Scalable and Effective Page-table and TLB management on NUMA Systems

Bin Gao, Qingxuan Kang, and Hao-Wei Tee, *National University of Singapore;* Kyle Timothy Ng Chu, *Horizon Quantum Computing;* Alireza Sanaee, *Queen Mary University of London;* Djordje Jevdjic, *National University of Singapore*

https://www.usenix.org/conference/atc24/presentation/gao-bin-scalable

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Scalable and Effective Page-Table and TLB Management on NUMA Systems

Bin Gao[1], Qingxuan Kang[1], Hao-Wei Tee[1], Kyle Timothy Ng Chu[2], Alireza Sanaee[3], and Djordje Jevdjic[1]

[1]*National University of Singapore* [2]*Horizon Quantum Computing* [3]*Queen Mary University of London*

Abstract

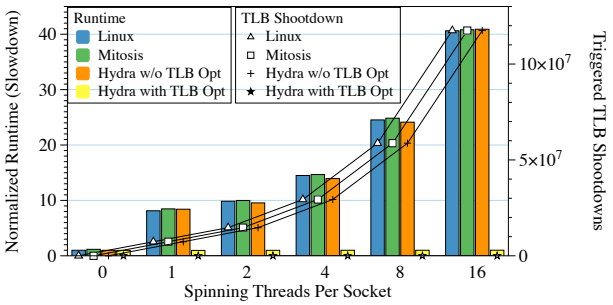
Memory management operations that modify page-tables, typically performed during memory allocation/deallocation, are infamous for their poor performance in highly threaded applications, largely due to process-wide TLB shootdowns that the OS must issue due to the lack of hardware support for TLB coherence. We study these operations in NUMA settings, where we observe up to 40x overhead for basic operations such as `munmap` or `mprotect`. The overhead further increases if page-table replication is used, where complete coherent copies of the page-tables are maintained across all NUMA nodes. While eager system-wide replication is extremely effective at localizing page-table reads during address translation, we find that it creates additional penalties upon any page-table changes due to the need to maintain all replicas coherent.

In this paper, we propose a novel page-table management mechanism, called *Hydra*, to enable transparent, on-demand, and partial page-table replication across NUMA nodes in order to perform address translation locally, while avoiding the overheads and scalability issues of system-wide full page-table replication. We then show that Hydra's precise knowledge of page-table sharers can be leveraged to significantly reduce the number of TLB shootdowns issued upon any memory-management operation. As a result, Hydra not only avoids replication-related slowdowns, but also provides significant speedup over the baseline on memory allocation/deallocation and access control operations. We implement Hydra in Linux on `x86_64`, evaluate it on 4- and 8-socket systems, and show that Hydra achieves the full benefits of eager page-table replication on a wide range of applications, while also achieving a 12% and 36% runtime improvement on Webserver and Memcached respectively due to a significant reduction in TLB shootdowns.

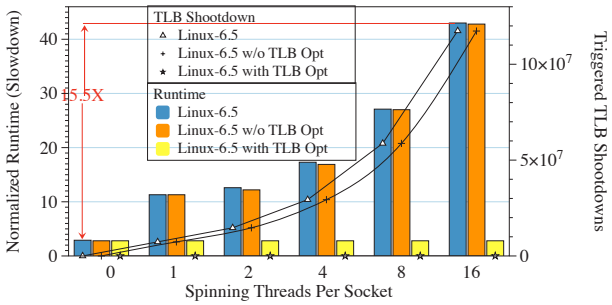
1 Introduction

Translation Lookaside Buffers (TLBs) are tiny hardware structures that cache recently used virtual-to-physical address mappings from the page-tables. Since TLBs must be accessed before every memory operation, their latency is critical to the performance of the entire system, which is why they must be small and tightly integrated into every CPU core. Given that every core has its own TLB that independently caches page-table entries (PTE), a mechanism that enforces coherence of PTEs across all TLBs that cache them is required for correctness. Unfortunately, most modern multi-core architectures do not provide hardware support for TLB coherence, but instead provide privileged instructions for invalidation of TLB entries, which the operating system (OS) calls upon any change to the page-tables. Such instructions can invalidate TLB entries only on the core that executes them; to invalidate TLBs on all other cores, OSes use an expensive, IPI-based mechanism to send an interrupt to each core individually due to the lack of support for flexible multi-cast delivery [38, 50], in a procedure that's called *TLB Shootdown*. Furthermore, when a PTE changes, the OS lacks the information about which TLB in the system currently caches the modified PTE, and thus must send shootdowns indiscriminately to all cores that currently run a thread of the same process, and it must do so synchronously for correctness, causing delays of several microseconds or even tens of microseconds on big machines [38].

The TLB coherence operations particularly affect multi-socket and multi-node systems, which data centers have shifted towards in order to continue to scale up the CPU performance and memory capacity in the post-Moore's law era. These systems connect multiple CPUs to multiple memory modules in a NUMA fashion. Unfortunately, this has a dramatic impact on TLB coherence. To illustrate, Figure 1 shows the performance of `mprotect`, a Linux syscall that changes the permission bits in page-table, when called for a 4KB page on an 8-socket machine. This experiment is performed using the testbed explained in Section 5.1. In this experiment, a single thread runs `mprotect` in a loop, repeatedly flipping a single bit in a single PTE. Additionally, we run a varying number of spinning threads on every socket. The spinning threads mimic the ideal behavior of scale-out workloads where numerous threads perform independent computations



a) `mprotect` on baseline Linux(v4.17), Mitosis, and Hydra



b) `mprotect` on baseline Linux(v6.5.7) and Hydra

Figure 1: Impact of page-table replication, and the TLB shootdown optimization on `mprotect`. Hydra reduces the run time slow down by up to 40x, and mitigates the TLB shootdown overhead by leveraging the information about page tables on each socket. All values in both plots are normalized to the baseline Linux v4.17 without replication.

with limited data-sharing and synchronization among them. As Figure 1 shows, the TLB shootdowns cause a 40x performance degradation for `mprotect` on the baseline Linux v4.17 when spinning threads are added to other sockets. This problem persists in newer kernels, such as v6.5.7, with up to 15.5x performance degradation as shown in Figure 1b. Note that the new kernel shows nominally better scalability, as it degrades performance by only 15.5x, vs. 40x with v4.17. However, this is due to the fact that the baseline `mprotect` performance (without spinning threads) is about 3x worse compared to Linux v4.17.[1] Also note that the impact of shootdowns sent to spinning threads on remote sockets is significantly higher compared to the shootdowns sent to threads that are spinning on the same socket as where `mprotect` runs, as Figure 2a shows. This suggests that the performance of virtual memory (VM) operations is undesirably held back by the number of threads, even in the idealized scale-out scenario wherein these threads do not exhibit any data sharing and perform no synchronization operations such as barriers or locks. As data centers scale beyond multi-socket systems to multi-node systems where a single process can span over multiple compute nodes connected via remote memory protocols like CXL [41, 45], ensuring TLB coherence across the logical process becomes increasingly expensive. Moreover, as VM abstraction continues to play an ever more important role in simplifying programming models for emerging systems such as heterogeneous and disaggregated memory architectures [29, 30, 60, 62], keeping the overhead of VM operations low is crucial to guarantee future performance.

Apart from being over-conservative on TLB coherence operations upon page-table updates, page-table reads, which happen during page walks as a result of TLB misses, are also significantly affected by the NUMA architecture. Mitosis [1] demonstrated that the performance penalty of the requested page-table entry (PTE) being allocated on a remote NUMA node is often higher than the penalty of the requested data

[1] Linux v4.17 performs better in absolute terms for all our workloads, and unless otherwise mentioned, all results we show are based on v4.17.

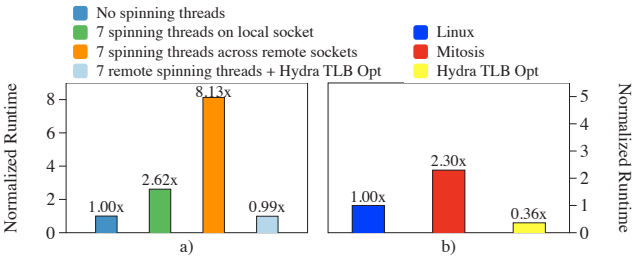


Figure 2: a) The slowdown of `mprotect` on Linux with local threads spinning on the local vs. remote sockets, b) The slowdown of Mitosis and Hydra over Linux when the range of `mprotect` is 512KB; note that Mitosis sees a slowdown, while Hydra experiences a speedup.

page being remote; this is particularly true for big data applications that experience high TLB miss rates [35, 62].

