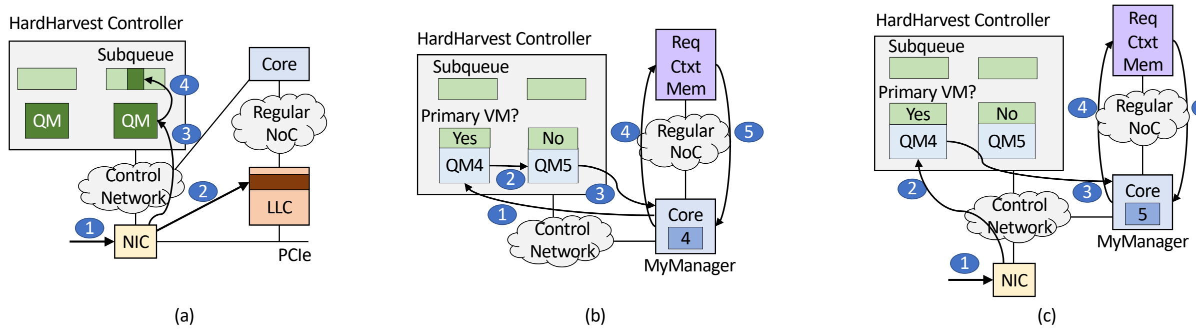

Figure 8: Path events in HardHarvest: (a) request arrival, (b) core re-assignment, and (c) core reclamation.

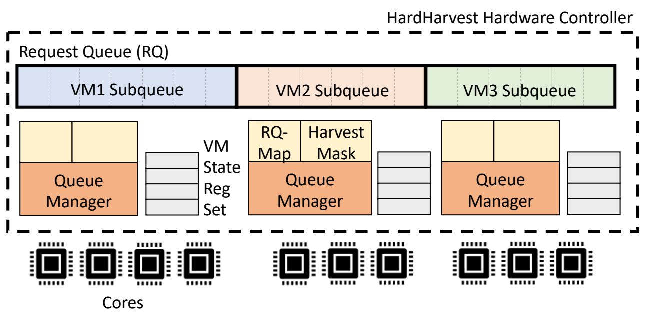

Figure 9: HardHarvest hardware controller.

full, the corresponding entries are moved to a software-based *In-memory Overflow Subqueue* for that VM. We discuss this structure later. Likewise, when a VM leaves the server, its chunks are assigned to the tails of the subqueues of the remaining VMs.

While a subqueue is logically contiguous, its chunks do not need to be physically contiguous. Every Queue Manager has an *RQ-Map* that maps the logical chunks of a VM's subqueue to their physical chunks. Therefore, when a VM's subqueue sheds a chunk, it simply invalidates the entry in its RQ-Map; when a VM's subqueue gets a new chunk, it inserts a new entry at the tail in its RQ-Map. In our implementation, the RQ has 32 chunks of 64 entries each. The total storage of an RQ-Map is 24B, i.e., up to 32 entries with 5 bits for the physical chunk ID and 1 valid bit.

To enable concurrency, each chunk of the RQ has its own access port. Since individual chunks are exclusively owned by a given Queue Manager, different Managers do not contend on such ports. All 32 chunks can be accessed in parallel.

4.1.3 Request Arrival and Processing

Every VM has its own network address. When the NIC receives a packet with a request (Figure 8(a) 1), it deposits the message payload in the LLC via DDIO 2, and reads the message's destination VM. Then, it checks a local software table that tells which Queue Manager (QM) is in charge of which VM. It informs the corresponding QM 3, which stores in its Request Subqueue a pointer to the request payload 4. If the Request Subqueue is full, the QM instead stores the pointer in the In-memory Overflow Subqueue of its VM. In either case, the request is marked as ready.

A core is bound to a QM through the *MyManager* register. Cores have instructions to spin on a Request Subqueue for work, to dequeue a request, and to inform the Request Subqueue when the request has been completed or when it is blocked on I/O. Such instructions access the QM corresponding to the core's *MyManager* register. This QM identifies the Request Subqueue to spin on, the request to dequeue, the request to remove as completed, and the request to mark as blocked, respectively. Moreover, each QM knows if it is managing a Primary or a Harvest VM and, if the former, which of its bound cores are currently "on loan" executing requests of a Harvest VM.

4.1.4 Operation of Core Reassignment

As a core bound to a Primary VM spins on the subqueue of its *MyManager* QM and there is no request to process (Figure 8(b) 1), the QM forwards the core's request to a Harvest VM's QM 2. A Harvest VM runs a batch application and is expected to always have available work. Hence, the Harvest VM's QM sends a process to the requesting core 3, together with the VM State Register Set associated with the QM. On receiving the message, the requesting core may have to save the state of its current process (if it was blocked). It also restores the state of the new Harvest VM process, loads the VM State Register Set of the new VM that it receives from the HardHarvest hardware controller, and proceeds to execute the new process.

To avoid entering the kernel in this case, HardHarvest can use hardware that automatically saves and restores the process register state on a context switch. Specifically, the current state is saved in a special Request Context Memory connected to the on-chip network 4 and the new state is restored from there 5. This hardware simply extends prior proposals for in-hardware context switching across requests [76], to additionally perform a VM context switch.

4.1.5 Reclaiming a Core by a Primary VM

A core is quickly reclaimed by its Primary VM on demand. When a new network packet arrives at the NIC (Figure 8(c) 1) and is either a new request for a Primary VM or a network response to a blocked request of a Primary VM, the NIC informs the corresponding Queue Manager (QM) 2. The QM checks if: (i) none of its bound cores is idle, and (ii) at least one of its bound cores is executing a request for a Harvest VM. If so, it interrupts one such core and passes it the new process and the correct VM State Register Set 3. Note that the interrupt is sent in hardware by the QM of a Primary VM—not by the VM itself. A Primary VM is never aware that another VM was running on one of its bound cores. The interrupted core immediately performs a context switch as described above: it saves the state of the current process 4, loads the new VM State Register Set received from the QM, and restores the state of the new process belonging to the Primary VM (5).

As the Harvest VM loses the core, the process that was running there (i.e., the vCPU) is returned to the queue of the Harvest VM vCPUs. The Harvest VM's QM multiplexes its vCPUs onto its remaining physical cores (pCPUs), similarly to an over-subscribed environment. Since Harvest VMs are configured with as many vCPUs as there are pCPUs in the server [88], the software running in the Harvest VM does not require any changes. However, as the Harvest VM knows the current number of pCPUs available to it, it can adapt dynamically, either by reducing parallelism or rescheduling tasks [88]. Importantly, there is no risk of deadlock from preempted Harvest VM threads holding locks as those threads will eventually run on the remaining pCPUs when they are scheduled again, ensuring forward progress.

