MEUNIK: Rethinking Virtual Machine Memory Resource Management for Unikernel-based VMs

Yongshu Bai[§]
Zhejiang Lab
Hangzhou, Zhejiang, China

Xin Zhang
School of Computing
Binghamton University
Binghamton, New York, USA

Yifan Zhang
School of Computing
Binghamton University
Binghamton, New York, USA

***Abstract*—In this paper, we investigate the problem of achieving efficient memory resource management for unikernel-based virtual machines (uVMs), where unikernels are running as the operating systems of the VMs. Through extensive experiments, we first demonstrate that existing VM memory management mechanisms are unsuitable for uVMs. Then, we propose MEUNIK, a system that aims to achieve high memory management efficiency or uVMs. The four key mechanisms of MEUNIK are designed to address the problems we find in existing solutions. We have implemented a prototype MEUNIK system based on the Xen hypervisor and performed extensive evaluation experiments on three groups of uVMs, which are constructed based on a variety of programs and applications, including fifteen small benchmark programs, four complex server applications, and two network-operation-heavy programs. The evaluation results show that our system achieves the design goals with minimal overhead.**

I. INTRODUCTION

Unikernels are a type of application-specific OSes which are constructed following the philosophy of library OS [1]–[3]. Each unikernel can be considered as a single-application OS where only the OS kernel functionalities needed by the application are compiled with the application code into the OS image. As a result, unikernels are small in image size and are extremely lightweight in OS bootstrapping and runtime performances [4]–[9].

Unikernels are highly promising for computing scenarios where the workload is simple, highly parallel, and demands a high level of security isolation. The potential of unikernels can be seen in the following two examples:

- In the emerging serverless computing paradigm [10]–[14], solutions that use conventional OSes often fall short when handling serverless workloads due to the complex nature of OS kernels [15]. Unikernels, on the other hand, are a more suitable solution because they offer two key advantages: short start-up and deconstruction times and ease of management and scaling, both due to their extremely lightweight nature [9], [15], [16].
- In multi-tenant edge (MTE), users run their workloads on edge computing nodes which can be embedded edge devices with limited computing resources [17]–[23]. VMs running conventional OSes cannot scale well in MTE due to the resource constraints on embedded edge devices and the multiplexing overhead of the OSes. In comparison, VMs running unikernels suit edge devices better because of their low demand for computing resources.

Due to the single-application nature, unikernels are typically deployed multiple per physical computing node. As a result, unikernels can be categorized as either VM-based [4]–[9], [24]–[33] or process-based [3], [34], [35]. VM-based unikernels run as guest OSes in individual VMs created by the underlying hypervisor, such as Xen [36] or KVM [37], [38]. Process-based unikernels run in special sandboxed processes, between which communication and interference are significantly limited. VM-based unikernels have received more attention in the literature due to their advantages of better security and scalability, which are enabled by the use of existing hypervisor technologies.

In this work, we first conduct experiments to study the suitability of existing VM memory resource management mechanisms for VMs that run unikernel as the guest OS, or *unikernel-based VMs* (uVMs). We make the following four observations, which suggest that existing solutions are not suitable for uVMs: (1) Existing hypervisors require a predetermined amount of memory to be configured to create VMs. We show that this requirement is not suitable for uVMs, which typically exhibit high memory usage variance. (2) Existing mechanisms for transferring memory between VMs do not work well for uVMs. (3) Existing cross-VM memory-sharing mechanisms are slow and costly for uVMs. (4) Existing VM working set estimation and idle memory reclamation mechanism incurs high overhead for uVM workload. The details of these observations are discussed in Section III.

To address the above problems, we propose MEUNIK, a framework that provides hypervisors with a means of managing the memory of uVMs efficiently. The primary objective of MEUNIK is to optimize memory availability within the system while taking into account the unique features of unikernels and uVMs. MEUNIK consists of four key components, which are summarized as follows:

First, our experiment shows that uVMs of the same type (i.e., uVMs running the same application) have a notably higher degree of shareable memory than traditional VMs. Therefore, to address the problem of memory over-allocation caused by predetermined uVM startup memory allocation, MEUNIK employs an on-demand memory allocation approach

[§]This work was completed during Yongshu's time as a student at Binghamton University.

2024 IEEE/ACM Symposium on Edge Computing (SEC) | 979-8-3503-7828-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SEC62691.2024.00030

2837-4827/24/$31.00 ©2024 IEEE
DOI 10.1109/SEC62691.2024.00030

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

based on copy-on-write (CoW) uVM cloning. This approach improves memory availability in the system by enabling intra-type uVM memory sharing, which allows uVMs of the same type to share most of their memory with an app-template uVM while requesting additional memory only as necessary. The details of the experiment and the on-demand memory allocation design are presented in Section IV-A.

Second, MEUNIK adopts a novel three-stage approach for uVM creation. uVMs created using this approach share memory with their corresponding app-template uVMs, which in turn share their memory with a generic-template uVM that uses only a single physical memory page. Through this design, the system achieves two goals at the same time: (1) minimizing the memory footprint of app-template uVMs, and (2) *cross-type uVM memory sharing* where uVMs running different applications share memory. More details are discussed in Section IV-B.

Third, MEUNIK uses a proactive approach to promptly release memory that has been freed by individual uVMs back to the available memory pool for reuse by other uVMs. The design addresses the unsuitability of existing VM memory transferring mechanisms on uVMs. However, due to the CoW-based memory allocation approach, this design can incur notable time overhead for uVMs with frequent memory operations. To address this issue, we propose two optimizations. The details of the proactive memory-releasing approach and its optimizations are discussed in Section IV-C.

Lastly, MEUNIK employs a simple yet effective approach based on the LRU (Least Recently Used) strategy to estimate the working set and reclaim idle memory for uVMs. We show that existing complex mechanisms for working set estimation and idle memory reclamation are overkill for uVMs, mainly due to the single-application nature of uVMs. We present the experiments and our insights in Section IV-D.

We implemented a prototype MEUNIK system with the Xen hypervisor [36]. We built three groups of unikernels based on the Rumprun platform: 15 Python benchmark program unikernels, 4 server application unikernels, and 2 ClickOS network middlebox unikernels. We then thoroughly evaluated the performance of the MEUNIK prototype system with these unikernels. The evaluation results show that MEUNIK effectively manages memory resources for individual uVMs based on their actual behavior and aggressively improves system memory availability with little overhead.

In summary, contributions in this paper are as follows:

- We demonstrate the limitations of existing VM memory management solutions for unikernel-based VMs through detailed real-world experiments and measurement studies.
- We propose MEUNIK, a solution specifically designed to address the limitations of the existing approaches.
- We implement a prototype MEUNIK system based on the Xen hypervisor.
- We create uVMs using various programs and applications, including 15 small benchmark programs, four complex server applications, and two network-intensive programs. We conduct extensive experiments to evaluate the performance of our prototype system, and the results indicate that our system achieves the intended design goals with minimal overhead.

II. RELATED WORK

VM forking and copy-on-write (CoW). There have been works that utilize the approach of cloning/forking/checkpointing an existing VM or container instance for various purposes [39]–[42]. For example, Potemkin [39] is a honeyfarm system that utilizes a large number of VMs to deploy decoy systems or services which attract attackers and malware. To speed up the VM creation, each new VM is forked from an initialized VM instance. SnowFlock [40] uses the VM fork approach to quickly clone a VM into multiple replicas on different hosts, which is a desired scenario in cloud computing. Zhi et al. utilize VM fork to start VMs in the cloud for more efficient system testing [41]. Catalyzer [42] proposes a sandbox fork mechanism that reduces the time needed to start a container in a serverless computing environment. All four works described previously also utilize the copy-on-write (CoW) technique to reduce VM startup time and conserve system resources. However, the works discussed above apply to conventional VMs, while we study the unique behaviors of uVMs and design solutions for efficient and effective uVM memory management in hypervisors.

Nephele [43] is a recent work that investigates how to systematically support uVM cloning. This work goes beyond duplicating address spaces to address other issues, such as I/O cloning and inter-uVM communication. While Nephele focuses on improving the CoW-based cloning technique for uVMs, our work aims to identify issues within current hypervisors regarding uVM memory management and develop solutions that consider the unique attributes of uVMs. In addition to leveraging the cloning technique, our work introduces three distinct and novel solutions to achieve the goal. In this sense, Nephele and our work complement each other.