In this work, we introduce a novel page-table management mechanism, called Hydra, that seeks to simultaneously improve the performance of both page-table *READ* and *UPDATE* operations in NUMA systems. Hydra enables transparent, partial, and on-demand page-table replication across NUMA nodes to ensure that address translations for local data is satisfied within the same NUMA node. Hydra achieves this by creating replicas of individual PTEs on the NUMA node that requests them. In doing so, Hydra avoids costly and ineffective system-wide page-table replication along with any coherence actions that would arise from such replication, as it limits the scope of replication and the related coherence actions only to nodes that actually share the same PTEs.

We further observe that Hydra's precise and coherently maintained knowledge of which sockets contain copies of any individual page-table can be leveraged to reduce the scope of TLB shootdowns sent upon any change to that page-table — an effective solution to the scalability bottleneck due to sending IPIs indiscriminately to all cores running the same process. Namely, Hydra's lazy on-demand replication of page-

System	Translation Performance	Selective Replication	Implicit Policy	Lazy Replication	Efficient PTE Updates	Efficient Migration	NUMA Scalability
Linux	✗	✗	N/A	N/A	✗	✗	✗
Mitosis	✓	✗	✗	✗	✗	✓	✗
Hydra	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of state-of-the-art solutions in NUMA support for page-table management.

tables ensures that the following invariant holds by design: if there is no replica of a given page-table on a given socket, there can be no thread running on that socket that currently has the corresponding PTE in its TLB; if any core on the socket has the PTE in its TLB, then the PTE must also exist in a local replica by design. As a consequence of this invariant, expensive TLB shootdowns do not need to be sent to any thread running on a socket that does not have a copy of the corresponding page-table. This allows Hydra to safely filter out many TLB shootdowns, dramatically improving the performance of memory management operations that change page-tables, as well as improving the performance of threads that avoid receiving unnecessary TLB shootdowns.

We implement Hydra in Linux (v4.17 and v6.57) and show that Hydra achieves the full benefits of page-table replication for page-table READs, as well as a 12% and 36% improvement in runtime on Apache Webserver and Memcached due to more efficient page-table UPDATES. Hydra minimizes the memory footprint and page-table coherence overheads, and avoids the scalability limitations of eager replication. As shown in Figure 1, Hydra is able to entirely eliminate the NUMA effect of operations such as `mprotect`, whose performance is improved by nearly 40x. We are open-sourcing Hydra to encourage more research in this area.

The rest of the paper is organized as follows. Section 2 presents the background on page-table management on NUMA systems. Sections 3 and 4 describe the design principles and implementation details of Hydra. Section 5 presents our evaluation methodology and results. Section 6 provides discussion and directions for future work. Section 7 discusses related work and Section 8 concludes the paper.

2 Background

2.1 Virtual Memory

Page-tables are a key component of most modern operating systems and are used to map the virtual address space of a process to the physical memory available on the hardware platform. As page-tables are hierarchically organized in multiple levels, conducting a full page-table walk usually requires multiple memory accesses [22] and on most systems it is done by hardware for performance reasons.

Translation Lookaside Buffers (TLBs) are used to accelerate the process of address translation by caching virtual-to-physical mappings that are frequently used. Unfortunately,

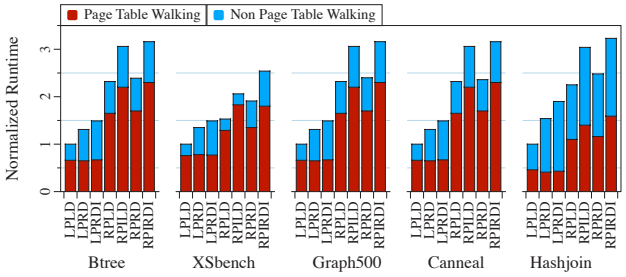


Figure 3: Impact of data and page-table placement on performance of various applications; L - local, R - remote, P - page-tables, D - data, I - with the interference of other applications on inter-socket traffic. The impact of page-table walks on the run time is significantly high often higher than data access. Detailed settings are show in Table 2.

the growth of main memory capacity has far outpaced the growth of TLB sizes in recent years. As a result, TLB coverage has stagnated which results in a higher TLB miss rate [6, 19, 23, 36, 54, 55]. When a TLB miss occurs, the Memory Management Unit (MMU) has to perform a page-table walk to retrieve the appropriate Page-Table Entry (PTE). This is a time-consuming process that can take several hundreds of cycles to complete [59]. As a result, it is not unusual to see applications spending anywhere from 10-93% of CPU cycles servicing TLB misses [6, 24, 36, 44], especially on NUMA systems [1, 51]. In addition to TLBs, modern CPUs also use Page Walk Caches (PWCs) to cache intermediate nodes of the page-tables that are frequently accessed. This allows the MMU to bypass some of the upper levels of the page-table. However, the number of entries in such caches is normally limited to several dozen entries due to hardware constraints, which limits their effectiveness [50, 64].

2.2 NUMA Architectures

NUMA architectures have historically come in many forms, but their defining feature is that accessing memory attached to a local node will have higher bandwidth and lower latency and energy compared to accessing memory on a remote node. NUMA systems are particularly popular in data centers and cloud systems in the form of multi-sockets, as they scale well with larger memory capacity. The NUMA paradigm is today further driven by the emerging architectures based on chiplets and multi-chip modules [3, 15, 17, 18, 21, 32, 34, 65].

Because of the large latency, bandwidth, and energy gap between accesses to local and remote NUMA nodes, the performance of NUMA systems largely depends on the ability of the system to maximize the chances that a piece of data is located in the same NUMA node as the thread requests it [11, 20, 24, 33], with most modern operating systems providing some form of support for optimizing data page placement on NUMA systems. For example, Linux employs AutoN-

UMA which migrates data pages to sockets that are closer to the threads that access them. Linux also provides *first-touch* and *interleaved* allocation policies which affect the initial placement of data. Under the first-touch policy, memory is allocated on the first node to access it while an interleaved policy alternates memory allocation across a set of nodes that can be defined by the user.

2.3 Eager Page-table Replication

Modern operating systems such as Linux offer little control over the page-table placement and mostly allocate page-tables and directories on the NUMA node that touched the corresponding data first [1], which is a policy known as *first touch* [40]. When any other NUMA node accesses the same or neighboring data, the resulting page-table traversal will lead to expensive cross-socket memory accesses, regardless of the location of the data. The hierarchical page-table organization with 4-5 levels present in most commercial architectures further exacerbates the problem; in the worst case, during a page walk, the requested entries at every page-table level could be allocated on a different NUMA node.

The impact of page-table placement on the performance of NUMA systems was first studied in Mitosis [1]. Their work showed that page-table placement can have a significant impact on how well a NUMA system performs and can even have a bigger impact than the placement of data pages in some workloads. We have successfully reproduced their results on a larger, 8-socket machine by running the same experiments using the provided scripts. The results can be seen in Figure 3, where we can see that page-tables being remote (RP) degrades performance almost as much as data being remote (RD), and the interference of other applications on the inter-socket traffic further increases the overhead of remote accesses dramatically.

To mitigate the overheads caused by poor placement of page-tables, Mitosis proposes the use of eager page-table replication where the page-table of a process is fully replicated across all NUMA nodes. While this method may work well for applications where almost all data is shared between all threads, it could also result in unnecessary overheads due to the need to maintain page-table coherence across all replicas even if a portion of the address space is not shared between threads. Furthermore, while the memory footprint of page-tables is usually negligible relative to the size of the data allocated (~0.2%), our experiments show that the memory footprint due to the additional replicas is usually around 1.5% of the data addressed which could translate to several gigabytes of additional memory overhead in some workloads and grows linearly with the number of sockets.

On the other hand, Hydra is able to achieve high scalability using selective and lazy replication, where it only replicates page-tables on the NUMA nodes that the accessing threads live on. Using this approach, Hydra does not require any explicit policies from the provider/developers, whose maintenance can be cumbersome in multi-tenant scenarios. Table 1 summarizes the key differences between Hydra compared Mitosis and the baseline Linux.

