Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines

Reto Achermann
ETH Zurich
reto.achermann@inf.ethz.ch

Ashish Panwar
IISc Bangalore
ashishpanwar@iisc.ac.in

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

Timothy Roscoe
ETH Zurich
troscoe@inf.ethz.ch

Jayneel Gandhi
VMware Research
gandhij@vmware.com

Abstract

Multi-socket machines with 1-100 TBs of physical memory are becoming prevalent. Applications running on such multi-socket machines suffer non-uniform bandwidth and latency when accessing physical memory. Decades of research have focused on data allocation and placement policies in NUMA settings, but there have been no studies on the question of how to place page-tables amongst sockets. We make the case for explicit page-table allocation policies and show that page-table placement is becoming crucial to overall performance.

We propose *Mitosis* to mitigate NUMA effects on page-table walks by transparently replicating and migrating page-tables across sockets without application changes. This reduces the frequency of accesses to remote NUMA nodes when performing page-table walks. *Mitosis* uses two components: (i) a mechanism to efficiently enable and (ii) policies to effectively control – page-table replication and migration.

We implement *Mitosis* in Linux and evaluate its benefits on real hardware. *Mitosis* improves performance for large-scale multi-socket workloads by up to 1.34x by replicating page-tables across sockets. Moreover, it improves performance by up to 3.24x in cases when the OS migrates a process across sockets by enabling cross-socket page-table migration.

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

Keywords NUMA, TLB, Linux, page-table replication, large pages, TLB miss overhead

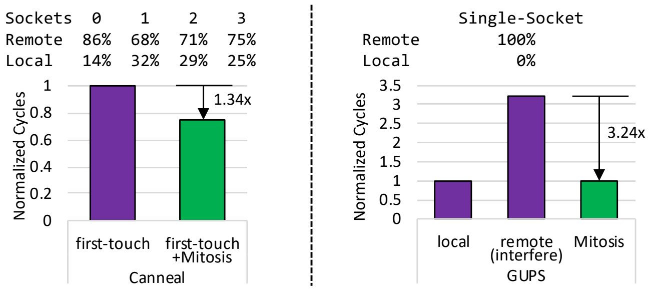
ACM Reference Format:
Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3373376.3378468

1 Introduction

In this paper, we investigate the performance issues in large NUMA systems caused by the sub-optimal placement not of program data, but of page-tables, and show how to mitigate them by replicating and migrating page-tables across sockets.

The importance of good data placement across sockets for performance on NUMA machines is well-known [15, 22, 31, 38]. However, the increase in main memory size is outpacing the growth of TLB capacity. Thus, TLB coverage (i.e. the size of memory that TLBs map) is stagnating and is causing more TLB misses [6, 41, 61, 62]. Unfortunately, the performance penalty due to a TLB miss is significant (up to 4 memory accesses on x86-64). Moreover, this penalty will grow to 5 memory accesses with Intel's new 5-level page- tables [34].

Our first contribution in this paper (§ 3) is to show by experimental measurements on a real system that *page-table* placement in large-memory NUMA machines poses performance challenges: a page-table walk may require multiple



Sockets	0	1	2	3
Remote	86%	68%	71%	75%
Local	14%	32%	29%	25%

	Single-Socket
Remote	100%
Local	0%

Figure 1. *Top Table:* Percentage of local and remote leaf PTEs as observed from each socket on a TLB miss and *Bottom Graph:* Normalized runtime cycles, for two workloads showing multi-socket (left) and workload migration (right) scenarios with their respective improvement using *Mitosis*.

remote DRAM accesses on a TLB miss and such misses are increasingly frequent. We show this effect due to page-table placement on a large-memory machine in two scenarios. The first is a *multi-socket scenario (§ 3.1)*, where large-scale multithreaded workloads execute across all sockets. In this case, the page-table is distributed across sockets by the OS as it sees fit. Such page placement results in multiple remote page-table accesses, degrading performance. We show the percentage of remote/local page-table entries (PTEs) on a TLB miss as observed from each socket in the top left table of Figure 1 for one workload (Canneal) from the multi-socket scenario. We observe that some sockets experience longer TLB misses since up to 86% of leaf PTEs are located remotely. Large-memory workloads like key-value stores and databases that stress TLB capacity are particularly susceptible to this behavior.

Our second analysis configuration focuses on a *workload migration scenario (§ 3.2)*, where the OS decides to migrate a workload from one socket to another. Such behavior arises for many reasons: the need to load balance, consolidate, improve cache behavior, or save power/energy [21, 49, 64]. A key question with migration is what happens to the data that the workload accesses. Existing NUMA policies in commodity OSes migrate data pages to the target socket where the workload has been migrated. Unfortunately, page-table migration is not supported [58], making future TLB misses expensive. Such misplacement of page-tables leads to performance degradation for the workload since 100% of TLB misses require remote memory access as shown in top right table of Figure 1 for one workload (GUPS). Workload migration is common in environments where virtual machines or containers are consolidated on large systems [49]. Ours is the first study to demonstrate the problem arising out of sub-optimal page-table placement on NUMA machines using these two commonly occurring scenarios.

Our second contribution (§ 4) is a technique, *Mitosis*, which replicates and migrates page-tables to eliminate NUMA effects of page-table walks. *Mitosis* works entirely within the OS and requires no change to application binaries. The design consists of a mechanism to enable efficient page-table replication and migration (§ 5), and associated policies for processes to effectively manage page-table replication and migration (§ 6). *Mitosis* builds on widely-used OS mechanisms like page faults and system calls and is hence applicable to most commodity OSes.

Our third contribution (§ 5, 6) is an implementation of *Mitosis* for an x86-64 Linux kernel. Instead of substantially re-writing the memory subsystem, we extend the Linux PV-Ops [44] interface to page-tables and provide policy extensions to Linux's standard user-level NUMA library, allowing users to control migration and replication of page-tables, and selectively enable it on a per-process basis. When a process is scheduled to run on a core, we load the core's page-table pointer with the physical address of the local page-table replica for the socket. Page-table updates are propagated to all replicas efficiently while page-table accesses return consistent values in *Mitosis* which sometimes requires consulting all replicas (e.g., for access and dirty bits).

An important feature of *Mitosis* is that it requires no changes to applications or hardware, and is easy to use on a per-application basis. For this reason, *Mitosis* is readily deployable and complementary to emerging hardware techniques to reduce address translation overheads like segmentation [6, 42], PTE coalescing [61, 62] and user-managed virtual memory [1]. We have open-sourced *Mitosis* to aid future research on page-table management [54, 55].

Our final contribution (§ 8) is a performance evaluation of *Mitosis* on real hardware. We show the effects of page-table replication and migration on a large-memory machine in the same two scenarios used before to analyze page-table placement. In the first, *multi-socket scenario*, we had observed that page-table placement results in multiple remote memory accesses, degrading performance for many workloads. The graph on the bottom left of Figure 1 shows the performance of a commonly used "first-touch" allocation policy which allocates data pages local to the socket that touches the data first. This policy is not ideal as it cannot allocate page-tables locally for all sockets. *Mitosis* replicates page-tables across sockets to improve performance by up to 1.34x in this scenario. These gains are achieved with a modest 0.6% memory overhead.

In the second, *workload migration scenario*, we observe that page-table migration is not supported in current OSes, which makes TLB misses even more expensive after a workload is migrated to a different socket. The graph on the bottom right in Figure 1 quantifies the worst-case performance impact of misplacing page-tables on memory that is remote with respect to the application socket (see remote (interfere) bar). The local bar shows the ideal execution time with locally allocated page-tables. *Mitosis* improves this situation by enabling cross-socket page-table migration, and boosts performance by up to 3.24x.

2 Background

2.1 Virtual Memory

Translation Lookaside Buffers (TLBs) enable fast address translation and are key to the performance of a virtual memory based system. Unfortunately, TLBs cover a tiny fraction of physical memory available on modern systems while workloads consume all memory for storing their large datasets. Hence, memory-intensive workloads incur frequent costly TLB misses requiring page-table lookup by hardware.

Research has shown that TLB miss processing is prohibitively expensive [6, 9–11, 30, 50] as walking page-tables (e.g., 4-level radix tree on x86-64) requires multiple memory accesses. Even worse, virtualized systems need two-levels of page-table lookups which can result in much higher TLB miss processing overheads (24 memory accesses instead of four on x86-64). Consequently, address translation overheads of 10–40% are not uncommon [6, 9, 10, 27, 28, 41], and will worsen with emerging 5-level page-tables [34].

In response, many research proposals improve address translation by reducing the frequency of TLB misses and/or accelerating page-table walks. Use of large pages to increase TLB-coverage [24–26, 56, 59, 67, 68, 70] and additional MMU structures to cache multiple levels of the page-tables [4, 9, 11] are some of the techniques widely adopted in commercial systems. In addition, researchers have also proposed TLB-speculation [5, 63], prefetching translations [39, 50, 66], eliminating or devirtualizing virtual memory [32], or exposing virtual memory system to applications to make the case for application-specific address translation [1].

We observe that prior works studied address translation on single-socket systems. However, page-tables are often placed across remote and local memories in large-memory systems. Given the sensitivity of large page placement on such systems [31], we were intrigued by the question of how page-table placement affects overall performance. In this paper, we present compelling evidence to show that optimizing page-table placement is as crucial as optimizing data placement.

2.2 NUMA Architectures

Multi-socket architectures, where CPUs are connected via a cache-coherent interconnect, offer scalable memory bandwidth even at high capacity and are frequently used in modern data centers and cloud deployments. Looking forward, this trend will only increase; large-memory (1-100 TBs) machines are integrating even more devices with different performance characteristics like Intel's Optane memory [20]. Furthermore, emerging architectures using chiplets and multi-chip modules [23, 35, 36, 40, 46, 51, 69, 72] will drive the multi-socket and NUMA paradigm: accessing memory attached to the local socket will have higher bandwidth and lower latency than accessing memory attached to a remote socket. Note that accessing remote memory can incur 2-4x higher latency than accessing local memory [43]. Given the non-uniformity of access latency and bandwidth, optimizing data placement in NUMA systems has been an active area of research.