Lightweight hypervisor designs. Designing hypervisors that are lightweight and secure has been the focus of many recent research work. Firecracker [44] is one such effort that is designed for serverless workloads. The central idea is to use VMs created by a hypervisor to run containers to achieve better security isolation. LightVM [45] is a Xen-based lightweight hypervisor that aims to boot a large number of VMs quickly. Similar to Firecracker, Kata Container [46] aims to provide a lightweight and secure virtualized runtime environment by running containers in VMs that are created and monitored by a hypervisor. Our work investigates and demonstrates the inadequacy of existing hypervisors' approach to managing memory resources for unikernel-based VMs. The proposed solutions in this work focus on the aspect of memory management and complement the lightweight hypervisor designs in the literature.

Improving memory availability in virtualized systems. Our work puts focus on maximizing memory sharing to improve memory availability for unikernel-based uVMs. Besides KSB-based [47]–[49] and TPS-based [50], [51] ways of performing

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

VM memory sharing/deduplication, there have been other work to improve memory sharing in virtualized environments [52]–[55]. Our work complements these works in that we are trying to solve this important problem in the context of uVMs.

III. Motivation

Due to the single-application nature of unikernels, a physical host needs to run a significantly higher number of uVM instances than with the case of traditional VMs (i.e., VMs running traditional OSes) in practice. We observe that existing VM memory management in hypervisors can cause memory to become a bottleneck which significantly limits the scalability of uVMs. We discuss our observations as follows.

Observation 1: Predetermined VM startup memory allocation is not suitable for uVMs. Existing hypervisors require that the size of memory assigned to a VM to be predetermined before starting it [56]–[59]. For example, with Xen, the administrator needs to specify either a fixed memory size or a range of memory sizes for a VM before launching it [56], [57]. If a fixed memory value is provided, the VM will be allocated with the requested amount of memory initially. If a range is specified, the hypervisor will keep the amount of memory allocated to the VM between the specified minimum and maximum memory values. With KVM, memory allocated to a VM is also manually specified, either when starting the VM or during runtime [58].

In practice, the static VM memory allocation described above is not suitable for uVMs because it is difficult for administrators to set a proper memory value for uVMs. The difficulty stems from the following two observations.
(1) *Cross-type uVM memory usage difference is high*: memory demands by uVMs of different types (i.e., uVMs that run different applications) can be drastically different.
(2) *Intra-type uVM memory usage variance is high*: for the same type of uVMs (i.e., uVMs that run the same application), the memory consumption may be highly variable depending on various factors, such as internal memory behavior of the application and different types of requests that the uVM needs to processes.

The two observations above were obtained through our experiments in which we compiled five applications into Rumprun unikernels [31], and run them with the Xen (version 4.10) VMs. The first two applications are `Node.js` [60] and `Ngnix` [61] web servers. Both web servers host a documentation website which consists of 1,000 static web pages of different sizes. The next two applications are popular in-memory key-value stores `Memcached` [62] and `Redis` [63]. The last one is on-disk key-value store `LevelDB` [64].

We first compared the memory demands of *uVMs of different types* (i.e., *cross-type* uVM memory usage). For the two web server uVMs (i.e., `Node.js` and `Ngnix`), we used the Apache `ab` benchmark tool to traverse all the web pages of the documentation website. For the two in-memory key-value store uVMs (i.e., `Memcached` and `Redis`), we used the `memtier` benchmark tool [65] to generate key-value store traffic which contained 20,000 requests with the set/get operation ratio set to be 1:1.

The experiment result shows that the two web server uVMs consumed drastically different memory: the `Node.js` uVM demanded 300 MB of memory while the `Nginx` uVM only required about 16 MB. The reason is that a substantial amount of memory was used to support the `Node.js` runtime while `Nginx` is known for its simplicity and lightweight. The two in-memory key-value store uVMs also exhibited notable differences in memory demand: `memcached` uVM consumed 35 MB of memory, and the `Redis` uVM needed 60 MB.

For *uVMs of the same type*, memory consumption can also vary significantly. For example, we sampled the memory usage of the `Node.js` uVM every second while it was running the workload described above. Figure 1(a) depicts the result, which shows that the memory usage of the uVM oscillated between roughly 300 MB and 180 MB while the workload was running. The oscillation is because of the fact that garbage collection of the `Node.js` runtime was invoked periodically (which caused the periodic drops in memory usage). After the workload completed the memory usage remained at around 180 MB. Besides internal memory behavior of the application, intra-type uVM memory consumption variance can also be caused by the different requests that the uVM needs to process. Figure 1(b) shows the per-second memory usage of the `Memcached` uVM when it dealt with two streams of key-value store traffic, both of which contained 20,000 requests. The difference was that the set/get operation ratio of one stream was 1/10 and the same ratio of the other was 10/1. It can be seen that the uVM consumed much more memory when processing the traffic with the 10/1 set/get operation ratio than processing the other one. Another example is shown in Figure 1(c), which showed the per-second memory usage of the `LevelDB` uVM. In this experiment, we compared the uVM memory consumption when the `LevelDB` application filled values in sequential key order ("fillseq") and random key order ("fillrandom"). The result showed that there was a 13% difference (278 MB for "fillseq" and 315 MB for "fillrandom").

The high cross-type and intra-type uVM memory usage variances demonstrated above render it difficult to statically set a proper memory value for uVMs when launching them or during runtime. To ensure uVMs to function properly, it is necessary to be conservative by choosing a high-end value when setting the amount of VM startup memory. However, this practice would cause memory waste which can devastate memory-constrained hosts such as in edge and embedded computing scenarios.

Observsation 2: Existing mechanisms for transferring memory between VMs do not suit uVMs. After VMs are started with the statically configured sizes of memory, existing hypervisors employ mechanisms to move memory across different VMs to support memory overcommitment [66]. Most of these mechanisms adopt the idea of memory ballooning [67], which utilizes "balloon drivers" in guest OSes

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

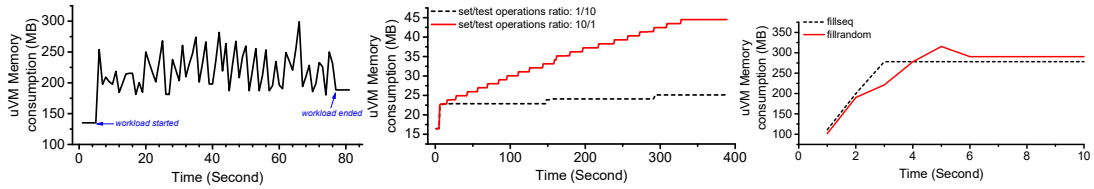

Fig. 1: Illustrations of high intra-type uVM memory usage variance. Memory consumption of (a) a `Node.js` uVM workload; (b) a `Memcached` uVM workload; and (c) a `LevelDB` uVM workload.

to transfer memory between VMs [51], [68]–[70]. However, memory ballooning is not suitable for uVMs for the following three reasons.

(1) Existing memory ballooning mechanisms require manual activation by administrators or system management tools for individual VMs [71], [72]. Although attempts have been made to enable automatic ballooning for KVM, such efforts remain incomplete [72]. Manual activation of memory ballooning for each uVM is inflexible, particularly considering the notably larger number of uVM instances in the system compared to traditional VMs.

(2) Even if memory ballooning can be efficiently activated for individual uVMs, recent studies indicate that it responds slowly to memory demand changes in VMs [69], [70], [73]. Given the substantial variance in memory consumption demonstrated by uVMs, memory ballooning is unlikely to be effective for them.

(3) Enabling memory ballooning for uVMs would require adding support to the unikernel guest OSes, such as incorporating balloon drivers into the unikernels. However, this would contradict the minimalism philosophy of unikernels.

Observation 3: Existing cross-VM memory sharing mechanisms are slow and costly for uVMs. The need of supporting a large number of uVM instances on a single physical host puts significant pressure on system memory consumption. Effective and efficient cross-VM memory sharing mechanisms are helpful to alleviate such pressure.

However, the existing VM memory sharing mechanisms are not suitable for uVMs because they are mostly slow and incur high costs. The existing mechanisms can be classified into two groups: Kernel Same-page Merging (KSM) [47]–[49] or Transparent Page Sharing (TPS) by VMware [50], [51]. Both KSM and TPS work by periodically scanning the entire system to identify and share identical memory pages. As a result, they are not suitable for uVMs because content-based scanning is slow and incurs high costs in deduplicating identical pages [49], [53], [54].