Config	CPU	Data	Interference	Kernel
LP-LD	0	0	N/A	Linux
LP-RD	0	1	N/A	Linux
LP-RDI	0	1	1	Linux
RP-LD	1	1	N/A	Linux
RPI-LD	1	1	0	Linux
RP-RD	1	0	N/A	Linux
RPI-RDI	1	0	0,1	Linux
RPI-LD-M	1	1	0	Mitosis
RPI-LD-H	$0 \Rightarrow 1$	1	0	Hydra
RPI-LD-HP	$0 \Rightarrow 1$	1	0	Hydra-Prefetching

Table 2: Configuration settings for benchmarks. L (local), R (remote) denote two different NUMA nodes in the system (e.g., L: socket 0, R: socket 1). D and P denote data and page-tables, respectively. Note that page-table is fixed on socket 0 if no replication involved. CPU/Data/Interference columns identify the socket where the application executes/where the data is located/where the interfering application executes, respectively. H: Hydra, M: Mitosis, HP: Hydra with Prefetching.

3 Hydra: Design Principles

In this section, we discuss the high-level goals, principles, and major design decisions behind Hydra.

3.1 Replication Policies

Eager replication of complete page-tables on all NUMA sockets, as done in Mitosis, is highly effective at avoiding any remote page-table accesses. This style of replication is also the least complex to implement. However, maintaining complete page-table trees in all sockets and keeping them coherent results in memory and coherence overheads that grow with the number of sockets.

To reduce these overheads in the future, Mitosis envisions the use of *explicit* page-table allocation policies, by which the user would be able to specify a subset of the NUMA nodes on which the page-tables will be fully replicated; the other NUMA nodes would not have any page-tables of that process. We call this style of replication *selective* replication, where the user (or the system) can select the NUMA nodes they want to limit the replication to. We argue against such policies, because they put the burden on the user and/or the system to ensure that both the execution threads and the corresponding page-tables are co-located on the same NUMA node throughout the execution of the process. In case of any deviation, e.g., when some threads migrate, the system would need to decide if, how, and when to establish a complete new

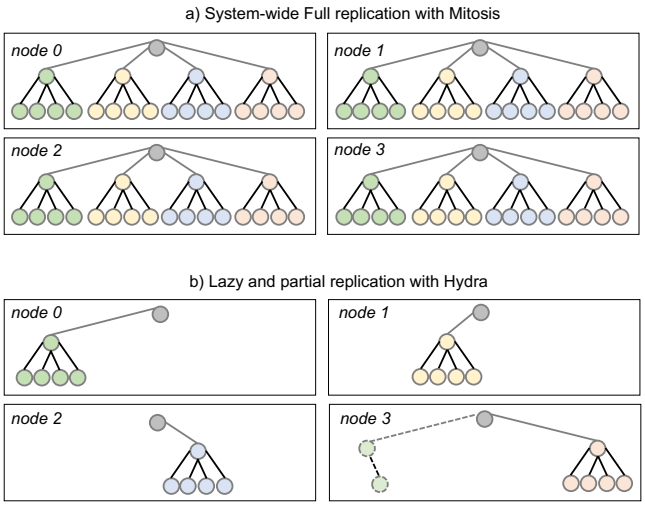


Figure 4: An abstract illustration of replication of hierarchical page-tables on Mitosis (a) and Hydra (b). Different memory allocations (VMAs) are colored differently. Mitosis eagerly replicates allocated page-tables on all NUMA nodes (sockets), whereas Hydra performs *lazy and partial* replication on-demand simultaneously instead.

copy of the page-tables on a new node, to support local address translation on the new socket. The system would also need to decide how and at which point to shut down a full copy of page-tables on a socket that a number of threads (or all threads) migrated from, in order to reduce memory and coherence overheads. While these policies could be implementable and would reduce some of the Mitosis overhead, they put a lot of burden on the user and/or the system to follow them and promptly adjust to any deviations. Although these policies could reduce the overheads compared to full-system replication by reducing the number of replicas, each replica still remains *complete*, resulting in memory and coherence overheads that eager replication cannot reduce any further.

Hydra, in contrast, uses lazy and *partial* replication, by which only the individual PTE entries that are demanded are allocated and copied to a new node. This automatically ensures 1) the minimum amount of memory spent on replication, 2) the minimum amount of activity needed to keep the replicas coherent, 3) the minimum overhead of aggregating dirty/accessed bits written by hardware in different PTE replicas, which happens when these bits are read by the OS. Importantly, our lazy approach that piggybacks on the standard page-fault procedure automatically ensures that PTEs are replicated wherever the thread is executed; when the thread is migrated the new PTE copies will be automatically established. In other words, Hydra enables selective (only on relevant nodes) and partial (only the relevant PTEs) replication in a lazy manner that automatically guarantees the co-location of execution threads and page-tables, sidestepping the problems of eager replication and explicit policies.

3.2 Page-Table Coherence Protocol

The challenge with lazy and partial page-table replication is that at the time when the requested PTE is missing, the OS needs to find out which node, if any, has a copy of the target PTE. Therefore, a mechanism that finds at least one existing sharer is needed, and if such a sharer does not exist, Hydra needs to know that.

One solution would be to statically designate one node to hold an entire copy of all page-tables, in which case partial replicas can be built lazily on the other nodes by copying the PTEs from the designated node on demand. However, the main problem with such a solution is that every page-table used has an extra sharer (the master node) that generally does not need that page-table for the translation of its own data, but nonetheless must be kept up to date. Other problems include load imbalance and the fact that the traffic to the destination node can easily become congested, either because of too much translation-related traffic or due to interference of other applications, slowing down the whole system. Furthermore, a single source node becomes a single point of failure and presents a scalability bottleneck.

We instead propose an efficient and decentralized page-table coherence protocol in which every memory allocation (e.g., virtual memory area or VMA in Linux) is assigned an *owner*; the owner of each allocation area is the NUMA socket that requested its allocation. We maintain the following invariant: if a valid PTE for a given page exists, the owner node must have it. This invariant is needed because a circular list of sharers is efficiently maintained at the level of individual page-tables [1], and we would not be able to reach that list of sharers if we do not know at least one node that is currently in the list. When the requested PTE does not exist on the local node, the entry is copied from the *owner* node as part of the page-fault handler, and the new entry is added to the list of sharers of that page-table. If the owner does not have that entry, then it means that the page has not yet been touched, so the PTE is created by taking a page fault on that page, and the entry is then inserted into both the owner's and replica's copy of the page-table and the two copies of the page-tables are linked in a circular list, if they were not linked before (for example, due to other PTEs in the same page-table).

In the case of system-wide full page-table replication [1], any changes to page-tables must be propagated to all replicas on every socket in the system. Because each replica is located on a different socket, updating all of them can take a significant amount of time. Furthermore, these updates must be performed while holding already contended memory-management locks, significantly affecting other page-table management operations happening concurrently, as we will show in Section 5. In the case of Hydra, upon any change to a page-table, only the replicas found in the list of sharers for that particular page-table are updated, if any. The coherence actions are therefore limited only to page-tables that are

actually shared, and the cost of the actions is limited by the number of nodes that actually share the page-table.

3.3 Replication and Partitioning

As an illustrative example, assume that a process with four threads runs on a 4-socket NUMA machine, one thread per socket. Also, assume that each thread is allocating a chunk of memory in its local memory and accesses only the data it allocated. This is an ideal scenario from a NUMA system point of view, as it enables perfect data parallelism. Figure 4a illustrates a potential state of page-tables in such a scenario in case of eager system-wide page-table replication, while Figure 4b shows the state of page-tables in case of Hydra. For simplicity of illustration, the page-table is depicted as a radix-tree with three levels of the hierarchy and an out-degree of four (in practice, there are 4-5 levels with an out-degree of 512). As shown in Figure 4a, full and eager replication will replicate all parts of the page-tables on all sockets. On the other hand, Hydra's protocol assigns ownership based on the allocation to the node that allocated the VMA, and as a result, each node is by default the owner of its own page-table replica. Unless a thread from one node tries to access data allocated by a thread on a different node, there will be no page-table replication in the system except for the root node. This effectively *partitions* the page-table into four independent partitions, each co-located with its data and accessed and managed locally without any cross-socket coherence activity between them, and this is all achieved automatically without intervention from the user or the system.

Apart from support for perfect data partitioning, Hydra also enables data sharing much more efficiently compared to Mitosis. Let's assume that node 3 accesses a piece of data located on node 0 for the first time. In Hydra, this will lead to the allocation of a single page-table on node 3 and the missing PTE will be copied, as shown in Figure 4b. Any updates to this page-table would limit the coherence activity only to two nodes involved, as opposed to system-wide.

