
# Fast Local Page-Tables for Virtualized NUMA Servers with *vMitosis*

Ashish Panwar
Indian Institute of Science
Bangalore, India

Reto Achermann
University of British Columbia
Vancouver, BC, Canada

Arkaprava Basu
Indian Institute of Science
Bangalore, India

Abhishek Bhattacharjee
Yale University
New Haven, CT, USA

K. Gopinath
Indian Institute of Science
Bangalore, India

Jayneel Gandhi
VMware Research
Palo Alto, CA, USA

## ABSTRACT

Increasing memory heterogeneity mandates careful data placement to hide the non-uniform memory access (NUMA) effects on applications. While NUMA optimizations have focused on application data for decades, they have ignored the placement of kernel data structures due to their small memory footprint; this is evident in typical OSes that pin kernel data structures in memory. In this paper, we show that careful placement of kernel data structures is gaining importance in the context of page-tables: their sub-optimal placement causes severe slowdown (up to 3.1×) on virtualized NUMA servers.

In response, we present *vMitosis* – a system for explicit management of two-level page-tables, i.e., the guest and extended page-tables, on virtualized NUMA servers. *vMitosis* enables faster address translation by *migrating* and *replicating* page-tables. It supports two prevalent virtualization configurations: first, where the hypervisor exposes the NUMA architecture to the guest OS, and second, where such information is hidden from the guest OS. *vMitosis* is implemented in Linux/KVM, and our evaluation on a recent 1.5TiB 4-socket server shows that it effectively eliminates NUMA effects on 2D page-table walks, resulting in a speedup of 1.8 − 3.1× for Thin (single-socket) and 1.06 − 1.6× for Wide (multi-socket) workloads.

## CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Virtual memory**.

## KEYWORDS

NUMA; TLB; Linux; KVM; 2D page-tables; replication; migration

**ACM Reference Format:**
Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. Fast Local Page-Tables for Virtualized NUMA Servers with *vMitosis*. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3445814.3446709


*ASPLOS '21, April 19–23, 2021, Virtual, USA*

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446709

## 1 INTRODUCTION

Applications suffer non-uniform memory access (NUMA) latencies on modern multi-tier memory systems. As computer systems embrace even more heterogeneity in the memory subsystem, with innovations in die-stacked DRAM, high-bandwidth HBM, more socket counts and multi-chip module-based designs, the speed differences between local and remote memory continue to grow and become more complex to reason about [52, 78]. Carefully placing, replicating, and migrating data among memory devices with variable latency and bandwidth is of paramount importance to the success of these technologies, and much work remains to be done on these topics.

However, while there is at least prior work on data placement for application pages [2, 18, 24, 41, 62], kernel data structures have been largely ignored from this discussion, primarily due to their small memory footprint. Consequently, most kernel objects are pinned and unmovable in typical OS designs [33, 58, 59]. We argue that the access latency of kernel objects is gaining importance. This paper focuses on one critical kernel data structure, *the page-table*, and shows that virtualized NUMA servers must carefully reason about its placement to enable high performance.

*Why focus on page-tables?* Page-tables are vital to overall system performance. First, big-memory workloads require DRAM accesses on frequent page-table walks due to high TLB miss rates [4, 14, 30]. As system memory capacities grow to meet ever-increasing workload data demand, page-tables grow proportionally outstripping the coverage of hardware TLBs. Larger address spaces require additional levels in page-tables (e.g., Intel's 5-level page-tables). TLB misses under virtualization are already expensive–a 2D page-table walk over guest page-tables (henceforth gPT) and extended page-tables (henceforth ePT) requires up to 24 memory accesses that will increase to 35 with 5-level page-tables. Finally, a page-table walk does not benefit from memory-level parallelism as it is an inherently serial process–each long DRAM access adds latency to address translation [14].

On a single socket machine, frequent 2D page-table walks have been shown to add 10-50% execution overhead on important applications [4, 30, 44]. However, in this paper, we show that these overheads are as high as 3.1× on multi-socket machines due to sub-optimal page-table placement.

Misplacement of page-tables occurs in two use-cases on NUMA machines. First, Thin workloads/VMs (i.e., those fitting within a single NUMA socket) are occasionally migrated across NUMA sockets. Commodity OSes and hypervisors use migration to improve

resource utilization and performance isolation. Some real-world examples include VMware vSphere that performs periodic NUMA re-balancing of VMs every two seconds [65], and Linux/KVM that migrates processes to improve load-balancing and performance under memory sharing on NUMA systems [22, 74]. While data pages are often migrated along with the threads, current systems usually pin kernel objects in memory [33, 59]. Consequently, workload/VM migration can make page-tables permanently remote. Second, Wide workloads/VMs (i.e., those spanning multiple NUMA sockets) experience remote page-table walks due to a single copy of the page-table; their virtual-to-physical address translations are requested from multiple sockets but each page-table entry is local to only one of the sockets.

We highlighted the importance of careful page-table placement, for *native* systems, in our recent work *Mitosis* [1]. In contrast, in this paper, we analyze the effect of NUMA on 2D page-tables and present *vMitosis* – a system that extends the design principles of *Mitosis* to virtualized environments.

*vMitosis* provides various mechanisms to mitigate NUMA effects on 2D page-table walks for both the use-cases discussed above. Our design applies *migration* and *replication* to page-tables to ensure that TLB misses are serviced from local memory. While replication and migration are well-known NUMA management techniques, virtualization-specific challenges make their practical realization non-trivial. For instance, a hypervisor may or may not expose the host platform's NUMA topology to a VM. We refer to VMs exposed to the host NUMA topology as NUMA-visible and to VMs not exposed to such information as NUMA-oblivious.

In the NUMA-oblivious configuration, the guest OS is exposed to a flat topology in which all memory and virtual CPUs (vCPUs) are grouped in a single virtual socket. This configuration provides great flexibility to cloud service providers as NUMA-oblivious VMs can be freely migrated for maintenance, power management, or better consolidation. Further, CPUs and memory can be added to or removed from NUMA-oblivious VMs dynamically, irrespective of NUMA locality. Most of the VMs on major cloud platforms are available under this configuration [67].

In contrast, NUMA-visible VMs mirror the host NUMA topology in the guest OS. It allows performance-critical services to tune their performance; some important workloads are NUMA-aware by design (e.g., databases [45, 71]), while others leverage OS-level optimizations. NUMA-visible VMs, however, disable hypervisor features such as vCPU hot-plugging, memory ballooning, and VM migration [43, 54]. This is because the current system software stack cannot adjust NUMA topology at runtime. Thus, NUMA-visible VMs limit the resource management capabilities of the hypervisor. However, the choice of a particular configuration is use-case specific, and hence we handle both configurations in our design.

We implement our design in Linux/KVM and evaluate it on a 4-socket NUMA server with 1.5TiB memory. We show that *vMitosis* improves performance by migrating and replicating gPT and ePT. The performance improvement is $1.8 - 3.1\times$ for Thin workloads with page-table migration, and $1.06 - 1.6\times$ for Wide workloads with page-table replication. Our evaluation shows that the page-table walks of many applications become less susceptible to the effect of NUMA while using 2MiB pages. However, some applications gain up to 1.47× speedup with *vMitosis* over using 2MiB pages.

**Table 1: NUMA support for page-tables in state-of-the-art systems. (*) Replication is possible in *Mitosis* only if the server's NUMA topology is exposed to the guest OS.**

| System | Guest page-tables | | Extended page-tables | |
|---|---|---|---|---|
| | Migration | Replication | Migration | Replication |
| Linux/KVM | No | No | No | No |
| *Mitosis* | via Replication* | Yes* | No | No |
| *vMitosis* | Yes | Yes | Yes | Yes |

*Contributions over Mitosis*: Our recent proposal *Mitosis* replicates the page-tables on native systems. *vMitosis* makes several novel contributions over *Mitosis* (see Table 1). First, *Mitosis* relies on the availability of the server's NUMA topology for replicating the page-tables. In bare-metal servers, the OS extracts platform specifications from ACPI tables. However, the hardware abstract layer in virtualized systems often hides platform details from the guest OS. Therefore, *Mitosis* can replicate gPT only in the NUMA-visible VMs. *vMitosis* supports both the VM configurations; it re-uses *Mitosis* in the NUMA-visible VMs but introduces two novel techniques for replicating gPT in the NUMA-oblivious VMs (§ 3.3).

Second, *Mitosis* does not provide ePT-level optimizations; *vMitosis* does. Finally, *vMitosis* handles page-table migration differently from *Mitosis*. To migrate page-tables, *Mitosis* first replicates them on the destination socket, configures the system to use the new replica, and then releases the old replica. In contrast, *vMitosis* incrementally migrates page-tables when the OS/hypervisor migrates data pages ( § 3.2). For single-socket workloads, incremental page-table migration of *vMitosis* provides similar address translation performance as the pre-replicated page-tables of *Mitosis* but with lower space and runtime overheads.

## 2 ANALYSIS OF 2D PAGE-TABLE PLACEMENT

We start by uncovering the sources of remote DRAM accesses during page-table walks and quantifying their impact on performance with a range of memory-intensive applications listed in Table 2. We focus on classes of workloads prevalent in data processing or virtual machine deployments that experience high TLB miss rates. Further, a non-negligible fraction of their page-table accesses is serviced from DRAM (i.e., miss in the cache hierarchy) due to their random access patterns. Many other big-memory workloads are known to exhibit such characteristics [14].

To simplify the discussion, we partition workloads into two groups to separately demonstrate the two use-cases that lead to remote page-table accesses. In the first use-case (referred to as Thin) a workload executes within a single NUMA socket. In the second use-case (referred to as Wide) a scale-out workload spans multiple NUMA sockets using all the CPUs and memory of the system. Our experimental platform is a 4-socket Intel Xeon based Cascade Lake server with 96 cores and 1.5TiB of DRAM (see § 4 for more details).

### 2.1 Analysis of Thin Workloads

Thin workloads are often migrated across NUMA sockets to reduce power consumption, improve load-balancing, and optimize performance when memory sharing is possible across VMs [22, 74]. We observe that migration of VMs or workloads makes page-tables remote. We first describe how the gPT and ePT become remote, and then quantify the performance impact of remote page-tables.

**Table 2: Detailed description of the workloads.**

| Workload | Description |
|---|---|
| *Memcached* | *A multi-threaded in-memory key-value store [50]* |
| Wide | 1280GB dataset, 4B keys, 24B queries 100% reads |
| Thin | 300GB dataset, 20GB slab, 9M queries |
| *XSBench* | *Monte Carlo neutron transport compute kernel [73]* |
| Wide | 1375GB input, g=2.8M, p=75M |
| Thin | 330GB input, g=0.68M, p=15M |
| *Canneal* | *Simulates routing cost optimization in chip design [15]* |
| Wide | 380GB dataset, # 1200M elements |
| Thin | 64GB dataset, # 240M elements |
| *Graph500* | *Generation, search and validation on large graphs [6]* |
| Wide | 1280GB, scale=30, edge=52, 4 iterations |
| *Redis* | *Single-threaded in-memory key-value store [64]* |
| Thin | 300GB dataset, 0.6B keys, 100% reads. |
| *GUPS* | *Measures the rate of random in-memory updates [38]* |
| Thin | 1 thread, 64 GB input, 1B updates. |
| *BTree* | *Measures index lookup performance [77]* |
| Thin | 1 thread, 330GB input, 3.4B keys, 50M lookups. |

Consider, for example, a case where the hypervisor migrates a VM from one NUMA socket to another. In this case, the hypervisor migrates the VM's memory to the new socket with NUMA-aware data migration. During such a migration, the hypervisor also migrates the gPT since gPT pages are like any other guest data pages to a hypervisor. However, hypervisors pin ePT pages in memory and thus the ePT becomes remote after VM migration.