2.3 Data Placement in NUMA machines

Modern OSes provide generic support for optimizing data placement on NUMA systems through various allocation and migration polices. For example, Linux provides first-touch vs. interleaved allocation to control the initial placement of data, and additionally employs AutoNUMA to migrate pages across sockets in order to place data closer to the threads accessing it. To further optimize data placement, Carrefour [22] proposed data-page replication along with migration. In addition, data replication has also been proposed at data structure level [15] and via NUMA-aware memory allocators [38] to further reduce the amount of remote memory accesses. In contrast, our work focuses on page-tables, not data pages.

Some prior research has proposed replicated data structures for address spaces. RadixVM [17] manages the process' address space using replicated radix trees to improve the scalability of virtual memory operations in the research-grade xv6 OS [19]. However, *RadixVM* does not replicate page-tables. Similarly, Corey [14] divides the address space into shared and private per-core regions where these explicitly shared regions share the page-table. In contrast, we replicate page-tables to manage NUMA effects on a TLB miss in an industry-grade OS.

Techniques for data vs. page-table pages: One may expect prior migration and replication techniques to extend readily to page-tables. In reality, subtle distinctions between data and page-table pages merit some discussion. First, data pages are replicated by simple *bytewise* copying of data, without any special reasoning of the contents of the pages. Page-table pages, however, require more care and cannot rely simply on bytewise copying – to semantically replicate virtual-to-physical mappings, upper page-table levels must hold pointers (physical addresses) to their replicated, lower level page-tables – which differ from replica to replica except at the leaf level. Moreover, data replication has high memory overheads and maintaining consistency across replicated pages (especially for write-intensive pages) can outweigh the benefits of replication. While data replication has its values, we show that page-table replication is equally important – it incurs negligible memory overhead, can be implemented efficiently and delivers substantial performance improvement.

3 Page-Table Placement Analysis

In this section, we first present an analysis of page-table distributions when running memory-intensive workloads on a large-memory machine (*multi-socket scenario* § 3.1) and then quantify the impact of NUMA effects on page-table walks (*workload migration scenario* § 3.2). Our experimental platform is a 4-socket Intel Xeon E7-4850v3 with 512 GB main memory (more detailed machine configuration in § 8).

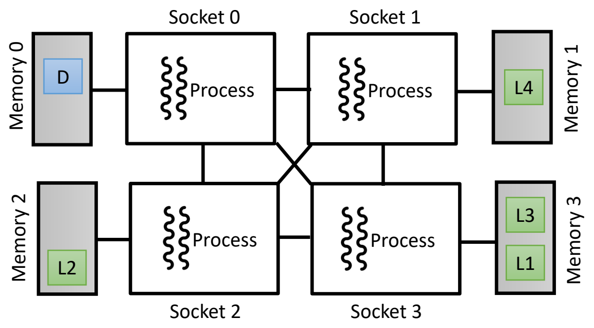


Figure 2. An illustration of current page-table and data placement for a multi-socket workload using 4-socket system.

3.1 Multi-Socket Scenario

We focus on page-table distributions where workloads use almost all resources in a multi-socket system. Consider the example in Figure 2. If a core in socket 0 has a TLB miss for data "D", which is local to the socket, it has to perform up to 4 remote accesses to resolve the TLB miss to ultimately discover that data was actually local to its socket. Even though MMU caches [4] help reduce some of the accesses, at least leaf-level PTEs have to be accessed. Since big-data workloads have large page-tables that are absent from the caches, main memory accesses are often unavoidable [10].

Methodology. We are interested in the distribution of pages for each level in the page-table; i.e., which sockets page-tables are allocated on. We've written an utility tool, consisting of a user-level CLI interface and kernel module, that dumps and analyzes the page-table contents of a process. We invoke the tool every 30 seconds while a multi-socket workload (e.g., Memcached) is running, producing a stream of page-table snapshots over time. We use 30 second time intervals as page-table allocation occurs relatively infrequently and smaller time interval does not change results significantly. We use first-touch or interleaved data allocation policy while enabling/disabling AutoNUMA [18] data page migration with different page sizes for multi-socket workloads in Table 1.

Analysis. We analyze the distribution of page-tables for each snapshot in time. For each page-table level, we summarize the number of per-socket physical pages and the number of valid PTEs pointing to page-table or data pages residing on local and remote sockets. From these snapshots, we collect a distribution of leaf PTEs and which sockets they are located on. We focus on leaf PTEs as there are orders of magnitude more of them than non-leaf PTEs and because they generally dominate address translation performance (upper-level PTEs can be cached in MMU caches [10]). These distributions indicate how many local and remote sockets a page-table walk may visit before resolving a TLB miss.

Results. Due to space limitations, we show a single, processed snapshot of the page-table for Memcached in Figure 3. This snapshot was collected using 4KB pages, local allocation, and AutoNUMA disabled. We studied 2MB pages as well and present observations from them later. The processed dump shows the distribution of all four levels of the page-table (L4 being the root, and L1 the leaf). The dump is organized in four columns representing the four-sockets in this system. In each cell, the first number is the total physical pages at that level-socket combination (e.g. socket 1 has the only L4 page-table page). Next is the distribution of pointers in square brackets of the valid PTEs at this level/socket (e.g. L4 on socket 1 has 8 pointers to L3 on socket 0, 3 pointers locally, and 1 pointer to socket 3). The percentage numbers in rounded brackets are the fraction of valid PTEs pointing to physical pages placed on remote sockets.

Figure 4 shows the percentage of remote leaf PTEs observed by a thread running on each socket for six different, multi-threaded workloads. We made the following observations based on analyzing the page-table dumps and the distribution of leaf PTEs:

1. Page-tables pages are allocated on the socket initializing the first data structures that the page-table pages point to. This is similar to data frame allocation but has important unintended performance consequences. Consider that each page-table page has 512 entries. This means that the placement of a page-table page is entirely dependent upon which of the 512 entries in the page-table page gets allocated first, and which socket the allocating thread runs on. If subsequently, other entries in the page-table page are used for threads on another socket, remote memory references for page-table walks become common.
2. With first touch policy, the number of page-tables tends to be skewed towards a single socket (e.g. socket 1 for Graph500 in Figure 4). This is especially the case when a single thread allocates and initializes all memory, a common practice in OpenMP programming.
3. The interleaved policy evenly distributes page-table pages across all sockets. This is due to the round-robin allocation of data pages.

Workload	Description	MS	WM
Memcached	a commercial distributed in-memory object caching system [53]. Params: keysize = 8, element size = 24, num elements = 576M, 100% reads.	363GB	–
Graph500	a benchmark for generation, compression and search of large graphs [2]. Params: scale factor = 30, edge factor = 16.	420GB	–
HashJoin	a benchmark for hash-table probing used in databases and other large applications. Params (MS): 2B elements. (WM): 128M elements.	455GB	33GB
Canneal	a benchmark for simulated cache-aware annealing to optimize routing cost of a chip design [12]. Params: generated netlist [60]. (MS): x = 120000, y = 11000, num elements = 1200000000. (WM): x = 10000, y = 11000, num elements = 100000000.	382GB	32GB
XSBench	a key computational kernel of the Monte Carlo neutronics application [71]. Params (MS): 112 threads, p factor = 25000000, g factor = 920000. (WM): 16 threads, p factor = 15000000, g factor = 180000.	440GB	85GB
BTree	a benchmarks for index lookups used in database and other large applications. Params: Order = 4. Element Size = 16. (MS): 112 threads, 1500M elements. (WM): 1 thread, 350M elements.	145GB	35GB
LibLinear	a linear classifier for data with millions of instances and features [47]. Params: 28 threads, dataset = kdd12 [16].	–	67GB
PageRank	a benchmark for page rank used to rank pages in search engines [8]. Params: 28 threads, nodes = 268435453, edges = 4236159837.	–	69GB
GUPS	a HPC Challenge benchmark to measure the rate of integer random updates of memory [33]. Params: 1 thread, table size = 64GB.	–	64GB
Redis	a commercial in-memory key-value store [65]. Params: key size = 25, element size = 64, num elements = 256M, 100% reads.	–	75GB

Table 1. Workloads used for analysis in multi-socket (MS) and workload migration (WM) scenarios. Scripts and sources available on GitHub [54, 55].

```
Level |         Socket 0            |         Socket 1            |          Socket 2           |          Socket 3
   L4  |   0 [  0   0   0   0] ( 0%) |   1 [  8   3   0   1] (75%) |   0 [  0   0   0   0] ( 0%) |   0 [  0   0   0   0] ( 0%)
   L3  |   1 [ 56  66  40  37] (72%) |   3 [ 33  43  26  26] (66%) |   0 [  0   0   0   0] ( 0%) |   0 [  0   0   0   0] ( 0%)
   L2  |  89 [11k 11k 11k 11k] (75%) | 109 [13k 13k 13k 13k] (75%) |  66 [ 8k  8k  8k  8k] (75%) |  63 [ 7k  7k  7k  8k] (75%)
   L1  | 40k [ 6M  4M  4M  4M] (67%) | 40k [ 4M  6M  4M  4M] (67%) | 40k [ 4M  4M  6M  4M] (67%) | 40k [ 4M  4M  4M  6M] (67%)
```

Figure 3. Analysis of page-table pointers from a page-table dump for a multi-socket workload: Memcached.

Config.	Workload	Page-Table	Data	Interference
(T)LP-LD	A	A: Local PT	A: Local Data	-
(T)LP-RD	A	A: Local PT	B: Remote Data	-
(T)RP-LD	A	B: Remote PT	A: Local Data	-
(T)RP-RD	A	B: Remote PT	B: Remote Data	-
(T)RPI-LD	A	B: Remote PT	A: Local Data	B: Interfere on PT
(T)LP-RDI	A	A: Local PT	B: Remote Data	B: Interfere on Data
(T)RPI-RDI	A	B: Remote PT	B: Remote Data	B: Interfere on PT&Data

Table 2. Configurations for workload migration scenario, where A and B denote different sockets. T denotes if THP in Linux is used for 2MB pages. Interference is another process that runs on a specified socket and hogs its local memory bandwidth. Figure 5 shows the 2-socket case.