3.4 Configuring Laziness

When the requested PTE entry does not exist on the local node, Hydra copies only the requested PTE entry from the owner. This is the laziest form of Hydra that we support, as no PTE is replicated unless demanded by a NUMA node. However, one could expect that such extreme laziness has a price, as the very first access to a page from a new node is guaranteed to result in a remote page-table access; this is in contrast to Mitosis, which eagerly replicates every page-table entry before it gets to be used for the first time and always ensures local page-table access.

Interestingly, we have found that extreme laziness incurs virtually no penalty in our workloads. The simple reason is that Hydra's laziness is penalized only upon the very first access to a given page by a new socket; any subsequent accesses to any part of that page by any core running on the same socket will find that PTE in the local memory. As the average number of times a page is accessed during its residence in memory is significantly higher than one for all the workloads we experimented with, we conclude that the performance penalty for laziness is negligible for most applications.

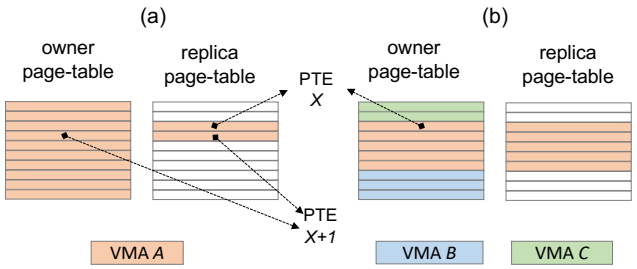


Figure 5: (a) Prefetching with a degree of 1 with PTE $X+1$ prefetched. (b) Maximum degree prefetching when the target page-table covers multiple VMAs is limited by both the page-table boundaries and boundaries of the encompassing VMA.

However, there are applications with low temporal reuse of pages that could benefit from a less lazy approach (we construct such a worst-case microbenchmark in Section 5). Furthermore, as we will show in Section 5, extreme sharing of data where every page ends up being shared on every socket, and every page-table is correspondingly allocated on every socket could effectively turn Hydra into Mitosis, with full replicas present on every socket. To improve Hydra's performance in such scenarios and reduce its laziness, we provide a low-cost support for prefetching a configurable number neighboring page-table entries together with the requested PTE, to maximize the chance of a local page-table access upon the very first access to that page by a new node. In our implementation, we provide an additional kernel parameter that allows the user to indicate the desired degree d of prefetching. Prefetching degree of d will fetch 2^d neighboring entries including the desired PTE (for d=0, in total 2^0=1 entry will be fetched, i.e., the requested PTE only). The prefetching is limited to the full page-table, as logically consecutive page-tables in the virtual address space are not necessarily physically adjacent. We also limit prefetching to the VMA boundaries, as other neighboring VMAs are not necessarily logically related, which is illustrated in Figure 5b.

3.4.1 Prefetching Overheads

The overhead of physically copying additional consecutive PTEs from the same page-table is negligible and we could not observe it in our measurements. At the same time, PTE prefetching also does not cause any PTE coherence overheads; to understand why, let's look at Figure 5a, which shows an example of one page-table that covers a fraction of single virtual memory area VMA A. Assume that the replica requests

PTE X that exists in the owner node but not in the replica. Also assume that the replica does not have this exact page-table allocated yet. Upon a page-fault on PTE X on the replica node, Hydra would by default fetch only PTE X from the owner, but in this example we choose a degree of d=1, with a total of 2^1=2 PTEs to be fetched, one of which, PTE $X+1$, is prefetched. Assume that PTE $X+1$ ends up never being used by the replica. The question is whether fetching PTE $X+1$ causes any additional coherence action that otherwise would not have happened.

Recall that Hydra does not keep the list of sharers at the level of individual PTE entries, which would be impractical and unnecessary, but at the level of individual page-tables, similar to Mitosis [1, 51]. As a result, when PTE $X+1$ is modified by the owner, the owner node cannot know if the replica has PTE $X+1$ or not; it only knows that the replica has *some* PTE entries from the page-table in question. Therefore, the replica must be updated regardless of whether or not PTE $X+1$ exists on the replica, and prefetching PTE $X+1$ does not cause any additional coherence actions. Given the negligible overheads of prefetching and potentially sizeable benefits in some applications, we suggest to Hydra users to set the degree of prefetching to maximum by default.

3.5 Reducing TLB shootdowns

When page-table entries are changed for any reason (e.g., due to the `mprotect` system call, `munmap`, a page is swapped out, etc.), the corresponding entries in the TLBs of each core that currently runs a thread of the same process have to be invalidated to ensure that stale cached page-table entries are not used in translation. This is done by sending an inter-processor interrupt (IPI) to each core that is running a thread from the same process, which causes significant overhead for all involved cores, particularly so for the initiating core. This process is known as a *TLB shootdown*. TLB shootdowns are known to be very expensive in general [38, 50] and particularly so for NUMA systems [4], where they pose a scalability challenge.

The reason that TLB shootdowns must be sent to every thread of the same process is that the OS cannot know which TLBs contain any given entry. However, Hydra has precise information about what which NUMA nodes contain a copy of any given page-table. Also note that, by design, it is not possible for any NUMA node to contain a given PTE entry in any of its TLBs unless the NUMA node is in the list of sharers for the page-table encompassing that PTE (if any TLB contained the PTE in question, then that TLB entry must be have been filled from a local copy, or the local copy would be filled together with the TLB). Therefore, Hydra can use the sharer information to safely reduce the scope of the TLB shootdowns and not issue them to any cores on those NUMA nodes that are not in the list of sharers for the particular page-table, because these nodes are guaranteed by design not to have the PTE in any of their TLBs.

4 Implementation Details

In this section, we discuss at a high level the modifications to the Linux kernel made to implement Hydra on *x86_64*.

4.1 Lazy replication

We added the following fields into the appropriate kernel structures:

- An *owner node* field in `struct vm_area_struct`, the structure that represents each allocation (VMA).
- A *next replica* field in `struct page`, the structure representing each physical page, similarly to Mitosis [1]. This field is used only in the structures representing physical pages containing page-table pages to construct a circular linked list of page-table replicas.
- A boolean flag controlling replication for a particular process in `struct mm_struct`, the structure containing all the information about a process's address space. This flag enables/disables replication for the process using a new system call that we added for this purpose.
- An array of pointers to page global descriptors (PGDs) in `struct mm_struct`, each pointing to the root of a page-table replica.

Next, we modify a few parts of the memory management component of Linux.

- When an allocation is created using `mmap`, a VMA is created representing that allocation in the function `mmap_region`. We extended this function to set the owner node of the VMA to be the node that created the allocation.
- When a context switch occurs, the page-table base register (the CR3 register on x86 and x86_64) is updated in the function `switch_mm_irqs_off`. We modified this function to set the base to the correct replica page-table (from the array mentioned earlier), if replication is enabled for the process.
- We modified the page fault mechanism in Linux to perform lazy replication. In `handle_pte_fault`, a core function in the page fault handling mechanism, if the faulting page is located in a VMA that is not owned by the current node, instead of performing a normal page fault, we attempt to copy the page-table entry from the owner node's page-table. If the entry is not present, the current node will perform the page fault on behalf of the owner node, and then copy the entry again from the owner. If prefetching is enabled, we may copy surrounding entries at this point as well, depending on the level of prefetching set, if this is the first time we are accessing memory mapped by the current page-table page. When this copying is done, we also check that both corresponding page-table pages are linked to each other

via the *next replica* field; if they are not, they are linked by merging the circular linked lists.

- Finally, we modified the functions that modify page-table entries to copy all modifications to replica page-tables by traversing the circular linked lists formed by the *next replica* fields. We also modified the functions that read the page-table entries for dirty and referenced bits written by hardware to take the union of those bits from all replicas, just like in Mitosis [1]. Note that this involves accessing memory in every socket that has the copy of the page-table that contains the same PTE. Also note that these bits are typically read in page replacement routines, which in modern operating systems are guarded by highly contended kernel locks [13, 37]. Compared to Mitosis, Hydra here has the advantage of not having to read PTE copies from all NUMA nodes, only the ones that share the page-table in question and therefore avoids additional lock contention in critical page replacement routines. However, we do not evaluate the differences between Hydra and Mitosis in out-of-memory scenarios due to the enormous amounts of memory on our evaluation platform.