Alternately, if a NUMA-visible guest OS migrates one of its workloads to another virtual NUMA socket, gPT accesses become remote. This happens because kernel data structures, including page-tables, are pinned in typical OS designs today. If ePT were populated by the hypervisor on a different NUMA socket prior to workload migration, then ePT also becomes remote post migration. In long-running cloud instances, it is therefore easy to observe that any combination of local/remote gPT and ePT can arise depending on how and when workloads are migrated.

It is important to highlight that ePT may become remote even without migration. ePT pages are allocated in response to virtualized page-faults that are referred to as ePT violations. A vCPU raises an ePT violation when a required translation is absent in the ePT. If fixing a virtualized page-fault requires an ePT page allocation, the hypervisor allocates the page from the local socket of the vCPU that raised the fault. However, ePT is shared across all vCPUs of a VM. Thus, it is possible that ePT pages are allocated on one NUMA socket by a guest workload, but the same translations are later re-used by other guest workloads running on other sockets.

We quantify the performance impact of remote page-tables with seven different configurations listed in Figure 1b: the first character denotes if gPT is allocated from local (L) or one of the remote (R) sockets while the second character denotes the same for ePT. For these experiments, we modify the guest OS (Linux) and the hypervisor (KVM) to control the placement of gPT and ePT on specific sockets. The workload threads and data pages are always co-located on the same NUMA socket. This allows us to measure the NUMA effects of page-table walks in isolation.

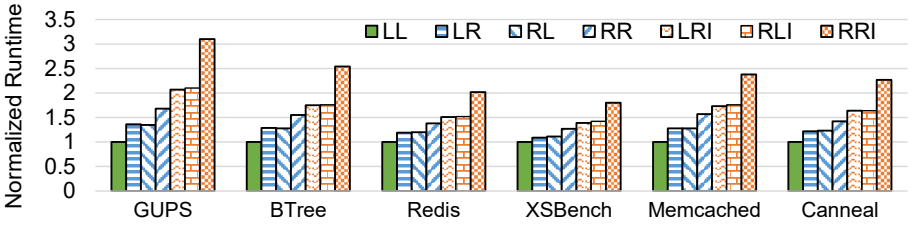
Figure 1a shows the runtime, normalized to the best-case configuration LL in which all page-tables are local. Considering the first four bars for each application, we observe that when one of the levels of page-table (LR, RL) is allocated on a remote socket, the runtime of the application increases by $1.1 - 1.4\times$: the impact of remote gPT is almost the same as remote ePT. As expected, performance drop is higher when gPT and ePT are both remote (RR).

In the experiments so far, we ensured that the remote socket is idle. This enables remote page-table accesses to experience uncontended (optimistic) latency. In a real execution scenario, however, the remote socket may be executing other independent application(s). Memory accesses from other processes would thus interfere with remote accesses to page-tables under consideration. To measure the impact of contended remote access latency, we add interference by executing STREAM micro-benchmark [49] on the remote socket. LRI, RLI, and RRI represent these configurations where ePT, gPT, or both experience contended remote accesses, respectively. As expected, the impact of remote page-table accesses is more pronounced under these configurations. In the worst case, remote page-tables can cause 1.8 - 3.1× slow down.

## 2.2 Analysis of Wide Workloads

A Wide workload uses resources from two or more NUMA sockets while sharing the same gPT and ePT. For these workloads, each translation entry is remote to all but one socket in the system, irrespective of where their page-table pages are placed. A single copy of the page-table therefore inevitably leads to remote page-table accesses for Wide workloads.

However, unlike Thin workloads, regular data accesses of Wide workloads are interspersed with their page-table walks that makes it hard to isolate the impact of remote gPT/ePT accesses. Hence, we adopt a different methodology to estimate the severity of remote page-table walks for Wide workloads: we perform an offline 2D

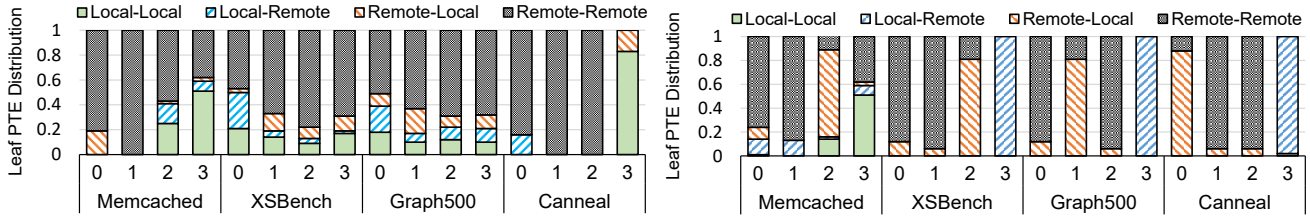



**(a) Performance impact of remote page-table walks.**

| Config | CPU | Data | gPT | ePT | Interference |
|---|---|---|---|---|---|
| LL | A | A | A | A | None |
| LR | A | A | A | B | None |
| RL | A | A | B | A | None |
| RR | A | A | B | B | None |
| LRI | A | A | A | B | B |
| RLI | A | A | B | A | B |
| RRI | A | A | B | B | B |

**(b) CPU, data, gPT and ePT placement.**

**Figure 1: Performance impact of misplaced gPT and ePT on Thin workloads (left) and details of the configurations (right). A, B represent two different sockets in the system (e.g., A=0, B=1). "I" represents interference due to a different workload.**




(a) NUMA-visible virtual instance

(b) NUMA-oblivious virtual instance

**Figure 2: Analysis of 2D page-table walk of Wide workloads on NUMA-visible and NUMA-oblivious VMs on a 4-socket machine. Bar for each socket (represented by the number) shows the fraction of 2D page-table walks that results in Local-Local, Local-Remote, Remote-Local or Remote-Remote leaf PTE access in gPT and ePT, when TLB misses are serviced for one of the threads running on that socket.**

page-table walk analysis to investigate: (1) what fraction of virtualized page-table walks results in one or more remote DRAM accesses and (2) how different configurations of virtualization, i.e., NUMA-visible and NUMA-oblivious, impact page-table placement. We discuss the methodology and summarize the observations.

Our analysis focuses on the placement of leaf page-table entries (PTEs) since their access latency dominates address translation performance; higher-level PTEs are more amenable to caching by the hardware. We run the workloads to completion and dump the gPT and ePT during their execution periodically once every 5 minutes. We analyze these dumps offline with a software 2D page-table walker. To estimate the local/remote access ratio of page-table walks, we perform address translation for each guest virtual address and record the NUMA socket on which the corresponding leaf PTEs from gPT and ePT are located. Depending on the placement of leaf PTEs, each 2D page-table walk is classified into one of the four groups: Local-Local, Local-Remote, Remote-Local and Remote-Remote. The first word denotes if the gPT leaf PTE is local or remote (for a particular socket), and the latter denotes the same for the ePT. We repeat this process on all NUMA sockets to estimate the ratio of local/remote DRAM accesses for 2D page-table walks.

Figure 2a shows the classification of 2D page-table walks in the NUMA-visible configuration. Best translation performance is expected when most page-table walks fall into Local-Local group. However, we find that $< 10\%$ page-table walks resulted in local memory accesses for both gPT and ePT. Intuitively, this is not surprising. In a system with $N$ NUMA sockets, each PTE (in either ePT or gPT) is local to only one and remote to the other $N-1$ sockets. Assuming a uniform distribution of PTEs, the probability of a 2D page-table walk resulting in local access in both levels is only $1/N^2$. Hence, on our 4-socket system, we expect only about $1/4^2 \approx 6\%$ page-table walks to fall into the Local-Local group. In fact, for a thread running on any socket, out of the 16 possible combinations of leaf PTE placement in gPT and ePT, only one configuration is Local-Local, while nine are Remote-Remote, and three in each Local-Remote and Remote-Local. Thus, more than 50% 2D page-table walks result in two remote memory accesses (one for gPT and ePT each) while more than 35% result in one remote access due to either gPT or ePT, in expectation.

Note that there can be exceptions to these observations. For example, Canneal's memory footprint is only 380GB that is slightly above the capacity of a single NUMA socket on our server (350GB). Further, it has a single-threaded memory allocation phase. Hence, almost all memory and page-tables were allocated from a single NUMA socket (i.e., Socket-3). In this case, more than 80% of the total 2D page-table walks are Local-Local for threads running on Socket-3 (25% of the total threads), while almost all page-table walks are Remote-Remote for the rest of the threads. This example also shows that the local (default) memory allocation policy can skew the placement of page-tables. Therefore, some threads may experience poorer locality than the others.

Figure 2b shows the same analysis for NUMA-oblivious VMs. In this case, Local-Local page-table walks are almost non-existent. This is not surprising–the invisibility of NUMA topology in the guest OS leads to an arbitrary placement of gPT pages. Consequently, even for a small workload like Canneal, almost all page-table walks involve at least one remote DRAM access. Hence, NUMA-oblivious deployments experience higher address translation overheads.

*Summary:* In this section, we analyzed how remote page-table accesses originate, resulting in up to 3.1× runtime overhead for Thin workloads. Moreover, we showed that a significant fraction of page-table accesses is remote for Wide workloads that span multiple NUMA sockets.

## 3 DESIGN AND IMPLEMENTATION

Our goal is to ensure that memory accesses in 2D page-table walks get serviced from local memory. We achieve this goal by applying two well-known NUMA management techniques – *migration* and *replication*. *vMitosis* supports three virtual machine configurations: one NUMA-visible (referred to as NV) and two variants of NUMA-oblivious VMs. We refer to the NUMA-oblivious variants as NO-P (para-virtualized) and NO-F (fully-virtualized). Table 3 provides a brief overview of *vMitosis* in the context of different configurations and the current state-of-the-art systems.

### 3.1 *vMitosis*: Design Overview

*Migration:* We propose page-table migration for Thin workloads. *vMitosis* takes a two-fold approach to enable the migration of gPT and ePT. First, we co-locate page-tables with data pages. Second, we integrate page-table migration with the data page migration policies of the OS/hypervisor. These mechanisms can be enabled independently in each layer.