4. While we observed data pages being migrated with AutoNUMA, page-table pages were never migrated. The fraction of data pages migrated over time depends on the workload and its access locality.
5. On all levels, a significant fraction of page-table entries points to remote sockets. In the case of interleave policy, this is $\frac{N-1}{N}$ for an N-socket system.
6. Due to the skew in page-table allocation, some sockets experience longer TLB misses since up to 99% of leaf PTEs are located remotely (e.g., BTree, HashJoin).

Summary On multi-socket systems, page-table allocation is skewed towards sockets that initialize the data structures. While data pages are migrated by OS policies, page-table pages remain on the socket they are allocated. Consequently, remote page-table walks are inevitable and multi-socket workloads suffer from longer TLB misses as their associated page-table walks require remote memory accesses.

3.2 Workload Migration Scenario

We now focus on the impact of NUMA on page-table walks in scenarios where a process on a single socket is migrated to another. Such situations arise frequently in commercial cloud deployments due to the need for load balancing and improving process-data affinity [13, 48]. Particularly, the prevalence of virtual machines and containers that rely on hypervisors and NUMA-aware schedulers to consolidate workloads in data centers are making inter-socket process migrations increasingly common. For e.g., VMware ESXi may migrate processes at a frequency of 2 seconds [49]. Today, data can be migrated across sockets but page-tables cannot, compromising performance.

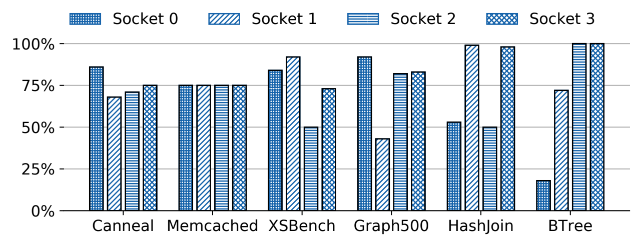


Figure 4. Percentage of remote leaf PTEs as observed from each socket for our multi-socket workloads.

Configurations. We run each workload in isolation while tightly controlling and changing *i)* the allocation policies for data pages and page-table pages, *ii)* whether or not the sockets are idle and *iii)* whether transparent, 2MB large pages (THP) are enabled. We disable NUMA migration. To study page-table allocations in a controlled manner, we modified Linux kernel to force page-table allocations on a fixed socket. We use the configurations shown in Table 2 and visualized in Figure 5. We use the STREAM benchmark [52] running on the socket indicated by interference to create a worst-case scenario of co-locating a memory-bandwidth heavy workload. Memory allocation and processor affinity are controlled by `numactl`.

Measurements. We use `perf` to obtain hardware performance counter values such as total execution cycles and TLB load and store miss walk cycles (i.e., the cycles that the page walker is active for).

Results. We run the eight workloads for all seven configurations. Figure 6 shows the normalized performance with 4KB pages. The base case is the LP-LD configuration where both page-tables and data pages are local. For each configuration, hashed part of the bar denotes the fraction of time spent on page-table walks. We make the following observations from this experiment:

1. All workloads spend a significant fraction of execution cycles (up to 90%) performing page-table walks. Parts of these walks may be overlapped with other work in modern out-of-order processors; nevertheless, they present a performance impediment.
2. LP-LD runs most efficiently in all cases.
3. The local page-table, remote data case (LP-RD and LP-RDI) suffers 3x slowdown versus the baseline. This is not surprising and has motivated prior research on data migration techniques in large-memory NUMA machines.
4. More surprisingly, the remote page-table, local data case (RP-LD and RPI-LD) suffers 3.3x slowdown. This slowdown can even be more severe than remote data accesses even though page-tables consume little memory.
5. When both page-tables and data pages are placed remotely (RP-RD and RPI-RDI), the slowdown is 3.6x and is the worst placement possible for all workloads.

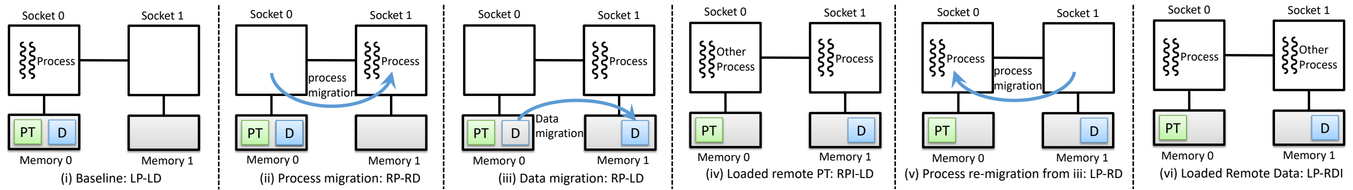

Socket 0
Socket 1
Process
Other Process
process migration
Data migration
PT
D
Memory 0
Memory 1
(i) Baseline: LP-LD
(ii) Process migration: RP-RD
(iii) Data migration: RP-LD
(iv) Loaded remote PT: RPI-LD
(v) Process re-migration from iii: LP-RD
(vi) Loaded Remote Data: LP-RDI

Figure 5. Different configurations for workload migration scenario. We show only 6 out of 7 configurations here. The 7th configuration (RPI-RDI) can be easily created from (ii) by running another process on Socket 0.

6. With 2MB page size (figure omitted for space), TLB reach improves and the number of memory accesses for a page-table walk decreases to 3 rather than 4. These two factors reduce the fraction of execution cycles devoted to page-table walks. Even so, overall performance is still vulnerable to remote page-table placement. We include best and worst cases for each workload in § 8.2 and show the effects of fragmentation on large pages in Figure 11.

Summary. The placement of page-table pages has a significant impact on the performance of memory-intensive workloads in NUMA systems. Remote page-tables can have similar, and in some cases even worse, slowdown than remote data pages accesses. Moreover, the slowdown is visible even with large pages.

4 Design Concept

Mitosis' key concept is a mechanism and its policies to replicate and migrate page-tables and reduce the frequency of remote memory accesses in page-table walks. *Mitosis* requires two components: *i)* a mechanism to support low-overhead page-table replication and migration and *ii)* policies for processes to efficiently manage and control page-table replication and migration. Figure 7 illustrates these concepts. Our discussion focuses on the multi-socket and workload migration scenarios used before in § 3.

4.1 Design Goals

Target Workloads: We design *Mitosis* for workloads that have high memory footprint beyond the reach of the processor's TLB coverage and that experience high TLB pressure. In other words, workloads that spend a good fraction of their time walking page-tables and many of those page-table walks miss the last-level cache.

Flexible Configuration: Not all tasks are equal. We want *Mitosis* to be enabled selectively for specific workloads, while using the default OS policy when *Mitosis* is disabled.

No slowdown: *Mitosis* does not target tasks that are short running, have small memory footprint or exhibit low TLB pressure. Activating *Mitosis* for such workloads should not result in high memory or runtime overheads.

4.2 Multi-socket Scenario

We showed in § 3.1 that multi-socket workloads will, assuming a uniform distribution of page-table pages, have $\frac{N-1}{N}$ PTEs pointing to remote pages for an N-socket system. Page-tables may be distributed among the sockets in a skewed fashion. Figure 7 (a)(i) shows a scenario where threads of the same workload running on different sockets have to make remote memory accesses during page-table walks.

From Figure 7 (a)(i) we can see that if a thread in socket 0 has a TLB miss for data "D" (which is local to the socket), it has to perform up to 4 remote accesses to resolve the TLB miss to only find out that the data was local to its socket.

With *Mitosis*, we replicate the page-tables on each socket where the process is running (shown in Figure 7 (a)(ii)). This results in up to 4 local accesses to the page-table, precluding the need for remote memory accesses in page-table walks.

4.3 Workload Migration Scenario

Single-socket workloads suffer performance loss when processes are migrated across sockets while page-tables are not (shown in Figure 7 (b)(ii)) The process is migrated from

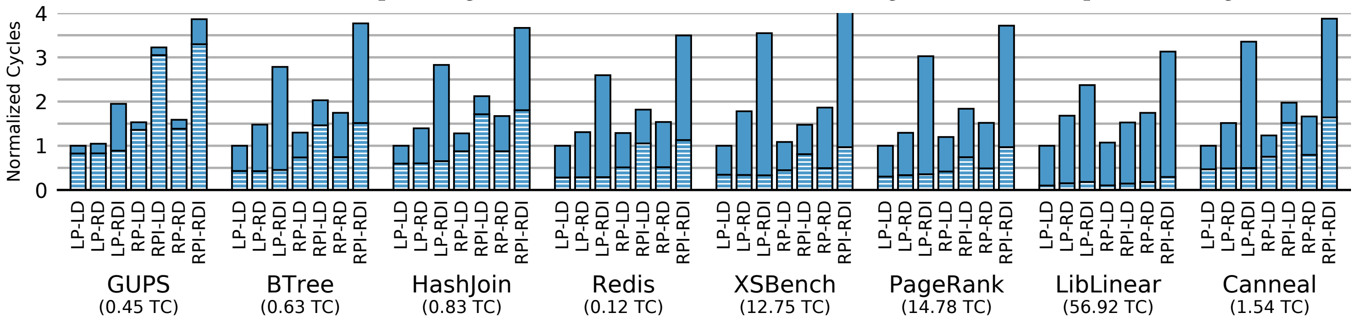


Figure 6. Normalized runtime of our workloads in workload migration scenario with 4KB page size. The lower hashed part of each bar is time spent in walking the page-tables. All configurations are shown in Table 2. Absolute runtime for the baseline in tera cycles ($\times 10^{12}$ cycles) below the workload.