Let us illustrate the replication mechanism with an example. Suppose there are two NUMA nodes, N_1 and N_2. Both nodes create a small allocation each, which we will call A_1 and A_2, respectively. Node N_1 is the owner of allocation A_1, and N_2 is the owner of A_2. At first, each node only accesses their own allocation, so each node's page-table only contains entries for the respective allocation. Note that at this point, even if the allocations fall within the same last-level page-table page, if there has not been any other cross-node access at this point, the corresponding page-table pages from both replicas will not yet be linked. Suppose now node N_1 accesses A_2 (owned by node N_2). There are no mappings for A_2 in N_1's page-table, so N_1 will take a page fault, and copy the relevant PTE from N_2's page-table. At this point, we check if the two corresponding page-table pages are linked, and if they are not, we link them together in a circular linked list using the *next replica* field, thereby ensuring any changes to the page-table entries are made to all replicas.

4.2 Reducing TLB shootdowns

By virtue of Hydra's lazy page-table replication, only nodes that have accessed a particular page will have entries for that page in their page-tables. Therefore, we implemented our TLB shootdown optimization by modifying the TLB shootdown function, `flush_tlb_mm_range`, to walk the replica lists of the pages of the page-table that map the region being shot down, and collect the nodes that have replicas. The TLB shootdown IPI is then only sent to CPUs in those nodes that are running a thread from the same process, as opposed to the Linux default, which is to send the shootdown to all CPUs running a thread from the same process.

Workload	**Description**
XSBench [61]	Monte Carlo neutron transport computational kernel applications. Dataset = 85GB, p=50M, g=200k. Only used in multi sockets experiment.
Graph500 [48]	Benchmark for generation, compression and search of large graphs. Dataset = 160GB, s=30, e=20.
Redis [39]	Single thread in-memory data structure store. Dataset = 256GB, key size = 25, element size = 64, element number = 1B, 100% reads. Only use in workload migration.
Btree [1]	Benchmarks for measuring the index lookup performance in large applications. Dataset = 110GB, 1M keys, 10B lookups.
HashJoin [12]	Benchmark for hash-table probing in database. Dataset = 145GB, 10B elements.
Canneal [8]	Simulates routing cost optimization in chip design. Dataset = 110GB, 400M elements.

Table 3: Detailed description of the workloads.

Workloads		Graph500	Btree	HashJoin	XSBench	Canneal
Program Size (GB)		160	110	145	85	110
page-table footprint (GB)	Linux	0.31	0.22	0.28	0.17	0.22
	Mitosis	2.51	1.72	2.27	1.33	1.72
	Hydra	0.67	0.44	0.4	1.32	0.32
page-table overhead (%)	Linux	0.2	0.2	0.2	0.2	0.2
	Mitosis	1.56	1.57	1.57	1.57	1.57
	Hydra	0.42	0.4	0.28	1.55	0.29

Table 4: Page-table footprint (GB) in the baseline (no replication), Mitosis [1], and Hydra for various benchmarks.

We only perform this optimization when the VMA in question falls within a single last-level page-table page, in order to avoid having to traverse the list of replicas for multiple page-table pages, which potentially means multiple remote memory accesses per node and which may end up costing more than simply sending the shootdowns.

5 Evaluation

5.1 Evaluation Platform

We conducted all measurements on an eight-socket NUMA machine with 8TB DDR4 physical memory in total. Every socket is equipped with 1TB DDR4 memory and one Intel Xeon E7-8890 v3 processor with 18 cores operating at a base frequency of 2.5 GHz and with two hyper-threads per core. Each processor has a unified 45MB L3 cache, a unified 256 KB L2 cache, a unified L2 TLB with 1024 entries and a private L1 TLB with 64 entries for each core. Hyper-threading is enabled and turbo-boost is disabled in all our experiments.

5.2 Page-table Footprint

Table 4 shows the program size and total page-table footprint for various workloads running on an 8-socket machine for three configurations: the baseline (no replication), Mitosis, and Hydra. As expected, Mitosis consistently results in 8x larger page-table footprint compared to the baseline (7x

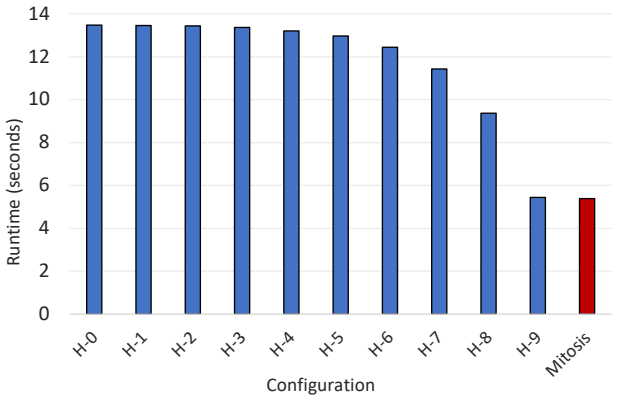


Figure 6: The time taken to traverse all pages in a large segment exactly once in a random order for various configurations. H-*i* refers to Hydra with a degree of prefetching *i*.

overhead). In contrast, the page-table footprint overhead in Hydra is only between 0.5x-1x and corresponds to the level of actual data sharing across sockets. The only exception is XSbench, which has extreme data sharing and every page-table is replicated on every socket, producing the same page-table footprint in the case of Hydra and Mitosis. In this case, Hydra effectively converges to Mitosis.

Note that Hydra's PTE prefetching has no impact whatsoever on page-table memory footprint, because the prefetching is limited to the page boundaries surrounding the requested PTE. At the time of prefetching the replica page for the requested PTE has already been allocated and the whole page is accounted for in the footprint, regardless of prefetching.

5.3 PTE Prefetching

To demonstrate the potential of PTE prefetching, we construct a microbenchmark that traverses, in a random order, a 1GB array such that every page is accessed exactly once, which is the worst case for Hydra. The array is set up and initialized on one node, and then it is accessed on the other. The benchmark is designed to achieve a near-zero hit ratio in caches and TLBs, and therefore nearly every data access results in a remote memory access for the purpose of translation.

Figure 6 shows the time it takes to traverse the array in the random order in the case of Hydra with multiple levels of PTE prefetching (from 0 - no prefetching, to 9 - maximum prefetching of 2^9=512 PTEs). We also show the results for Mitosis for comparison. We can see that prefetching within a page-table is enough to eliminate any laziness penalty that Hydra experiences by lazily copying PTEs one by one. Also note that subsequent traversals of this array would lead to identical behavior in the case of Hydra and Mitosis regardless of the level of prefetching, because at that point all page-tables are constructed and replicated where they are needed.

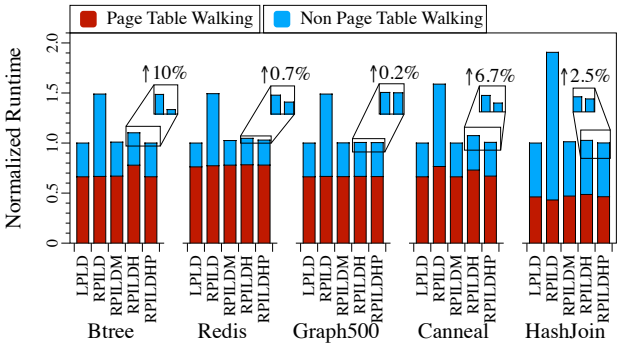


Figure 7: Normalized performance in the workload migration scenario for Mitosis, Hydra, and Hydra with a prefetching degree of 9. All of the configurations shown on X-axis are listed in Table 2.

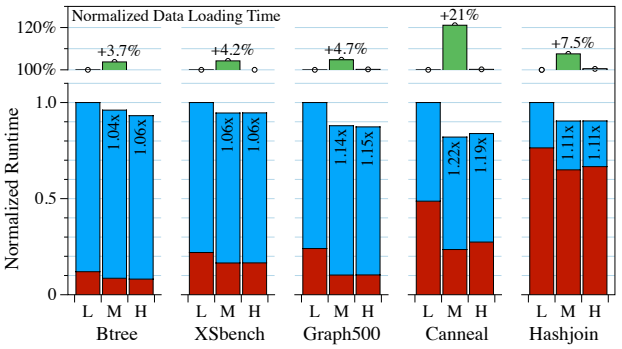


Figure 8: The lower part shows performance of Mitosis and Hydra for the workloads in a multi-socket setting, normalized to the baseline Linux. L, M and H indicate Linux, Mitosis, and Hydra, respectively. The upper part shows the data loading time in the same setting, also normalized to the base Linux.