**Table 3: Migration and replication of 2D page-tables in *vMitosis* as compared to the state-of-the-art virtualized systems.**

| | Config. | State-of-the-art | *vMitosis* |
|---|---|---|---|
| **Page-Table Migration** | NV | gPT: replicate and delete old replica in guest [1]<br>ePT: no migration | gPT: migrate incrementally with data migration in the guest OS<br>ePT: allocated to be co-located with data pages in the hypervisor |
| | NO-P<br>NO-F | gPT: migrates with data migration in the hypervisor<br>ePT: no migration | gPT: migrates with data migration in the hypervisor<br>ePT: migrate incrementally with data migration in the hypervisor |
| **Page-Table Replication** | NV | gPT: replicate in the guest OS [1]<br>ePT: no replication | gPT: replicate in the guest OS [1]<br>ePT: replicate in the hypervisor |
| | NO-P | gPT: no replication<br>ePT: no replication | gPT: replicate in the guest with hypercalls<br>ePT: replicate in the hypervisor |
| | NO-F | gPT: no replication<br>ePT: no replication | gPT: replicate in the guest OS with data migration in hypervisor<br>ePT: replicate in the hypervisor |

*Replication:* We propose page-table replication for Wide workloads and VMs. A hypervisor has direct access to the NUMA topology of the system. Therefore, ePT can be replicated by extending the *Mitosis* design [1]. For replicating gPT, the guest OS needs to: (1) know the number of NUMA sockets the VM is using, (2) allocate gPT replicas on different sockets, and (3) identify the scheduler mapping of vCPUs to NUMA sockets to load each vCPU with its local gPT replica. A NUMA-visible guest OS replicates gPT easily since NUMA topology is exposed to the guest OS. However, a NUMA-oblivious guest OS requires additional techniques to fulfill these requirements. We propose two different techniques to handle this: the first technique is based on para-virtualization while the second technique is fully-virtualized. We discuss the trade-offs involved in these techniques in § 3.3

## 3.2 *vMitosis*: Page-Table Migration

*General design:* We start by allocating page-tables from the local NUMA socket of the workload (similar to current systems) but additionally use a simple policy to determine when to migrate them. First, we maintain some metadata for each page-table page to decide whether it is placed well or needs to be migrated. Note that a page-table is a tree of physical address pointers wherein page-table entries (PTEs) in the internal levels point to the next-level page-table pages while leaf PTEs point to the application data pages. For each page-table page, we maintain an array with an entry for each NUMA socket; each array element represents the number of valid PTEs that point to its NUMA socket.

Based on this metadata, we say that a page-table page is placed well if it is co-located with most of its children. While we proactively try to allocate page-tables close to data, the system software runtime can migrate workloads and data pages at runtime. To account for dynamic scheduling activities in the system, we track page-table placement by piggybacking on PTE updates that happen in the page migration path. Since current systems use sophisticated techniques to co-locate data and threads, PTE updates due to data page migration serve timely hints to *vMitosis* to trigger the migration of page-tables as soon as they become remote. We now discuss how gPT and ePT migration works under different modes of virtualization.

*3.2.1 Page-Table Migration in NV Configuration.* If a Thin workload is running in a NUMA-visible VM, the guest OS's NUMA-aware scheduler may move the workload from one socket to another. As a result, both the gPT and the ePT may get misplaced and remain remote for the rest of the workload life cycle starting from the point of migration. For NUMA-visible VMs, we expect that the guest OS employs automatic NUMA balancing to co-locate the data and threads of its workloads. In this case, the guest OS will migrate data pages to the new socket but not the gPT.

We leverage the fact that leaf PTEs in gPT get updated when the guest OS migrates application data pages. *vMitosis* tracks these migrations and updates the counter values in the corresponding page-table pages. As soon as most of the PTEs in a leaf gPT page point to a remote socket, *vMitosis* notices the misplacement of the page and migrates it. Hence, incremental data migration in *vMitosis* automatically triggers the migration of leaf gPT pages first. Further, the migration of leaf gPT pages results in updated counter values for the internal (higher) level gPT pages that in turn triggers their migration. This way, page-table migration is automatically propagated from the leaf level to the root of the gPT tree.

ePT migration works similarly in the hypervisor. However, optimizing ePT placement requires an additional consideration. Note that a single vCPU may allocate the entire memory for its VM, e.g., when the VM boots with pre-allocated memory or a single guest thread initializes all the memory. Similar to current hypervisors, *vMitosis* allocates ePT pages on the local socket of the vCPU that requests memory. For a Wide VM, all ePT pages may therefore get consolidated on a single socket while data pages are distributed across multiple sockets. In these cases, the guest OS can migrate data pages at runtime to improve memory access locality. However, NUMA migrations of the guest OS may not be visible to the hypervisor. For example, a guest OS's data page migration does not trigger an ePT violation if the ePT entries of both the old and new data pages are already allocated. The invisibility of guest NUMA migrations can therefore lead to misplaced ePT. To handle such cases, we occasionally invoke automatic page-table migration to verify the co-location invariant and migrate misplaced ePT pages.

*3.2.2 Page-Table Migration in NO-P and NO-F Configurations.* Under NUMA-oblivious deployments, we expect the hypervisor to co-locate the threads and data pages of guest applications. Note that when the hypervisor migrates guest data pages, gPT is automatically migrated since a gPT page is like a regular VM data page for the hypervisor. Therefore, we do not need to consider a separate gPT migration mechanism for NUMA-oblivious VMs. However, ePT becomes remote if the hypervisor migrates guest

applications or the entire VM. We use the same technique here as discussed above: migration of guest physical pages trigger ePT migration in *vMitosis* from the leaf level to the top of the ePT tree. This way *vMitosis* achieves local page walks for both ePT and gPT in all configurations.

*3.2.3 Linux/KVM Implementation.* We implement ePT and gPT migration in Linux/KVM as an extension to the pre-existing automatic NUMA balancing technique called AutoNUMA [21]. AutoNUMA periodically invalidates PTEs in a process's page-table to induce minor page faults. These faults act as a hint for the OS to assess whether a remote socket dominates memory accesses for a data page. In addition to allocating page-tables from the local socket during workload initialization, we rely on AutoNUMA-driven data page migration to drive the migration of page-tables.

In our implementation, we avoid interfering with regular page-table updates by implementing page-table migration as another pass on top of AutoNUMA. To do so, we first wait for AutoNUMA to complete fixing the placement of data pages in a specific virtual address space range, and then scan the corresponding page-tables to update the counters and migrate the page-tables if necessary. This allows *vMitosis* to benefit from AutoNUMA's dynamic rate limiting heuristics that adjust the frequency of scanning based on the rate of data page migration. In the normal case where no page-table migration is needed, we rarely scan page-tables causing no interference in the common case.

To ensure correctness while migrating a gPT page, we acquire a write lock on the per-process mmap_sem semaphore. This is needed to avoid consistency issues in the presence of split page-table locks in Linux where each page-table page can be locked independently by different threads [26]. We acquire and release the lock for each gPT page migration separately to avoid latency issues. However, we do not expect this to be a performance issue as page-table migration is an infrequent operation and migrating a page-table page takes only a few microseconds. In KVM, all ePT updates are already protected by a per-VM spin lock. Therefore, *vMitosis* does not require additional locking for ePT migration.

## 3.3 *vMitosis*: Page-Table Replication

*General design:* We enable local address translation for Wide workloads by replicating their page-tables. We recently proposed page-table replication for native execution on NUMA machines in *Mitosis*. However, two levels of the page-tables and the hardware abstraction layer of the hypervisor make it non-trivial to extend *Mitosis* to virtualized environments. This subsection introduces our replication support in *vMitosis* which builds on the *Mitosis* design while highlighting the subtle differences and challenges involved in extending such a design.

We extend *Mitosis* to replicate ePT in both NUMA-visible and NUMA-oblivious configurations. In the NUMA-visible configuration, we also re-use *Mitosis* to replicate gPT. However, the gPT replication technique of *Mitosis* is insufficient for NUMA-oblivious VMs since the guest OS has no visibility into the NUMA topology. We propose two new techniques to replicate gPT for such VMs. The first technique replicates gPT via para-virtualization. In this approach, the guest OS relies on the hypervisor to identify NUMA topology, vCPU to NUMA socket mapping, and allocate gPT replicas. The second technique is fully-virtualized wherein the guest OS replicates gPT by discovering the NUMA topology and vCPU scheduling information with a micro-benchmark. Additionally, it leverages the commonly-used *"local"* memory allocation policy of the hypervisor [72] to allocate gPT replicas from different sockets.

*3.3.1 ePT Replication.* The design and implementation of ePT replication is common across all VM configurations. We introduce the following four components in the hypervisor for replicating ePT:

(1) *Allocating ePT replicas:* We extend the ePT violation handler to allocate replicas on all NUMA sockets eagerly, i.e., each ePT page allocation is followed by the allocation of its replicas. To allocate ePT replicas from the desired sockets, we introduce a per-socket "page-cache" that reserves some pages on each socket and uses them to allocate ePT pages. When the free memory pool in a NUMA socket falls below a threshold, the page-cache reclaims memory from the socket by migrating some data pages to another socket or by swapping them out.
(2) *Ensuring translation coherence:* The updates to ePT are managed solely by the hypervisor. ePT updates occur when a VM allocates a new data page or due to various hypervisor actions like page-sharing, live-migration, working set detection, etc. These updates are performed by the hypervisor on the ePT. We eagerly update all replicas when an ePT entry is modified, followed by a TLB flush to ensure translation coherence for the entire VM.
(3) *Assigning local ePT replica:* Each virtual CPU (vCPU) of a VM is managed by the hypervisor as a user-level thread scheduled on any physical CPU (pCPU). When a vCPU is scheduled, we provide it with the local ePT replica to ensure that ePT page-table walks are performed entirely within the local socket.
(4) *Preserving the semantics of access and dirty bits:* ePT is referenced by the hardware on a TLB miss. Recent architectures have also introduced access and dirty bits on ePT [39]. The hardware page-table-walker sets these bits when a physical page is accessed or modified; the hypervisor is not involved in their updates. For these bits, ePT replicas may be inconsistent since a hardware page-table walker will set them only on its local replica. However, this inconsistency does not compromise correctness. Hypervisors use these bits in various contexts e.g., to decide whether a page needs to be flushed before it can be released. To ensure correctness, we OR the value of these bits on all replicas when the hypervisor queries them; the return value is the same as it would be if all replicas were always consistent. Similarly, if the hypervisor clears the access or dirty bits, we reset them on all the replicas.

With these four components, derived from the native *Mitosis* design and adapted to virtualized systems, we enable ePT replication for virtual machines. Next, we discuss how gPT is replicated under different virtualization scenarios.

*3.3.2 gPT Replication in NV Configuration.* This is the simplest case for replicating gPT since the physical topology is exposed to the guest OS. We modify the guest OS to reserve the page-cache, allocate gPT replicas on different sockets and program each thread's page-table base register with its local replica. We leverage the open-source version of *Mitosis* [61] to replicate gPT in this configuration.