Observation 4: Existing VM working set estimation and idle memory reclamation mechanisms incur high overhead. In addition to memory ballooning and cross-VM memory sharing, idle memory reclamation is another way to improve memory availability for VMs. Idle memory reclamation works by first estimating VM working set which is the set of memory pages that are being actively used by the VMs, and then swapping non-working-set pages out to the swap area [51]. However, the existing ways of estimating VM working set, such as random TLB invalidation followed by TLB misses checking to identify working sets [51], incur high overhead to achieve the goal [69], [74].

TABLE I: VM sharable physical memory pages comparison: uVMs vs. traditional VMs.

	Rumprun (unikernel) VMs			Lubuntu (lightweight Linux) VMs		
	U	**S**	**F**	**U**	**S**	**F**
After boot	2%	10%	88%	58%	17%	25%
`qsort`	3%	16%	81%	62%	14%	24%
`jpeg`	3%	13%	84%	65%	13%	22%
`sha`	3%	17%	80%	59%	17%	24%

U: Unique pages | **S**: Shareable base pages | **F**: Freeable pages.

IV. System Design

A. On-demand uVM memory allocation and intra-type uVM memory sharing

As demonstrated in Section III, uVMs exhibit a wide and instantaneous variance in memory usage when compared to VMs running traditional OSes. Therefore, the predetermined VM startup memory allocation approach that is commonly adopted by most hypervisors often leads to memory over-allocation when hosting uVMs.

To address this issue, MEUNIK employs an on-demand memory allocation approach which is based on the principles of *VM cloning* and *copy-on-write (CoW)*. Our experiments show that the combination of these techniques provides greater benefits to uVMs than to traditional VMs in terms of improving memory availability in the system.

Intra-type uVM memory sharing opportunities. In our experiments, we compiled three benchmark programs, `qsort`, `jpeg`, and `sha`, from the MiBench suite [75] into three types of Rumprun unikernels [31]. For each type of unikernel, we ran it as the guest OS of two Xen VMs, which therefore are two uVMs of the same type, on an Odroid XU4 single-board computer [76]. We measured the similarity among the two uVMs' physical memory pages at two time points: immediately after the VM booted and after the benchmark program had run for one minute. We also ran each of the programs on two conventional Xen VMs, which used the lightweight Linux distribution Lubuntu [77] as the guest OS, and performed the same measurements for the two VMs. The

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

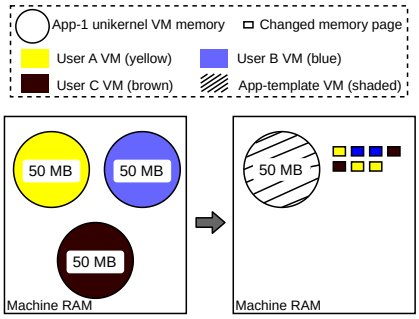

Fig. 2: On-demand memory allocation and intra-type uVM memory sharing.

physical memory sizes of a Rumprun uVM and a Lubuntu VM were configured to 20 MB and 256 MB, respectively.

By comparing the similarity of the physical memory pages between the two VMs, we classified the memory pages of a VM into three groups: A page is a *unique page* if it is different from any other pages across the two VMs. If the two VMs have pages that are identical, one of those identical pages is marked as a *shareable base page*, and the remaining pages are considered as *freeable pages*. Table I shows the percentages of the three categories of memory pages in the uVMs and the conventional VMs. The results show that Rumprun uVMs have a much higher potential for memory sharing than the two conventional VMs. The main reason is that traditional operating systems typically have many computing tasks belonging to either the OS kernel or the user, whereas unikernels only have a single task. This reduces the number of memory modifications in uVMs, which increases the opportunities for memory sharing. As a result, uVMs of the same type can benefit significantly from memory sharing.

uVM cloning and on-demand memory allocation. To take advantage of intra-type uVM memory sharing opportunity, we designed a uVM cloning mechanism that utilizes the principle of copy-on-write (CoW) to achieve memory sharing among uVMs of the same type and on-demand uVM memory allocation. Using this mechanism, a new uVM of an app is launched by cloning the app-template uVM of the same app. An app-template uVM is created by following the same process as starting a uVM normally, except that it is paused after the initialization phase, which includes loading the code and data of the unikernel. The memory pages of the app-template uVM are then made read-only. Instead of having their own memory allocated by the hypervisor from the beginning, new uVMs are cloned from the corresponding app-template uVMs and share memory pages with them. This allows individual memory allocation needs from the new uVMs to be met via the CoW process [78]. Specifically, when a new uVM needs to write to a memory page, a page fault is generated due to the attempt to modify a read-only memory page. The control is transferred to the hypervisor, which then allocates a new page for the new uVM, copies the content of the faulting page to the new page, and marks the new page writable to the new uVM.

Figure 2 illustrates the advantages of the above mechanism using an example scenario. Let's consider three users, A, B, and C, each launching a uVM running the same application. The estimated memory consumption of a uVM, depending on the application's operations, ranges from 20 MB to 50 MB (such as the `Memcached` uVM demonstrated in Figure 1 (b)). To ensure the normal operation of the application, it is advisable to assign a conservative 50 MB of memory when launching each uVM. With the existing predetermined VM startup memory allocation, a total of 150 MB of memory would be allocated to the three uVMs (left part of Figure 2). However, considering the varying application demands, a large portion of this allocated memory is likely to be underutilized. In contrast, by utilizing the proposed uVM cloning and on-demand memory allocation mechanism, only 50 MB of memory is initially required to launch the app-template uVM, along with additional memory that is actually needed by the uVMs during runtime (right part of Figure 2).

B. Single-page generic-template uVM and three-stage uVM creation

Given the single-application nature of uVMs, it is expected that many uVMs of different types, each running a distinct application, are present in the system concurrently. Despite the mechanisms described in Section IV-A, it is still necessary to preallocate memory conservatively for a considerable number of app-template uVMs. Furthermore, the mechanisms do not utilize the potential memory sharing opportunities among uVMs of different types.

Three-stage uVM creation. To address the above limitations, we extend the uVM cloning mechanism to encompass the creation of app-template uVMs. As a result, a three-stage approach is used to launch individual uVMs.

(1) *Stage-1: single-page generic-template uVM creation*. During its bootstrapping process, MEUNIK creates a special VM called the generic-template uVM, which is used as a template for creating app-template uVMs.

To create the generic-template uVM, the only real work to perform is to set up the uVM's page table such that every virtual memory page (VMP) in the address space is linked to a sole machine memory page (MMP) allocated for the generic-template uVM. In regular VMs, VMPs in an address space are not associated with any MMPs at the beginning. As a result, all the page table entries (PTEs) are marked as invalid. When an invalid PTE is referenced during runtime, the page fault handling process takes place. This process maps an MMP from the (conservatively) preallocated VM memory to the PTE, and marks the PTE as valid. However, with our design for the generic-template uVM, MEUNIK allocates just one MMP, and associates it with all the PTEs. Subsequently, all the PTEs are marked as valid from the start. This design eliminates the need to preallocate memory for the generic-template uVM, as well as all future app-template uVMs. MEUNIK then marks the PTES to set all the VMPs as read-only and suspends the VM. The generic-template uVM is now ready for future app-template uVM creation.

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

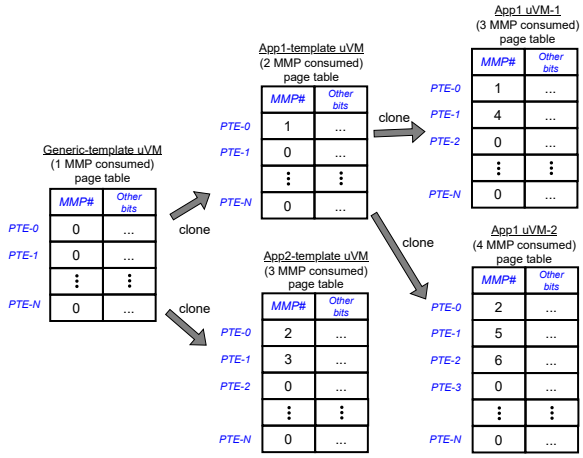

Fig. 3: An illustration of three-stage uVM creation (PTE stands for "page table entry", MMP stands for "machine memory page").