5.4 Workload Migration

Figure 7 shows the behavior of Linux, Mitosis and Hydra in a scenario where a thread migrates to a different socket, where the data resides. In the case of Linux, the page-tables will remain on a remote socket, and the thread will be accessing the remote socket only for translation. This causes a significant slowdown in the presence of applications that interfere with inter-socket traffic (RPILD). However, Mitosis doesn't suffer from this problem as it pre-replicates the page-tables system-wide. Hydra suffers from a small performance penalty due to lazy replication (RPILDH), but that penalty is largely eliminated through prefetching.

5.5 General Applications

Figure 8 shows the performance of multiple real-world applications, listed in Table 3, running on our 8-socket machine. The workloads consist of two phases: 1) data setup/loading, which exercises page-table updates, and 2) the execution

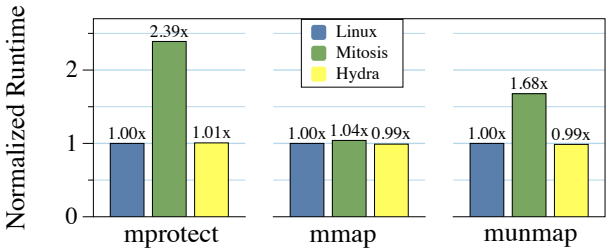


Figure 9: `mprotect`, `mmap`, and `munmap` overhead on Linux, Mitosis and Hydra when operating on a 128K memory range.

phase with the pre-loaded in-memory data, which stresses page-table reads. During the loading time (the upper plot), all page-tables are constructed, and in the case of Mitosis, replicated system-wide. The execution phase (the lower plot) exhibits a significant amount of data sharing and benefits from page-table replication across sockets; once the data is loaded, there are very few page-table modifications. Note that the data setup phase is time-consuming and takes several hours, whereas the subsequent execution is benchmarked for 5 minutes. We show three configurations: baseline (no replication), Hydra (maximum PTE prefetching enabled), and Mitosis.

From the lower part of Figure 8 we can see that in all benchmarks the performance of Hydra matches that of Mitosis despite its laziness. In some workloads, such as Btree and Graph500, Hydra even achieves a tiny speedup compared to Mitosis due to more efficient page-table coherence, to the extent to which these workloads exercise memory management. The biggest advantage Mitosis has over Hydra is for Canneal, where Mitosis achieves 1.22x speedup over Linux, whereas Hydra achieves 1.19x. However, this advantage seems to be coming at the expense of the data loading time, during which Mitosis creates 5x more replicas than needed, according to Table 4 which compares the page-table footprints of all applications. This unnecessary replication results in a 21% slowdown compared to Linux. Hydra, on the other hand, always matches the performance of Linux when it comes to data loading time, because it does not perform any replication in that phase.

5.6 Memory Management

Figure 9 shows the overhead of the basic memory management operations, `mmap`, `munmap`, and `mprotect` as a function of the size of the input range when executed on an 8-socket machine with three different designs: baseline (no replication), Mitosis (system-wide eager replication), and Hydra (partial and lazy replication). Note that there are no interfering/spinning threads. We see that `mmap` is largely unaffected by replication, as page-table updates are a small part of its functionality. In contrast, for mprotect and munmap, Mitosis pays a significant cost for page-table coherence, which Hydra avoids.

5.6.1 TLB Shootdowns

We next measure the overhead of Mitosis, Hydra without the TLB optimization, and Hydra with the TLB optimization, on the performance of the `mprotect`. The size of the `mprotect` range is a single 4KB page. The `mprotect` syscall simply flips a single bit in one PTE, and does this in a loop. Additionally, we run a varying number of spinning threads on every socket. The spinning threads have nothing to do with the `mprotect` thread; they simply increase a private counter in an infinite loop, and we measure the impact of `mprotect` on the performance of spinning threads and vice versa. The results are shown in Figure 1.

When there are no spinning threads, Mitosis results in a 25% slowdown on 8-sockets, because it must update all replicas upon every `mprotect` operation, whereas Hydra has no overhead due to the absence of coherence activity. However, as we add spinning threads and increase their number per socket, all systems, including the baseline Linux, Mitosis, and Hydra (without the TLB optimization) will result in significant overheads, up to 40x. This is despite the fact that the spinning threads have nothing to do with data covered by `mprotect`. Hydra without TLB optimization is only slightly better than Mitosis, as it avoids updating any replicas, but still suffers a significant slowdown. This is due to the TLB shootdowns that must be sent to every running thread of the process, despite the fact that these are spinning threads. In contrast, Hydra enabled with the TLB shootdown optimization avoids sending shootdowns to any other socket, leading to substantial performance improvements.

Figure 10 shows similar results in the same setup, except that instead of `mprotect` we use `munmap`, the underlying implementation of memory freeing. The range of `munmap` is set to a single 4KB page. With no spinning threads, Hydra does not experience any slowdowns, but Mitosis experiences a 23% slowdown. As we add spinning threads, the overhead of Mitosis grows to almost 30x, whereas Hydra with the TLB shootdown optimization results in only 2.6x overhead. The effect of replication on `mmap` is less pronounced due to the additional work that `mmap` does that overshadows the coherence overhead.

5.6.2 Case Study: Memory Allocation

Figure 11 and Figure 12 show the overall impact of replication on memory allocation (`malloc`) on various numbers of sockets, with one thread per socket. We use three prominent `malloc` implementations: `mmap`, `glibc`, and `tcmalloc`. We develop two `malloc` benchmarks. The first benchmark is *stateless*: the benchmark will repeatedly allocate a segment of random size following the Gamma distribution with a mean allocation size of about 3.3MB, and then free the allocated segment. The second benchmark follows the same allocation size distribution, but it first allocates 256 segments, and then in a loop frees one segment and allocates another, such that

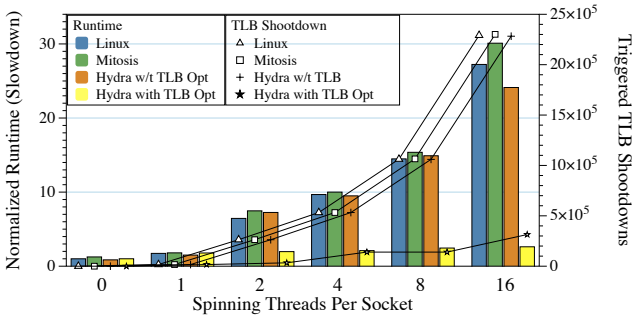


Figure 10: Impact of page-table replication and the TLB shootdown optimization on `munmap`. Existing solution are unable to quickly deallocate page-tables while Hydra does this efficiently. All values are normalized to the baseline without replication.

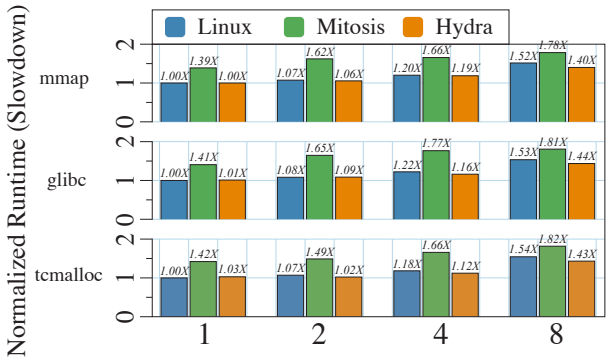


Figure 11: The impact of replication on *stateless* memory allocation for various configurations. The *x* axis indicates the socket number.

there are 256 concurrent allocations at any time for every thread. We can see that Mitosis results in an overhead that ranges from 1.4X to 1.9X in both malloc benchmarks. At the same time, Hydra achieves a speedup compared to both Linux and Mitosis, thanks to Hydra's minimal page-table coherence.