*3.3.3 gPT Replication in NO-P Configuration.* In this configuration, the NUMA topology is not visible inside the VM. The guest OS requires two main pieces of information to replicate gPT: (1) how many sockets are being used by the VM and (2) how vCPUs are scheduled on these sockets. This information is required to identify how many replicas the guest OS should allocate and configure each vCPU with its local replica. We use para-virtualization to resolve both these challenges where the guest OS relies on explicit hypervisor support as discussed below:

(1) *Identifying socket IDs of vCPUs:* The guest OS queries the physical socket ID from the hypervisor for all its vCPUs to determine how many replicas need to be allocated.
(2) *Allocating gPT page-caches on specific sockets:* The guest OS populates a per-socket "page-cache" based on the number of sockets that the VM is currently using on the host. To ensure local allocation of each page-cache on the physical server, the guest OS requests the hypervisor to *pin* these page-cache pages onto their intended sockets.

This design allows the hypervisor to perform NUMA-aware scheduling and change the vCPU to pCPU mapping. To adapt to such scheduling changes, the guest OS queries the vCPU to socket ID mapping at regular intervals and updates the vCPU to gPT replica mapping as required.

*3.3.4 gPT Replication in NO-F Configuration.* The goal is to replicate gPT entirely within the guest OS without support from the hypervisor and para-virtualization. We achieve this by exploiting two common properties of NUMA systems: (1) two hardware threads from different sockets exhibit higher communication latency compared to threads within the same socket [17, 40], and (2) OS/hypervisors commonly use local memory allocation policy wherein memory is preferably allocated from the same NUMA socket where the requesting application thread is running [72].

We exploit the first property to construct virtual NUMA groups within the guest OS using a micro-benchmark that measures the pair-wise cache-line transfer latency between all vCPUs. Based on these measurements, *vMitosis* assigns vCPUs to virtual NUMA groups such that the communication latency is low between any two vCPUs in the same group and high for any two vCPUs from different groups.

For example, consider Table 4 that shows the cost of transferring a cache line between different vCPU pairs on our experimental platform. Given this cost metric, *vMitosis* forms four groups of vCPUs (0,4,8), (1,5,9), (2,6,10), and (3,7,11) where each tuple represents a virtual NUMA group. These virtual groups have a one-to-one correspondence with our physical server topology, and hence, we identify the affinity groups of the vCPUs without relying on para-virtualization. In general, we find that the virtual NUMA groups constructed by our micro-benchmark always mirror the host topology, even under interference from other VMs and workloads.

We next leverage the hypervisor's local memory allocation policy to allocate gPT replicas from the local physical socket of each virtual NUMA group. For this, we select one vCPU from each group in the guest to allocate memory for its page-cache immediately upon boot. The vCPU allocates and accesses its page-cache to enforce page allocation in the hypervisor via ePT violations. From this point, each virtual NUMA group references its replica, and gPT replication works as discussed before. When a gPT page is released, we add it back to its original page-cache pool.

**Table 4: Time to transfer a cache line (in ns) between different vCPU pairs. The table is shown partially from the 192x192 matrix we profiled on our system.**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | 125 | 125 | 126 | **50** | 125 | 126 | 126 | **55** | 125 | 125 | 126 |
| 1 | - | - | 125 | 126 | 126 | **50** | 125 | 125 | 125 | **52** | 126 | 125 |
| 2 | - | - | - | 126 | 125 | 126 | **62** | 126 | 125 | 125 | **55** | 125 |
| 3 | - | - | - | - | 125 | 125 | 126 | **50** | 125 | 126 | 125 | **55** |
| 4 | - | - | - | - | - | 125 | 125 | 126 | **62** | 126 | 125 | 125 |
| 5 | - | - | - | - | - | - | 125 | 126 | 125 | **55** | 125 | 126 |
| 6 | - | - | - | - | - | - | - | 126 | 126 | 126 | **50** | 125 |
| 7 | - | - | - | - | - | - | - | - | 125 | 126 | 125 | **52** |

In this approach, it is possible that a replica page could not be allocated locally e.g., due to unavailability of free memory on the local socket. For these cases, we expect the hypervisor's NUMA-balancing technique to migrate misplaced replica pages to their expected sockets. Note that different NUMA groups reference different copies of gPT replicas. Therefore, all accesses for each replica page originate from the same socket. It makes it easier for the hypervisor to identify which of the gPT replica pages are misplaced (if any) and migrate them. In our evaluation, we show that the overheads of misplaced gPT replicas are moderate even in the worst case (when all gPT accesses are remote) because most gPT accesses are already remote in existing systems.

Note that replicating gPT via NO-P or NO-F involves a trade-off regarding the ease of deployment and performance guarantees. NO-P guarantees high performance by providing explicit hypervisor support to the guest OS for satisfying all the requirements of gPT replication. However, cross-layer communication makes NO-P harder to deploy. NO-F is easy to deploy but may lead to suboptimal performance in rare cases when non-local replicas get assigned to vCPUs. Our evaluation shows that NO-F and NO-P provide similar performance in the common case (see § 4.2.2).

*3.3.5 Linux/KVM Implementation.* KVM maintains a descriptor to store the attributes of each ePT page. We use the original ePT pages as the *master* copy and store references to the corresponding replicas within their descriptors. We then intercept writes to the master ePT and propagate them to all the replicas within the same acquisition of a per-VM spin lock to ensure eager consistency. If a vCPU is rescheduled to a different NUMA socket, we invalidate the old ePT for the vCPU and assign a new replica based on its new socket ID.

We replicate gPT using the open-source implementation of *Mitosis* in the NV configuration. We also use *Mitosis* as the core gPT replication engine for NO-P and NO-F configurations but augment it with two different guest modules. In NO-P, the guest module issues hypercalls to the hypervisor to determine the physical socket ID of all its vCPUs and allocate local gPT replica for each vCPU group. Under the NO-F configuration, the kernel module builds the necessary virtual NUMA groups with a micro-benchmark and allocates one replica page-cache for each group. These modules periodically update their vCPU to NUMA group mappings to adapt to hypervisor level scheduling changes.

### 3.4 Deploying *vMitosis*

*vMitosis* supports per-process/per-VM migration and replication of page-tables. Users can enable or disable page-table migration at runtime (enabled system-wide, by default), while replication requires explicit selection by the user.

It is important to highlight that that the choice of migration or replication depends on the classification of a workload as either Thin (migration) or Wide (replication). Our paper primarily focuses on "mechanisms" for various real-world scenarios. Hence, we used simple heuristics (e.g., number of requested CPUs and memory size) and user inputs (e.g., `numactl`) to classify VMs/processes as Thin or Wide. We leave investigating more sophisticated policies as future work, e.g., based on the cpuset allocation, hardware performance counters etc.

## 4 EVALUATION

We evaluate *vMitosis* on real hardware with a selection of memory-intensive workloads. We conduct page-table migration experiments for Thin (§ 4.1) and page-table replication experiments for Wide workloads (§ 4.2). We further explore the trade-offs between replication and migration (§ 4.3). In all cases, we exclude workload initialization time from performance measurements.

*Evaluation platform:* We conduct all measurements on an Intel 4x24x2 Xeon Gold 6252 (Cascade Lake) server with 1.5TiB DDR4 physical memory in total, divided equally among four NUMA sockets. The processor runs at a base frequency of 2.10 GHz with a per-socket 35.75MiB L3 cache. It contains a private two-level TLB per core with 64 and 32 L1 entries for 4KiB and 2MiB sized pages, and a unified L2 TLB with 1536 entries. We enable hyperthreading and disable turboboost.

*Software configuration:* We use Linux v4.17-*vMitosis* as both the host and the guest OS [61], and KVM as the hypervisor. We pin vCPUs to pCPUs, use `numactl` to select the memory allocation policy, and selectively enable/disable automatic NUMA-balancing and transparent huge pages (THP) depending on the configuration being tested. When enabled, THP is used in both–the guest OS and the hypervisor.

*Virtual machines:* We configure two VMs using `libvirt` for the KVM hypervisor, each with 192 vCPUs and 1.4TiB of DRAM. NUMA-visible VM divides the DRAM and vCPUs into four virtual sockets with a one-to-one mapping between physical and virtual sockets. NUMA-oblivious VM exports the entire server as a single socket system.

### 4.1 Evaluation with Page-Table Migration

This subsection focuses on Thin workloads that fit into one NUMA socket. We show that *vMitosis* mitigates the effects of remote page-table walks when the workload is migrated and scheduled on another NUMA socket.

*Evaluation methodology:* We select the NUMA-visible VM configuration to explore various page-table configurations for these experiments. In the NUMA-visible case, NUMA controls reside in the guest OS and the hypervisor maintains a 1:1 mapping between virtual to physical sockets. We focus on the worst-case situation that occurs when the guest OS migrates one of its workloads; in this case, gPT and ePT both become remote as discussed in § 2.1.

We profile the execution for five configurations listed in the table at the top of Figure 3. LL represents the best-case performance with local page-tables. RRI represents Linux/KVM where ePT and gPT are both remote after workload migration. We measure *vMitosis* in three configurations; RRI+e replicates only ePT, RRI+g replicates only gPT while RRI+M replicates both ePT and gPT. Additionally, we execute workloads with 4KiB and 2MiB pages.

*Description of results:* We show the benchmark results in Figure 3. The base case (LL) has both the ePT and gPT on the local NUMA socket. With 4KiB pages, all workloads experience performance loss when either ePT or gPT is remote. The worst-case occurs when both are remote, resulting in a slowdown of 1.8 – 3.1×. For all six workloads, *vMitosis* eliminates the overhead of remote page-table walks by migrating both levels of the page-tables (RRI+M), achieving the same performance as the best case. The effect of ePT or gPT migration is similar as RRI+e and RRI+g contribute roughly half to the overall speedup.