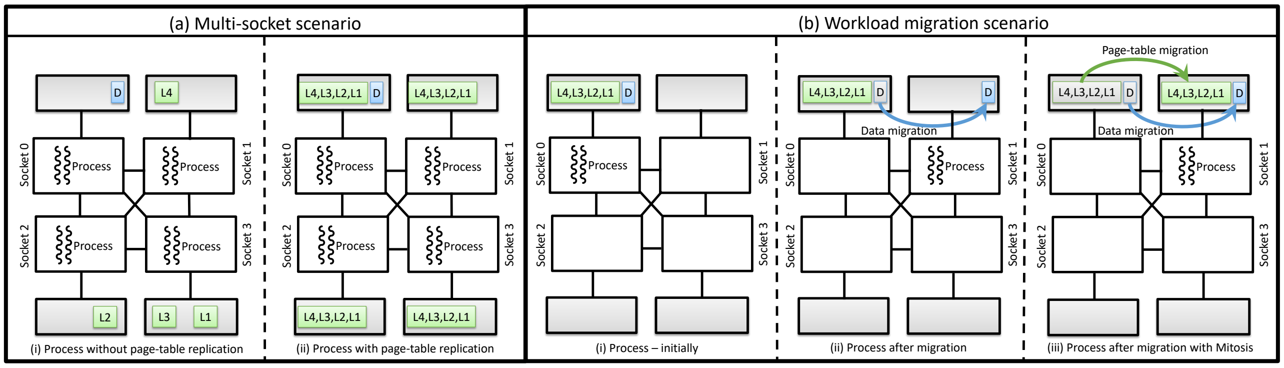

(a) Multi-socket scenario
(b) Workload migration scenario
Socket 0
Socket 1
Socket 2
Socket 3
Process
L4
D
L2
L3
L1
L4,L3,L2,L1
Page-table migration
Data migration
(i) Process without page-table replication
(ii) Process with page-table replication
(i) Process – initially
(ii) Process after migration
(iii) Process after migration with Mitosis

Figure 7. *Mitosis*: Page-table migration and replication on large-memory machines.

socket 0 to socket 1, the NUMA memory manager transparently migrates data pages, but page-table pages remain on socket 1. In contrast, *Mitosis* migrates the page-tables along with the data (Figure 7 (b)(iii)). This eliminates remote memory accesses for page-table walks, improving performance.

5 Mechanism

Replication and migration are inherently similar. We first describe the building blocks for page-table replication, and later show how we can leverage the replication infrastructure to achieve page-table migration.

Mitosis enables per-process replication; the virtual memory subsystem needs to maintain multiple copies of page-tables for a single process. Efficient replication of page-tables can be divided into three sub-tasks: *i)* strict memory allocation to hold the replicated page-tables, *ii)* managing and keeping the replicas consistent, and *iii)* using replicas when the process is scheduled. We now describe each sub-task in detail by providing a generalized design and our Linux implementation. We also discuss how *Mitosis* handles accessed and dirty bits.

5.1 Allocating Memory for Storing Replicas

General design: All page-table allocations are performed by the OS on a page fault–an explicit mapping request can be viewed as an eager call to the page fault handler for the given memory area. *Mitosis* extends the same mechanism to allocate memory across sockets for different replicas.

Such allocation is strict, i.e. it has to occur on a particular list of sockets at allocation time. It is, therefore, possible that it may fail due to the unavailability of memory on those sockets. There are multiple ways to sidestep this problem. First, the OS can reserve enough pages on each socket for page-table allocations using per-socket *page-cache*. These pages can be explicitly reserved through a system call or automatically when a process allocates a virtual memory region. Alternatively, the OS can reclaim physical memory through demand paging mechanisms or evicting a data page onto another socket.

Linux implementation: We rely on the existing page allocation functionality in Linux to implement *Mitosis*. When

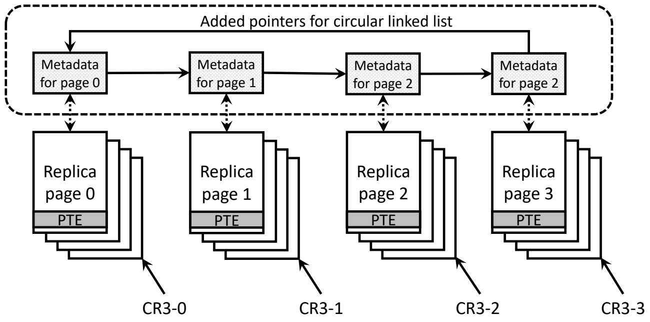


Figure 8. Circular linked list to locate all replicas efficiently (implemented in Linux with `struct page`).

allocating page-table pages, we explicitly supply the list of target sockets for page-table replication. Since strict allocation can fail, we implemented per-socket *page-caches* to reserve pages for page-table allocations. The size of this page-cache is explicitly controlled using a `sysctl` interface. The current prototype returns an out-of-memory error when the page-table cache is depleted. This allows evaluating our prototype and keep dynamic approaches like page-stealing for future work.

5.2 Management of Updates to Replicas

General design: For security, OSes usually do not allow user processes to directly manage their own page-tables. Instead, OSes export an interface through which page-table modifications are handled, e.g. map/unmap/protect of pages. *Mitosis* extends the same interfaces for updates to page-tables to keep all replicas consistent. One way to implement this is to *eagerly* update all replicas at the same time via this standard interface when the kernel modifies the page-table.

On an eager update, the OS finds the physical location to update in the local replica by walking the local replica of the page-table. It is required to walk other replicas of the page-table to locate the physical location to update all the replicas at the same time. Therefore, an N-socket system in x86_64 will need 4*N* memory accesses with replication on a page-table update: 4 memory accesses to walk the page-table on each of the N sockets. To reduce this overhead, we designed a circular linked-list of all replicas. The meta-data about each physical page is utilized to store the pointers to the next physical page holding the replica of the page-table. Figure 8 shows an illustration with 4-way replication. This allows updates to proceed without walking the page-tables. With this optimization, the update of all N replicas takes $2N$ memory references (N for updating the N replicas and N for reading the pointers to the next replica).

Linux implementation: We implemented eager updates to the replica page-tables in Linux. This required intercepting any writes to the page-tables and propagate updates accordingly. But instead of revamping the full-memory subsystem, we used an already existing interface in Linux, PV-Ops [44], which is required to support para-virtualization environments such as Xen [3]. The Linux kernel shipped with major distributions like Ubuntu has para-virtualization support enabled by default, and all page-table updates propagate through this interface.

Conceptually, this is done by indirect calls to the native or Xen handler functions. To avoid the overheads of indirect calls, the PV-Ops subsystem patches the call sites with direct call code to the respective handler function (in our case the *Mitosis* variants) during initialization. The PV-Ops subsystem interface consists of functions to allocate and free page-tables of all levels, reading and writing the translation base register (CR3 on x86_64), and writing page-table entries. Some example functions from the PV-Ops interface can be seen in Listing 1.

```
void write_cr3(unsigned long x);
void paravirt_alloc_pte(struct mm_struct *mm, unsigned
    long pfn);
void paravirt_release_pte(unsigned long pfn);
void set_pte(pte_t *ptep, pte_t pte);
```

Listing 1. Excerpt of the PV-Ops interface.

We implemented *Mitosis* as a new backend for PV-Ops alongside with the native and Xen backends. When the kernel is compiled with *Mitosis*, the default PV-Ops is switched to the *Mitosis* backend. We implemented the *Mitosis* backend with great care to ensure identical behavior to the native backend when *Mitosis* is turned off. Besides, note that replication is generally not enabled by default, and thus the behavior is the same as the native interface.

The PV-Ops subsystem provides an efficient way for *Mitosis* to track any writes to the page-tables in the system. Propagating those updates efficiently requires a fast way to find the replica page-tables based solely on the information provided through the PV-Ops interface (Listing 1) which is either the kernel virtual address (KVA) or a physical frame number (PFN) of the page-table or an entry.

We augment the page meta-data to keep track of replicas with our circular linked list. The Linux kernel keeps track of each 4KB physical frame in the system using `struct page`. Moreover, each frame has a unique KVA and PFN. Linux provides functions to convert between `struct page` and it's corresponding KVA/PFN, which is typically done by adding, subtracting or shifting the respective values and are hence efficient operations. We can, therefore, obtain the `struct page` directly from the information passed through the PV-Ops interface and update all replicas efficiently.

5.3 Efficiently Utilizing Page-Table Replicas

General design: When the OS schedules a process or task, it performs a context switch, restores processor registers and resumes execution of the new process or task. The context switch involves programming the page-table base register of the MMU with the base address of the process' page-table and flushing the TLB. With *Mitosis*, we extend the context switch functionality, to select and set the base address of the socket's local page-table replica efficiently. This enables a task or process to use the local page-table replica if present.

Linux implementation: For each process, we maintain an array of root page-table pointers which allows directly selecting the local replica by indexing this array using the socket id. Initializing this array with pointers to the very same root page-table is equivalent to the native behavior.

5.4 Handling of Bits Written by Hardware

General design: A page-table is mostly managed by software (the OS) most of the time and read by the hardware (on a TLB miss). On x86, however, hardware–namely the page-walker–reports whenever a page has been accessed or written to by setting the accessed and dirty bits in the PTEs. In other words, page-table is modified without direct OS involvement. Thus, accessed and dirty bits do not use the standard software interface to update the PTE and cannot be replicated easily without hardware support. Note, that these two bits are typically set by the hardware and reset by the OS, which uses them for system-level operations like swapping or writing back memory-mapped files if they are modified in memory. With *Mitosis* when replicated, we logically OR accessed and dirty bits of all replicas when read by the OS.

Linux implementation: We need to read accessed/dirty bits from all replicas as well as reset them in all replicas. Unfortunately, the PV-Ops interface doesn't provide functions to read a page-table entry, worse we have found code in the Linux kernel which even writes to the page-table entry without going through the PV-Ops interface. We augmented with the corresponding `get` functions to PV-Ops which consult all copies of page-table entry and make sure the flags are returned correctly. The new function reads all the replicas and ORs the bits in all replicas to get the correct information.

5.5 Page-Table Migration

We use replication to perform migration in the following way: we use *Mitosis* to replicate the page-table on the socket to which the process has been migrated. The first replica can be eagerly freed after migration, or alternatively kept up-to-date in the case the process gets migrated back and lazily deallocated in case physical memory is becoming scarce.

6 Policy

The policies we implement with *Mitosis* control when page-tables are replicated and determine the processes and sockets for which replicas are created. As with NUMA policies, page-table replication policies can be applied system-wide or upon user request. We discuss both in this section.

6.1 System-wide Policies

General design: System-wide policies can range from simple on/off knobs for all processes to policies that actively monitor performance counter events provided by the hardware to dynamically enable or disable *Mitosis*.