Figure 10 summarizes core reclamation. In Figure 10(a), the red core bound to the Primary VM is currently executing request *ID5* of the Harvest VM, and a new request *ID6* of the Primary VM arrives. In Figure 10(b), the core is interrupted and forced to execute *ID6*, leaving *ID5* in a ready state for another core to take.

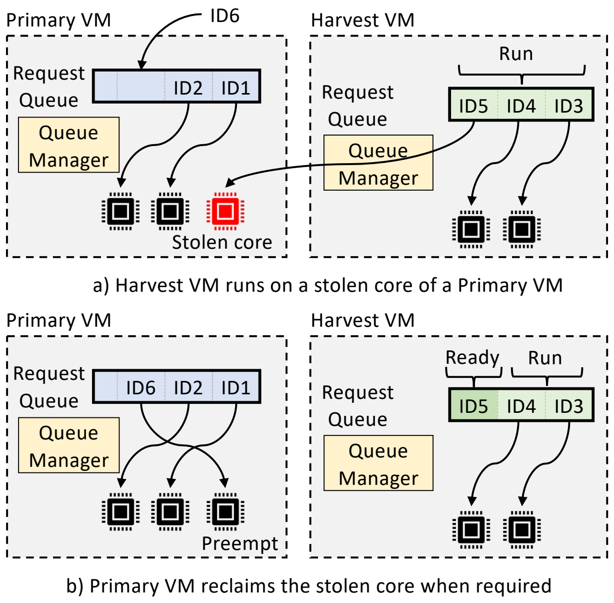

Figure 10: In-hardware core reassignment between VMs.

When a core running a request for a Primary VM blocks on I/O, it may be stolen. However, the pointer to the request is kept in the corresponding Request Subqueue. Later, when the NIC receives the network response to the blocked request, the NIC informs the QM. The QM marks the request in the Request Subqueue as ready and the procedure described above is followed to reclaim the core.

Our HardHarvest design uses FIFO scheduling within a VM. Future work could explore customized scheduling policies, either based on application-provided information, or learned by the system. For example, the application could identify periods when bursts are expected or when the SLO is likely to be violated frequently. In response to this, the system could reduce the harvesting aggressiveness by, for example, keeping a buffer of idle cores ready for Primary VM bursts. Alternatively, the system could monitor events such as when requests spend a very short time blocked on I/O. In this case, the system could dynamically switch from harvesting on blocking call to harvesting only on request completion.

4.1.6 The Need for a Hardware Scheduler

HardHarvest uses a hardware scheduler instead of CPU-centric software scheduling for two reasons. First, a hardware scheduler allows fast core notification when a Primary VM request enters the queue, as the QM checks if the core needed by the VM is on-loan to a Harvest VM and instantly alerts it. In contrast, a software scheduler would require cores to poll memory locations, lowering throughput by diverting core cycles from application logic to checking for new requests. Second, a hardware scheduler minimizes queue contention, eliminating the need for locking mechanisms when multiple cores access the same subqueue, which software scheduling requires.

A hardware scheduler can use memory-mapped queues or hardware queues. Memory-mapped queues are inexpensive and flexible. However, hardware queues have better performance for two reasons. First, dedicating special SRAM queues for incoming requests reduces the contention between (i) the scheduler and (ii) the cores and NIC on the cache hierarchy. Indeed, accesses from the hardware scheduler to the special SRAM queues do not compete with regular cache hierarchy accesses by cores or NIC accesses that deposit requests using DDIO. Second, hardware queues and their network can be designed to have low access latency. Hence, in HardHarvest, we design the RQ (Figure 9) as a dedicated SRAM hardware queue. However, HardHarvest could be easily integrated with memory-mapped queues as well.

4.1.7 Limitations of Prior Hardware Queues for Microservices

Prior work has used hardware queues in microservice environments [76, 96]. However, these designs lack four important supports that HardHarvest provides. First, they assume that all cores can pick requests from the same queue, without any security concerns. In a cloud setup, one must isolate different users. Thus, in HardHarvest, the RQ is split into per-VM subqueues that are managed by different hardware Queue Managers (QMs). These QMs operate in parallel and on distinct subqueues, avoiding any sharing or contention.

Second, prior designs assume non-virtualized environments. Thus, a request's state is only the state of a Linux process. However, cloud workloads run inside sandboxed VMs. Thus, when executing a request, a core needs to load both request and VM contexts. In HardHarvest, each QM keeps the VM state in the VM State Register Set (Figure 9).

Third, prior designs do not have the notion of a Harvest VM process being pre-emptable by a Primary VM process. In HardHarvest, a Harvest VM running on a stolen core may be immediately preempted when a new request for the owner Primary VM is received. The QMs have logic to detect such scenarios and enforce the reassignment across VMs.

Finally, prior proposals are inflexible in that they work with fixed-size queues. Instead, in HardHarvest, per-VM subqueues can dynamically change their size as new VMs are allocated or old VMs depart the server. Further, each subqueue has a software In-memory Overflow Subqueue in main memory that holds requests that overflowed the subqueue.

4.1.8 Implementation Details

The HardHarvest controller of Figure 9 is a centralized hardware module in the chip that is accessed with a dedicated network. We use a special network for two reasons. First, accesses to the controller should not compete with the regular workload traffic. Second, the control network has different needs than the regular network. As it transfers mostly control messages, and it is latency-, not bandwidth-sensitive, it has thin links. In our design, we use a tree topology.

Cores communicate only with the QMs and not with the Request Subqueues. This approach is both secure and avoids data races. The enqueue, dequeue, and other instructions are user-level instructions that are embedded in libraries. These instructions are transparent to the application developers. For example, `CompletionQueue::Next` [29] in gRPC and `TServerSocket::listen` [6] in Thrift are augmented with the HardHarvest dequeue instruction.

The processor chip also includes the special Request Context Memory where the hardware for fast context switch proposed by μManycore [76] saves and restores the core state in a context switch. Such memory is connected to the regular NoC. As saving and restoring is done in hardware, there are no new instructions.