(2) *Stage-2: app-template uVM creation*. When a new uVM is launched to run an application that is different from any current uVMs, an app-template uVM is created by cloning the generic-template uVM. The cloned uVM is then resumed and continues the booting process. which includes initializing other VM data structures, such as virtual CPUs (vCPUs) and I/O devices, and setting up the guest OS using the unikernel of the new app. Since page tables are not shareable, the page table is also copied, and the copied version is loaded into the hardware page table base register (PTBR) of the cloned uVM. The cloned uVM is paused before the first instruction of the app code is executed. At this point, the cloned uVM is ready to be used as a template for regular uVM creation.

(3) *Stage-3: regular uVM creation*. A regular uVM is created by cloning the corresponding app-template uVM. The cloned uVM is then resumed, and a copy of the app-template uVM's page table is loaded into the cloned uVM's PBTR. The cloned uVM then continues and functions as a regular uVM.

An illustrative example. Figure 3 illustrates how the three-stage uVM creation works. As part of the system booting process, a single-page generic-template uVM is created. Suppose there are N VMPs in a uVM's address space, the leftmost part of Figure 3 shows the content of the generic-template uVM's page table. In this page table, the MMP number field of all page table entries (PTEs) is set to 0, meaning all VMPs are backed by the same MMP (i.e., MMP #0).

The generic-template uVM is then cloned into two app-template uVMs. Suppose app1-template uVM only modifies VMP0 during the initialization process, the VMP0 of this uVM is allocated with a new MMP (#1). The remaining VMPs of app1-template uVM are backed by MMP0, which is the single MMP allocated for the generic-template uVM. Similarly, for app2-template uVM, if the first two VMPs are modified during its initialization process, only these two VMPs are allocated with new MMPs.

Regular uVMs are cloned from the corresponding app-template uVM. In the example, two regular app1 uVMs are created. In the beginning, uVM-1 shares MMP1 with the app-template uVM and shares MMP0 with the generic-template uVM. Then, new MMPs are allocated to it as individual VMPs are written.

Benefits analysis. The three-stage uVM creation design provides the following three main advantages.

First, each app-template uVM is cloned from the generic-template uVM, where all the VMPs are mapped to the same read-only MMP. This approach allows for on-demand memory allocation via CoW as individual VMPs are written, ensuring that only the necessary amount of memory is allocated for each app-template uVM. In contrast, the conventional way to create an app-template uVM would require allocating a predetermined amount of memory, which can add up to a significant amount considering the potentially high number of app-template uVMs in the system. Worse, most of this preallocated memory would remain unused as app-template uVMs do not execute actual applications during runtime.

Second, each regular uVM is cloned from its corresponding app-template uVMs. Therefore, uVMs of the same type share the majority of their memory with the app-template uVM and only request additional memory pages as needed due to the usage of CoW-based page allocation.

Third, with the proposed three-stage uVM creation approach, unused VMPs in uVMs are backed by the same MMP and considered to be valid. This approach allows unused memory to be shared across and within individual uVMs, resulting in improved memory availability in the system. In essence, MEUNIK enables lazy page allocation at the hypervisor level, which is typically a functionality of the guest OS kernel. This reduces the complexity of the uVM guest OS, conforming to the minimalism philosophy of unikernels.

It is worth noting that our three-stage uVM creation design incorporates CoW without incurring the usual copying overhead associated with typical CoW scenarios. This is because the copying phase can be bypassed if CoW is triggered as the result of writing to the sole MMP allocated to the generic-template uVM (which suggests that the page being written to was unused). As a result, the CoW-based on-demand memory allocation during the uVM creation phases exhibits similar performance in terms of allocation time to the conventional memory allocation approach.

C. Proactive memory releasing

As explained in Section III, existing approaches for returning memory back to the hypervisor when system memory pressure is high, such as memory ballooning, are not suitable for uVMs. This is because the memory ballooning approach (1) needs to be activated manually, (2) responds slowly to memory demand changes in uVMs, and (3) requires substantial changes to the unikernel. To address this issue, our solution comprises two parts: automatic freed memory releasing (this section) and idle memory reclamation (Section IV-D).

Automatic freed memory releasing (AFMR). During periods of high memory pressure, AFMR automatically releases memory back to the hypervisor as soon as a uVM indicates that it

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

no longer needs it, such as when memory is explicitly freed by the uVM. It can be achieved by adding a single hypercall which is invoked whenever memory is freed by unikernel code. The hypercall simply marks the MMPs corresponding to the VMPs that have been freed as available. Additionally, it maps the sole MMP allocated for the generic-template uVM to the freed VMPs and sets the VMPs as read-only. This ensures that the CoW-based on-demand page allocation mechanism can be activated when the freed VMPs are reallocated and referenced again.

AFMR can notably improve system memory availability when the system is hosting unikernels that frequently perform memory allocation and free operations. However, it may also cause noticeable time overhead in certain scenarios. For example, if a large amount of memory pages are reallocated and written to shortly after being freed, an equivalent amount of CoW page faults can occur. While the copying phase can be skipped when the faulting page is backed by the sole MMP allocated to the generic-template uVM, handling CoW page faults can still be time-consuming, especially when handling a large amount. To address this problem, we propose two optimizations to complement AFMR.

Optimization 1: speculative allocation (SA). The first optimization aims to improve the efficiency of memory reallocation. When a CoW page fault occurs on VMP, it speculates which additional VMPs are likely to be written to and preallocates MMPs for all those VMPs within one CoW page fault handling process. As a result, the overhead associated with multiple individual CoW page fault operations is reduced.

The SA algorithm leverages the principles of spatial and temporal locality in memory references to predict whether to preallocate an MMP for a VMP. It maintains two counters for each VMP in an address space: the *allocation count* and the *hit count*. The *allocation count* increases each time the VMP is allocated with an MMP through either CoW or preallocation. The *hit count* increases each time when the VMP is written to for the first time since being allocated with an MMP. When handling a CoW page fault, for each VMP in a window of VMPs that follow the faulting page, the algorithm speculatively preallocates an MMP for the VMP if it satisfies all the conditions below:

- *Cond-1:* The VMP is mapped with the sole MMP allocated to the generic-template VM. In other words, the VMP is not currently allocated with an actual MMP.
- *Cond-2:* The VMP has been allocated with an actual MMP before. This is because a VMP that has never been touched does not provide any temporal information for prediction.
- *Cond-3:* The time elapsed since the last time the VMP was freed is smaller than a threshold. In our prototype system implementation, we use a threshold of 5 seconds.
- *Cond-4:* The current prediction hit ratio on the VMP, which is the ratio between the VMP's hit count and allocation count, is greater than a threshold. For our prototype system, we set this threshold to 95%.

If any VMP in the prediction window does not to meet one or more of these conditions one VMP, the SA algorithm aborts for all the VMPs behind it.

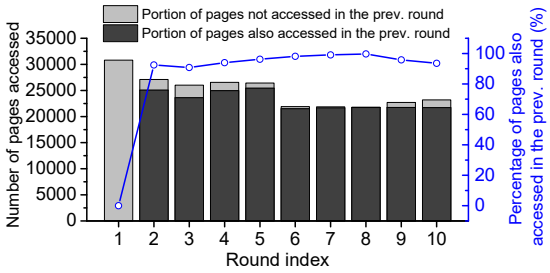

Fig. 4: Memory pages accessed for the `Node.js` uVM.

Optimization 2: delayed releasing (DR). The second optimization aims to address the potential problem of AFMR by delaying the release of MMPs back to the hypervisor. In Section V, we discuss how the DR optimization was be implemented while taking system efficiency into account.

D. uVM-aware working set estimation and idle memory reclamation

If memory pressure continues to be high even after proactive memory releasing is enabled, MEUNIK takes the approach of reclaiming memory that is not being actively used by the VMs. In other words, it targets idle memory that does not belong to the VMs' working sets.

Existing VM idle memory reclamation solutions typically involve two steps. First, the hypervisor determines which VMs should have their MMPs reclaimed. A common approach is to calculate the "price" that a VM is paying for each MMP, and then reclaim MMPs from VMs that are paying a lower price and reallocate them to VMs that are willing to pay a higher price [51]. In the second step, the hypervisor estimates the working sets of the target VMs and reclaims memory pages that are not part of the working sets.

However, existing idle memory reclamation solutions do not fit uVMs well for two reasons: *First*, calculating page price for every VM in the system is prohibitively expensive considering the much larger number of VMs in the case of uVMs. *Second*, the existing methods for working set estimation, such as random TLB invalidation followed by TLB misses checking to identify the working sets [51], are complex and incur high overhead [69], [74].