Event-based triggers can be developed for page-table migration and replication within the OS. For instance, the OS can obtain TLB miss rates or cycles spent walking page-tables through performance counters that are available on modern processors and then apply policy decisions automatically [57]. A high TLB miss rate suggests that a process can benefit from page-table replication or migration. The ratio between the time spent to serve TLB misses and the number of TLB misses can indicate a replication candidate. Processes with a low TLB miss rate may not benefit from replication.

Even if the OS makes a decision to migrate or replicate the page-tables, it may be costly to copy the entire page-table as big memory workloads easily achieve page-tables of multiple GB in size. By using additional threads or even DMA engines on modern processors, the creation of a replica can happen in the background and the application regains full performance when the replica or migration has completed.

Mitosis primarily targets long-running big-memory workloads with high TLB pressure, and therefore we disable page-table replication for short-running processes since the memory and runtime overhead of replicating page-tables for short-running processes cannot be amortized (§ 8.3).

Linux implementation: We support a straightforward, system-wide policy with four states: *i)* completely disable *Mitosis*, *ii)* enable per-process basis, *iii)* fix the allocation of page-tables on a particular socket, and *iv)* enabled for all processes in the system. This system-wide policy can be set through the `sysctl` interface of Linux. We leave it as future work to implement an automatic, counter-based approach.

6.2 User-controlled Policies

General design: System-wide policies usually imply a one-size-fits-all approach for all processes, but user-controlled policies allow programmers to use their understanding of their workloads and to select policies explicitly. These user-defined replication and migration policies can be combined with data and process placement primitives. Such policies can be selected when starting the program by defining the CPU set and replication set, or at runtime using corresponding system calls to set affinities and replication policies. All of these policies can be set per-process so that users have fine-grained control on replication and migration.

```
numactl [--pgtablerepl= | -r <sockets>]
void numa_set_pgtable_replication_mask(struct bitmask *);
```

Listing 2. Additions to libnuma and numactl

Linux implementation: We implement user-defined policies as an additional API call to `libnuma` and corresponding parameters of `numactl`. Similar to setting the allocation policy, we can supply node-mask or a list of sockets to replicate the page-tables (Listing 2). Applications can thus select the replication policy at runtime, or we can use `numactl` to select the policy without changing the program.

Both, `libnuma` and `numactl` use two additional system calls to set and get the page-table replication bitmask. Whenever a new mask is set, *Mitosis* will walk the existing page-table and create replicas according to the new bitmask. The bitmask effectively specifies the replication factor: N bits set corresponds to copies on N sockets and by passing an empty bitmask, the default behavior is restored.

7 Discussion

7.1 Why Linux Implementation?

As a proof-of-concept, we implement *Mitosis* in the widely-used Linux OS. Choosing Linux as our testbed allows us to prototype our ideas on a complex and complete OS where subtle interactions of many systems features and *Mitosis* stress-tests its evaluation. Specifically, we implemented *Mitosis* on top of Linux kernel v4.17 for the x86_64 architecture.

7.2 Huge/Large Pages Support?

Large pages mitigate address translation overheads by increasing the amount of memory that each TLB entry map by orders of magnitude. Even with 2MB and 1GB page size support in x86-64 on an Intel Haswell processor, the TLB reach is still less than 1%, assuming 1TB of main memory. Moreover, many commodity processors provide limited numbers of large page TLB entries especially for 1GB pages, which limits their benefit [6, 27, 42]. Additionally large pages are not always the best choice, particularly on NUMA systems [31].

Since, address translation overheads are non-negligible even with large pages, they are also susceptible to NUMA effects on page-table walks. *Mitosis* supports larger page sizes by extending page-table replication support to transparent huge pages (THP) in Linux.

7.3 Applicability to Virtualized Systems?

Virtualized systems widely use hardware-based nested paging to virtualize memory [29]. This requires two-levels of page-table translation:

1. gVA to gPA: guest virtual address to guest physical address via a per-process guest OS page-table (gPT)
2. gPA to hPA: guest physical address to host physical address via a per-VM nested page-table (nPT)

In the best case, the virtualized address translation hits in the TLB to directly translate from gVA to hPA with no overheads. In the worst case, a TLB miss needs to perform a 2D page walk that multiplies overheads because accesses to the guest page-table also require translation by the nested page-table. For x86-64, a nested page-table walk requires up to 24 memory accesses compared to four in native. The 2D page-table walk also comes with additional hardware complexity.

Understanding page-table placement in virtualized systems is a major undertaking and requires a separate study. We anticipate that higher performance gains can be obtained by employing *Mitosis* to replicate both levels of page-tables in virtualized systems. We believe our design can be extended to replicate both guest page-tables and nested page-tables independently if the underlying NUMA architecture is exposed to the guest OS. To extend the design, we can rely on setting accessed and dirty bits at both gPT and nPT by the nested page-table walk hardware available since Haswell [37]. Thus, we can extend our OS extension for or-ing the access and dirty bits across replicas to get the correct information at both levels independently. However, the main challenge is that most cloud systems prefer not to expose the underlying architecture to the guest OS. This calls for novel techniques to replicate and migrate both levels of page-tables in a virtualized environment.

7.4 Consistency Across Page-Table Replicas?

Coherence between hardware TLBs is maintained by the OS with the help of TLB flush IPIs and updates to the page-table are already thread-safe as they are performed within a critical section. In Linux, a lock is taken whenever the page-table of a process is modified and thus ensuring mutual exclusion. The updates to the page-table structure are made visible after releasing the lock. When an entry is modified, its effect is made visible to other cores through a global TLB flush as the old entry might still be cached.

Mitosis provides the same consistency guarantees as Linux by updating all page-table replicas eagerly while being in the critical section. Thus, only one thread can modify the page-table at a time. Hardware may read the page-table while updates are being carried out. The critical section ensures correctness while serving page faults or other VM operations while TLB flushes ensure translation coherence after modification of page-table entries.

7.5 Applicability to Library OS

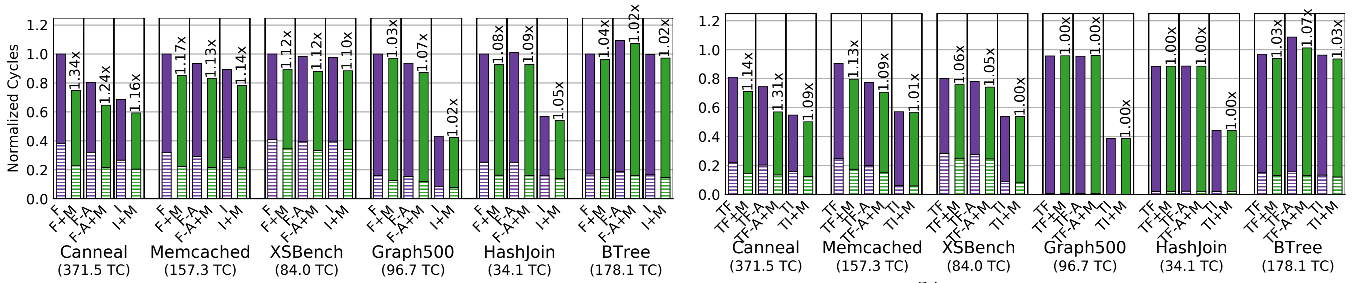
We have chosen to implement the prototype of *Mitosis* in Linux. However, the concept of *Mitosis* is applicable to other operating systems. Microkernels, for instance, push most of their memory management functionality into user-space libraries while the kernel enforces security and isolation. In Barrelfish [7], for example, processes manage their own address space by explicit capability invocations to update page-tables with new mappings.

In such a system, one could implement *Mitosis* purely in user-space by linking to a *Mitosis*-enabled libraryOS, and the kernel itself would not need to be modified. The library can keep track of the address space, including page-tables, replicas etc. Those data-structures can be augmented to include an array of page-table capabilities instead of a single such table. This would allow policies to be defined at application level by using an appropriate policy library. Updates to page-tables might need to be converted to explicit update messages to other sockets, which avoid the need for global locks and propagates updates lazily. On a page fault, updates can be processed and applied accordingly in the page fault handling routine. We leave such an implementation to future work, but believe it to be straightforward.

8 Evaluation

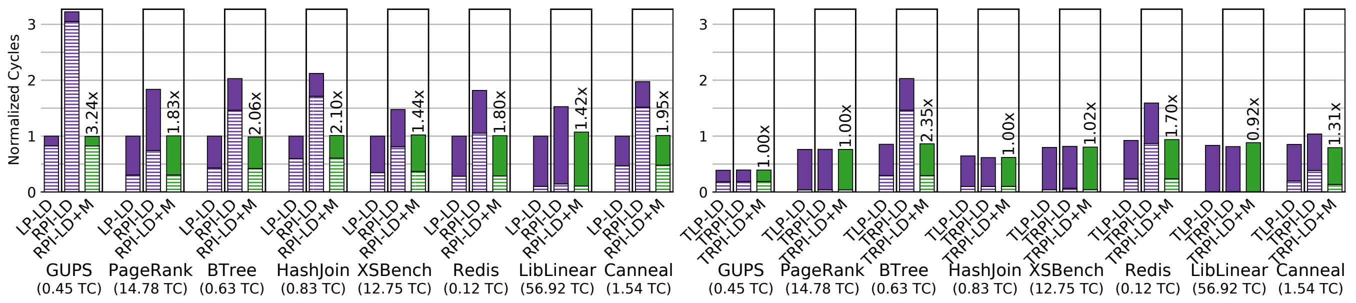
We evaluate *Mitosis* using a set of big-memory workloads and micro-benchmarks. We show: (1) how multi-threaded programs benefit from *Mitosis* (§ 8.1), (2) how *Mitosis* eliminates NUMA effects of page-walks when page-tables are placed on remote sockets due to task migration (§ 8.2) and (3), the memory and runtime overheads of *Mitosis* (§ 8.3).

Hardware Configuration We used a four-socket Intel Xeon E7-4850v3 with 14 cores and 128GB memory per-socket (512 GB total memory) with 2-way hyper-threading running at 2.20GHz. The L3 cache is 35MB in size and the processor has



(a) 4KB Pages **(b)** 2MB Large Pages

Figure 9. Normalized performance with *Mitosis* for multi-socket workloads with 4KB and 2MB page size. The lower hashed part of each bar is execution time spent in walking the page-tables. Absolute runtime for the baseline in tera cycles ($\times 10^{12}$ cycles) below the workload.