4.2 Cache/TLB Flush/Inval & Cold Restart

4.2.1 TLB and Cache Partitioning

Following existing practice [7, 9, 24, 71, 82, 84, 85, 94], HardHarvest would need to flush and invalidate the L1/L2 caches and L1/L2 TLBs of a core when the core is re-assigned across VMs, to ensure that the structures do not leak information. The LLC does not need to be flushed because it is partitioned using Intel's CAT [34]. To minimize this overhead, HardHarvest uses the fact that microservice invocations have relatively small working sets for both data and instructions [23, 72]. Specifically, HardHarvest proposes to partition the L1 caches (D and I), L2 cache, L1 TLBs (D and I), and L2 TLB in a way that minimizes the overhead for Primary VMs and still allows Harvest VMs to attain good performance.

Each of these structures is way-partitioned into one *Harvest Region* and one *Non-Harvest Region*. When a Primary VM runs, it uses the whole structure; when a Harvest VM runs, it can only use the harvest region—which may be, e.g., 1/2 or 1/3 of the ways of the structure. The non-harvest region is inaccessible to the Harvest VMs, and keeps state of the Primary VM that will be reused when the core is returned to the Primary VM.

When a core transitions from a Primary to a Harvest VM, the harvest region is flushed and invalidated. The Harvest VM is not allowed to start execution until a certain time has elapsed equal to the longest possible duration of the flush/invalidate operation. This is done to eliminate a timing side-channel.

When a core transitions from a Harvest to a Primary VM, again only the harvest region is flushed and invalidated. However, the Primary VM restarts execution *right away* when the core is reclaimed, reusing the data stashed away in the warmed-up non-harvest region. In the background, the harvest region is flushed and invalidated. When the invalidation is completed and the worst-case time has elapsed to ensure there is no timing side-channel, the empty ways also become visible to the Primary VM. Hence, flushing and invalidating the harvest region is not in the critical path.

For a Primary VM, what fraction of the ways in the L1/L2 caches and TLBs are the harvest region can be a default value or specified by the software. This information is saved in a *HarvestMask* hardware register in the Queue Manager of the VM (Figure 9). HarvestMask contains a bit per way for each of the structures, for a total of 5B. A bit is set if the way is in the harvest region. When a core is assigned (or re-assigned) to a VM, the system knows whether the VM is Primary or Harvest, and obtains the VM's HarvestMask. Then, before the core starts executing, the HarvestMask is used to reconfigure the private caches/TLBs in a way similar to CAT [34]. For example, if the core is executing a Harvest VM, the non-harvest ways of the L1/L2 caches and TLBs are inaccessible to the requests. Coherence messages such as invalidations are still received for data in either the harvest or the non-harvest ways, since data is not remapped. To improve performance, HardHarvest could profile a workload and recommend an initial non-harvest region size. Then, the system could transparently learn during execution the best non-harvest region size by opportunistically trying to change it and seeing the performance impact.

A Harvest VM such an ML training job could make use of more space than the harvest region partition. However, HardHarvest is still attractive to these workloads because, while the harvested cores have reduced cache capacity, renting them has a lower price—thus improving cost-efficiency for latency-tolerant applications.

4.2.2 Improving Cache Allocation for Primary VMs

To improve the performance of Primary VMs, HardHarvest tries to keep in the non-harvest region the state that is most likely to be reused when a core is reclaimed back by a Primary VM. To understand what this state is, we consider two types of pages: those that are *shared* across different invocations of the same service, and those that are *private* to a particular invocation of the service. Shared pages include program code, libraries, read-only input data and, generally, data pages that are allocated in a microservice before forking a process to execute a particular invocation of the microservice. Private pages are those allocated after forking a process for a particular microservice invocation. Generally, shared pages are more likely to be reused across core reassignments. So, HardHarvest tries to keep entries from shared pages in the non-harvest region.

We can assume that all instructions and all data objects allocated by the process that initializes a microservice are potentially shared. For example, in microservices implemented using the Thrift [78] or gRPC [28] frameworks, HardHarvest assumes that all data allocated from the start of the microservice until executing `server.serve()`[17, 19] is shared data. If the shared data gets reallocated to expand its size, the new pages are also assumed shared. On the other hand, data allocated by the threads in individual invocations [18] is assumed to be private. Our profiling of more than 60 microservices from open-source DeathStarBench [23], TrainTicket [97], and μSuite [73] benchmark suites confirms this behavior.

Based on these assumptions, when a page is allocated by a Primary VM, HardHarvest sets a *Shared* bit to 1 or 0 in its page table entry. Such bit is copied to the TLB entries, and determines, on an access, the *preferred* ways where a TLB entry or a cache line is placed.

4.2.3 Cache/TLB Replacement Algorithm

HardHarvest changes the algorithm that picks the victim way when inserting an entry in private caches or TLBs. The aim is to steer shared entries to the non-harvest region (*Non-Harv*) ways and private entries to the harvest region (*Harv*) ways, while keeping the algorithm simple. Algorithm 1 shows which way to pick when inserting entry E in Set S. In the cases when the algorithm can pick one of multiple victim candidates, it picks the victim using the default replacement algorithm, which is LRU.

Algorithm 1: Inserting an entry in a private cache or TLB.

```
Inputs: E = entry to insert; S = set where E maps
Result: Cache/TLB way that takes E
if S has empty slots then
    if empty slots in both Non-Harv and Harv then
        if E is shared then
            Take an empty slot in Non-Harv
        else // E is private
            Take an empty slot in Harv
    else
        Take an empty slot
else
    if E is shared then
        if any Non-Harv slot has a private entry then
            Take the slot of one of them
        else
            if any Harv slot has a private entry then
                Take the slot of one of them
            else // All S slots have shared entries
                Take a slot
    else // E is private
        if any Harv slot has a private entry then
            Take the slot of one of them
        else
            if any Non-Harv slot has a private entry then
                Take the slot of one of them
            else // All S slots have shared entries
                Take a slot
```

First, assume that there are empty slots in S. If there are both *Non-Harv* and *Harv* empty slots, a shared entry takes a *Non-Harv* empty slot and a private entry takes a *Harv* empty slot; otherwise, E takes an empty slot. If, instead, there is no empty slot in S, the action depends on whether E is shared or private. If E is shared, the algorithm proceeds as follows. First, it checks *Non-Harv* for any slots with a private entry. If there are any, E evicts one of them. Otherwise, *Harv* is checked for any slots with a private entry. If there are any, E evicts one of them. Otherwise, all the slots in S have shared entries, and the algorithm picks one of them as the victim.