We observed that the working sets of uVMs are more stable over time than those of conventional VMs. To illustrate this observation, we conducted an experiment in which a uVM web server running Node.js was set up to host a documentation website consisting of 1,000 static web pages. We used the Apache `ab` benchmark tool on a separate machine to browse through the entire wedbsite. We repeated the experiment for 10 rounds, and tracked how memory pages of the uVM were accessed during each round. Figure 4 shows the results of the experiment. The working set of the uVM remained stable throughout the 10 rounds of the experiment, with an average of 95% of the memory pages accessed in each round also

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

being accessed in the previous round. The reason for highly stable working sets in uVMs is that a uVM only runs a single application only the necessary OS kernel functionalities. In contrast, a conventional VM typically runs multiple processes alongside with the full OS kernel. As a result, working sets of conventional VMs are more prone to change as active processes change over time.

Given the observation that uVMs tend to have stable working sets, we propose a uVM-aware working set estimation and idle memory reclamation mechanism based on the simple least recently used (LRU) heuristic. With our solution, the hypervisor maintains a queue of all the MMPs that have been allocated to the uVMs in the system. When an MMP is referenced, it is moved to the back of the queue. When the system experiences high memory pressure, the hypervisor reclaims a certain amount of MMPs from the front of the queue, as configured by the user. Our solution is lightweight compared to existing solutions in two ways. *First*, it treats all the MMPs in the system as a whole, rather than checking individual uVMs to determine which ones should have their memory reclaimed. *Second*, the LRU heuristic is much more lightweight than existing approaches such as random TLB invalidation followed by TLB misses checking.

V. System Implementation

Implementation setup. We have implemented the proposed MEUNIK system on Xen hypervisor version 4.10 [79]. In Xen, each VM is referred to as a domain. For the remainder of this paper, we will use the terms "domain" and "VM" interchangeably. The Dom0, which is the privileged management domain for the unprivileged domains (referred to as DomUs), runs on Linux 4.4 as the operating system.

We chose paravirtualization (PV) as the virtualization technique for our prototype system. There are two main reasons for this decision. First, our design involves participation from VMs, such as proactive memory releasing, which requires modifications to the VM guest OS. Second, unikernels, which are the guest OSes of unikernel VMs, are typically built before deployment due to their single-application nature. As a result, changing unikernels to add MEUNIK support would not incur much deployment overhead.

The unikernels that have been integrated with our prototype system are Rumprun [31], which is a NetBSD-based library OS that supports the development and execution of existing application code as unikernels, and ClickOS [30], which is a virtualized software router platform based on the Click Modular Router architecture [80].

Xen Background. To speed up memory translation, Xen's PV MMU (memory management unit) model lets a guest OS map its virtual memory pages (VMPs) directly to machine memory pages (MMPs) in its page tables, instead of mapping them to the guest's physical memory pages (PMPs) [81]. Thus, guest OSes' page tables are also called V2M tables. A V2M table can be updated by the hypervisor and the guest OS, but the guest must use hypercalls provided by the hypervisor to do so. The hypervisor maintains another page table called the P2M table, which records the mappings from the guest's PMPs to MMPs. The P2M table is readable to the guest, and it helps the guest to populate the V2M table as unmapped VMPs are referenced.

CoW-based page allocation. To implement the CoW-based page allocation mechanism, an MMP to be shared is first set as read-only by clearing the writable bit of the corresponding V2M table entry and then marked as shared in its page descriptor. This can be done by either the hypervisor (when setting up a generic-template uVM or an app-template uVM), or the guest OS (when exercising AFMR). When a shared MMP is modified, a page fault is generated. The hypervisor handles this page fault by allocating a new MMP for the uVM that attempted to modify the faulting MMP. The hypervisor then copies the content of the faulting MMP to the newly allocated MMP and updates the uVM's corresponding V2M entry accordingly. If a page fault is not due to an attempt to modify a read-only page, the hypervisor injects the page fault back to the uVM and lets the uVM handle it.

Three-stage uVM creation. To create the generic-template uVM, the hypervisor only needs to initialize the V2M table by mapping all the VMPs of the guest OS to a single MMP dedicated to the generic-template uVM and setting the VMPs as shared and read-only. To create an app-template uVM, the hypervisor takes a copy of the generic-template uVM, duplicates its V2M table, and continues the uVM startup process. This includes loading the unikernel image, setting up resources related to the vCPUs and I/O devices, and finishing the application initialization. During the startup process, all newly allocated MMPs due to CoW page allocation are marked as read-only, except for those that cannot be shared (e.g., MMPs for the page table, and resources related to vCPUs and I/O devices). The app-template uVM is suspended when it is ready to execute the first instruction of the application code. Regular uVMs are created by cloning the corresponding app-template uVMs. During this process, the hypervisor only needs to allocate and initialize the memory for contents that cannot be shared. The regular uVM shares the majority of its memory with the app-template uVM and only requests new memory as needed.

AFMR and its optimizations. The AFMR (automatic freed memory releasing) mechanism was implemented by instrumenting the Rumprun framework so that a hypervisor hypercall is made whenever a memory-freeing operation is performed in the guest OS. The details of the AFMR hypercall actions are discussed in Section IV-C.

The SA algorithm maintains two counters for each VMP in the address space: the allocation count and the hit count. The allocation count is increased by one when the VMP is allocated with an MMP, through either CoW or SA. The hit count is incremented by one each time the VMP is first modified since it was last allocated with an MMP. In our implementation, we set the hit count of a VMP when it is released back to the hypervisor via AFMR. Specifically, when serving an AFMR hypercall, the hypervisor checks the dirty bit of the V2M table

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

entry which contains the VMP being freed. If the dirty bit is set, the hypervisor increments the hit count of the VMP by one.

To implement the DR mechanism, we used a queue to track the MMPs that have been recently freed by the guest OS. We modified the buddy page allocator of the Rumprun platform such that MMPs in the queue are used to satisfy memory allocation requests in the guest OS first. If an MMP in the queue is selected to satisfy a memory allocation request, it is removed from the queue. A kernel thread in the guest is then used to periodically examine the MMPs in the queue and invoke the AFMR hypercall to release the MMPs that have remained in the queue for more than a threshold period of time (which is set to 5 seconds in our prototype system).

uVM-aware idle memory reclamation. The uVM-aware idle memory reclamation mechanism described in Section IV-D treats all the MMPs that have been allocated to the individual uVMs as a whole and applies a simple heuristic of LRU (Least Recently Used) to select pages for reclamation. In our implementation, we used the clock algorithm [82] to approximate LRU. The clock algorithm uses the dirty bit of individual MMPs (which can be found in the corresponding V2M table entry) to determine which MMP should be reclaimed next. Due to space constraints, we omit the details of the algorithm. One notable issue that occurred in our implementation is that the clock algorithm clears the dirty bit of all the allocated MMPs periodically. This created a problem for our system because the dirty bit of individual MMPs is also used in the speculative allocation mechanism. To address this problem, we duplicated the dirty bit of all the allocated MMPs into a bitmap (which is called the shadow dirty bitmap) each time the clock algorithm clears the MMPs' dirty bit. By doing this, we can safely use the original dirty bit of individual MMPs for the clock algorithm while using the shadow dirty bitmap for speculative allocation.

Dealing with network devices. Since the DomU uVMs are cloned from the app-template uVMs, uVMs of the same type all have the same MAC and IP addresses. Existing solutions use network tools that are readily available in conventional OSes to change VM MAC/IP addresses at runtime [39]. However, these solutions are not applicable for uVMs because of the app-specific nature of unikernels. Our solution is to assign new MAC/IP addresses to newly cloned uVMs on the back-end network device driver, while keeping the MAC/IP addresses on the front-end which are inherited from the app-template uVM unchanged. Since the shared ring buffer between Dom0 and DomU is placed at front-end driver in DomU, the back-end driver in Dom0 is able to differentiate ownership of network packets and acts as a proxy accordingly. For outgoing packets that a uVM (i.e., DomU) put into the ring buffer, the back-end driver in Dom0 fetches them and replaces the source MAC/IP addresses with the new ones assigned to that uVM. For incoming packets, the back-end driver is able to tell which uVM is the receiver by examining the destination MAC/IP address. It then replaces the packet destination MAC/IP addresses with the original ones and places the packets in the ring buffer of the receiving uVM.