(a) 4KB Pages

(b) 2MB Large Pages

Figure 10. Normalized performance with *Mitosis* for workloads in workload migration scenario with 4KB and 2MB page size. The lower hashed part of each bar is execution time spent in walking the page-tables. Absolute runtime for the baseline in tera cycles ($\times 10^{12}$ cycles) below the workload.

a per-core two-level TLB with 64+1024 entries. Accessing memory on the local NUMA socket has about 280 cycles latency and throughput of 28GB/s. For a remote NUMA socket, this is 580 cycles and 11GB/s respectively.

8.1 Multi-socket Scenario

In this part of the evaluation, we focus on multi-threaded workloads running in parallel on all sockets in the system. For a machine with N NUMA sockets, in expectation $\frac{N-1}{N}$ of page-table accesses will be remote while the remote sockets are busy themselves. We evaluate six workloads (see § 3.1), for all commonly used configurations that influence data and page-table placement (see Table 3). Performance is presented as an average of three runs, excluding the initialization phase.

The results are shown in Figure 9a for 4KB pages and Figure 9b with 2MB large pages respectively. All bars are normalized to 4KB first-touch allocation policy (bar: F). Bars with the same allocation policy are grouped in boxes for comparison. The number on top of *Mitosis* bars (green) shows improvement from corresponding non-*Mitosis* bars (purple) within a box. Note that data allocation policy impacts performance and is shown across boxes for each workload. The results for 2MB pages are normalized to 4KB (bar: F) to show performance impact with increase in page size.

We observe that with 4KB pages, up to 40% of the total runtime is spent in servicing TLB misses. *Mitosis* reduces the overall runtime for all applications with the best-case improvement of 1.34x for Canneal. Most of the improvements

Config.	Data pages	Page-table pages
(T)F	First-touch allocation	First-touch allocation (bar: purple)
(T)F+M		*Mitosis replication (bar: green)*
(T)F-A	First-touch allocation	First-touch allocation (bar: purple)
(T)F-A+M	+ Auto page migration	*Mitosis replication (bar: green)*
(T)I	Interleaved allocation	Interleaved allocation (bar: purple)
(T)I+M		*Mitosis replication (bar:green)*

Table 3. Configurations for multi-socket scenario where workload runs on all sockets. T denotes Linux with THP. M denotes that *Mitosis* is enabled in addition.

can be noted in the reduction of page-walk cycles due to replication of page-tables.

Large pages can significantly reduce translation overheads for many workloads. However, NUMA effects of page-table walks are still noticeable, even if all workload memory is backed by large pages. Hence, *Mitosis* provides significant speedup, e.g. 1.14x, 1.13x, and 1.07x for Canneal, Memcached, and BTree, respectively. Note that large pages can create performance bottleneck on NUMA systems and hence may not be used for many systems and workloads [31].

Using various data page placement policies improves performance for our workloads as expected. In combination with all policies, *Mitosis* consistently improves performance.

We have provided evidence that highly parallel workloads experience NUMA effects of remote-memory accesses due to page-table walks. Yet, running a workload concurrently means we cannot inspect a thread in isolation: a TLB miss on one core may populate the cache with the PTE needed to serve the TLB miss on another core of the same socket. Moreover, accessing a remote last-level cache may be faster than accessing DRAM. Nevertheless, we have shown that using replicated page-tables for address translation leads to higher performance on large multi-socket systems, providing up to 1.34x speed up over a single page-table. *Mitosis* does not cause any slowdown for these workloads.

8.2 Workload Migration Scenario

As we observed in § 3.2, NUMA schedulers can move processes from one socket to another under various constraints. In this part of the evaluation, we show that *Mitosis* eliminates NUMA effects of page-walks originating due to data and threads migrating to a different socket while page-tables remain on the socket where workload was first initialized.

We execute the same workloads used for workload migration scenario in § 3.2. As an additional configuration, we enabled *Mitosis* when the page-table is allocated on a remote socket. Recall, we disabled Linux' AutoNUMA migration, and pre-allocated and initialized the working set (17-85GB).

The results are shown in Figure 10a and Figure 10b with 4KB and 2MB page sizes respectively. Table 2 in § 3.2 showed

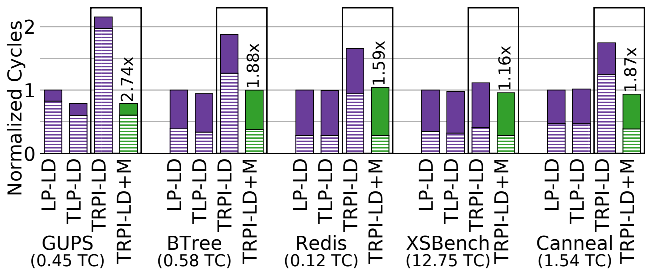


Figure 11. Performance of *Mitosis* in workload migration scenario with 2MB pages under heavy memory fragmentation. Absolute runtime for the baseline in tera cycles ($\times 10^{12}$ cycles) below the workload.

the configurations used for evaluation: LP-LD (Local PT - Local Data) and RPI-LD (Remote PT with interference - Local Data). RPI-LD+M shows the improvement with page-table migration enabled by *Mitosis* when RPI-LD case arises in the system. The boxes denote the bars to compare to see the improvement due to page-table migration. The number on top of the bar denotes the improvement due to *Mitosis* (green bar) as compared to non-mitosis bar (purple bar) within the same box. All bars are normalized to 4KB LP-LD configuration. The results for 2MB pages are normalized to 4KB (LP-LD) to show performance impact of large pages.

With 4KB pages (Figure 10a), remote page-tables cause 1.4x to 3.2x slowdown (bar: RPI-LD) relative to the baseline (LP-LD). *Mitosis* can mitigate this overhead and has the same performance as the baseline by migrating the page-tables with process migration.

With 2MB large pages (Figure 10b), we see that the page walk overheads are comparatively lower, nevertheless we observe a slowdown of up to 2.3x for TRPI-LD over TLP-LD configuration. Again, *Mitosis* can mitigate this overhead and has the same performance as the TLP-LD configuration. Note that for certain workloads page-tables are cached well in the memory hierarchy and thus there is no difference in runtime. For example, in the case of GUPS, we observe roughly one TLB miss per data access–two cache-line requests in total per data array access. By breaking this down, we obtain that each leaf page-table cache-line covers about 16MB of memory which corresponds to 256k cache-lines of the data array. Therefore, the page-table cache-lines are accessed 256k more often than the data array cache-lines, and there are less than 500k page-table cache lines which can easily be cached in L3 cache of the socket. In summary, page-table entries are likely to be present in the sockets processor cache.

Memory Fragmentation: Physical memory fragmentation limits the availability of large pages as the system ages, leading to higher page-walk overheads [45, 58]. Figure 11 shows the performance of Mitosis under heavy fragmentation while using 2MB pages in Linux. We use FMFI (Free Memory Fragmentation Index [45]) to quantify fragmentation. The value of FMFI lies between 0 (no fragmentation) and 1 (severe fragmentation). We fragment physical memory prior to running the workload using a custom benchmark that allocates/frees memory repeatedly and populates the OS page cache until FMFI reaches a high value (> 0.9). Kernel services `khugepaged` and `kcompactd` remain active (default) to generate large pages in the background.

We observe that all workloads, including those that did not show performance improvement with *Mitosis* while using 2MB pages in Figure 10b, show dramatic improvement with *Mitosis* in this case. This is due to workloads falling back to 4KB pages under fragmentation – which we have already shown to be susceptible to NUMA effects of page-table walks. Note that we present this experiment under heavy fragmentation to demonstrate that even if large pages are enabled, page-walk overheads can approach that of 4KB pages. In practice, the actual state of memory fragmentation may depend on several factors and these overheads will be proportional to the failure rate of large page allocations.

Summary: With this evaluation, we have shown that *Mitosis* completely avoids resulting overheads due to page-tables being misplaced on remote NUMA sockets.

		Number of Replicas				
Footprint	**PT Size**	**1**	**2**	**4**	**8**	**16**
1 MB	0.02 MB	1.0	1.015	1.046	1.108	1.231
1 GB	2.01 MB	1.0	1.002	1.006	1.014	1.029
1 TB	2.00 GB	1.0	1.002	1.006	1.014	1.029
16 TB	32.0 GB	1.0	1.002	1.006	1.014	1.029

Table 4. Memory footprint overhead in *Mitosis*.

8.3 Space and Runtime Overheads

Enabling *Mitosis* implies maintaining replicas which consume memory and use CPU cycles to ensure consistency. We evaluate these overheads by estimating the additional memory requirement, then perform micro-benchmarks on the virtual memory operations and wrap up by running applications end-to-end to set those overheads into perspective.

8.3.1 Memory Overheads

We estimate the overhead of the additional memory used to store the page-table replicas when *Mitosis* is enabled. We define the two-dimensional function

$$mem\_overhead(Footprint, Replicas) = Overhead\%$$

that calculates memory overhead relative to the single page-table baseline and evaluate it using different values for the application's memory footprint and the number of replicas. For this estimation, we assume 4-level x86 paging with a compact address space e.g. the application uses addresses $0..FootPrint$. Each level has at least one page-table allocated and a page-table is 4KB in size.

Table 4 shows the memory overheads of *Mitosis* for small to large applications using up to 16 replicas. We use the single page-table case as the baseline. The page-table accounts for about 0.19% of the total footprint, except for the 1MB case where it accounts for 1.5%. With an increasing memory

Operation	4KB region			8MB region			4GB region		
mmap	2.3k	2.4k	(1.0x)	2.6M	2.6M	(1.0x)	1.4G	1.4G	(1.0x)
mprotect	0.9k	1.0k	(1.1x)	1.1k	3.4k	(3.2x)	5.5M	17.9M	(3.2x)
munmap	2.0k	2.1k	(1.0x)	6.4k	8.7k	(1.3x)	0.3G	0.4G	(1.4x)

Table 5. Overhead (in brackets) of *Mitosis* for virtual memory operation using 4-way Replication in cycles.

footprint used by the application, *Mitosis* requires less than 2.9% of additional memory for 16-replicas, whereas our four-socket machine used just 0.6% additional memory.