Instead, if E is private, the algorithm swaps steps one and two above. First, it checks *Harv* for any slots with a private entry. If there are any, E evicts one of them. Otherwise, the algorithm checks *Non-Harv* for any slots with a private entry. If there are any, E evicts one of them. Otherwise, all the slots in S have shared entries, and the algorithm picks one of them as the victim. The algorithm swaps steps when E is private because a private entry in *Non-Harv* will not remain cached for too long. A shared entry may soon evict it.

Since *Shared=1* for all instruction pages, this algorithm does not change the default behavior of the L1 instruction cache/TLB. The L2 cache/TLB have instructions and data. There, the algorithm prioritizes instructions over private data. In practice, both data and instructions in microservices have small working sets [23, 72]. Thus, assigning about half of the ways to *Harv* during harvesting is typically tolerable.

Continuously prioritizing shared entries within a set can penalize the performance of private entries. For example, if a set at some point has only shared entries, from that point on, all private entries will compete for a single way, making the set appear as direct-mapped. To avoid this issue, HardHarvest uses Algorithm 1 to pick an eviction victim only among the M least-recently used entries in the set. These entries are called *Eviction Candidates*, and can be, e.g., 1/2 or 3/4 of all the entries of the set. The other, more-recently-used entries are not considered. In this way, popular private data avoids eviction, allowing services that have large private memory footprints to operate with high performance.

With this design, HardHarvest makes a best effort to keep shared entries in the *Non-Harv* ways without fully sacrificing the associativity for private entries.

4.2.4 Hardware Implementation of the Replacement Algorithm

We implement Algorithm 1 in a simple manner. Specifically, each TLB/cache entry has a *Shared* bit that is set to 1 if the entry has Shared state. Also, each way has a *Harvest* bit, which is set to 1 if the way is a harvest way. These bits, together with the *Invalid* bit of an entry, are used as inputs to two priority multiplexers that determine the victim entry that should be replaced. One of the multiplexers is used for incoming shared entries and the other for private ones.

The multiplexer for incoming shared entries selects the victim entry based on this decreasing priority: Invalid and Not Harvest; Invalid; Not Harvest and Not Shared; Harvest and Not Shared. The multiplexer for incoming private entries uses this decreasing priority: Invalid and Harvest; Invalid; Harvest and Not Shared; Not Harvest and Not Shared.

4.2.5 Other Issues

Current processors flush and invalidate caches inefficiently (Section 3). Hence, for HardHarvest's mechanisms to be effective, they must be coupled with support for efficient flush/invalidate as proposed elsewhere [30, 51]. In our evaluation, we will assume such support and evaluate its contribution to performance separately.

5 Experimental Setup

Modeled Architectures. We model a cluster of 8 servers, where each server has 36 beefy cores and 128GB of memory. Cores and caches are modeled after the Intel Sunny Cove microarchitecture [13, 14, 89] present in the IceLake server processors [35]. Each core has private L1 and L2 caches and TLBs, and a shared, physically distributed L3 cache. Each server has 8 Primary VMs, each with 4 cores, and 1 Harvest VM, which starts with 4 cores and harvests additional cores from Primary VMs. Detailed architecture parameters are shown in Table 1. We evaluate five systems:

• *NoHarvest* is a conventional system where no VM performs core harvesting. As a result, many cores remain idle.

• *Harvest-Term* is a state-of-the-art *software* core harvesting system as described in SmartHarvest [88]. In here, a Harvest VM harvests cores only when the core is idle because it terminated a request, and based on load prediction. We use *Harvest-Term* as the baseline. We also model a more aggressive software design where, in addition, a

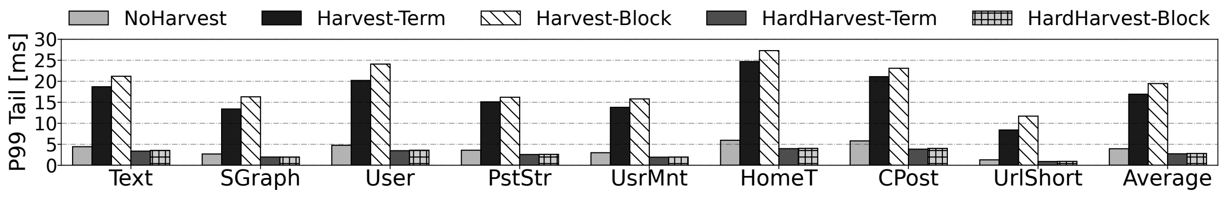

Figure 11: P99 tail latency of microservices running in Primary VMs for the 5 evaluated architectures (lower is better).

Table 1: Architectural parameters used in the evaluation.

System and Processor Parameters	
Machine	Cluster of 8 servers
Server processor	36 6-issue cores at 3GHz
OoO Execution	352-entry ROB, 200-entry LSQ
L1 D-Cache	48KB, 12-way, 5 cyc. round trip (RT), 64B line
L1 I-Cache	32KB, 8-way, 5 cyc. RT, 64B line
L2 Cache	512KB, 8-way, 13 cycles RT, 32 MSHRs
L3 Cache	Per core: 2MB, 16-way, 36 cyc. RT, 32 MSHRs
L1 TLB	128 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
Network	
Intra Server	2D mesh, 5 cycles/hop
Inter Server	$1\mu s$ RT; 200GB/s
Virtual Machines	
Primary VMs	8 VMs/server, each with 4 cores (fixed)
Harvest VMs	1 VM/server, with 4 cores + harvested cores
Main Memory per Server	
Capacity; Rate	128GB; DDR4-3200; 4 memory controllers
Mem. Bandwidth	102.4GB/s per socket
HardHarvest Parameters	
Num chunks in RQ	32
Num entries/chunk	64
Num Queue Manag.	16
Num regs in VM State Regs	16
Ways in Harv. Region	50% of all ways
Eviction Candidates (M)	75% of all ways
Flus+Inv HarvRegion	1000 cycles

Harvest VM also harvests cores when a request issues a blocking call and the core it was running on becomes idle (*Harvest-Block*).
• *HardHarvest* is the design proposed in this paper. We consider two versions: one where a Harvest VM harvests idle cores only when they are idle because a service request terminates (*HardHarvest-Term*), and one where, in addition, a Harvest VM also harvests cores when they are idle because a request issues a blocking call and the core becomes idle (*HardHarvest-Block*). The latter is our proposal.