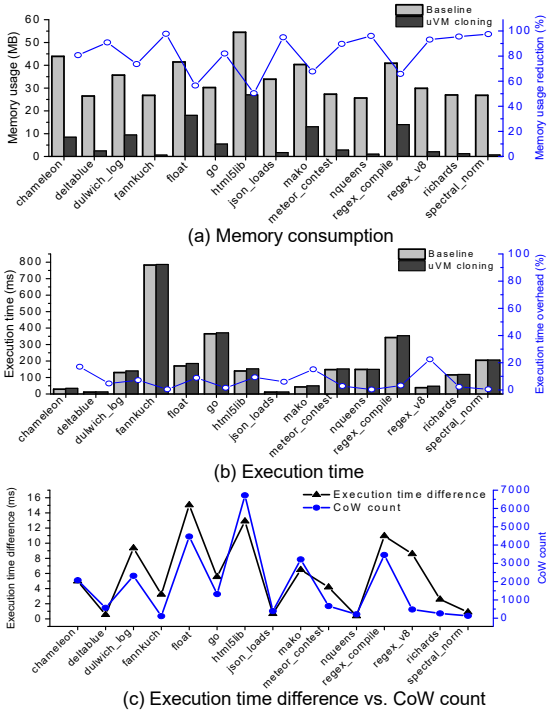

Fig. 5: Python benchmark uVMs experiment results.

VI. System Evaluation

We evaluate our prototype system on a desktop server with an 8-core 3.3 GHz CPU and 32 GB of physical memory. The unikernels used in the experiments were compiled from the following three groups of programs and applications:

- *Python benchmark program unikernels*: To evaluate MEUNIK's performance when running unikernels built from small programs, we selected and compiled benchmark programs from the `pyperformance` Python performance benchmark suite [83] into Python benchmark unikernels. We skipped programs that involve user interaction, as we were only interested in evaluating uVM execution time performance. We also skipped programs that were similar to the ones that had already been selected. As a result, we selected 15 programs, as shown in the evaluation results later. To build a Python benchmark uVM, we first fed the code of a selected program and the Python interpreter to the Rumprun platform. Rumprun then generated the unikernel, which was used as the guest OS to start a uVM.
- *Server application unikernels*: We compiled four popular server applications, `Node.js`, `Nginx`, `Memcached`, and `Redis` into Rumprun unikernels, and built four app uVMs using these unikernels.
- *ClickOS middlebox unikernels*: We ran two ClickOS middleboxes, IP router and Firewall, as uVMs, and evaluated MEUNIK's performance of hosting network-oriented uVMs.

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

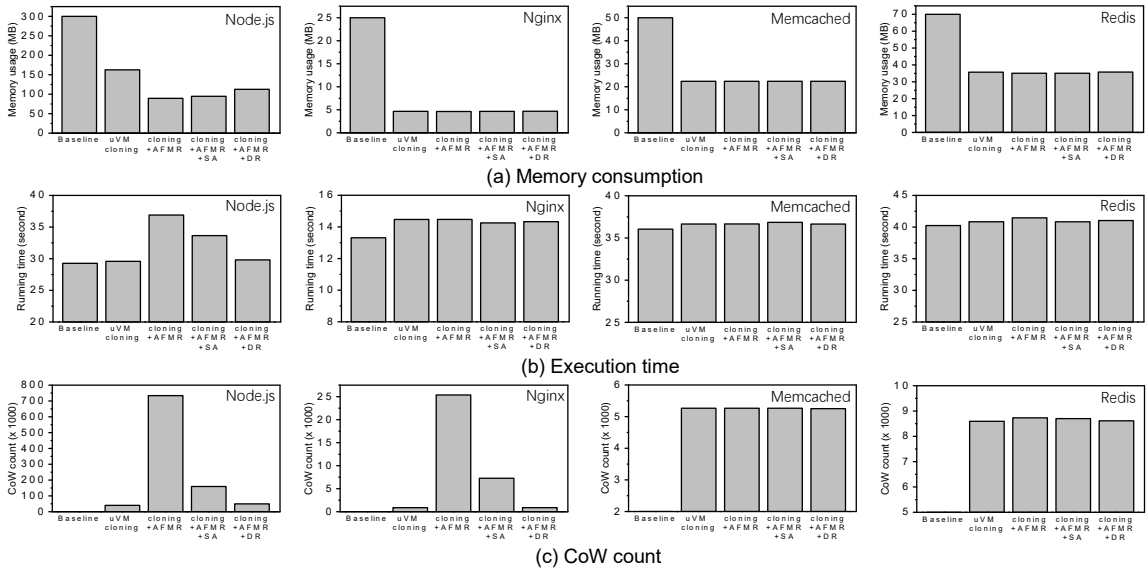

Fig. 6: Application uVMs experiment results.

TABLE II: Memory pages distribution of Python microbenchmark uVMs (page size is 4KB)

	chameleon		deltablue		dulwich_log		fannkuch		float	
	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique
text section	1,462	0	1,462	0	1,462	0	1,462	0	1,462	0
data sections	1,410	87	1,444	53	1,376	121	1,470	27	1,415	82
kernel stack	1,600	0	1,600	0	1,587	13	1,600	0	1,600	0
user stack & dyn. alloc. mem.	4,315	1,998	1,455	510	2,063	2194	1,954	86	1,312	4384
VM-specific info	0	72	0	72	0	72	0	72	0	72
Total	8,787	2,157	5,961	635	6,488	2,400	6,486	185	5,789	4,538
(percentage)	(80.29%)	(19.71%)	(90.37%)	(9.63%)	(73.00%)	(27.00%)	(97.23%)	(2.77%)	(56.06%)	(43.94%)
	go		**html5lib**		**json_loads**		**mako**		**meteor_contest**	
	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique
text section	1,462	0	1,462	0	1,462	0	1,462	0	1,462	0
data sections	1,425	72	1,334	163	1,444	53	1,401	96	1,452	45
kernel stack	1,600	0	1,587	13	1,600	0	1,600	0	1,600	0
user stack & dyn. alloc. mem.	1,650	1,242	2,404	6,553	3,469	341	2,307	3,108	1,550	618
VM-specific info	0	72	0	72	0	72	0	72	0	72
Total	6,137	1,386	6,787	6,801	7,975	466	6,770	3,276	6,064	735
(percentage)	(81.58%)	(18.42%)	(49.95%)	(50.05%)	(94.48%)	(5.52%)	(67.39%)	(32.61%)	(89.19%)	(10.81%)
	nqueens		**regex_compile**		**regex_v8**		**richards**		**spectral_norm**	
	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique
text section	1,462	0	1,462	0	1,462	0	1,462	0	1462	0
data sections	1,465	32	1,379	118	1,450	47	1,451	46	1467	30
kernel stack	1,600	0	1,587	13	1,600	0	1,600	0	1600	0
user stack & dyn. alloc. mem.	1,563	182	2,246	3,312	2,387	423	1,868	213	1945	97
VM-specific info	0	72	0	72	0	72	0	m72	0	72
Total	6,090	286	6,674	3,515	6,899	542	6,381	331	6,474	199
(percentage)	(95.51%)	(4.49%)	(65.50%)	(34.50%)	(92.72%)	(7.28%)	(95.07%)	(4.93%)	(97.02%)	(2.98%)

TABLE III: Memory pages distribution of application uVMs (page size is 4KB)

	Node.js		Nginx		Memcached		Redis	
	Shared	Unique	Shared	Unique	Shared	Unique	Shared	Unique
text section	2,512	0	879	0	551	0	618	0
data sections	2,718	388	824	61	896	92	844	169
kernel stack	1,650	14	1,650	14	1,587	13	1651	13
user stack & dyn. alloc. mem.	27,458	39,740	1,732	1,046	3,871	5,354	5462	8,607
VM-specific info	0	520	0	44	0	136	0	136
Total	34,338	40,662	5,085	1,165	6,905	5,595	8575	8925
(percentage)	(45.78%)	(54.22%)	(81.36%)	(18.64%)	(55.24%)	(44.76%)	(49.00%)	(51.00%)

A. uVM memory consumption reduction

We first evaluated how MEUNIK improves system memory availability with its uVM cloning and on-demand page allocation mechanisms.

Python benchmark program uVMs. In the first experiment, we measured the baseline cases where each Python benchmark

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

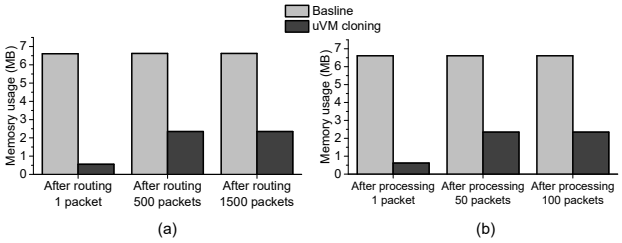

Fig. 7: ClickOS middlebox uVMs, (a) IP router and (b) Firewall, experiment results.