The page-tables use a small fraction of the total memory footprint of the application. For small programs, the fraction is higher because there is a hard minimum of at least 16KB of page-tables–a 4KB page for each level. This is reflected by the large 23.1% increase in memory consumption for small programs. However, putting this into perspective we advocate not to use *Mitosis* in this case as the 1MB memory footprint falls within the TLB coverage.

In summary, we showed that even with a 16-socket NUMA machine, *Mitosis* adds just 2.9% memory overhead and this overhead drops to 0.6% for our four-socket machine.

8.3.2 VMA Operation Overheads

In this part of the evaluation, we are interested in the overheads of managing consistency across page-table replicas. In particular, we evaluate the overheads of page-table replication on low-level virtual memory operations such as *mmap*, *mprotect* and *munmap*.

We conducted a micro-benchmark that repeatedly calls the VMA operations and measured the execution time of the corresponding system calls. For each operation, we enforce that the page-table modifications are carried out e.g. by passing the `MAP_POPULATE` flat to `mmap`. We varied the number of affected pages from a single page to a large multi-GB region of memory. We ran the micro-benchmark with and without *Mitosis* using 4KB pages and 4-way replication.

The results of this micro-benchmark are shown in Table 5. The table shows CPU cycles required to perform the operation on a memory region of size 4KB, 8MB, or 4GB with *Mitosis* being on or off. Further, we calculate the overheads of *Mitosis* by dividing the 4-way replicated case (*Mitosis* on) with the base case, *Mitosis* off. For `mmap`, we observe an overhead of less than 2%. For `unmap`, the overhead grows to 35% while *Mitosis* adds more than 3x overheads for `mprotect`.

With 4-way replication, there are four sets of page-tables that need to be updated resulting in four times the work. We attribute the rather low overhead for `mmap` to the allocation and zeroing of new data pages during the system call. Likewise, when performing the `unmap` the freed pages are handed back to the allocator, but not zeroed resulting in less work per page and thus higher overhead of replication. *Mitosis* experiences a large overhead for `mprotect` which is still smaller than the replication factor. The `mprotect` operation does a read-modify-write cycle on the affected page-table

Workload	*Mitosis* Off	*Mitosis* On	Overhead
GUPS	270.93 (0.43)	272.18 (0.00)	0.46%
Redis	633.94 (0.34)	636.31 (0.86)	0.37%

Table 6. Runtimes with LP-LD setting, including initialization with and without *Mitosis*. (Standard Deviation).

entries. This process is efficient with no replicas as it results in sequential access within a page-table. However, with the PV-Ops interface, for each written entry all replicas are updated accordingly which leads to poor locality. We expect this to be improved by adapting the PV-Ops interface to update multiple, consecutive entries in a single invocation. Moreover, the integration of the updates with Linux' TLB shootdown mechanism and *Mitosis*-aware page fault handlers may further reduce the latency of these operations by deferring updates until they are really needed. This requires further investigation.

8.3.3 No End-to-End Slowdown

We now set the VMA operations micro-benchmark of the previous section into the perspective of real-world applications. We show that our modifications to the Linux kernel to support *Mitosis* has negligible end-to-end overhead for applications. For this, we execute workloads with and without *Mitosis* and measure overall execution time, including allocation and initialization phase using the LP-LD configuration, i.e., everything is locally allocated. THP is deactivated.

The results are shown in Table 6. We observe that in both cases, GUPS and Redis, the overheads of *Mitosis* are less than half a percent, which is small compared to the improvements we have demonstrated earlier.

Summary: *Mitosis* targets long-running, memory-intensive workloads that suffer from high TLB pressure due to frequent page-table walks missing the last-level cache (e.g. GUPS, Redis, Canneal). Our evaluation shows that, for such workloads, *Mitosis* is able to eliminate NUMA effects of page-table walks without degrading performance in other cases.

9 Conclusion

We presented *Mitosis*: a technique that transparently replicates page-tables on large-memory machines, and provides the first platform to systematically evaluate page-table allocation policies inside the OS. With strong empirical evidence, we made the case for taking the allocation and placement of page-tables to a first-class consideration. We open-source the tools used in this work to inspire further research on page-table management [55], and plan to work with the Linux community to integrate *Mitosis* into the mainline kernel.

Acknowledgments

We would like to thank our anonymous reviewers, Chris Rossbach, Dan Tsafrir, Mark Hill, Michael Swift and our shepherd Tim Harris for their insightful comments and feedback on the paper.

References

[1] 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

[2] James Ang, Brian Barrett, Kyle Wheeler, and Richard Murphy. 2010. Introducing the Graph500. (01 2010). https://graph500.org/

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Bolton Landing, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[4] 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

[5] 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

[6] 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

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, Montana, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[8] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[9] 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

[10] 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

[11] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. San Antonio, Texas, USA, 62–73. https://doi.org/10.1109/HPCA.2011.5749717

[12] 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*.

[13] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. 2018. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. Boston, MA, USA, 85–96. https://www.usenix.org/conference/atc18/presentation/bouron

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. San Diego, California, 43–57. https://www.usenix.org/legacy/event/osdi08/tech/full$_p$apers/boyd-wickizer/boyd$_w$ickizer.pdf

[15] 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

[16] Chih-Chung Chang and Chih-Jen Lin. 2019. Dataset for LibLinear Classifier. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#kdd2012.

[17] 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)*. 211–224. https://doi.org/10.1145/2465351.2465373

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

[19] Russ Cox, Frans Kaashoek, and Robert Morris. 2019. Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2019/xv6.html.

[20] Ian Cutress. 2019. Intel's Enterprise Extravaganza 2019: Launching Cascade Lake, Optane DCPMM, Agilex FPGAs, 100G Ethernet, and Xeon D-1600. https://www.anandtech.com/show/14155/intels-enterprise-extravaganza-2019-roundup.

[21] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (May 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[22] 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

[23] Yigit Demir, Yan Pan, Seukwoo Song, Nikos Hardavellas, John Kim, and Gokhan Memik. 2014. Galaxy: A High-performance Energy-efficient Multi-chip Architecture Using Photonic Interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. Munich, Germany, 303–312. https://doi.org/10.1145/2597652.2597664

[24] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. 2015. Supporting Superpages in Non-Contiguous Physical Memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. Burlingame, CA, USA, 223–234. https://doi.org/10.1109/HPCA.2015.7056035

[25] Zhen Fang, Lixin Zhang, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. 2001. Reevaluating Online Superpage Promotion with Hardware Support. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*. Nuevo Leone, Mexico, 63–72. https://doi.org/10.1109/HPCA.2001.903252

[26] Narayanan Ganapathy and Curt Schimmel. 1998. General Purpose Operating System Support for Multiple Page Sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC '98)*. New Orleans, Louisiana. https://www.usenix.org/conference/1998-usenix-annual-technical-conference/general-purpose-operating-system-support-multiple

[27] 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

[28] 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

[29] J. Gandhi, M. D. Hill, and M. M. Swift. 2017. Agile Paging for Efficient Memory Virtualization. *IEEE Micro* 37, 3 (2017), 80–86. https://doi.org/10.1109/MM.2017.67

[30] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (May 2016), 118–126. https://doi.org/10.1109/MM.2016.10

[31] 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

[32] 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

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

[34] Intel Corporation. 2017. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level$_p$aging$_w$hite$_p$aper.pdf.

[35] Intel Corporation. 2017. New Intel Core Processor Combines High-Performance CPU with Custom Discrete Graphics from AMD to Enable Sleeker, Thinner Devices. https://newsroom.intel.com/editorials/new-intel-core-processor-combine-high-performance-cpu-discrete-graphics-sleek-thin-devices/.

[36] S. S. Iyer. 2016. Heterogeneous Integration for Performance and Scaling. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 6, 7 (July 2016), 973–982. https://doi.org/10.1109/TCPMT.2015.2511626

[37] Sunil Jain. 2014. Are You Ready to Innovate? Four New Virtualization Technologies on the Latest Intel Xeon Product Family. https://software.intel.com/en-us/blogs/2014/09/08/four-new-virtualization-technologies-on-the-latest-intel-xeon-are-you-ready-to.

[38] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. Santa Clara, CA, 263–276. https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle

[39] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. Anchorage, Alaska, 195–206. https://doi.org/10.1109/ISCA.2002.1003578

[40] A. Kannan, N. E. Jerger, and G. H. Loh. 2015. Enabling interposer-based disintegration of multi-core processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Waikiki, HI, USA, 546–558. https://doi.org/10.1145/2830772.2830808

[41] 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

[42] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. 2014. Performance analysis of the memory management unit under scale-out workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. Raleigh, NC, USA, 1–12. https://doi.org/10.1109/IISWC.2014.6983034

[43] Patrick Kennedy. 2017. AMD EPYC Infinity Fabric Latency DDR4 2400 v 2666: A Snapshot. https://www.servethehome.com/amd-epyc-infinity-fabric-latency-ddr4-2400-v-2666-a-snapshot/.

[44] Kernel.org. 2019. Paravirt_ops. https://www.kernel.org/doc/Documentation/virtual/paravirt$_o$ps.txt.

[45] 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)*. 705–721. https://doi.org/10.1145/3139645.3139659

[46] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, and Sam Naffziger. 2018. The Next Generation AMD Enterprise Server Product Architecture. https://www.hotchips.org/wp-content/uploads/hc$_a$rchives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.921-EPYC-Lepak-AMD-v2.pdf.

[47] Chih-Jen Lin. 2019. LIBLINEAR – A Library for Large Linear Classification. https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[48] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. London, United Kingdom, 1–16. https://doi.org/10.1145/2901318.2901326

[49] Xunjia Lu. 2017. Extreme Performance Series: vSphere Compute & Memory Schedulers. https://static.rainfocus.com/vmware/vmworldus17/sess/1489512432328001AfWH/finalpresentationPDF/SER2343BU$_F$ORMATTED$_F$INAL$_1$507912874739001gpDS.pdf.