All schemes use Intel's DDIO technology. In the baseline schemes (*NoHarvest* and *Harvest-Term/Block*), the NIC deposits the whole request in the LLC.

Simulation Infrastructure. We evaluate the architectures with full-system simulations using QEMU [70] and SST [67]. QEMU captures both user-space and kernel-space instructions, memory accesses, and system calls. QEMU forwards all the events to SST, which models the architectures and performs cycle-level simulations. Thus, the simulation models the whole software stack: OS (Ubuntu 20.04), hypervisor, harvesting agents, and application logic. The main memory system is modeled with DRAM-Sim2 [68].

Our modeled cluster is comprised of 8 servers to test a different type of Harvest VM workload in each server, as we discuss later. Such servers execute without requiring any communication between them. This is because microservices do not communicate across servers. Specifically, each server hosts an instance of each of the evaluated microservices, but microservices only communicate with other microservices that are placed on the same server. We use the $1\mu s$ inter-server communication latency to model the network latency when a service accesses remote caches (e.g., Memcached), key-value stores (e.g., Redis), or databases (e.g., MongoDB). These backend services (Memcached, Redis, and MongoDB) run on dedicated servers. We do not simulate the execution of the queries on the backend services. Instead, we use the execution times obtained by profiling them on a real server.

To make the simulation time tolerable, we run the simulations in parallel. Each of the 8 servers is simulated on a different physical machine because servers do not communicate with each other. In addition, within each simulated server, SST also runs in parallel. With this design, the longest simulation (corresponding to 30 seconds of wall-clock time) takes 4 days.

Applications. For the latency-critical Primary VMs, we use 8 SocialNet microservices from DeathStarBench [23]. For the batch workloads executed in the 8 Harvest VMs, we use graph applications from GraphBIG (BFS, CC, DC, and PRank) [60], ML training from FunctionBench (LRTrain and RndFTrain) [41], data analytics from CloudSuite (Hadoop) [64] and bioinformatics from BioBench (MUMmer) [1]. We deploy each Primary VM with 4 cores because this is the most common size for Alibaba's microservice instances [87]. Each Harvest VM also starts with 4 cores. As indicated before, each server has one Harvest VM running one of the batch applications, and 8 Primary VMs, each running one of the microservices.

We pick 8 representative services from Alibaba's production-level open source traces [48] and we mimic their behavior with our 8 DeathStarBench services. Hence, *we execute with real-world invocation rates*, using an open-loop load generator that keeps the load the same across all systems (i.e., the client is independent of the server) [73]. The average load per Primary VM core is 65-250 requests per second (RPS). We report average and tail latency after executing 100K microservice invocations across all 64 Primary VMs. As done in prior work [39, 81], we match a service in the production trace to the service in the DeathStarBench suite that has the most similar service execution time.

6 Evaluation

6.1 End-to-End Tail Latency of Primary VMs

Figure 11 shows the P99 tail latency of microservices running in Primary VMs for the 5 evaluated architectures. Software harvesting schemes (*Harvest-Term* and *Harvest-Block*) have a high tail latency due to software overheads. The average tail latency in *Harvest-Term*

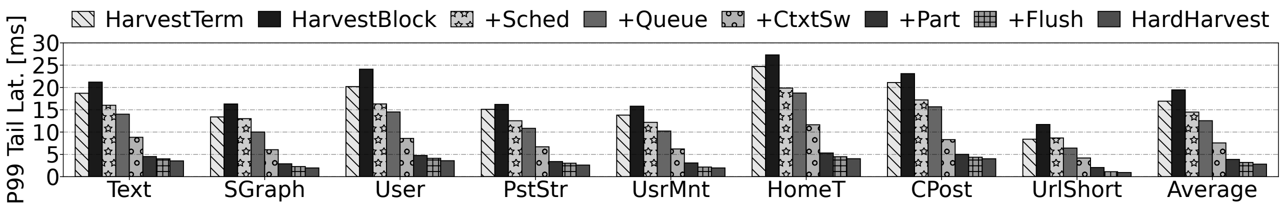

Figure 12: Cumulative impact of individual optimizations in *HardHarvest* on the P99 tail latency of Primary VMs. Core harvesting is enabled.

and *Harvest-Block* is 3.4× and 4.1× higher, respectively, than in *NoHarvest*. On the other hand, *HardHarvest* reduces the tail latency substantially. Compared to *Harvest-Term*, *HardHarvest-Term* and *HardHarvest-Block* reduce the tail latency by 83.3%. In fact, the tail latency in these architectures is even lower than in *NoHarvest*: the average tail latency in *HardHarvest-Term* and *HardHarvest-Block* is 30.5% and 28.4% lower, respectively, than in *NoHarvest*. The reason is that some *HardHarvest* optimizations such as improved cache/TLB replacement and request queuing in hardware not only speed-up harvesting, but also microservices in general as well. The reductions relative to *Harvest-Term* and *Harvest-Block* are more significant in services that *(i)* operate mostly on shared pages such as *HomeT*, or *(ii)* frequently block on I/O such as *User*.

6.2 Tail Latency Reduction Breakdown

Figure 12 shows the *cumulative* impact of individual optimizations on the tail latency of Primary VMs. The figure starts with the tail latency of software harvesting with *Harvest-Term* and *Harvest-Block*. It then applies the following optimizations to *Harvest-Block* one by one, in order: hardware request scheduler (*+Sched*), hardware request queues (*+Queue*), in-hardware context switching (*+CtxtSw*), cache/TLB partitioning with LRU replacement (*+Part*), efficient cache/TLB flushing (*+Flush*), and optimized replacement policy (*HardHarvest*). Recall that we borrow *CxtSw* [76] and *Flush* [30, 51] from the literature, as they are needed to take full advantage of our optimizations.

All techniques help reduce tail latency. On average, the gradual application of these optimizations reduces the tail latency of *Harvest-Block* by 25.6%, 35.5%, 61.1%, 80.1%, 83.6%, and 85.6%, respectively. In-hardware request scheduling (*+Sched*) is effective because, e.g., when a service that is blocked on I/O receives the response, the scheduler ensures that it is scheduled right away. Without the scheduler, a polling core would discover the ready service much later, which hurts tail latency. Hardware queuing (*+Queue*) is effective because it reduces contention on the cache hierarchy relative to memory-mapped queues, and also reduces the latency of request fetching. Cache/TLB partitioning (*+Part*) also helps, even with LRU replacement and without advanced hardware flushing. Advanced hardware flushing (*+Flush*) has a small impact because, after a Primary VM resumes, flushing occurs in the background while the Primary VM is already running. Finally, our proposed cache line replacement (*HardHarvest*) further reduces the tail.