With THP enabled, the difference between the base case and the remote configurations is less visible due to fewer TLB misses

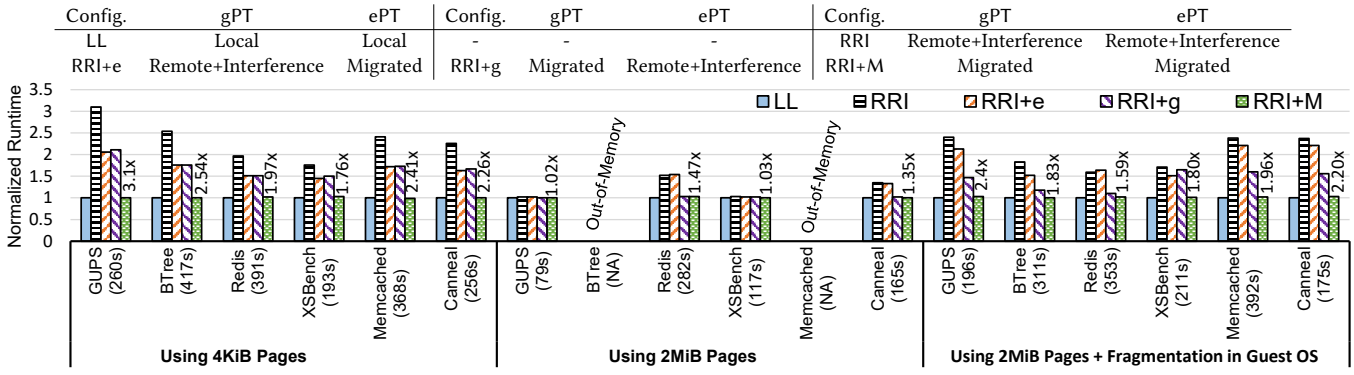
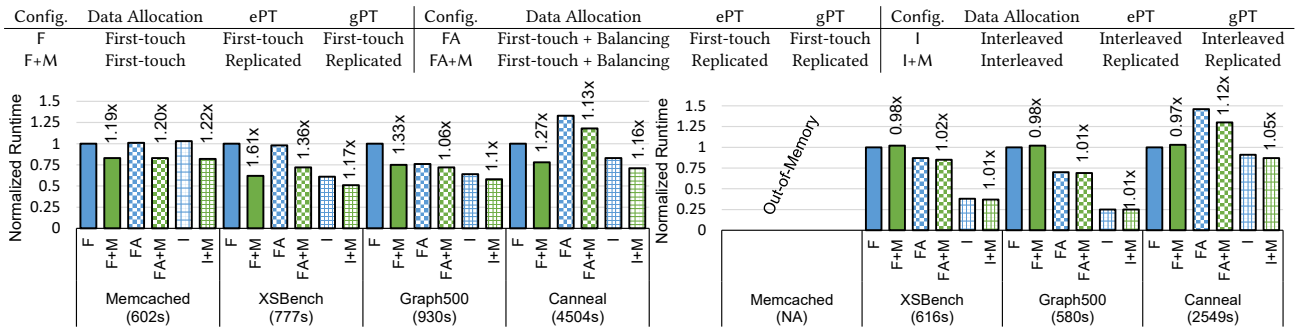



**Figure 3: Workload performance with and without ePT and gPT migration. Bars are normalized to base case (LL). Absolute runtime for the base case in brackets. Numbers at the top show speedup with *vMitosis* over the worst-case setting (RRI).**




**(a) Using 4KiB pages** **(b) Using 2MiB pages**

**Figure 4: NUMA-visible: Workload performance with and without *vMitosis*, normalized to the base case (F). Runtime (in seconds) for the base case are in brackets. Numbers at the top show speedup with *vMitosis* over the corresponding memory allocation policy of Linux/KVM.**

with 2MiB pages. Therefore, *vMitosis* provides a relatively modest speedup, except for Redis and Canneal that gain 1.47× and 1.35× improvement. Memcached and BTree result in out-of-memory (OOM) due to the well-known internal fragmentation problem with 2MiB pages that leads to memory bloat (discussed in § 5).

We also measure performance with THP where the guest OS's physical memory is fragmented. Fragmentation is well-known to limit 2MiB page allocations, increasing TLB pressure. To fragment the guest OS's memory, we first warm up the page-cache by reading two large files into memory. The total size of these files exceeds memory capacity of the socket where applications execute. We then access random offsets within these files for 20 minutes. This process randomizes the guest OS's LRU-based page-reclamation lists. When the application allocates memory, the guest OS invokes its page replacement algorithm to evict inactive pages. Since we accessed files at random offsets, the eviction usually frees up non-contiguous blocks of memory, forcing the allocator to use 4KiB pages. However, background services for compacting memory and promoting 4KiB pages into 2MiB pages remain active.

With THP enabled and fragmented guest OS, *vMitosis* recovers the performance that was lost due to the lack of 2MiB page allocations, resulting in up to 2.4× speedup. Memcached and BTree were able to complete their execution in this case since fewer 2MiB pages were allocated due to fragmentation. Both these applications also observed significant performance gain with *vMitosis*. Note that host physical memory is not fragmented and ePT maps guest physical to host physical memory with 2MiB pages. Therefore, the speed up here is lower than when both layers use 4KiB pages.

*Summary: vMitosis* effectively mitigates the slowdown caused by remote page-table walks by integrating the migration of ePT and gPT with that of data pages. Overall, *vMitosis* provides up to 3.1× speedup without THP. With THP, we gain up to 2.4× and 1.47× speedup with and without fragmentation.

## 4.2 Evaluation with Page-Table Replication

We evaluate the performance benefits of ePT and gPT replication in two settings: NUMA-visible and NUMA-oblivious.

*4.2.1 Replication in a* NUMA-Visible *Scenario.* We first measure the speedup due to gPT and ePT replication by giving guest OS access to the NUMA topology.

*Evaluation methodology:* We set up the guest OS to replicate gPT in our NUMA-visible VM and the hypervisor to replicate ePT. We execute Wide workloads (shown in Table 2) inside the VM. We use local memory allocation on the host to match guest memory mappings to the host's NUMA mappings, and pin each vCPU to a pCPU of the respective socket. We use different memory allocation policies in the guest and run each configuration with and without *vMitosis*. Moreover, we run each workload with and without THP.

The table on top of Figure 4 shows the configurations. F represents workload execution with first-touch (local) memory allocation in the guest OS while FA represents auto page migration enabled on top of first-touch memory allocation. Configuration I represents interleaved memory allocation wherein pages (including gPT and ePT pages) are allocated from all four sockets in round-robin. The *vMitosis* counterpart for each of these configurations is represented with the suffix +M. For example, configuration F+M represents replicated gPT and ePT combined with the first-touch allocation policy for data pages.

*Description of results:* Figure 4a shows the relative runtime of workload execution under six configurations, normalized to the base case (F). We show the absolute runtime (in seconds) for the base case below the workload name. Replicating both gPT and ePT with *vMitosis* provides 1.06 – 1.6× speedup without any workload changes. Performance improvements due to *vMitosis* are generally higher in configurations with local memory allocation (i.e., F and FA)–this is because local allocation leads to skewed page-table walk traffic in Linux/KVM. Even with a balanced distribution of page-tables (configuration I), replicating page-tables via *vMitosis* provides more than 1.10× speedup for all workloads.

We run the same workloads with THP enabled and show the results in Figure 4b. All workloads benefit from THP except Memcached that resulted in OOM. Further, we have found that replicating the ePT alone does not make much of a difference when THP is enabled. Except for Canneal, improvements due to *vMitosis* are negligible. Canneal gains 1.12× and 1.05× speedup in first-touch with NUMA balancing and interleaved policies, respectively.

*4.2.2 Replication in a* NUMA-Oblivious *Scenario.* We next evaluate how *vMitosis* can improve performance when the NUMA topology is not exposed to the guest OS.

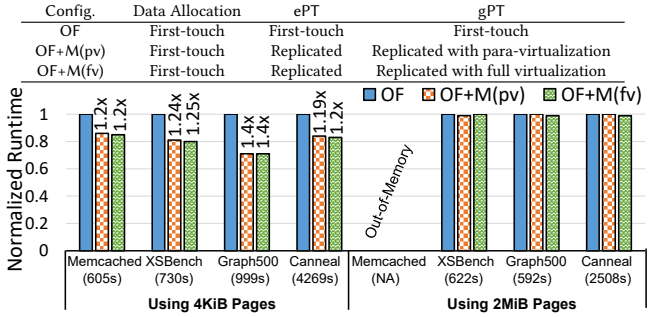



**Figure 5: NUMA-oblivious: Workload performance, normalized to the base case (OF). Runtime for the base case in brackets. Numbers at the top show speedup with *vMitosis* wherever significant.**

*Evaluation methodology:* For these experiments, we use Wide workloads in three different configurations as listed in the table on top of Figure 5 using the first-touch allocation policy at the hypervisor and pinning vCPUs to pCPUs to ensure stable performance. These settings are consistent with standard virtualization practices [72].

A NUMA-oblivious VM views the system as a single virtual socket. Therefore, only the local memory allocation policy is possible in the guest OS, unlike the NUMA-visible scenario. The base case (OF) represents vanilla Linux/KVM with first-touch memory allocation. We evaluate the two *vMitosis* variants against the baseline. Configurations OF+M(pv) and OF+M(fv) represent our para-virtualized and fully-virtualized solutions, respectively. We enable ePT replication in both the variants.

*Description of results:* Results are shown in Figure 5, normalized to the base case (OF) with its runtime beneath the workload name. All configurations benefit from *vMitosis*: ePT and gPT replication provides performance improvements of $1.16 - 1.4\times$ over the baseline using 4KiB pages. Enabling THP, we see similar performance characteristics for all configurations. Due to fewer TLB misses and reduced cache footprint of page-tables, we see a statistically insignificant speedup of up to 1% with *vMitosis*.

The performance of both *vMitosis* variants is roughly similar in all cases. This is an important result highlighting that our fully-virtualized approach of replicating gPT entirely within the guest OS can deliver similar performance as the para-virtualization based approach. Hence, in common cases, a guest OS integrated with *vMitosis* can experience the same performance benefits of gPT replication as if they were replicated with explicit hypervisor support.

*Impact of misplaced gPT replicas (4KiB pages):* We notice that our fully-virtualized approach may fail to achieve the best-case performance in some cases. While a *vMitosis* enabled guest OS can always discover the NUMA mappings of its vCPU's, the placement of gPT replica pages depends on the state of the hypervisor. Therefore, if the hypervisor fails to allocate gPT replicas from a vCPU's local socket (e.g., if free memory is not available), *vMitosis* may assign non-local gPT pages to some vCPUs. While we expect these cases to be rare, we evaluate the worst-case overhead of non-local replicas in our fully-virtualized approach OF+M(fv).

For these experiments, we artificially create a situation that mimics misplaced gPT replicas by configuring each thread's page-table base register *cr3* to point to one of the remote replicas. For example, we configure threads running on socket-0 to use socket-1's copy of the gPT and so on. This leads to 100% remote memory accesses for gPT. We then evaluate performance with and without replicating ePT.

Disabling ePT replication isolates the impact of misplaced gPT replicas. Our experiments showed a moderate 2%, 4%, and 5% slowdown over Linux/KVM for Graph500, XSBench, and Memcached, respectively. These results are in-line with our expectations: we do not expect high overheads since non-replicated page-tables in Linux/KVM already result in about 75% remote gPT accesses on a 4-socket system, on average. If ePT replication is enabled, *vMitosis* outperforms Linux/KVM even if all gPT replicas are misplaced. This is also expected since *vMitosis* reduces the overall number of remote page-walk accesses in this case – misplaced gPT adds 25% remote accesses but replicated ePT eliminates 75% remote accesses, on average.

*Summary:* We have shown that *vMitosis* improves application performance by replicating gPT and ePT in both NUMA-visible and NUMA-oblivious configurations. While 2MiB pages perform well even without replicated page-tables, they are susceptible to out-of-memory errors due to internal fragmentation.