5.6.3 Case Study: Web Server

Webserver, such as Apache, is an important and widely used application that is known to have problems with TLB shootdowns. Webserver application (e.g., Apache Webserver) spawns a large number of threads, each of them serving a web request, which in our case is a web page the same as in [4, 38]. For each request, the webserver serving thread allocates memory for the web page using `mmap` and subsequently frees it using `munmap`, generating many unnecessary TLB shootdowns along the way. Previous work has demonstrated that techniques that reduce TLB shootdowns can significantly improve the Apache's throughput [38]. Because of the complex NUMA impact of the NICs [10, 49], Apache Webserver scales very poorly beyond a single socket, which

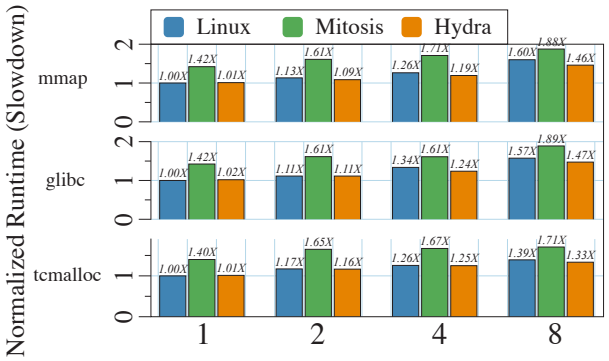


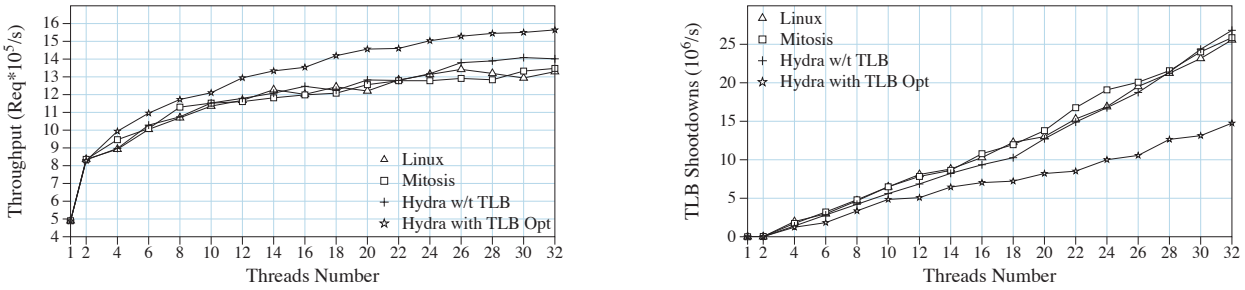
Figure 12: The impact of replication on *stateful* memory allocation for various configurations. The *x* axis indicates the socket number.

is why constructed a synthetic benchmark that performs the webserver functionality without using a NIC, similar to prior work [10, 49].

Figure 13a shows the throughput of the webserver benchmark on unmodified Linux, Mitosis, Hydra without the TLB optimization, and Hydra with the TLB optimization. We run the benchmark with a varying number of threads (up to 32) uniformly distributed across four sockets. Figure 13b shows the number of TLB shootdowns (in millions per second) as we vary the number of threads. Because this application does not exhibit any data sharing, the impact of page-table replication on the performance of page-table reads is negligible. Similarly, we see that Linux, Mitosis, and Hydra experience a similar rate of TLB shootdowns, indicating a similar overhead of page-table updates. As such, Mitosis and Hydra (without the TLB optimization) achieve similar performance to Linux. However, as we can see in Figure Figure 13b, Hydra with the TLB optimization incurs about 45% reduction in TLB shootdowns. This reduction in TLB shootdowns results in about 18-20% increase in throughput, as shown in Figure 13a.

5.6.4 Case Study: In-memory key-value store.

In-memory key-value stores, such as Memcached [46], are widely used in storing keys and values in memory to achieve low latency and high throughput. To protect the data store from data leakage, sensitive information corruption, or arbitrary accesses, `mprotect` is applied to the critical data section to protect the stored data [28, 53, 66]. We create 8 client threads that use libMemcached to send SET/GET requests. The proportion of SET and GET are 0.1 and 0.9. The size of the keys and values are 32B and 256B, respectively. The Memcached instances allow up to 1024 concurrent connections. The memory used for Memcached storage is 10GB in this experiment. To maximize the scalability, we employ a varying number of Memcached processes, with two threads per process. The threads are evenly distributed across four sockets.



a) b)

Figure 13: a, b: Impact of page-table replication and the TLB shootdown optimization on webserver, normalized to the baseline without replication. The threads are evenly distributed across four sockets.

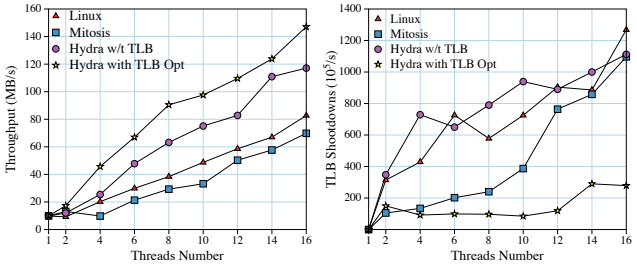


Figure 14: Impact of page-table replication and TLB shootdown optimization on Memcached running on four sockets.

Figure 14 shows the normalized Memcached throughput and TLB shootdown rate of Mitosis and Hydra over unmodified Linux. Hydra with TLB optimization always gains consistent speed-ups with a geomean improvement of 36% across all thread counts, while Mitosis suffers a slowdown, due to the overhead of synchronously keeping all page replicas coherent upon every page-table modification. The performance improvements of Hydra come from the reduction in TLB shootdowns, shown in Figure 14. Hydra with TLB optimization reduces the occurrence of shootdowns by 50% to 96%.

6 Discussion and Future Work

Support for Transparent Huge Pages. Transparent Huge Pages (THPs) are known to improve system performance by reducing the number of page-table levels that need to be traversed when performing a translation and increasing the amount of memory that can be mapped by a single TLB entry. However, THPs will still suffer from limited TLB reach in systems with large amounts of main memory. As a result, THPs would still benefit from the faster address translations that Hydra provides. Extending Hydra to support THPs is a relatively straightforward task as we would simply need to implement the same duplication and coherence mechanisms that we had previously implemented on the THPs. However, support for THPs was not considered a priority for this version of Hydra as prior work has shown that THPs may not be the best choice [2, 16, 47, 52, 58], especially for NUMA systems [27, 63].

Support for Virtualization. Virtualized systems are another potential area for future work to explore. These systems typically make use of hardware-based nested paging [25, 51] to translate the guest virtual address to the guest physical address and from the guest physical address down to the host physical address. Recent work has shown that virtualized systems are particularly sensitive to poor page-table placement [51]. While Mitosis has been extended to support 2D page-tables under vMitosis, vMitosis still works based on explicit policies given by the developers or providers. Unfortunately, it is challenging to maintain particular policies per applications and workloads. Furthermore, vMitosis also suffers from the same performance and memory overheads that Mitosis has since they both make use of eager replication. Thus, we believe that we will continue to see positive results once Hydra is extended to support the lazy replication of both the guest and host page-tables.

Reducing intra-socket TLB shootdowns. While the remote (inter-socket) shootdowns are significantly more costly, the overhead of local (intra-socket) shootdowns remains non-negligible (Figure 2). Hydra can be adapted to solve this problem by logically partitioning the cores of a socket into several domains based on their physical proximity (similarly to how scale-out processors physically partition a chip [42, 43]), while being mindful of the underlying cache architecture, and enabling lazy replication across logical domains.

7 Related Work

In this section we discuss the related work that has not been covered in earlier sections. Surprisingly, the problem of remote page-table accesses in NUMA systems is an issue that has not received much attention. However, there is a body of related work that indirectly addresses the problem by either (1) minimizing the need for address translations or by (2) speeding up the address translation process.

7.0.1 Segmented Address Spaces

Corey [9] is an experimental operating system designed for many-processor systems. In Corey, the address space of each process is partitioned into two a shared region and a private one. While this method ensures that page-tables for the private partition are located locally, accesses to the shared partition might still result in a remote page-table access since the shared page-tables are not replicated. In addition, Corey requires the application to explicitly partition the address space which places an unnecessary burden on the programmer that makes unlikely for this technique to see widespread adoption.

7.0.2 Minimizing the Need for Address Translations

The related work below aims to reduce the need for address translations through various means, which can help reduce the performance penalties of remote pages by reducing the need to do page-table walks. However, these approaches are generally either (1) too disruptive and compromise key functionalities of existing VM implementations or (2) incremental changes whose effectiveness is fundamentally limited by the existing system design.