To see the relative impact of *Sched* and *CxtSw*, we perform an ablation study in Figure 13. The figure takes *Harvest-Block* and applies only *CtxtSw*, then only *Sched*, and then both *CtxtSw* and

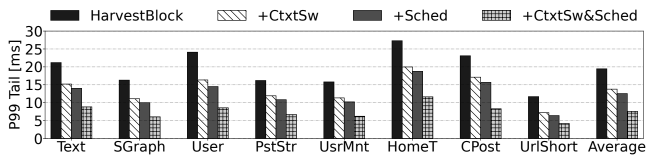

Figure 13: Ablation study on the effectiveness of in-hardware context switching and hardware request scheduling.

Sched. We see that both *Sched* and *CxtSw* have a similar impact, and when applied together, they have a partially additive effect.

6.3 Impact of the Optimized Cache Replacement Policy

To understand the impact of the HardHarvest cache replacement policy, Figure 14 shows the measured L2 cache hit rate in four different environments: vanilla LRU, the RRIP advanced replacement [37], our proposed policy (Algorithm 1), and an ideal cache replacement policy (Belady [36]). We see that, on average, our algorithm (*HardHarvest*) increases the L2 cache hit rate over LRU and RRIP by 11.3% and 8.2%, respectively. Since RRIP does not differentiate between Primary and Harvest processes accessing the same cache, its re-reference interval calculations get polluted, leading to sub-optimal performance. *HardHarvest* is within 3.1% of the ideal replacement algorithm. Results are similar for L1 and TLBs.

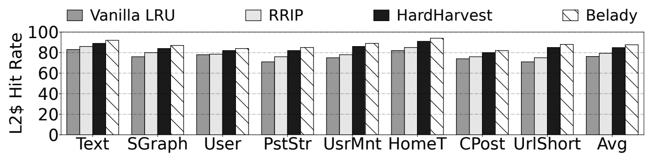

Figure 14: L2 hit rate with different replacement policies.

Since shared pages across invocations of the same service include both code and data, we investigate whether prioritizing instruction over data pages in the replacement algorithms of caches and TLBs improves performance. We performed this experiment in *HardHarvest* via Code-Data-Prioritization (CDP) [61] in Intel's CAT. We find that such an approach is not beneficial. It increases the tail latency by 8% over our proposed *HardHarvest* replacement policy.

6.4 HardHarvest Optimizations without Core Harvesting

Figure 15 shows the tail latency of Primary VMs as we add HardHarvest optimizations to the NoHarvest baseline without performing core harvesting. We evaluate *+Sched*, *+Queue*, *+CtxtSw*, and *+ReplPolicy* (which is our optimized replacement policy). Since there is no harvesting, cache partitioning and cache flushing are not relevant and not evaluated. We see that all four techniques are effective: they cumulatively reduce the tail latency by 14.5%, 20.1%, 28.6% and 33.6%, respectively. In-hardware request scheduling (*+Sched*) is effective because, when a service that is blocked on I/O receives the response, the scheduler ensures that it is scheduled right away. The scheduler offloads CPU polling. Hardware queues (*+Queue*) reduce contention in the cache hierarchy and the latency to fetch the request. Finally, our replacement policy (*+ReplPolicy*) helps by preserving shared entries in the caches/TLBs across invocations.

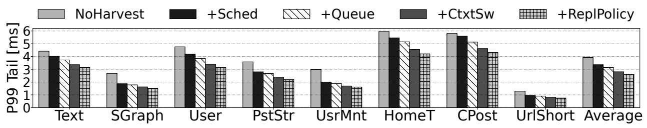

Figure 15: Cumulative impact of optimizations on the P99 tail latency of Primary VMs. Core harvesting is disabled.

6.5 Median Latency of Primary VMs

Figure 16 shows the median latency of microservices in the five evaluated architectures. Although we saw that software harvesting significantly degrades tail latency, it has a modest impact on the median latency. The median latency of *Harvest-Term* is only 7.9% higher than *NoHarvest*. On the other hand, *HardHarvest* not only reduces tail latency but is also effective at reducing the median latency as well: *HardHarvest-Block* reduces the median latency by 26.1% over *NoHarvest*.

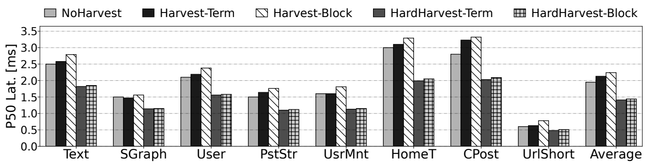

Figure 16: Median latency of microservices in Primary VMs.

6.6 Throughput of Harvest VMs

The target metric for Harvest VMs is throughput (i.e., the number of jobs executed per unit of time). Figure 17 shows the throughput of Harvest VMs with the evaluated architectures normalized to *NoHarvest*. On average, *Harvest-Term* [88] and *HardHarvest-Block* (our proposal) improve throughput by 1.7× and 3.1×, respectively. Memory-intensive applications, e.g., RndFTrain, see slightly lower throughput gains. *HardHarvest-Block* improves the throughput over *Harvest-Term* because it (i) steals cores whose service is blocked on I/O (i.e., it harvests more cores), and (ii) reduces the overheads of core reassignment (i.e., Harvest VMs start running on stolen cores sooner).

6.7 Core Utilization

HardHarvest benefits cloud providers as it increases the utilization of the cores. It can be shown that *NoHarvest*, *Harvest-Term*,

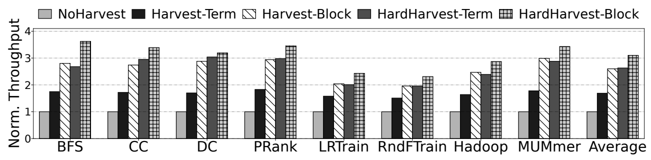

Figure 17: Throughput of Harvest VMs with the five evaluated architectures normalized to *NoHarvest*.

Harvest-Block, *HardHarvest-Term*, and *HardHarvest-Block* have an average utilization of 10.3, 23.8, 26.5, 28.7, and 34.8 cores out of the available 36 cores in the server, respectively. *HardHarvest* has higher core utilization because it performs core harvesting in hardware, using cores efficiently and eliminating emergency buffers. Overall, *HardHarvest-Block* increases core utilization by 1.5× over *Harvest-Term*.