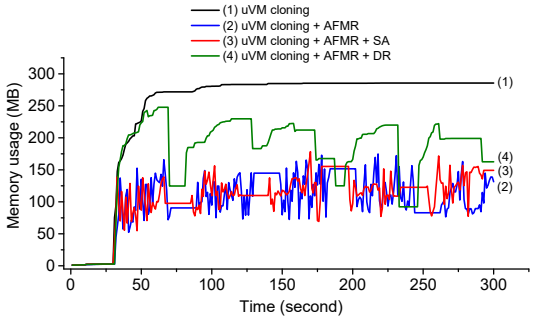

Fig. 8: Memory usage of `Node.js` uVM under a simulated long periodic web usage scenario.

uVM was run with the unmodified Xen hypervisor. Since the uVM was running as a conventional VM, it was conservatively configured with 64 MB of machine memory. We then ran each Python benchmark uVM with our MEUNIK system. We measured the actual memory consumption of each uVM at the end of each run. We repeated the experiment three times and calculated the average memory consumption. Figure 5(a) shows the results of this experiment. As can be seen, the uVM cloning and on-demand memory allocation mechanisms significantly reduce memory consumption for each uVM. Five Python benchmark uVMs (`fannkuch`, `json_loads`, `nqueens`, `richards`, and `spectral_norm`) see over 95% memory consumption reduction compared to the baseline cases. The uVM that has the smallest consumption reduction (`html5lib`) still enjoys about 50% reduction.

It is worth noting that in the baseline cases, each uVM was assigned 64 MB of machine memory, even though the actual memory consumption varied between 26 MB and 55 MB. This means that a significant amount of memory was left unused. In contrast, with MEUNIK, only the memory that is actually consumed by each uVM is allocated. This results in significant improvement in system memory availability.

We then measured the time needed to run each benchmark once with the uVM. Figure 5(b) shows the results. We can see that uVM workloads run with MEUNIK take more time than those with the original hypervisor. However, the time overhead is generally low, with 12 of the 15 uVMs having an overhead of less than 10%. The reason for the execution time overhead is because MEUNIK allocates memory pages on demand using the CoW mechanism. Figure 5(c) plots the execution time overhead (black line, left Y) and the CoW page fault count (blue line, right Y). It shows that the two lines roughly overlap, which supports our explanation above.

Sever application uVMs. We performed the same experiment as above for the four app uVMs. In the experiment, the app uVMs performed the workloads described in Section III "Observation 1". The first (the leftmost) bar in each plot of Figure 6 shows the results for the baseline cases, and the second bar shows the results for cases with uVM cloning and on-demand memory allocation enabled. Similar to the Python benchmark uVMs, all four app uVMs see consumption reduction at 45% or above with MEUNIK. The execution time overhead for `Node.js`, `Memcached`, and `Redis` is all less than 2%, while the overhead for `Nginx` is 8%.

ClickOS network middlebox uVMs. We also measured memory consumption reduction for the two ClickOS middlebox uVMs. For the IP router middlebox uVM, we measured its actual memory consumption after 1, 500, and 1,500 network packets had been routed. For the Firewall uVM, we took the measurement after 1, 50, and 100 network packets have been processed by the firewall. Figure 7 shows the results from which two observations can be made. First, the uVM cloning mechanism significantly reduced the memory consumption for the two middlebox uVMs. Second, in the baseline cases, memory consumption remains constant across the three measurement timings. However, in the cases with MEUNIK, uVM memory consumption increases as more packets are processed but remains significantly lower than in the baseline cases. To understand the reason for the second observation, we examined the source code of the two middlebox programs. We found that both programs request a large amount of heap memory on startup. Therefore, the explanation is that with the original hypervisor, all the requested memory is actually allocated immediately upon request. With MEUNIK's three-stage uVM creation design, memory pages are allocated lazily by the hypervisor (Section IV-B), which results in improved system memory availability.

B. uVM memory pages distribution

The major source of uVM memory consumption reduction by MEUNIK is memory page sharing between uVMs of the same type and the app-template uVM. During our experiments, we collected statistics about how page sharing among uVMs is distributed among different types of memory pages, such as text, data, stack/heap of kernel code, stack/heap of app code, and VM-specific info. Table II shows the results of the 15 Python benchmark program uVMs, and Table III presents the results of the 4 server application uVMs. In the tables, the memory pages of each uVM are broken down into two parts: the memory pages that are shared with the corresponding app-template uVM and therefore saved compared to the baseline case (listed in the "Shared" columns) and memory pages that are unique (listed in the "Unique" columns). The results show that uVM memory page changes are mostly made on user program stacks and heaps. This is also the reason why the server application uVMs saw lower percentages of memory

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

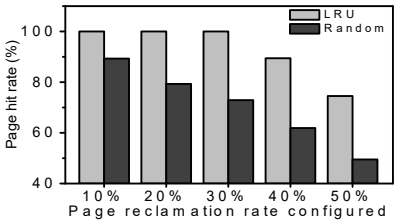

Fig. 9: Evaluating LRU-based idle memory reclamation.

saving - server applications run longer and have more complex operations than the Python benchmark programs.

C. Effectiveness of AFMR and its optimizations

AFMR (automatic freed memory releasing) is designed to achieve more aggressive memory savings when the system is under high memory pressure. We conducted two experiments to evaluate the effectiveness of AFMR and its two optimizations SA (speculative allocation) and DR (delayed releasing).

In the first experiment, we evaluated the performance of AFMR and the two optimizations for the four app uVMs executing the workloads described in Section III "Observation 1". The rightmost 3 bars in each plot of Figure 6 show the results of this experiment. As shown in Figure 6(a), the `Node.js` uVM benefited the most from AFMR, consuming 70.2% less memory than the baseline case, compared to 45.8% with uVM cloning alone. However, the memory saving enabled by AFMR comes at the price of execution time overhead. As shown in Figure 6(b), the time overhead for the `Node.js` uVM is 26.1% with AFMR enabled, compared to 1.1% with uVM cloning alone. The increase in time overhead is due to the increased number of CoW page faults caused by AFMR. As shown in Figure 6(c), the CoW count with AFMR was 18 times that of uVM cloning alone. SA and DR are designed to address the high time overhead issue. Both of them work well for the `Node.js` uVM. AFMR with SA achieves a 68.5% memory consumption reduction while incurring 14% of time overhead compared with the baseline. AFMR with DR achieves a 62% memory consumption reduction with a time overhead of just 2%. AFMR achieves insignificant improvement in memory consumption reduction for `Nginx`, `Memcached`, and `Redis` uVMs. This is because these three apss have much less memory activities than `Node.js`.

The `Node.js` workload used in the first experiment completes in less than 4 seconds. We conducted a more practical evaluation of the different mechanisms of MEUNIK. We simulated a long-term periodic web usage scenario by having a web client sequentially request 500 files with different sizes from the `Node.js` web server uVM. The first request was made 30 seconds after the experiment started, and there was a 20-second idle period between each batch of file requests. The whole experiment lasted for 300 seconds and was repeated 5 times. Figure 8 shows the experiment results. As can be seen, AFMR is effective in achieving significantly more memory consumption reduction than with uVM cloning alone. SA is able to accurately predict memory activities when a new batch of requests starts and preallocate memory pages accordingly. As a result, SA maintains the memory savings achieved by AFMR while incurring fewer memory usage fluctuations.

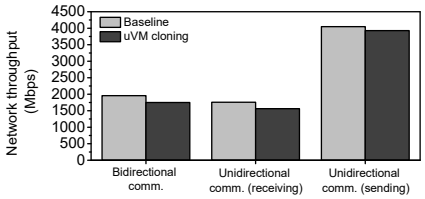

Fig. 10: Dom0-DomU network overhead.

D. LRU-based idle memory reclamation

We compared the performance of MEUNIK's LRU-based idle memory reclamation to that of a random reclamation policy. The workload used in this experiment was the same as the one described in Section IV-D. Memory reclamation for a configured percentage was performed after 9 rounds of the experiment. We measured the memory page hit rate during the 10th round of the experiment. Figure 9 shows the results of the experiment. It can be seen that the LRU implementation achieves a 100% hit rate when the reclamation percentage is less than 30%. The hit rates for 40% and 50% reclamation percentages are 89% and 75% respectively, all of which are significantly higher than the random reclamation approach.