[50] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. https://doi.org/10.1145/2445572.2445574

[51] Marvell Corporation. 2016. MoChi Architecture. http://www.marvell.com/architecture/mochi/.

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

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

[54] Mitosis. 2019. Artifact Repository. https://github.com/mitosis-project/mitosis-asplos20-artifact.

[55] Mitosis. 2019. Open Source Code Repository. https://github.com/mitosis-project/.

[56] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 89–104. https://doi.org/10.1145/844128.844138

[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, 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

[59] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. Burlingame, CA, USA, 210–222. https://doi.org/10.1109/HPCA.2015.7056034

[60] PARSEC. 2012. Canneal Netlist Generator. https://parsec.cs.princeton.edu/download/other/canneal$_n$etlist.pl.

[61] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*. Orlando, FL, USA, 558–567. https:

//doi.org/10.1109/HPCA.2014.6835964

[62] 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

[63] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. Waikiki, Hawaii, 1–12. https://doi.org/10.1145/2830772.2830773

[64] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. 2009. Thread Motion: Fine-grained Power Management for Multi-core Systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Austin, TX, USA, 302–313. https://doi.org/10.1145/1555754.1555793

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

[66] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. Vancouver, British Columbia, Canada, 117–127. https://doi.org/10.1145/339647.339666

[67] A. Seznec. 2004. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Trans. Comput.* 53, 7 (July 2004), 924–927. https://doi.org/10.1109/TC.2004.21

[68] Mark Swanson, Leigh Stoller, and John Carter. 1998. Increasing TLB Reach Using Superpages Backed by Shadow Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. Barcelona, Spain, 204–213. https://doi.org/10.1145/279358.279388

[69] Taiwan Semiconductor Manufacturing Company Limited. 2019. CoWoS Services. http://www.tsmc.com/english/dedicatedFoundry/services/cowos.htm.

[70] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. San Jose, California, USA, 171–182. https://doi.org/10.1145/195473.195531

[71] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - 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

[72] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. Shoaib Bin Altaf, N. Enright Jerger, and G. H. Loh. 2018. Modular Routing Design for Chiplet-Based Systems. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 726–738. https://doi.org/10.1109/ISCA.2018.00066

A Artifact Appendix

A.1 Abstract

Our artifact provides x86_64 binaries of our modified Linux kernel v4.17, user-space control libraries (libnuma) and evaluated benchmarks with their input files where appropriate. We further provide source code of our Linux modifications, 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 for a multi-socket Intel Haswell machine (or similar microarchitecture) with at least 512GB of main memory.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Page-table replication.
- **Programs:** Canneal, Memcached, XSBench, Graph500, HashJoin, BTree, GUPS, Redis, PageRank, LibLinear.
- **Compilation:** GCC version 7.4.0 (gcc-7 7.4.0-1ubuntu1~18.04.1)
- **Transformations:** Page-table replication implemented as a Linux kernel extension.
- **Binary:** Included for x86_64. Source code and scripts included to regenerate binaries.
- **Data set:** Generated netlist for Canneal. kdd12 dataset for LibLinear. There are scripts to obtain those.
- **Run-time environment:** Provided by the supplied Linux kernel binaries for x86_64 hardware, source code given.
- **Hardware:** We recommend a 4 socket Intel Xeon E7-4850v3 with 14 cores and 128GB memory per-socket (512 GB total memory) to reproduce the paper results. Other multi-socket x86_64 servers with at least 512GB main memory / 128GB memory per NUMA node should give similar results.
- **Run-time state:** Workloads populate their runtime state themselves.
- **Execution:** Natively on Mitosis-Linux using bash-scripts.
- **Output:** The artifact produces the graphs for each figure used in the paper.
- **Experiments:** All of the programs above with different NUMA policies, page sizes and replication settings.
- **How much disk space required:** 200 GB for workloads and datasets etc.
- **How much time is needed to prepare workflow:** 30-60 mins.
- **How much time is needed to complete experiments:** 3-4 days (1 day excluding Canneal benchmark).
- **Publicly available:** Private GitHub repository. Public relase in the work.
- **Workflow framework used?:** No.
- **Archived:** Yes. DOI: 10.5281/zenodo.3605382.

A.3 Description

A.3.1 How delivered

All scripts are available in the GitHub repository https://github.com/mitosis-project/asplos20-ae. Source code are currently in a *private* GitHub repository to which we will share access. We are working on a public release in meantime. Pre-compiled binaries are available at https://zenodo.org/record/3560200.

A.3.2 Hardware dependencies

We recommend a 4 socket Intel Xeon E7-4850v3 with 14 cores and 128GB memory per-socket (512 GB total memory) to reproduce the paper results. Other multi-socket x86_64 servers with at least 512GB main memory / 128GB memory per NUMA node should give similar results. Note, the state memory sizes are a hard requirement as some workloads use hardcoded data structure scaling sizes.

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 may also work. In addition to the packages shipped with Ubunty 18.04 LTS, we require the following packages:

```
apt-get install build-essential libncurses-dev \
                bison flex libssl-dev libelf-dev \
                libnuma-dev python3 python3 \
                python3-pip python3-matplotlib \
                python3-numpy git wget \
                kernel-package fakeroot ccache \
                libncurses5-dev wget pandoc \
                libevent-dev libreadline-dev
```

See https://www.kernel.org/doc/html/v4.17/process/changes.html to get a detailed explanation for Linux kernel build dependencies

In addition, install the following python libraries with pip:

```
pip3 install zenodo-get
```

There is a docker image which you can use to compile. You can do `make docker-shell` to obtain a shell in the docker container, or just to compile everything type `make docker-compile`.

A.3.4 Data sets

The data sets are automatically obtained when executing the run scripts. The generation and download scripts are present in the `datasets/`.

A.4 Installation

To install, either download the complete artifact from Zenodo (https://zenodo.org/record/3605382) (DOI 10.5281/zenodo.3605382), or clone the GitHub repository from https://github.com/mitosis-project/mitosis-asplos20-artifact. The GitHub repository contains all needed scripts to run all artifacts, it does not contain any source code or binaries. There are scripts which download the pre-compiled binaries, or source code for compilation.

A.4.1 Obtaining pre-compiled binaries

To obtain the pre-compiled binaries execute:

```
./scripts/download_binaries.sh
```

The pre-compiled binaries are available on Zenodo.org (https://zenodo.org/record/3560200) You can download them manually and place them in the `precompiled` directory.

A.4.2 Compiling Binaries

If you plan to use the pre-compiled binaries, you can skip this step. Otherwise use the following commands to compile the binaries from the sources.

The source code for the Linux kernel and evaluated worloads are available on GitHub (https://github.com/mitosis-project. To obtain the source code you can initialize the corresponding git submodules. *Note*: contact us for the workload source code.

To compile everything just type `make` in the root directory. There is a separate make target for each workload.

A.4.3 Installing Mitosis-Linux

Either place the `vmlinux` binary and boot from it or use the debian packages to install Mitosis-Linux on your machine under test. See the hardware dependencies for the minimum requirements for this machine.

To install the kernel module for page-table dumping you need to execute:

```
make install lkml
```

It's best to compile it on the machine runnig Mitosis-Linux.

```
make mitosis-page-table-dump
```

A.4.4 Preparing Datasets

Canneal and LibLinear workloads require datasets to run. Scripts to download or generate the datasets are placed in `datasets/` and require approximately 100GB of disk space. Datasets are generated as part of experimental runs, if not already present. You can also prepare datasets before running the experiments as:

```
./datasets/prepare_liblinear_dataset.sh
./datasets/prepare_canneal_datasets.sh
```

A.5 Experiment workflow

Once you have obtained the artifact sources or compiled the binaries, you can execute the workloads as follows. We provide scripts for each workload used in the paper, and to run all of them in one go.

A.5.1 Deployment

You can download the artifact to the test machine and execute the scripts directly.

Alternatively, you can deploy it to a separate test machine. To do so, set your target host-name and directory in `scripts/site_config.sh` and use the deploy script to copy the scripts, binaries and datasets to your test machine.

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

A.5.2 Running all experiments

We advise to run the experiments natively and exclusively on a machine i.e. no virtual machines or compute/memory-intensive application running concurrently. A test script (`scripts/run_test.sh`) is provided to verify the experimental setup. To run all experiments, execute:

```
./scripts/run_all.sh
```

To run experiments for individual figures, do:

- Figure 6 - `./scripts/run_f6_all.sh`
- Figure 9a - `./scripts/run_f9a_all.sh`
- Figure 9b - `./scripts/run_f9b_all.sh`
- Figure 10a - `./scripts/run_10a_all.sh`
- Figure 10b - `./scripts/run_10b_all.sh`
- Figure 11 - `./scripts/run_f11.sh`
- Table 5 - `./scripts/run_t5.sh`

You can also execute each bar of the figures independently (refer to `scripts/run_f6_all.sh` and `scripts/run_f10a_all.sh` for more examples) as:

```
./scripts/run_f6f10_one.sh BENCHMARK CONFIG
./scripts/run_f9_one.sh BENCHMARK CONFIG
```

All output logs are redirected to `evaluation/measured/FIGURENUM`. Logs can be processed by executing:

```
./scripts/process_logs_core.py
```

A.6 Evaluation and expected result

Once you've ran *all* experiments above, you can compare the outcome with the expected results. The reference data set (as shown in the paper) is obtained on a four-socket Intel Xeon E7-4850v3 with 14 cores and 128GB memory per-socket (512 GB total memory) with

2-way hyper-threading running at 2.20GHz). The data is located in the folder `evaluation/reference/`.

A.6.1 Collecting the Results

If you used the deploy script to copy your data to the test machine, you can collect the runtime results by executing

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

A.6.2 Generate the Report

You can execute the report with the reference and measured data and graphs using the following commands.

```
./scripts/compile_report.sh
```

This will produce a pdf and website to display the comparison.

A.7 Experiment customization

Experiments are not customizable and tailored to run on a machine with at least 512GB of main memory, with 128GB of memory per NUMA node.

A.8 Methodology

The artifact of this paper was submitted and reviewed following the guidelines stated in http://cTuning.org/ae/submission-20190109.html and http://cTuning.org/ae/reviewing-20190109.html.