6.8 Storage Cost

HardHarvest adds the hardware controller in Figure 9 to each server. The storage cost of a controller is a 2K-entry RQ, where each entry has 66 bits (2 bits for the request status and 64 bits for a pointer to the request payload) and, for each of the 16 pairs of QMs and VM State Register Sets: 1) 16 VM State registers of 8B each, 2) a 24B RQ-Map, and 3) a 5B HarvestMask register. The total storage cost per controller is 18.9KB (or 0.53KB per core). On top of that, each entry in the TLBs, L1 D-caches, and L2 caches has an extra *Shared* bit, which results in a total storage cost per server of 67.8KB (or 1.9KB per core). We use McPAT [46] to estimate the power and area overheads of these storage structures. Scaling to 7nm technology [74], the resulting overheads are only 0.19% and 0.16% increases in area and power (dynamic plus static), respectively, of the multicore.

6.9 Sensitivity to LLC Size

Throughout the evaluation, we used an LLC with 2MB per core. In this section, we perform a sensitivity study to see the impact of different LLC sizes on the effectiveness of HardHarvest. Note that the LLC is non-inclusive of the L2. Figure 18 shows the P99 tail latency of microservices running Primary VMs in HardHarvest-Block with different LLC sizes. When we increase the LLC size to 2.5MB per core, the tail latency reduces because there are fewer misses, while when we decrease the LLC size, the tail latency increases. Overall, the changes in latency are small because microservices have relatively modest footprints.

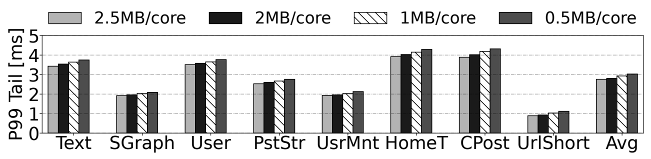

Figure 18: P99 tail latency of microservices running in Primary VMs with HardHarvest-Block and different LLC sizes.

6.10 Sensitivity to Eviction Candidate Set Size

Throughout the evaluation, we set the eviction candidate set to be 75% of all ways in a set. In this section, we perform a sensitivity study to see the impact of different sizes of the eviction candidate set on the effectiveness of HardHarvest. Figure 19 shows the P99 tail latency of microservices running Primary VMs in HardHarvest-Block with different sizes of the eviction candidate set. We see that, when we decrease the eviction candidate set size (to *25%* and *50%*), the tail latency increases because the algorithm is unable to preserve some shared lines. On the other hand, when we increase the eviction candidate set size to *100%*, the tail latency again increases because the algorithm keeps evicting needed private cache lines.

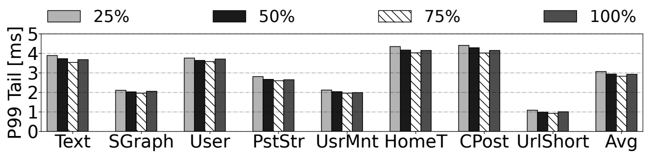

Figure 19: P99 tail latency of microservices in HardHarvest with different sizes of the eviction candidate set.

7 Related Work

Resource Harvesting. Currently, spare cloud resources are harvested via software techniques [5, 49, 69, 77, 88, 90, 92, 93]. To minimize the interference on latency-critical tasks, co-located workloads can be isolated via cache partitioning and power control [47], or memory bandwidth control [50]. We show that software-only harvesting techniques introduce overheads that are not tolerable for microservices workloads. HardHarvest proposes a hardware solution for core harvesting with much lower overheads.

Software for Scheduling and Context Switching. A large body of work proposed software solutions for fast scheduling and context switching [8, 21, 31, 38, 49, 52, 53, 63, 65, 91, 98]. ZygOS [65] allows cores to steal requests from other cores for load balance. Shenango [63] dedicates a core for scheduling. In cloud environments, on every cross-VM context switch, these systems perform expensive cache flushes and core reassignments.

Hardware Support for Microservices. Researchers have proposed hardware support for microservices [33, 45, 57–59, 75, 76, 96]. RPCValet [15] uses the on-chip NICs to perform in-hardware load balancing. μManycore [76] proposes a manycore architecture specialized for microservices. Duplexity [59] re-configures between latency-critical and batch modes of execution. These schemes do not consider core harvesting and could be combined with HardHarvest. Hyperplane [57] proposes a hardware solution to avoid fruitless core spinning on empty queues. However, while the core is busy, Hyperplane does not detect the arrival of higher priority requests and does not notify/interrupt the core. Thus, it cannot be directly used for core harvesting.

8 Conclusion

This paper proposed *HardHarvest*, the first architecture for core harvesting in hardware. HardHarvest eliminates software-based overheads by using hardware request queues to speed-up core reassignment, and by partitioning private caches/TLBs while using a smart replacement algorithm. Compared to state-of-the-art core harvesting, HardHarvest increased core utilization by 1.5×, increased Harvest VM throughput by 1.8×, and reduced Primary VM tail latency by 6.0×.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007, CCF 2107470, CNS 2211381, and CCF 2316233; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. 2005. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '05)*.
[2] Amazon AWS. 2024. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.
[3] Amazon AWS. 2024. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot.
[4] Amazon AWS. 2024. AWS Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstableperformance-instances.html.
[5] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.
[6] Apache Thrift. 2024. TServerSocket: Listen(). https://github.com/apache/thrift/blob/1252cf3a2f3b1d942c8c4713ed7b2cf35c64e547/lib/cpp/src/thrift/transport/TServerSocket.cpp#L381.
[7] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. 2017. Side-channels beyond the cloud edge: New isolation threats and solutions. In *Proceedings of the 1st Cyber Security in Networking Conference (CSNet '17)*.
[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.
[9] Benjamin A. Braun, Suman Jana, and Dan Boneh. 2015. Robust and Efficient Elimination of Cache and Timing Side Channels. *CoRR* abs/1506.00189 (2015). arXiv:1506.00189 http://arxiv.org/abs/1506.00189
[10] Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*.
[11] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. 2018. Taming the Killer Microsecond. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*.
[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.
[13] Ian Cutress. 2024. Examinining Intel's Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3.
[14] Ian Cutress. 2024. The Ice Lake Benchmark Preview: Inside Intel's 10nm. https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2.
[15] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.
[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80.
[17] DeathStarBench. 2024. RateService Initialization. https://github.com/delimitrou/DeathStarBench/blob/master/hotelReservation/services/rate/server.go#L81.
[18] DeathStarBench. 2024. TextService Handler. https://github.com/delimitrou/DeathStarBench/blob/master/socialNetwork/src/TextService/TextHandler.h#L41.
[19] DeathStarBench. 2024. TextService Initialization. https://github.com/delimitrou/DeathStarBench/blob/master/socialNetwork/src/TextService/TextService.cpp#L59.
[20] Felix Cloutier. 2024. WBINVD — Write Back and Invalidate Cache. https://www.felixcloutier.com/x86/wbinvd.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[22] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-Harvesting VMs in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[24] Michael Godfrey and Mohammad Zulkernine. 2013. A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud. In *Proceedings of the IEEE Sixth International Conference on Cloud Computing (CLOUD '13)*.