Memory Segmentation - Memory segmentation is a memory management technique that was used in early x86 machines and allowed the OS to allocate variable sized segments to programs. While it has long since fallen out of popularity in favour of paging, some recent works have decided to explore the idea as a possible way to extend the reach of TLBs [6, 36]. However, such methods prevent the use of mechanisms such as Copy-on-Write, demand-paging and per-page protection. Furthermore, these works may experience fragmentation as they require the segments to be allocated in contiguous blocks.

Direct and Set-associative mappings - Set-associative mappings restrict the range of physical addresses a particular virtual address can map to [56, 57]. In more extreme cases, some of these methods directly map physical to virtual addresses [31]. While these methods virtually eliminate the overhead associated with address translation, they do not support important mechanisms like Copy-on-Write and demand paging in some cases [31]. In addition, such methods may also compromise the security of a system as it might impact the functionality of memory-protection mechanisms like address space layout randomization.

Multi-page mapping - Multi-page mappings allow translations with contiguous physical addresses to be mapped to a single entry in the TLB [54, 55]. While these methods do increase the reach of TLBs, the improvement is still not sufficient to support modern systems that have several hundreds of gigabytes of memory [26].

7.0.3 Speeding Up Address Translation

The following body of related work seeks to improve the speed of address translation in various ways and is largely orthogonal to our work.

RadixVM [14] proposes a number of radical changes of the memory management, including lazy replication of memory management structures across all cores, which allows it to eliminate unnecessary TLB shootdowns, similarly to Hydra. Unfortunately, RadixVM is not scalable due to the per-core page-table replication, which results in huge memory overheads. For example, on our machine with 288 cores, around 60% of the entire memory would be occupied by page-tables. RadixVM also requires a complete redesign of the kernel architecture and cannot be easily integrated into commercial OSes such as Linux. **LATR** [38] modifies a number of memory management system calls to introduces lazy and batched issuance of TLB shootdowns and is orthogonal to our work.

Prefetched Address Translation [44] takes advantage of the structure of page-tables to prefetch the lower levels of the page-table during a TLB miss. This reduces page-table walk latency without significant changes to existing systems. **Barrelfish** [7] relies on message passing instead of IPIs to shoot down TLBs in remote cores, eliminating inter-processor interrupts and lowering the latency of TLB shootdowns. Amit et al. [5] present a solution to accelerate the TLB shootdown performance by avoiding synchronous TLB shootdown calls mainly. Unfortunately, this approach is entirely oblivious to page-table replication design, and it won't be able to prevent extra TLB shootdown calls introduced by replicas. Nevertheless, Hydra can achieve even higher performance if it integrates such TLB shootdown acceleration solutions.

8 Conclusions

We have presented *Hydra*, a novel on-demand partial page-table replication mechanism for NUMA systems that simultaneously optimizes both read and update accesses to page-tables. We have shown that page-table replication, if done lazily, can be leveraged to improve rather than degrade the performance of memory management operations. Using the extra knowledge of the sharers of a page-table afforded by our lazy replication, Hydra is also to reduce unnecessary TLB shootdowns and improve the runtime of memory management operations by up to 40x. At the same time, Hydra reaps all the benefits of page-table replication, providing a 20% improvement over a wide range of applications, and provides scalability to any number of NUMA nodes.

Acknowledgments

This research was supported by the Advanced Research and Technology Innovation Centre (ARTIC) at the National University of Singapore under grant FCT-RP1 A-0008129-00-00, and by the Ministry of Education in Singapore grants A-0008143-00-00 and A-0008024-00-00.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.

[2] Ibrar Ahmed. Settling the myth of transparent hugepages for databases. https://www.percona.com/blog/2019/03/06/settling-the-myth-of-transparent-hugepages-for-databases/.

[3] AMD. The next generation amd enterprise server product architecture. https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.921-EPYC-Lepak-AMD-v2.pdf.

[4] Nadav Amit. Optimizing the tlb shootdown algorithm with page access tracking. In *USENIX Annual Technical Conference*, ATC, 2017.

[5] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shootdowns! In *ACM European Conference on Computer Systems*, EuroSys, 2020.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 2013.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles*, SOSP, 2009.

[8] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Annual Workshop on Modeling, Benchmarking and Simulation*, AWMB, 2009.

[9] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Tappan Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2008.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM, 2021.

[11] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2017.

[12] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems*, 2007.

[13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2012.

[14] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *ACM European Conference on Computer Systems*, EuroSys, 2013.

[15] Taiwan Semiconductor Manufacturing Company. Cowos services. http://www.tsmc.com/english/dedicatedFoundry/services/cowos.htm.

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

[17] Intel Corporation. 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-intelcore-processor-combine-high-performance-cpu-discrete-graphicssleek-thin-devices/.

[18] Marvell Corporation. Mochi architecture. http://www.marvell.com/architecture/mochi/.

[19] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *nternational Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2017.

[20] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGARCH Computer Architecture News*, 2013.

[21] Y Demir, Y Pan, S Song, N Hardavellas, J Kim, and G Memi. Galaxy: A high-performance energy-efficient multi-chip architecture using photonic interconnects. In *ACM International Conference on Supercomputing*, ICS, 2014.

[22] Peter J Denning. Virtual memory. *ACM Computing Surveys*, 1970.

[23] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *International Symposium on High Performance Computer Architecture*, HPCA, 2015.

[24] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2014.

[25] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging for efficient memory virtualization. *IEEE Micro*, 2017.

[26] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 2016.

[27] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. Large pages may be harmful on NUMA systems. In *Annual Technical Conference*, ATC, 2014.

[28] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. {EPK}: Scalable and efficient memory protection keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 609–624, 2022.

[29] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 649–667, USA, 2017. USENIX Association.

[30] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing memory in heterogeneous systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 637–650, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Swapnil Haria, Mark D Hill, and Michael M Swift. Devirtualizing memory in heterogeneous systems. *ACM SIGPLAN Notices*, 2018.

[32] S Iyer. Heterogeneous integration for performance and scaling. *IEEE Transactions on Components*, 2016.

[33] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *Usenix Annual Technical Conference*, ATC, 2015.

[34] A Kannan, N Jerger, and G Loh. Enabling interposer-based disintegration of multi-core processors. In *International Symposium on Microarchitecture*, MICRO, 2015.

[35] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. *SIGARCH Comput. Archit. News*, 43(3S):66–78, jun 2015.

[36] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. *ACM SIGARCH Computer Architecture News*, 2015.

[37] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *European Conference on Computer Systems*, EuroSys, 2020.

[38] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2018.

[39] Redis Labs. Redis. https://redis.io.

[40] Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 2013.

[41] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS, 2023.

[42] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[43] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer,

and Babak Falsafi. Retrospective: Scale-out processors. In *ISCA@50 25-Year Retrospective: 1996-2020*. ACM SIGARCH and IEEE TCCA, 2023.

[44] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[45] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS, 2023.

[46] Memcached. A distributed memory object caching store. https://memcached.org.

[47] mongoDB. Disable transparent huge pages. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/.

[48] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group*, 2010.

[49] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM, 2018.

[50] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *2015 International Conference on Parallel Architecture and Compilation*, PACT, 2015.

[51] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2021.

[52] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2018.

[53] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[54] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. Increasing tlb reach by exploiting clustering in page translations. In *International Symposium on High Performance Computer Architecture*, HPCA, 2014.

[55] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2012.

[56] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. Near-memory address translation. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2017.

[57] Javier Picorel, Seyed Alireza Sanaee Kohroudi, Zi Yan, Abhishek Bhattacharjee, Babak Falsafi, and Djordje Jevdjic. Sparta: A divide and conquer approach to address translation for accelerators. *arXiv preprint arXiv:2001.07045*, 2020.

[58] redis. Latency induced by transparent huge pages. https://redis.io/topics/latency.

[59] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. *SIGARCH Computer Architecture News*, 2017.

[60] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[61] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future*, 2014.

[62] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Lightweight virtual memory support for many-core accelerators in heterogeneous embedded socs. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 45–54, 2015.

[63] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[64] Idan Yaniv and Dan Tsafrir. Hash, don't cache the page table. *ACM SIGMETRICS Performance Evaluation Review*, 2016.

[65] J Yin, Z Lin, O Kayiran, M Poremba, M Altaf, N Jerger, and G Loh. Modular routing design for chiplet-based systems. In *International Symposium on Computer Architecture*, ISCA, 2018.

[66] Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, and Ion Stoica. Rearchitecting in-memory object stores for low latency. *Proceedings of the VLDB Endowment*, 15(3):555–568, 2021.