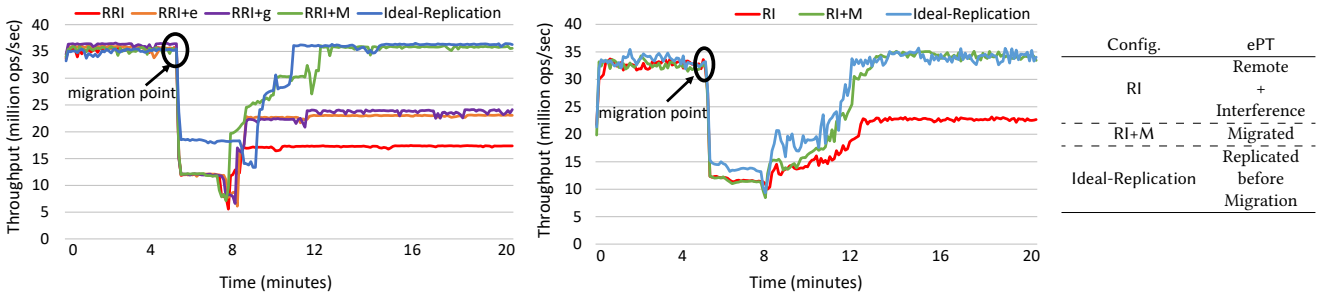
## 4.3 Replication vs. Migration of Page-Tables

In § 4.1, we profiled execution in a static setting by allocating data and page-tables on different sockets and later migrating page-tables close to data. However, scheduling policies of the OS/hypervisor have a dynamic effect on the placement of page-tables. In this subsection, we demonstrate a live migration example using Memcached as a representative workload. We demonstrate live migration in both the guest OS and the hypervisor, and compare the effect of page-table replication and migration in both these cases.

*Evaluation methodology:* For this evaluation, we select a Thin Memcached instance with a 30GiB dataset and execute it in both NUMA-oblivious and NUMA-visible configurations. We initialize the cache and start querying the key-value store while measuring the throughput over time. After about five minutes, we migrate Memcached from Socket-0 to Socket-1. At this point, all memory accesses become remote for a few minutes until NUMA balancing starts migrating data pages.

*Description of results:* We first consider the NUMA-visible case (Figure 6a) and evaluate five configurations. Configurations RRI, RRI+e, RRI+g and RRIM+M are similar to the ones used in § 4.1. Ideal-Replication represents pre-replicated page-tables wherein page-table accesses are always local.

Initially, all five configurations operate at 35M operations per second, and then they experience a sharp drop in the throughput after Memcached gets migrated to another NUMA socket. In vanilla Linux/KVM (RRI) even after NUMA balancing has co-located the dataset of Memcached, the throughput is restored to only about 50% of its pre-migration level. When either ePT or gPT migration is enabled via *vMitosis* (RRI+e or RRI+g), we experience a similar pattern initially but reach 65% of the initial throughput in a




**(a) Workload migration by a NUMA-visible guest OS** **(b) VM migration by KVM hypervisor**

**Figure 6: Throughput of a Thin Memcached instance before, during and after migration. In the NUMA-visible case (a), the guest OS migrates Memcached. In the NUMA-oblivious case (b), the hypervisor migrates Memcached's VM.**

few minutes. The best outcome is obtained with the migration of both ePT and gPT (RRI+M), where 100% of the throughput is regained. While the initial drop is less dramatic when using ideal pre-replicated page-tables, our page-table migration technique also quickly restores the throughput by migrating page-tables with data pages. In the long run, *vMitosis* achieves the same behavior as ideal page-table replication.

In the NUMA-oblivious case, the hypervisor-level NUMA balancing migrates guest data pages as well as the gPT when it migrates the VM. Hence, there are only three configurations in Figure 6b. Configuration RI represents the baseline Linux/KVM system where ePT is remote after VM migration. We evaluate *vMitosis* in two configuration where RI+M represents *vMitosis* with ePT migration and Ideal-Replication represents pre-replicated ePT.

Since gPT is automatically migrated by the hypervisor, the loss in Memcached's throughput in Linux/KVM in Figure 6b (RI) is lesser than the loss in the NUMA-visible case of Figure 6a (RRI): RI experiences ≈ 35% drop (local gPT, remote ePT) compared to 50% of RRI (remote gPT, remote ePT). However, this is still a significant performance loss considering that VM migration has completed from the system point of view. *vMitosis* restores the full performance by migrating ePT (RI+M); this behavior is also close to an ideal ePT replication scenario.

## 4.4 Memory and Runtime Overheads

We quantify the runtime overhead of our implementation with a micro-benchmark that invokes common memory management related system calls mmap, mprotect, and munmap – similar to the methodology discussed in *Mitosis* [1]. The micro-benchmark repeatedly invokes these system calls with different virtual memory region sizes. We measure throughput as the number of PTEs updated per second for each system call when invoked at 4KiB, 4MiB and 4GiB granularity on three system configurations: Linux/KVM, *vMitosis* with migration, and *vMitosis* with replication. In *vMitosis* configurations, replication or migration of ePT and gPT is enabled simultaneously. Table 5 shows our measurements. We make the following observations based on these results.

First, the exact overhead of replication depends on the specifics of a particular system call. For example, the cost of mmap and munmap system calls is dominated by the time taken to allocate and free pages, respectively. In contrast, mprotect only updates certain page-table bit, and therefore experiences significantly higher overhead due to replication. The overhead also depends on the size of the memory region. For smaller virtual memory area per system call, the overhead of context switching dominates that of updating page-table replicas. Hence, the overheads are low in the 4KiB case but more pronounced in the 4MiB and 4GiB cases.

Secondly, Linux/KVM and *vMitosis* (in its default migration mode) both maintain a single copy of the page-tables. Hence, their throughput is roughly similar in all cases. Thin workloads, therefore, do not experience runtime overhead in *vMitosis* before or after migration or when they are never migrated across sockets. This is an important benefit that justifies the value of integrating page-table migration with that of data pages. In contrast, replication-based page-table migration (as in *Mitosis*) involves expensive page-table updates. Furthermore, the overhead of replication increases linearly with the number of replicas.

Third, through separate profiling of these overheads for ePT and gPT, we observe that the cost of updating gPT replicas dominates the overall replication overheads. In general, ePT updates are infrequent – ePT is updated when memory pages are first allocated to a VM which is a one time operation in the common case. Furthermore, the cost of updating ePT is dominated by VM-exits. Hence, ePT replication contributes only a marginal overhead in Table 5.

**Table 5: Throughput (measured as million PTEs updated per second) of different system calls when invoked with different virtual memory region sizes using 4KiB mappings. Numbers in parentheses represent throughput normalized to Linux/KVM.**

| Syscall | Size | Linux/KVM | *vMitosis* (migration) | *vMitosis* (replication) |
|---|---|---|---|---|
| **mmap** | 4KiB | 0.44 | 0.44 (1.0×) | 0.40 (0.91×) |
| | 4MiB | 1.10 | 1.10 (1.0×) | 1.08 (0.98×) |
| | 4GiB | 1.11 | 1.10 (1.0×) | 1.08 (0.98×) |
| **mprotect** | 4KiB | 0.82 | 0.83 (1.01×) | 0.69 (0.84×) |
| | 4MiB | 30.88 | 31.34 (1.01×) | 9.05 (0.29×) |
| | 4GiB | 31.82 | 31.81 (1.0×) | 8.97 (0.28×) |
| **munmap** | 4KiB | 0.34 | 0.34 (1.0×) | 0.30 (0.88×) |
| | 4MiB | 6.40 | 6.60 (1.03×) | 4.92 (0.75×) |
| | 4GiB | 6.62 | 6.64 (1.0×) | 4.75 (0.72×) |

**Table 6: Memory footprint of 2D page-tables for a 1.5TiB workload using 4KiB pages with different replication factors. Numbers in parentheses represent memory consumption of page-tables as a fraction of workload size.**

| # replicas | ePT | gPT | Total |
|---|---|---|---|
| 1 | 3GB | 3GB | 6GB (0.4%) |
| 2 | 6GB | 6GB | 12GB (0.8%) |
| 4 | 12GB | 12GB | 24GB (1.6%) |

Finally, Table 6 shows the space overhead of *vMitosis* with different replication factors for a representative 1.5TiB workload using 4KiB pages. For a typical densely populated address space, page-tables consume a small fraction of overall memory (i.e., 0.2% – since a 4KiB page-table page maps 2MiB of address space). Therefore, each 2D page-table replica adds 0.4% memory overhead on virtualized systems, resulting in an overall 1.2% memory overhead on our four-socket system. With 2MiB large pages, the space overhead of 4-way replication reduces to 36MiB i.e., a negligible 0.003% of workload memory footprint. Overall, these space overheads are moderate compared to the performance benefits of *vMitosis*.

### 4.5 Summary of Results

We have shown that *vMitosis* fully mitigates the overheads of remote page-table walks providing $1.8 - 3.1\times$ speedup for Thin workloads. It improves the performance of Wide workloads by $1.06 - 1.6\times$ in the NUMA-visible case and by $1.16 - 1.4\times$ in the NUMA-oblivious case. We have also shown that *vMitosis* can restore the performance of the Memcached server to its initial state within a few minutes of VM/workload migration. In a few cases, *vMitosis* provides significant improvement over Linux THP.

## 5 DISCUSSION

### 5.1 Large Pages

Large pages can effectively reduce page-table walk overheads under ideal conditions, and hence our improvements over Linux THP are moderate. However, integration of large pages into existing systems has been extremely challenging [20, 23]. The difficulty arises due to some fundamental properties of large pages including but not limited to memory bloat due to internal fragmentation [27, 70], unbalanced memory traffic on NUMA systems due to coarse granularity of data placement [32], latency and OS jitter due to systems overhead involved in managing large pages [3, 37], their negative impact on memory consolidation for VMs [34], and having to deal with the inevitable external memory fragmentation problem in long-running systems [33]. Large page support on heterogeneous platforms adds further complexity in memory management [7, 8]. While researchers have tried to overcome these challenges [44, 55, 57, 59, 79], the magnitude of the problem has forced many application developers to recommend against using THP [25, 53, 63]. Consequently, the need for large page alternatives is growing [5, 12, 35, 36, 42, 47, 48, 66, 68]. In this paper, we have only shown THP causing out-of-memory (OOM) error for Memcached and BTree due to memory bloat, and their below-par performance in a fragmented system. Readers are referred to well-documented literature on other THP-related problems (cited above).

However, we do not claim that large pages are useless as many important applications benefit from them. We only point out that their utility is use-case specific, and that *vMitosis* provide significant performance boost when large pages are unsuitable. Further, *vMitosis* is compatible with Linux THP and even improves performance when used in tandem.

### 5.2 Shadow Page-Tables

Under virtualization, address translation overheads can be reduced by replacing 2D page-tables with shadow page-tables. In shadow paging, hypervisor-managed shadow page-tables translate guest virtual addresses directly to host physical addresses [76]. This reduces the maximium number of memory references involved in an address translation to only four (similar to native systems), as compared to 24 with two-level paging. However, shadow page-tables must be kept consistent with guest page-tables; for this, a typical hypervisor write protects gPT pages and applies gPT modifications to its shadow page-tables. This process involves an expensive VM exit on every gPT update. Shadow page-tables, therefore, involve a complicated trade-off i.e., optimizing hardware page-table walks at the expense of high software memory management overheads.