E. Network overhead

As discussed in Section V, we addressed the issue of duplicated MAC/IP addresses in uVMs of the same type by having the network device back-end driver act as a proxy to properly substitute MAC/IP addresses for outgoing and incoming network packets. To evaluate the overhead of this approach, we compiled the `iperf` [84] server program as a uVM and ran it in a Xen DomU. We then ran the `iperf` client program in Dom0 to measure the network throughput between Dom0 and DomU. The results (Figure 10) show that the average network throughput reduction caused by our implementation is about 10%.

VII. CONCLUSION

In this paper, we demonstrated the issues of existing solutions for uVM memory management using thorough experimental results. We then proposed MEUNIK a framework for hypervisors to manage memory resources of unikernel-based VMs (uVMs). The evaluation results on our prototype system suggest that our solutions are effective and incur marginal overhead.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF Award #1943269.

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[2] G. Ammons, J. Appavoo, M. A. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. S. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski, "Libra: a library operating system for a jvm in a virtualized execution environment," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2007.

[3] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[4] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[5] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. M. Leslie, "Jitsu: Just-in-time summoning of unikernels," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[6] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici, "Unikernels everywhere: The case for elastic cdns," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2017.

[7] Y. Zhang, J. Crowcroft, D. Li, C. Zhang, H. Li, Y. Wang, K. Yu, Y. Xiong, and G. Chen, "Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, 2018.

[8] H. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *EuroSys*, 2020.

[9] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[10] P. C. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, 2019.

[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX Annual Technical Conference (ATC)*, 2018.

[12] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[13] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[14] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference (ATC)*, 2018.

[15] R. Koller and D. Williams, "Will serverless end the dominance of linux in the cloud?" in *Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[16] H. Fingler, A. Akshintala, and C. J. Rossbach, "USETL: unikernels for serverless extract transform and load why should you settle for less?" in *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2019.

[17] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2016.

[18] K. Bhardwaj, M. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2016.

[19] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: agile VM handoff for edge computing," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.

[20] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.

[21] Y. Ren, V. Nitu, G. Liu, G. Parmer, T. Wood, A. Tchana, and R. Kennedy, "Efficient, dynamic multi-tenant edge computation in edgeos," *CoRR*, 2019.

[22] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "EdgeCourier: An Edge-hosted Personal Service for Low-bandwidth Document Synchronization in Mobile Cloud Storage Services," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.

[23] Y. Bai, P. Hao, and Y. Zhang, "A Case for Web Service Bandwidth Reduction on Mobile Devices with Edge-hosted Personal Services," in *IEEE Infocom*, 2018.

[24] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. M. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[25] lsub.org, "Removing (most of) the software stack from the cloud," http://lsub.org/ls/clive.html.

[26] Galois, Inc., "Halvm," https://galois.com/project/halvm/.

[27] K. Stengel, F. Schmaus, and R. Kapitza, "Esseos: Haskell-based tailored services for the cloud," in *International Workshop on Adaptive and Reflective Middleware (ARM)*, 2013.

[28] runtimejs.org, "Javascript library operating system for the cloud," http://runtimejs.org/.

[29] erlangonxen.org, "Ling," https://erlangonxen.org/.

[30] J. Martins, M. Ahmed, C. Raiciu, V. A. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[31] rumpkernel, "Rumprun," https://github.com/rumpkernel/rumprun.

[32] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2019.

[33] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv - optimizing the operating system for virtual machines," in *USENIX Annual Technical Conference (ATC)*, 2014.

[34] C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. A. S. de Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *EuroSys*, 2014.

[35] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference (ATC)*, 2017.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SOSP*, 2003.

[37] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Linux Symposium*, 2007.

[38] Xen.org, "Xen Project Software Overview," https://wiki.xen.org/wiki/Xen_Project_Software_Overview.

[39] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[40] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *EuroSys*, 2009.

[41] J. Zhi, S. Suneja, and E. de Lara, "The case for system testing with swift hierarchical VM fork," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[42] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[43] C. Lupu, A. Albisoru, R. Nichita, D. Blânzeanu, M. Pogonaru, R. Deaconescu, and C. Raiciu, "Nephele: Extending virtualization environments for cloning unikernel-based vms," in *EuroSys*, 2023.

[44] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *USENIX NSDI*, 2020.

[45] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.

your container," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
[46] OpenStack Foundation, "Kata Containers," https://katacontainers.io/collateral/kata-containers-1pager.pdf.
[47] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Ottawa Linux Symposium (OLS)*, 2009.
[48] P. Sharma and P. Kulkarni, "Singleton: system-wide page deduplication in virtual environments," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2012.
[49] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: classification-based memory deduplication through page access characteristics," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2014.
[50] I. Banerjee, P. Moltmann, K. Tati, and R. Venkatasubramanian, "Esx memory resource management: Transparent page sharing," 2013.
[51] C. A. Waldspurger, "Memory resource management in vmware ESX server," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
[52] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
[53] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *USENIX Annual Technical Conference (ATC)*, 2009.
[54] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: more effective memory deduplication scanners through cross-layer hints," in *USENIX Annual Technical Conference (ATC)*, 2013.
[55] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.
[56] Citrix Systems, Inc., "Citrix Hypervisor VM memory," https://docs.citrix.com/en-us/citrix-hypervisor/vms/vm-memory.html.
[57] ——, "Citrix Hypervisor Configuring VM Memory," https://docs.citrix.com/en-us/xencenter/current-release/vms-memory.html#dynamic-memory-control-dmc.
[58] Jose DelaRosa, "KVM Virtualization in RHEL 7 Made Easy," Dell Linux Engineering White Paper, 2014.
[59] VMware, "VMware vSphere Documentation:Memory Virtualization Basics:Virtual Machine Memory," https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-C25A8823-F595-4322-BD0D-4FD5B081F877.html.
[60] OpenJS Foundation, "Node.js," https://nodejs.org/en/.
[61] NGINX Inc., "Nginx," https://www.nginx.com/.
[62] Danga Interactive, "Memcached," https://memcached.org/.
[63] Redis Labs, "Redis," https://redis.io/.
[64] Wikipedia, "LevelDB," https://en.wikipedia.org/wiki/LevelDB.
[65] RedisLabs, "memtier benchmark," https://github.com/RedisLabs/memtier_benchmark.
[66] Wikipedia, "Memory overcommitment," https://en.wikipedia.org/wiki/Memory_overcommitment.
[67] ——, "Memory ballooning," https://en.wikipedia.org/wiki/Memory_ballooning.
[68] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.
[69] J. Kim, V. V. Fedorov, P. V. Gratz, and A. L. N. Reddy, "Dynamic memory pressure aware ballooning," in *International Symposium on Memory Systems (MEMSYS)*, 2015.
[70] T. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *EuroSys*, 2013.
[71] P. Hahn, "VirtIO Memory Ballooning," https://pmhahn.github.io/virtio-balloon/.
[72] linux kvm.org, "Automatic Ballooning," https://www.linux-kvm.org/page/Projects/auto-ballooning.
[73] N. Amit, D. Tsafrir, and A. Schuster, "Vswapper: a memory swapper for virtualized environments," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
[74] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. V. Astsatryan, "Working set size estimation techniques in virtualized environments: One size does not fit all," *POMACS*, 2018.
[75] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.
[76] Hardkernel, "ODROID XU4," http://www.hardkernel.com/main/products/prdt_info.php.
[77] lubuntu.net, "Lubuntu," http://lubuntu.net/.
[78] Wikipedia, "Copy-on-write," https://en.wikipedia.org/wiki/Copy-on-write.
[79] Xen.org, "Xen Project 4.10 Release Notes," https://wiki.xenproject.org/wiki/Xen_Project_4.10_Release_Notes.
[80] R. T. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1999.
[81] Xen.org, "X86 Paravirtualised Memory Management," https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management.
[82] R. Arpaci-Dusseau and A. Arpaci-Dusseau, "Chapter 22: Beyond physical memory: Policies," *Operating Systems: Three Easy Pieces*, 2018.
[83] Victor Stinner, "The Python Performance Benchmark Suite," https://pyperformance.readthedocs.io/index.html.
[84] iPerf, "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/.

Authorized licensed use limited to: STATE UNIV NY BINGHAMTON. Downloaded on August 14,2025 at 20:08:14 UTC from IEEE Xplore. Restrictions apply.