[25] Google Cloud. 2024. Machine families resource and comparison guide. https://cloud.google.com/compute/docs/machine-resource.

[26] Google Cloud. 2024. Spot Virtual Machines. https://cloud.google.com/spot-vms.

[27] Google Cloud. 2024. What is Microservices Architecture? https://cloud.google.com/learn/what-is-microservices-architecture.

[28] gRPC. 2024. An RPC library and framework. https://github.com/grpc/grpc.

[29] gRPC. 2024. Completion Queue: Next(). https://grpc.github.io/grpc/cpp/classgrpc_1_1_completion_queue.html#a86d9810ced694e50f7987ac90b9f8c1a.

[30] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data Speculation Support for a Chip Multiprocessor. In *Conference on Architectural Support for Programming Languages and Operating Systems*.

[31] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

[32] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*.

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*.

[34] Intel. 2016. Intel® CAT: Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[35] Intel. 2024. Intel Xeon Platinum 8380 Processor. https://ark.intel.com/content/www/us/en/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html.

[36] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*.

[37] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*.

[38] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Maziéres, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[39] Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, and Nectarios Koziris. 2024. FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*.

[40] Kernel Virtual Machine. 2017. *https://www.linux-kvm.org*.

[41] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD'19)*.

[42] KVM Hypervisor. 2024. KVM Lock Overview. https://docs.kernel.org/virt/kvm/locking.html.

[43] KVM Hypervisor. 2024. KVM/Linux Kernel Scheduler. https://elixir.bootlin.com/linux/v4.14/source/kernel/sched/core.c#L479.

[44] KVM Hypervisor. 2024. Linux KVM Hypercalls. https://www.kernel.org/doc/html/v5.9/virt/kvm/hypercalls.html.

[45] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*.

[46] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*.

[47] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

[49] Zhihong Luo, Silvery Fu, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2023. Out of Hand for Hardware? Within Reach for Software!. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*.

[50] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '11)*.

[51] Jose Martinez, Jose Renau, Michael Huang, Milos Prvulovic, and Josep Torrellas. 2002. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *International Symposium on Microarchitecture (MICRO)*.

[52] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*.

[53] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*.

[54] Microsoft Azure. 2024. Azure Burstable VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable.

[55] Microsoft Azure. 2024. Use Azure Spot Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/spot-vms.

[56] Microsoft Azure. 2024. Virtual Machine series. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.

[57] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. 2020. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*.

[58] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2019. Express-Lane Scheduling and Multithreading to Minimize the Tail Latency of Microservices. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC '19)*.

[59] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. 2019. Enhancing Server Efficiency in the Face of Killer Microseconds. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*.

[60] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*.

[61] Khang Nguyen. 2016. Code and Data Prioritization - Introduction and Usage Models in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-code-and-data-prioritization-with-usage-models.html.

[62] Old GigaOm. 2011. The Biggest Thing Amazon Got Right: The Platform. https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

[63] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[64] Tapti Palit, Yongming Shen, and Michael Ferdman. 2016. Demystifying Cloud Benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '16)*.

[65] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[66] Chris Richardson. 2024. What are microservices? https://microservices.io/.

[67] Arun F. Rodrigues, K. Scott Hemmert, Brian W. Barrett, Chad D. Kersey, Ron A. Oldfield, M. Weston, R. Risen, J. Cook, Paul Rosenfeld, E. Cooper-Balis, and

Bruce L. Jacob. 2011. The structural simulation toolkit. *SIGMETRICS Performance Evaluation Reviews* 38, 4 (2011).

[68] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011).

[69] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. 2023. UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*.

[70] Software Freedom Conservancy. 2024. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[71] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and Roy Campbell. 2018. Scheduling, Isolation, and Cache Allocation: A Side-Channel Defense. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E '18)*.

[72] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA '19)*.

[73] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization (IISWC '18)*.

[74] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration the VLSI journal* (2017).

[75] Jovan Stojkovic, Esha Choukse, Enrique Saurez, Íñigo Goiri, and Josep Torrellas. 2024. Mosaic: Harnessing the Micro-Architectural Resources of Servers in Serverless Environments. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*.

[76] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. 2023. μManycore: A Cloud-Native CPU for Tail at Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*.

[77] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

[78] The Apache Software Foundation. 2024. Apache Thrift. https://thrift.apache.org/.

[79] Think Software. 2021. Microservices Architecture of Twitter Service. https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca.

[80] Uber. 2020. Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/blog/microservice-architecture/.

[81] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. 2023. Enabling In-Vitro Serverless Systems Research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*.

[82] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*.

[83] Ketan Varshneya. 2021. Understanding design of microservices architecture at Netflix. https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/.

[84] Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. 2024. Principled Microarchitectural Isolation on Cloud CPUs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[85] Stavros Volos and Boris Kopf. 2024. Preventing side-channels in the cloud. https://www.microsoft.com/en-us/research/blog/preventing-side-channels-in-the-cloud/.

[86] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. 2020. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*.

[87] Kangjin Wang, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou Hou, Jie Yao, Liping Zhang Zhang, and Ying Li Li. 2022. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *51st International Conference on Parallel Processing (ICPP '22)*.

[88] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*.

[89] Wikipedia. 2024. Sunny Cove. https://en.wikipedia.org/wiki/Sunny_Cove_(microarchitecture).

[90] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating Serverless Computing by Harvesting Idle Resources. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*.

[91] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

[92] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*.

[93] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[94] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*.

[95] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *USENIX Annual Technical Conference (USENIX ATC '22)*.

[96] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, and Natalie Enright Jerger. 2022. ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*.

[97] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*.

[98] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.