Research has shown that TLB-intensive workloads that allocate memory once can benefit from shadow-paging [30]. *vMitosis* supports migration and replication of shadow page-tables in KVM. Our experiences with shadow page-tables have been mixed. In the best-case (when page-table updates are infrequent), shadow paging combined with migration and replication with *vMitosis* improves performance by up to 2× over 2D page-tables, at the expense of $2 - 6\times$ higher initialization time. In the worst-case, shadow paging degraded performance by more than 5×. We also observed extreme overheads due to guest kernel's services that update page-tables (e.g., some of the workloads did not complete even in 24 hours when we enabled AutoNUMA in the guest OS). In general, shadow paging combined with the techniques of *vMitosis* could be useful for workloads that involve little kernel activities (e.g., HPC applications). The use of shadow paging in its current form does not seem fruitful, and consequently some hypervisors have abandoned it. However, techniques that exploit the best of shadow and extended paging have been explored in the literature [9, 30]; combined with *vMitosis*, such techniques could prove to be more powerful on heterogeneous memory systems.

## 6 RELATED WORK

Technique to improve NUMA systems performance and address translation optimizations have been explored extensively in the literature, albeit independently.

*NUMA optimizations:* OS-level NUMA optimizations have been proposed for decades [16, 75]. Applications with complex NUMA properties can directly leverage system calls to retain control over the placement of their threads and data [46]. Other applications can rely on built-in system heuristics that automatically drive NUMA optimizations [21, 28, 56]. While OS and library support thus far has focused only on user data pages, we have shown that the placement of important kernel structures such as page-tables is also crucial for performance.

Replication and migration are popular techniques that have been applied in various contexts [69]. Carrefour is a kernel design that transparently replicates and migrates application data on NUMA systems [24]. Shoal provides an abstraction to replicate, partition, and distribute arrays across NUMA domains. Carrefour-LP optimized large page performance on NUMA systems by judiciously allocating/demoting large pages [32]. RadixVM used replication to improve the scalability of the xv6 kernel [19]. Various locking techniques also employ replication for reducing cross-CPU traffic to achieve high scalability [18]. In contrast, we apply replication and migration to improve address translation performance. Our technique of discovering the server's NUMA topology is inspired by Smelt that derived efficient communication patterns on multi-core platforms via online measurements [40].

*Address translation optimizations:* These include techniques aimed at minimizing TLB misses [14, 31, 35], reducing page-table walk lengths [13], or avoiding memory lookups on TLB misses [10]. Some of these techniques have been incorporated in commercial systems in various forms including large multi-level TLBs [51], large pages [44, 55], and hardware page-walk caches [11]. Prior research has shown that despite these optimizations, large scale systems are still susceptible to page-table walk latency [12, 44, 57].

Some of the other techniques for reducing TLB miss overheads include contiguity-aware TLBs that increase translation coverage without increasing TLB or page size [13, 60] and direct segments that try to eliminate TLB lookups for most memory accesses [12]. Various application specific and machine learning techniques have also been explored for accelerating address translation [4, 47]. Especially for virtualized systems, POM-TLBs were proposed to resolve TLB misses using a single memory lookup using an in-memory TLB structure [66] while virtualized direct segments were used to translate guest virtual addresses directly to host physical addresses [29]. OS-level address translation optimizations have explored various policies to overcome the fundamental limitations of large pages [44, 55, 57, 59, 79]. Different from prior techniques, *vMitosis* explores heterogeneity on virtualized page-table walks. While prior techniques are effective, *vMitosis* can be used to further improve their efficiency on NUMA-like systems.

## 7 CONCLUSION

We highlight that the placement of kernel objects is becoming important in NUMA systems. Our detailed analysis on a real platform shows that remote DRAM accesses due to misplaced guest and extended page-tables cause up to 3.1× slowdown. We present *vMitosis* – a system design to explicitly manage 2D page-tables in different virtualized environments. *vMitosis* leverages well-known replication and migration techniques and effectively eliminates NUMA effects on 2D page-table walks.

## ACKNOWLEDGMENTS

We thank our anonymous reviewers and our shepherd Ding Yuan for their thoughtful feedback. This work was partly done when Ashish Panwar and Reto Achermann were interns at VMware Research. Ashish Panwar is also supported by the Prime Minister's Fellowship Scheme for Doctoral Research, co-sponsored by CII, Government of India and Microsoft Research Lab India. Abhishek Bhattacharjee is supported by NSF career award number 1916817. Arkaprava Basu's research is partially supported by Pratiksha Trust, Bangalore, and by a research grant from VMware.

## A ARTIFACT APPENDIX

### A.1 Abstract

Our artifact provides x86_64 binaries of our modified Linux kernel v4.17 and KVM hypervisor, user-space control libraries (libnuma) and evaluated benchmarks with their input files where appropriate. We further provide source code of our Linux modifications, benchmarks, user-space libraries and scripts to compile the binaries.

The exact invocation arguments and measurement infrastructure is provided through bash and python scripts which allow reproducing the data and graphs in the paper on a four-socket Intel Cascade Lake (or similar) machine with 1.5TiB of main memory.

### A.2 Artifact Check-List (Meta-information)

- **Algorithm:** Migration and replication of the Linux guest OS kernel and KVM extended page-tables.
- **Programs:** GUPS, BTree, XSBench, Graph500, Redis, Memcached, Canneal.
- **Compilation:** GCC version 7.4.0.
- **Binary:** Included for x86_64. In addition, source code and scripts are provided to compile binaries.
- **Data set:** A netlist is required for Canneal which is automatically generated by the run scripts.
- **Run-time environment:** Provided by the supplied Linux kernel binaries for x86_64 hardware, source code given. Scripts require sudo privileges.
- **Hardware:** We recommend a four-socket Intel Xeon Gold 6252 with 24 cores (48 threads) and 384GB memory per-socket (1.5TB total memory) to reproduce results reported in the paper. Other four-socket x86_64 servers with similar memory and compute capability are expected to produce comparable results.
- **Run-time state:** Populated by the workloads themselves.
- **Execution:** Using bash-scripts on a Linux/KVM platform. Scripts to be executed on the host.
- **Output:** The artifact scripts produce the graphs for each figure used in the paper.
- **Disk space required:** 600GB including all datasets. 200GB without fragmentation related experiments.
- **Time needed to prepare workflow:** 2-3 hours.
- **Time needed to complete experiments:** 2 weeks excluding Canneal. 6-7 weeks including Canneal.
- **Publicly available:** Yes.
- **Workflow framework used:** None.
- **Archived:** Yes. DOI: 10.5281/zenodo.4321310

### A.3 Description

*A.3.1 Availability.* All scripts are available in the GitHub repository https://github.com/mitosis-project/vmitosis-asplos21-artifact. All sources including Linux/KVM modifications, benchmarks and other utilities are included as public git submodules. The artifact with pre-compiled binaries is also available at https://zenodo.org/record/4321310.

*A.3.2 Hardware Dependencies.* We recommend a four-socket Intel Xeon Gold 6252 with 24 cores and 384GB memory per-socket to reproduce results reported in the paper. Other multi-socket x86_64 servers with similar memory and compute capability should produce comparable results. Memory sizes are hardcoded in the binaries but they can be adjusted by modifying the workload source, if required.

*A.3.3 Software Dependencies.* The compilation environment and our provided binaries and scripts assume Ubuntu 18.04 LTS, which also uses the Linux Kernel v4.17. Similar Linux distributions are also expected to work. In addition to the packages shipped with Ubunty 18.04 LTS, additional packages need to be installed as follows:

```
$ sudo apt-get install build-essential flex \
        libncurses-dev bison libssl-dev \
        libelf-dev libnuma-dev python3 git \
        python3-pip python3-matplotlib \
        python3-numpy wget kernel-package \
        fakeroot ccache libncurses5-dev \
        pandoc libevent-dev libreadline-dev \
        python3-setuptools qemu-kvm virtinst \
        bridge-utils libvirt-bin virt-manager
```

*A.3.4 Datasets.* The datasets (required only for Canneal) are automatically generated when executing the run scripts. The scripts to generate them are present in `./datasets/`.

## A.4 Installation

To install, either download the complete artifact from Zenodo (https://zenodo.org/record/4321310) (DOI 10.5281/zenodo.4321310), or clone the GitHub repository from https://github.com/mitosis-project/vmitosis-asplos21-artifact. The GitHub repository contains all the scripts required to run the artifact along with all pre-compiled binaries. All sources are included as public submodules in `./sources/`. Pre-compiled binaries are placed in `./precompiled/`.

*A.4.1 Compiling Binaries.* If you plan to use the pre-compiled binaries, you can skip this step. Otherwise use the following commands to compile binaries from the sources:

```
$ cd vmitosis-asplos21-artifact/
$ git submodule init
$ git submodule update
$ make
```

*A.4.2 Deployment.* If you are running everything on the test machine, you can skip this step. Alternatively, you can deploy the artifact from your local system to the target machine under test. To do so, set your target host-name and directory in `./scripts/configs.sh` and deploy the artifact to the test machine as follows:

```
$ ./scripts/deploy.sh
```

*A.4.3 Installing vMitosis-Linux.* On your test machine, compile and install the vmlinux binary from `./sources/vmitosis-linux/` and boot from it. Alternatively, use Debian packages provided in `./precompiled/` to install vMitosis-Linux headers and kernel image on your machine under test.

*A.4.4 Installing and Configuring a Virtual Machine.* Install a virtual machine using libvirt on your test machine. An example using command line installation is provided below (choose ssh-server when prompted for package installation). Once installed, run the second command to generate three VM configurations (i.e., NUMA-visible, NUMA-oblivious and Thin). The appropriate configuration will be loaded by run scripts themselves.

```
$ virt-install --name vmitosis --ram 4096 \
   --disk path=~/vmitosis.qcow2,size=50 \
   --vcpus 4 --os-type linux --os-variant \
   generic --network bridge=virbr0  \
   --graphics none --console pty, \
   target_type=serial --location `URL' \
  --extra-args `console=ttyS0,115200n8 serial'

 $ ./scripts/gen_vmconfigs.py vmitosis
```

We recommend installing Ubuntu18.04 in the VM. To do so, replace URL with http://archive.ubuntu.com/ubuntu/dists/bionic/main/installer-amd64/ in the installation command above. In addition, do the following:

- In `./scripts/configs.sh`, edit user names (HOSTUSER, GUESTUSER), IP addresses (HOSTADDR, GUESTADDR) and `libvirt`'s ID of the VM image (VMIMAGE) as per your installation.
- Setup password-less authentication between guest and host. This can be done, for example, by adding the RSA key of the host user in `$HOME/.ssh/authorized_keys` in the guest and vice-versa.
- Set up guest Linux to auto-mount the artifact directory in the same path as the host using a network file system such as SSHFS (e.g., user's home directory).
- Grant sudo privileges to users in both host and guest; they should be able to execute `sudo` without entering password.
- Add your user name to `libvirt` group in the host.

If you face any issues (e.g., VM fails to boot) with configuration files generated by `./scripts/gen_vmconfigs.py`, create three VM configurations in `./vmconfigs/` manually – following detailed instructions provided in README.md on GitHub repository. Refer to `./vmconfigs/samples/` for XML configurations that were used in the paper.

## A.5 Experimental Workflow

We provide individual scripts for each figure in the paper, and a common script to launch all of them in one go. Once you have prepared the target machine, you can execute the workloads as described below.

*Running experiments:* We advise to run the experiments exclusively on a machine i.e., no other compute or memory-intensive application should be running concurrently. To run all experiments, individual figures or individual bars in the figures, execute:

```
$ ./scripts/run_all.sh
          OR
$ ./scripts/run_figure-{1..6}.sh
          OR
$ ./scripts/run_figure-{1..6}.sh BENCH CONFIG
```

Refer to individual scripts for the list of supported benchmarks and configurations. The output logs are redirected to `./evaluation/measured/data/`, in a sub-directory named after the benchmark. They can be processed into CSV files and PDF plots by executing:

```
$ ./scripts/compile_report.py --all
```

Processing 2D page-table dumps for Figure-2 takes some time (more than 20 minutes for each configuration). To avoid processing them while compiling the report, remove the `--all` argument. The final report will be generated in `./vmitosis-artifact-report/`.

## A.6 Evaluation and Expected Results

Once you've completed *all* or *partial* experiments, you can compare the outcomes with the expected results. The reference output logs that were used in the paper are located in the folder `./evaluation/reference/data/`. They will also be processed into CSV files and PDF plots while compiling the report.

*Collecting results:* Copy the final report to your local machine. If you used the deploy script to copy this artifact to a different target machine, you can collect the report by executing:

```
$ ./scripts/collect-results.sh
```

Finally, open `./vmitosis-artifact-report/vmitosis.html` in your web browser to see the reference and measured plots side-by-side. Additionally, you can also check the CSV files of each figure (both reference and measured) in the report.

## REFERENCES


[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland, 2020) *(ASPLOS '20)*. Lausanne, Switzerland, 283–300. https://doi.org/10.1145/3373376.3378468

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Xi'an, China, 631–644. https://doi.org/10.1145/3037697.3037706

[3] Ibrar Ahmed. 2019. Settling the Myth of Transparent HugePages for Databases. Online https://www.percona.com/blog/2019/03/06/settling-the-myth-of-transparent-hugepages-for-databases/.

[4] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Toronto, ON, Canada, 457–468. https://doi.org/10.1145/3079856.3080209

[5] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios I. Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. 515–528. https://doi.org/10.1109/ISCA45697.2020.00050

[6] James Ang, Brian Barrett, Kyle Wheeler, and Richard Murphy. 2010. Introducing the Graph500. https://graph500.org/

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Cambridge, Massachusetts, 136–150. https://doi.org/10.1145/3123939.3123975

[8] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Williamsburg, VA, USA, 503–518. https://doi.org/10.1145/3173162.3173169

[9] Chang S. Bae, John R. Lange, and Peter A. Dinda. 2011. Enhancing Virtualized Application Performance through Dynamic Adaptive Paging Mode Selection. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. Karlsruhe, Germany, 255–264. https://doi.org/10.1145/1998582.1998639

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. Saint-Malo, France, 48–59. https://doi.org/10.1145/1815961.1815970

[11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. San Jose, CA, USA, 307–317. https://doi.org/10.1145/2000064.2000101

[12] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Tel-Aviv, Israel, 237–248. https://doi.org/10.1145/2485922.2485943

[13] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. Davis, California, 383–394. https://doi.org/10.1145/2540708.2540741

[14] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Xi'an, China, 63–76. https://doi.org/10.1145/3037697.3037705

[15] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[16] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. 1989. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Litchfield Park, AZ, USA, 19–31. https://doi.org/10.1145/74850.74854

[17] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-Aware Reader-Writer Locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Shenzhen, China, 157–166. https://doi.org/10.1145/2442516.2442532

[18] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Xi'an, China, 207–221. https://doi.org/10.1145/3037697.3037721

[19] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. Prague, Czech Republic, 211–224. https://doi.org/10.1145/2465351.2465373

[20] Jonathan Corbet. 2007. Large pages, large blocks, and large problems. Online https://lwn.net/Articles/250335/.

[21] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/.

[22] Jonathan Corbet. 2014. NUMA scheduling progress. Online https://lwn.net/Articles/568870/.

[23] Jonathan Corbet. 2019. Transparent huge pages, NUMA locality, and performance regressions. Online https://lwn.net/Articles/787434/.

[24] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Houston, Texas, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[25] Advanced Micro Devices. 2012. Hadoop Performance Tuning Guide. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[26] Linux Kernel Documentation. 2020. Split page table lock. Online https://www.kernel.org/doc/html/latest/vm/split_page_table_lock.html.

[27] Nelson Elhage. 2010. Disable Transparent Hugepages. https://blog.nelhage.com/post/transparent-hugepages/.

[28] FreeBSD [n.d.]. FreeBSD - NUMA. https://wiki.freebsd.org/NUMA.

[29] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Cambridge, United Kingdom, 178–189. https://doi.org/10.1109/MICRO.2014.37

[30] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. Seoul, Republic of Korea, 707–718. https://doi.org/10.1109/ISCA.2016.67

[31] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrian Cristal, Mark Hill, Kathryn McKinley, Mario Nemirovsky, Michael Swift, and Osman Unsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (05 2016). https://doi.org/10.1109/MM.2016.10

[32] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. Philadelphia, PA, 231–242. https://www.usenix.org/node/183962

[33] Mel Gorman and Patrick Healy. 2008. Supporting Superpage Allocation Without Additional Hardware Support. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. Tucson, AZ, USA, 41–50. https://doi.org/10.1145/1375634.1375641

[34] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. Istanbul, Turkey, 39–51. https://doi.org/10.1145/2731186.2731187

[35] Faruk Guvenilir and Yale N. Patt. 2020. Tailored Page Sizes. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. Virtual Event, 900–912. https://doi.org/10.1109/ISCA45697.2020.00078

[36] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Williamsburg, VA, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[37] Todd Hoff. 2015. The Black Magic Of Systematically Reducing Linux OS Jitter. http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html.

[38] HPCCHALLENGE. 2019. RandomAccess: GUPS (Giga Updates Per Second). https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/.

[39] Intel. 2020. Intel® 64 and IA-32 Architectures Developer's Manual, Vol. 3C. Online https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html.

[40] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. 2016. Machine-Aware Atomic Broadcast Trees for Multicores. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA, 33–48. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kaestle

[41] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Toronto, ON, Canada, 521–534. https://doi.org/10.1145/3079856.3080245

[42] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Portland, Oregon, 66–78. https://doi.org/10.1145/2749469.2749471

[43] Joe Kozlowicz. 2018. Checking Your Virtual Machine NUMA Configuration. Online https://www.greenhousedata.com/blog/dont-turn-numa-numa-yay-into-numa-numa-nay-checking-virtual-machine-numa.

[44] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA, 705–721. https://doi.org/10.1145/3139645.3139659

[45] Kieran Laffan. 2020. SQL Server Best Practices, Part I: Configuration. Online https://www.varonis.com/blog/sql-server-best-practices-part-configuration/.

[46] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages. https://doi.org/10.1145/2508834.2513149

[47] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2018. Virtual Address Translation via Learned Page Table Indexes. In *Proceedings of the Workshop on ML for Systems at NeurIPS*.

[48] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Columbus, OH, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[49] John D. McCalpin. 2019. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/.

[50] memcached. 2019. memcached: a distributed memory object caching system. https://memcached.org.

[51] Timothy Merrifield and H. Reza Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. Atlanta, Georgia, USA, 25–35. https://doi.org/10.1145/2892242.2892258

[52] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (Monte Verità, Switzerland) *(HotOS'09)*. Monte Verità, Switzerland, 14.

[53] MongoDB, Inc. [n.d.]. Recommendation to disable huge pages for MongoDB. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/.

[54] Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, and Noel De Palma. 2019. Memory flipping: a threat to NUMA virtual machines in the Cloud. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. 325–333. https://doi.org/10.1109/INFOCOM.2019.8737548

[55] Juan Navarro, Sitaram Iyer, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. Boston, MA. https://www.usenix.org/conference/osdi-02/practical-transparent-operating-system-support-superpages

[56] Oracle. 2019. Solaris 11.4 - Locality Groups and Thread and Memory Placement. https://docs.oracle.com/cd/E37838_01/html/E61059/lgroups-32.html.

[57] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Providence, RI, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[58] Ashish Panwar, Naman Patel, and K. Gopinath. 2016. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*. Hong Kong, Hong Kong, Article 15, 8 pages. https://doi.org/10.1145/2967360.2967371

[59] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Williamsburg, VA, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[60] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. Vancouver, B.C., CANADA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[61] Mitosis Linux Project. 2020. Mitosis Linux. Online https://github.com/mitosis-project/mitosis-linux/. Accessed 20. May 2020.

[62] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Tucson, Arizona, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[63] Redis Labs. [n.d.]. Recommendation to disable huge pages for Redis. http://redis.io/topics/latency.

[64] Redis Labs. 2019. Redis. https://redis.io.

[65] Breno Leitao Rodrigo Ceron, Rafael Folco and Humberto Tsubamoto. 2012. Online https://static.rainfocus.com/vmware/vmworldus17/sess/1489512432328001AfWH/finalpresentationPDF/SER2343BU_FORMATTED_FINAL_1507912874739001gpDS.pdf.

[66] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Toronto, ON, Canada, 469–480. https://doi.org/10.1145/3079856.3080210

[67] Amazon Web Services. 2020. Amazon EC2 Instance Types. Online https://aws.amazon.com/ec2/instance-types/.

[68] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Lausanne, Switzerland, 1093–1108. https://doi.org/10.1145/3373376.3378493

[69] Vijayaraghavan Soundararajan, Mark Heinrich, Ben Verghese, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1998. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. Barcelona, Spain, 342–355. https://doi.org/10.1145/279358.279403

[70] Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath. 1998. Implementation of Multiple Pagesize Support in HP-UX. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '98)*. New Orleans, Louisiana.

[71] Oracle Support. 2019. Enable Oracle NUMA support with Oracle Server. Online https://support.oracle.com/knowledge/Oracle%20Cloud/864633_1.html.

[72] Andrew Theurer. 2012. KVM and Big VMs. Online https://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf.

[73] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[74] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA Balancing. Online https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[75] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. Cambridge, Massachusetts, USA, 279–289. https://doi.org/10.1145/237090.237205

[76] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. http://www.usenix.org/events/osdi02/tech/waldspurger.html

[77] Mitosis workload BTree. 2019. Open Source Code Repository. https://github.com/mitosis-project/mitosis-workload-btree.

[78] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Providence, RI, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[79] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi