Radiant: Efficient Page Table Management for Tiered Memory Systems

Sandeep Kumar*
Processor Architecture Research Lab, Intel Labs, India
sandeep.kumar@cse.iitd.ac.in

Aravinda Prasad
Processor Architecture Research Lab, Intel Labs, India
aravinda.prasad@intel.com

Smruti R. Sarangi
IIT Delhi, India
srsarangi@cse.iitd.ac.in

Sreenivas Subramoney
Processor Architecture Research Lab, Intel Labs, India
sreenivas.subramoney@intel.com

Abstract

Modern enterprise servers are increasingly embracing tiered memory systems with a combination of low latency DRAMs and large capacity but high latency non-volatile main memories (NVMMs) such as Intel's Optane DC PMM. Prior works have focused on the efficient placement and migration of data on a tiered memory system, but have not studied the optimal placement of page tables.

Explicit and efficient placement of page tables is crucial for large memory footprint applications with high TLB miss rates because they incur dramatically higher page walk latency when page table pages are placed in NVMM. We show that (i) page table pages can end up on NVMM even when enough DRAM memory is available and (ii) page table pages that spill over to NVMM due to DRAM memory pressure are not migrated back later when memory is available in DRAM.

We study the performance impact of page table placement in a tiered memory system and propose *Radiant*, an efficient and transparent page table management technique that (i) applies different placement policies for data and page table pages, (ii) introduces a differentiating policy for page table pages by placing a small but critical part of the page table in DRAM, and (iii) dynamically and judiciously manages the rest of the page table by transparently migrating the page table pages between DRAM and NVMM. Our implementation on a real system equipped with Intel's Optane NVMM running Linux reduces the page table walk cycles by 12% and total cycles by 20% on an average. This improves the runtime by 20% on an average for a set of synthetic and real-world large memory footprint applications when compared with various default Linux kernel techniques.

CCS Concepts: • **Software and its engineering → Memory management**.

Keywords: Page Tables, NVMM, Intel Optane DC

ACM Reference Format:
Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3459898.3463907

1 Introduction

The performance of the memory subsystem, both at the software and the hardware layer, is getting increasingly important in the digital era due to the explosive growth in the amount of data generated, processed and stored. This along with DRAM scaling challenges [19, 22, 24] has led to the exploration of several new hardware memory technologies with diverse capabilities and capacities such as Intel's Optane PMM non-volatile main memory (NVMM) [20].

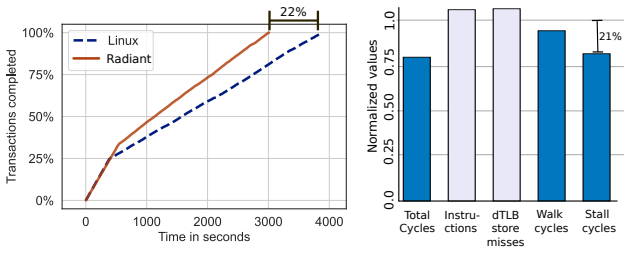


Figure 1. Redis populating 1 TB of key-value pairs. The inflection at around 500 seconds is when Linux starts allocating both data and page table pages on NVMM. In contrast, Radiant efficiently manages the placement of page table pages between DRAM and NVMM.

Modern servers typically use both DRAM and NVMMs to exploit the low latency capabilities of DRAM and high capacities of NVMMs [16, 18, 39]. Such tiered memory systems

*Work done during his internship at Intel Labs, Bengaluru, India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISMM '21, June 22, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8448-3/21/06…$15.00
https://doi.org/10.1145/3459898.3463907

bring in additional challenges in terms of managing or tiering the placement and migration of data between DRAM and NVMM. Several prior works [12, 21, 27, 35, 41] have studied these challenges for data pages and proposed solutions to identify and migrate hot data pages from NVMM to DRAM. However, they have not studied this in the context of page table pages. We argue that explicit and efficient management of page table pages is crucial for system performance for the following reasons.

❶ First, large memory footprint applications with terabytes of memory incur frequent TLB misses [5, 32, 36] as TLBs cover only a small portion of the total physical memory (covering few MBs of physical memory with 4K page size and covering up to 3 GB with 2 M pages). As a consequence, a significant fraction of the memory accesses require a page table walk.

❷ Second, the access latency of NVMMs is significantly higher than DRAM. For example, on Intel's Optane DC PMM, the read latency is 3× higher than DRAM, mainly due to the Optane's longer media latency [42]. Consequently, a hardware page table walk incurs higher walk latency when a page table page is placed in NVMM. As a page table walk requires up to 4 memory accesses upon a TLB miss (for a 4-level page table), the page table walk latency can be significantly higher in such cases which negatively impacts the application's performance (as shown in Figure 1). Radiant efficiently places the page table pages between DRAM and NVMM to reduce cycles spent in page table walks which in turn improves the start-up time of Redis by 22% (Figure 1).

❸ Third, a typical page table occupies a small fraction of memory. For example, the page table size of an application with 2 TB memory footprint is around 4 GB which is around 1% of DRAM on our evaluation system. Despite its relatively small size, page table pages can end up on NVMM even when there is enough free memory in DRAM. For instance, existing operating systems do not differentiate between page table and data page allocations; they apply the same allocation policy for both of them [3, 12, 15]. Hence, when memory interleave policy [15] is selected for data pages, page table pages are also allocated in a round robin order on all nodes, including NVMM nodes, even when DRAM has free memory.

❹ Lastly, operating systems do not support migration of page table pages [3]. Once the page table pages are allocated, they remain fixed for their lifetime; they are reclaimed only when either the corresponding data pages are freed or the process is terminated. In contrast, data pages enjoy the flexibility of migration between DRAM and NVMM based on the application's memory access pattern.

A simple and straight forward approach to avoid page table pages spilling to NVMM is to bind the page table to DRAM. However, this approach results in pathological behavior where applications are killed by the out-of-memory (OOM) handler even when significant amount of free memory is available in the system (details in §3.5). In addition, as all the page table pages are not frequently accessed, placing the complete page table on high-performance DRAM memory is not merited. Hence, we argue for judiciously managing the placement of page table pages across DRAM and NVMM.

In this paper, we propose Radiant, an efficient and transparent page table management technique for tiered memory systems. Radiant differentiates between a data and a page table page allocation by applying different placement policies to them. It also considers the underlying memory heterogeneity while deciding on the placement of the page table pages.

Additionally, Radiant employs the following techniques for efficient page table management:

- **Placement:** introduces a differentiating placement policy within the page table by placing a small but critical part of the page table in DRAM. This differentiating placement strategy is based on the observation that the top three levels of a page table tree forms a small portion of the page table but are frequently accessed during a page table walk (3 out of 4 accesses during a page walk are from the higher levels of a page table).
- **Migration:** efficiently identifies and transparently migrates the last level page table pages between memory tiers by employing a novel data-page-migration triggered page table migration technique.

We implement Radiant in the Linux kernel and evaluate the performance benefits on a real system equipped with Intel's Optane PMM persistent memory. Radiant reduces the page table walk cycles by 12% and total cycles by 20% on an average. This improves the runtime by 20% on an average for a set of synthetic and real-world large memory footprint applications when compared with the techniques employed in the Linux kernel.

The main contributions of the paper are as follows:

- Based on extensive characterization and experimentation on a diverse set of workloads, we argue that different placement and migration policies are required for data and page table pages in tiered memory systems.
- To the best of our knowledge, this is the first work that focuses on efficient placement and migration of page tables on tiered memory systems.
- A differentiating placement policy within the page table where a small but critical part of page table pages are allocated on DRAM while the rest of the page table pages are dynamically managed by migrating between memory tiers.

The rest of the paper is organized as follows: we provide the necessary background in Section 2 followed by the motivation for the paper in Section 3. We present our design in Section 4 and implementation details in Section 5. We evaluate the performance of Radiant in Section 6. We briefly

discuss related works in Section 7 and finally, conclude in Section 8.

2 Background

In this section, we cover the necessary background required for the rest of the paper.

2.1 Optane Persistent Memory

Intel's Optane Persistent Memory Module is a high-capacity non-volatile main memory (NVMM) that is DDR4 socket compatible and fits into standard DIMM slots [20]. Optane can be used either as a high-capacity volatile main memory (Memory Mode and Flat Mode) or as a persistent memory (App Direct Mode) [33, 42]. Large memory footprint applications can exploit the additional memory capacity when Optane is configured as a high-capacity volatile memory. For example, Optane can seamlessly enable large-scale in-memory graph analytics for graphs with billions of edges [16]

In this work, we use Optane as a high-capacity volatile memory in Flat Mode (also referred to as DRAM-NVMM hybrid mode [33]). The difference between Memory Mode and Flat Mode is that in Memory Mode, Optane acts as a byte-addressable volatile main memory while DRAM acts as a cache; software has no control on the data placement. In Flat Mode, both DRAM and Optane memory can be accessed as a unified, but heterogeneous, byte-addressable memory. The advantage with Flat Mode is that the software can control and optimize the placement of data between low latency DRAM and high latency Optane [18, 39].

We configure the system in Flat Mode using `ndctl` tool [31] and `daxctl` utility [13]. Step by step guide to configure Optane as a hot-plugged main memory is available in Persistent Memory Development Kit (PMDK) [34]. Once configured in Flat Mode, Optane memory is reflected as "no-CPU" NUMA nodes in the system as shown in Figure 2 (node 2 and node 3). Support for Flat Mode is already part of the Linux kernel [17] and hence, all the NUMA features (e.g., placement and balancing) in Linux are readily available for Optane-backed NUMA nodes as well.

2.2 Page Tables

A page table maintains virtual address (VA) to physical address (PA) translations and is organized as a multi-leveled tree (`x86_64` supports both 4-level and 5-level page tables; we use 4-level page table for the discussions in the rest of the paper[1]) where a page global directory (PGD or L1) is the root of the tree. Each active entry in PGD points to a physical page containing an array of page upper directory (PUD or L2) entries. Similarly, each active entry in PUD points to a physical page containing an array of page middle directory (PMD or L3) entries. PMDs in turn point to a physical page (PTE or L4) containing an array of page table entries. A PTE

[1] A 4-level page table can map up to 256 TB of memory.

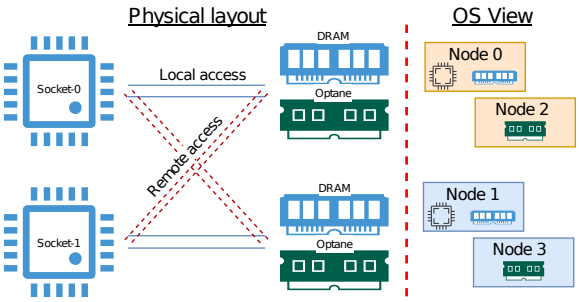


Figure 2. A 2-socket system equipped with Intel's Optane memory. The two sockets are logically divided into four NUMA nodes in Linux. Node 0 and Node 1 are backed by DRAM while Node 2 and Node 3 are backed by Optane.

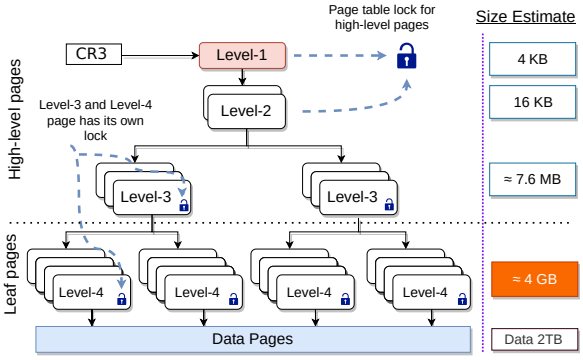


Figure 3. Figure depicting the structure of a 4-level page table.

entry contains the physical page address of the data page corresponding to the virtual address as shown in Figure 3.

Upon a CPU TLB (Translation Lookaside Buffer) miss, the hardware – being aware of the page table tree layout – performs a page table walk to insert an entry in the TLB. As TLBs cover only a small portion of the total physical memory, most of the memory accesses by large memory footprint workloads cause a TLB miss requiring a page table walk.

In modern operating systems, page tables are dynamically allocated: the root of the page table tree for a process is allocated when the process is created. The physical pages to store the intermediate and leaf-level pages of the page table are allocated whenever the process page-faults on a valid virtual address for the first time.

2.3 Userspace Data Page Allocation and Migration

Modern operating systems such as Linux provide a stable and transparent technique for data page allocation on a multi-socket system. Additionally, they also provide mature interfaces or APIs for applications to explicitly control data page allocation. By default, Linux employs a first-touch policy [3, 15], which allocates data pages on a local NUMA node

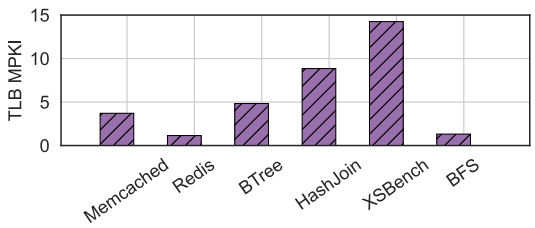


Figure 4. TLB MPKI for applications with large memory footprint. Benchmark details in Table 2.

and falls back to remote nodes when there is not enough memory on the local node. Apart from this, an interleaved allocation policy [15] is also available where the data pages are allocated on all NUMA nodes in a round robin order. This improves memory bandwidth utilization by distributing the data pages across nodes and thus, avoids skewed allocation to a set of nodes [15].

In a NUMA system, accessing data from a remote node causes significant memory overheads incurring 2–4× higher latency than accessing the data from a local node [3]. Many solutions have been proposed over the last few decades to mitigate such performance issues, including migration of the data pages from the remote NUMA node to a local NUMA node [12, 25, 44].

Operating systems such as Linux provides well defined userspace APIs to trigger data page migrations between NUMA nodes [26]. In addition, operating systems are capable of transparently migrating frequently accessed data pages between NUMA nodes (e.g., AutoNUMA in Linux [10]). However, it is important to note that the page migration support is only available for userspace data pages and not for kernel pages.

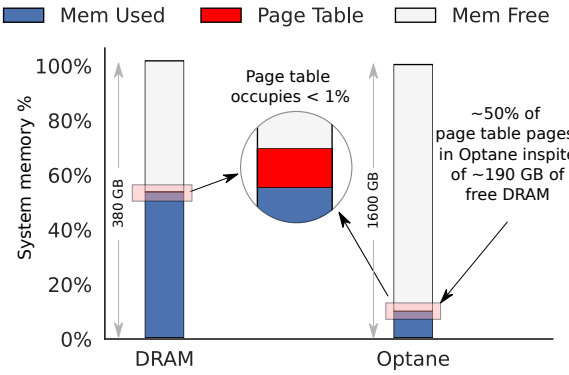
3 Motivation

In this section, we present page table analysis for large memory footprint applications including the placement and distribution of page table pages, migration of page table pages and performance impact of page table placement. System and configuration details are in Table 1.

3.1 TLB Misses

Large memory footprint applications using terabytes of memory incur frequent TLB misses as TLBs cover only a small portion of the total physical memory. Figure 4 shows the TLB Misses-Per-Kilo-Instructions (MPKI) for applications with large memory footprint (600 GB to 1 TB). A higher MPKI implies that a significant fraction of the memory accesses incurs TLB misses, thus requiring page table walks.

It is important to note that MMU employs caching techniques to cache the page table entries to reduce page walk overheads. Additionally, page table entries are also cached in system memory caches as MMU units access the page table



	DRAM		Optane	
Memcached stats	Node 0	Node 1	Node 2	Node 3
Data (in GBs)	79.00	90.00	84.00	84.00
Page table (in GBs)	00.17	00.17	00.16	00.16

Figure 5. Page table distribution for Memcached when around 338 GB of data has been populated with interleaved allocation policy. Around 50% of page table pages end up in NVMM even when 190 GB of DRAM is free.

through the memory hierarchy. Despite MMU caching and other TLB optimization techniques, we observe that large memory footprint applications spend up to 68% of the total execution cycles in page table walks. This observation is also consistent with previous findings [3, 4, 6, 7, 28, 43].

3.2 Page Table Placement

Operating systems dynamically allocate pages for all the four levels of page table on-demand, i.e., when the corresponding virtual address page faults for the first time. However, the NUMA node on which a page table page is allocated depends on multiple factors including the socket on which the allocating thread is running and the memory allocation policy of the application [12, 15]. It is important to note that operating systems employ the same allocation and placement policy for both data and page table pages.

Figure 5 shows the placement of page table pages and data pages when around 338 GB of data has been populated in Memcached using memory interleave policy (round-robin allocation of data and page table pages across all NUMA nodes). It can be observed that around 50% (0.32 GB) of page table pages are allocated in Optane despite having around 190 GB free memory in DRAM.

But in first-touch allocation policy, allocation of both data and page table pages spills over to Optane only when DRAM is almost full. Later when a part of DRAM memory is freed, data pages are migrated from Optane to DRAM. However, page table pages remain in Optane as they cannot be migrated.

As a result, in one scenario, page table pages can be allocated in NVMM even when enough free memory is available in DRAM and in another scenario page table pages allocated

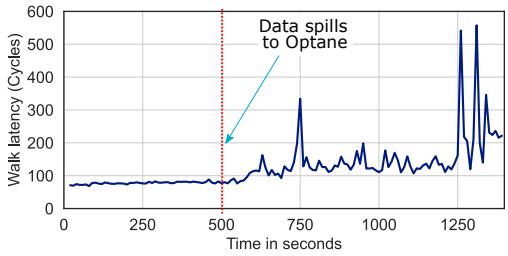


Figure 6. Page walk latency when populating Redis with 1 TB of key-value pairs (we plot the first 1400 seconds, but the trend continues). Page walk latency increases when the page table page allocation spills to Optane.

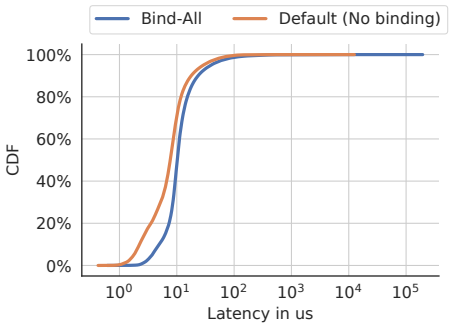


Figure 7. L4 page table page allocation latency in case of the default Linux kernel and when the entire page table is binded to DRAM.

on NVMM remains on NVMM even when enough memory is freed on DRAM (**Observation 1**).

3.3 Page Walk Latency

The access latency of NVMMs are significantly higher than DRAM mainly due to the longer media latency. Hence, a hardware page table walk incurs higher walk latency when a page table page is placed in NVMM. Additionally, a page table walk requires up to 4 memory accesses to NVMM when all the four levels of page table pages are allocated in NVMM. This further increases the page walk latency. It has also been observed that concurrent access to NVMMs, especially Optane, from multiple CPUs in a multi-core system can degrade performance due to limited internal buffers [42].

We measure the page walk latency when populating Redis with 1 TB of key-value pairs using the default first-touch policy. Page walk latency increases significantly (Figure 6) when the page table page allocation spills to NVMM (**Observation 2**).

3.4 Migration Support

Techniques employed by operating systems and userspace applications to identify and migrate frequently accessed pages from NVMM to DRAM to improve application performance are restricted to data pages and cannot be directly extended to migrate page table pages. Because, the design of most modern operating systems does not allow migration of kernel data (which includes page tables). As a consequence, once page table pages are allocated, they remain fixed for their lifetime; they are reclaimed only when either the corresponding data pages are freed or the process is terminated. As a result, page table pages that are allocated on NVMM remain in NVMM.

Furthermore, enhancing the kernel to enable page table page migration is a non-trivial operation as it requires fixing the page table tree structure to ensure that the virtual to physical address mappings are intact. In addition, page table page migration on a multi-core system requires careful handling of race conditions. For example, the page table page under migration can either be accessed by hardware during a page walk or can be accessed/modified by other CPUs to serve a page fault.

3.5 Page Table Binding

A simple and straight forward approach to avoid page table pages spilling to NVMM is to bind the page table to DRAM. Even though this looks like a viable option, it results in pathological behaviours as we demonstrate by evaluating the Linux kernel patches [40] that propose to bind the page table to DRAM.[2]

We start populating Memcached in-memory database with the default first-touch allocation policy on a freshly booted system. Initially, all data and page table page allocations for the in-memory database are directed to DRAM (as per first-touch policy) resulting in DRAM nodes filling up before Optane nodes.

Once DRAM is almost full, all new data page allocations are directed to Optane nodes, while the page table pages are still directed to DRAM due to DRAM binding. Forcing the page table page allocations on almost-full DRAM nodes results in higher allocation latencies (Figure 7) as the buddy allocator falls back to slowpath function that performs additional work of compaction and page reclamation.

Interestingly, reclaimed free pages in DRAM are used to allocate both data and page table pages as per first-touch policy. This quickly fills up DRAM triggering another round of reclamation for a page table page allocation request. As DRAM is just 19% of the total memory on our system, the cycle of reclaiming DRAM memory and filling it up again (a thrashing kind of situation) starts early during the initialization of in-memory database and continues as we populate key-value pairs in the database.

[2]These patches are not included in the Linux kernel; Linux kernel v5.6 still allows allocation of page table pages on Optane NUMA nodes.

However, after a while, the Linux kernel fails to reclaim enough DRAM pages to serve page table page allocation requests and as a result triggers the out-of-memory (OOM) handler. OOM handler kills the Memcached server even when 700 GB of free memory is available in Optane NUMA nodes.

Out-of-memory issues can be mitigated to some extent by employing aggressive page reclamation heuristics, but mitigating high page table allocation latencies and thrashing issues require complex changes to the kernel. We address these challenges fundamentally by efficient allocation and placement of page table pages across memory tiers.

3.6 Summary

To summarize, we argue that with the growing relevance of large tiered memory systems, it is important to explore efficient page table allocation and placement technique across memory tiers, which has received least attention till now.

4 Radiant Design

We propose an efficient and transparent page table management technique to reduce page walk overheads on tiered memory systems. In this section, we present the design of Radiant.

4.1 Design Considerations

Differentiate between data and page table pages: Large memory footprint applications with terabytes of memory incur frequent TLB misses. The performance of such applications is sensitive to the placement of page table pages in a tiered memory system. Hence, it is necessary to consider different allocation and placement policies for data and page table pages.

Differentiate between NVMM and DRAM memory: Carefully consider the underlying memory heterogeneity (e.g., capacity, latency) while deciding on the placement of page table pages.

We propose the following two techniques that incorporate the above design considerations along with the observations made during page table analysis in §3.

4.2 Binding Critical Page Table Pages to DRAM

The read latency on NVMM is 3× higher than DRAM mainly due to the longer media latency. As a page table walk requires 4 memory accesses, the page table walk latency is significantly higher when all the four levels of the page table pages are allocated on NVMM. Even though a typical page table for a large memory footprint application can occupy a small fraction of DRAM, binding the entire page table to DRAM can result in pathological behaviours as demonstrated in §3.5.

We observe that a majority of the page table memory is consumed by leaf level or L4 page table pages; L1, L2 and L3 page table pages together consume insignificant amount of memory. For example, an application with around 2 TB memory footprint requires around 4 GB memory for L4 pages and collectively requires around 7.62 MB for L1, L2 and L3 page table pages (size estimation in Figure 3). We exploit this insight to significantly reduce the amount of time spent on page table walks.

Our placement strategy is to dynamically allocate and bind L1, L2, and L3 page table pages in DRAM. With such a placement technique, during a 4-level page walk, 3 out of 4 memory accesses are guaranteed from low latency DRAM thus drastically reducing the page walk cycles. It is important to note that we achieve this by strategically placing less than 0.18% of page table pages in DRAM.

Such a policy not only improves the application execution time but also improves startup or initialization time for large memory footprint applications. For example, when populating initial key-values in an in-memory database, initializing a large graph, or restoring a VM snapshot, a large portion of L1, L2, and L3 page table pages are initialized and accessed (e.g., zeroing a newly allocated page table page). Hence, placing them in DRAM reduces the startup time of applications.

Our strategy, as opposed to placing the entire page table in DRAM [40] has several advantages. ❶ First, we drastically minimize the amount of page table pages that requires binding to DRAM. For example, we bind only 7.62 MB for a 2 TB workload which is less than 0.0019% of DRAM on our evaluation system. In contrast binding the entire page table requires 4 GB of DRAM. ❷ Second, by using less than 0.0019% of DRAM for binding we guarantee 75% of page table walks from DRAM. ❸ Finally, even under extreme memory pressure operating systems can allocate L1, L2 and L3 page table pages in DRAM by reclaiming a small amount of DRAM memory. While binding the entire page table requires reclaiming few GBs of DRAM memory which can trigger out-of-memory handler.

4.3 Page Table Migration

We allow allocation of L4 page table pages, which constitutes the majority of the page table pages, on both DRAM and NVMM. Further, we use data-page-migration triggered page table migration technique to efficiently identify and migrate L4 pages between DRAM and NVMM. With this technique we derive hot/cold page table pages from the hotness of the data pages, thus eliminating explicit page table tracking overheads.

The rationale behind such an approach is that a data page migration provides crucial hint on the placement of the corresponding L4 page table page. For example, migration of a hot data page from NVMM to DRAM hints that the corresponding L4 page table page, if present on NVMM, should also be migrated. Because, for a large memory footprint application with terabytes of memory even a hot data page incurs frequent TLB misses (as the amount of hot data far

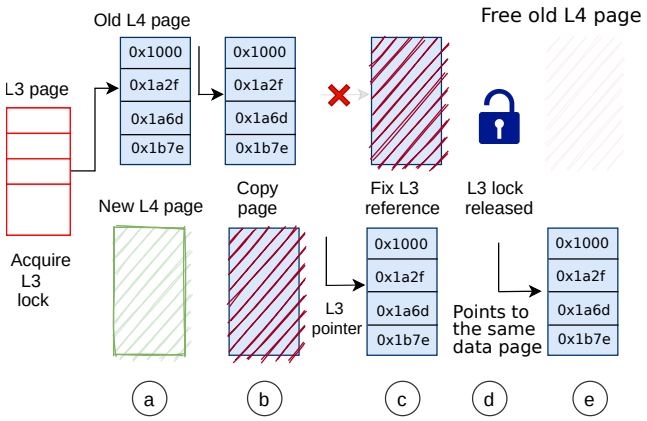


Figure 8. Steps for a page table page migration. A shaded page indicates an un-referenced page.

more exceeds the TLB reach) resulting in frequent accesses to L4 page by the hardware page walker. Therefore, when a data page is migrated between memory tiers we trigger the migration of the corresponding L4 page table page.

Operating systems such as Linux provides a well defined userspace API [26] to trigger data page migrations to enable novel userspace techniques to efficiently identify and migrate data pages between memory tier. For example, identifying and migrating hot and cold data pages between memory tiers or speculatively pre-migrating a set of data pages between DRAM and NVMM based on the application's memory access patterns. In addition, operating systems are capable of transparently migrating frequently accessed data pages between NUMA nodes (e.g., AutoNUMA in Linux). We exploit such existing data migration techniques to trigger an L4 page table page migration between DRAM and NVMM.

We migrate an L4 page from NVMM to DRAM upon the migration of the corresponding data page, however, we migrate an L4 page from DRAM to NVMM only when the last data page it is pointing to is migrated to NVMM. This is to ensure that an L4 page is in DRAM if any data page it is pointing to is in DRAM.

4.4 Page Table Migration Details

As mentioned before, the core design of many operating systems does not allow migration of kernel data which includes page table pages. We exploit the page table tree structure to enable migration without changing the core kernel design.

Algorithm 1 and Figure 8 show the steps involved in migrating an L4 page table page. Whenever a data page migration is initiated either by a userspace program or by the kernel (e.g., AutoNUMA), we trigger the migration of the corresponding page table page. The L4 page migration is initiated after the corresponding data page migration is successfully completed (Line 4).

Algorithm 1 Algorithm to migrate an L4 page

```
1:  procedure MIGRATE_DATA(data_page, dest_node)
2:      data_page_new ← alloc_page(dest_node)
3:      rc←migrate_data_page(data_page, data_page_new,
    dest_node)
4:      if rc==SUCCESS then
5:          migrate_L4(data_page_new, dest_node)      ▷Migrate L4 page
6:      end if
7:  end procedure
8:  ---------------------------------------------------------
9:  procedure MIGRATE_L4(data_page_new, dest_node)
10:     /* Walk the page table */
11:     (L4, L3) ← get_pt_entries(data_page_new)
12:     L4_node ← page_node(L4)                          ▷Get L4's node
13:     if L4_node == dest_node then
14:         return                                       ▷Already in destination
15:     else if L4_node in DRAM and dest_node in DRAM then
16:         return                                  ▷Already in DRAM (similarly for NVMM)
17:     else if dest_node in NVMM then
18:         if any data page pointed by L4 in DRAM then
19:             return                               ▷L4 pointing to a page in DRAM
20:         end if
21:     end if
22:     if lock(L4 and L3) then                          ▷Lock L4 and L3 pages
23:         /*Allocate a L4 page on the destination NUMA node*/
24:         L4_new ← alloc_page(dest_node)
25:         tlb_flush()                                  ▷Invalidate L4_old mappings
26:         memcpy(L4_new, L4, 4096)                     ▷Copy the L4 page
27:         update_L3(L4_new)                            ▷Sync. point
28:         unlock(L3 and L4)                            ▷Unlock the L3 and L4 pages
29:     end if
30: end procedure
```

To migrate a page table page we first fetch L4 and L3 pages corresponding to the new data page ($data\_page_{new}$) by performing a software page table walk (Line 11). Once we have L4 page, we get its NUMA node. We skip the migration if L4 page is already in the destination NUMA node (Line 14) or if the migration is from one DRAM (or NVMM) node to another DRAM (or NVMM) node (Line 16). We also skip the migration of L4 page from DRAM to NVMM if any data page pointed by L4 is in DRAM (Line 19).

On meeting all the necessary conditions, we start the migration by locking L4 and L3 page table pages. Locking is required to synchronize between parallel data or L4 migrations, which is common in multi-core systems. Now we allocate a new L4 page ($L4_{new}$) on the destination NUMA node. If successful, we flush the TLB and MMU caches to invalidate any entries pointing to old L4 page and then copy the contents from old L4 page to $L4_{new}$ and update L3 to point to $L4_{new}$ (Line 27).

TLB flushing forces a hardware page walk on CPUs that concurrently attempts to access the old L4 page under migration, while an invalid old L4 entry triggers a page fault. The operating system's page fault handler being aware of the ongoing L4 migration waits for the migration to complete before inserting the updated mapping.

4.4.1 Page Table Consistency. In a multi-core system, multiple CPUs can concurrently try to access an L4 page under migration in the software page fault handler. Furthermore, similar to a data page migration, an L4 page migration can also be triggered simultaneously, thus, requiring explicit synchronization during a page table migration. We also need to ensure that the hardware page table walker sees a consistent state of the page table at all the times.

Even though Algorithm 1 provides generic steps to migrate an L4 page, the actual implementation and sequence of steps (e.g., when to flush TLB entries) may vary depending on the underlying architecture and the operating system.

5 Implementation

In this section, we explain the implementation details of Radiant for x86_64 architecture in the Linux kernel. We use the Linux kernel's terminology to refer to different levels of a page table; L1 is referred as PGD, L2 as PUD, L3 as PMD, and L4 as PTE.

As explained before, the default kernel only migrates data pages during a migration. Enabling PTE migration on a multi-core system is not trivial; a simple pointer flip at the PMD-level and freeing of the old PTE page is not enough. We list down a few challenges in implementing PTE migrations on a production-class operating system such as Linux:

❶ Multiple CPUs in a multi-core system, upon a TLB miss, can concurrently perform page walk by accessing the page table pages using the physical addresses. Hence, we need to ensure that the hardware always sees a consistent page table.

❷ As a PTE page points to 512 data pages, it is possible to have multiple concurrent migrations of these data pages to different NUMA nodes. Every such instance of successful data migration triggers a PTE page migration. We need to ensure that the page table is consistent without causing a significant performance overhead.

In the subsequent sections, we explain implementation details including challenges and solutions.

5.1 Binding the High-Level Page Table Pages

The default Linux kernel allows us to specify memory policies for applications to bind to specific NUMA nodes. However, Linux does not support binding page table pages independent of the data pages. We modify the page table page allocation functions in the kernel, `pgd_alloc()`, `pud_alloc()`, and `pmd_alloc()`, to add support to bind PGD, PUD, and PMD pages in DRAM.

We extend the `numactl` utility [8] to enable the processes for which the high-level pages of a page table should be placed in DRAM. Placement of high-level page table pages is independent of data page placement for processes enabled with `numactl` binding. Rest of the processes in the system follow the data page placement policy for page table pages.

5.2 PTE Migrations

The Linux kernel ensures that a data page under migration is completely isolated from the rest of the system. Any page fault on this page waits either on the locked PTE or the locked data page until the migration is complete.

As shown in Figure 8, we first try to acquire the PMD lock. If successful, a new PTE page is allocated on the destination NUMA node using `alloc_pages_node()` function. Then, we copy the page content from the old PTE page to the new PTE page and fix the page table (update the PMD entry to point to this new PTE).

We also flush the TLB entries and MMU cache to clear the old PMD to PTE mappings. But, the PTE to data page mappings are still valid as we copy the contents of old PTE page to the new PTE page (see Figure 8). After the PMD to new PTE page mapping is updated in the page table, any TLB miss will use the new PTE page instead of the old PTE page; the hardware need not wait for the release of the lock on the old PTE page.

5.3 Performance Implications

The page table of a process has three types of locks; a page table lock, a per-PMD page lock, and a per-PTE page lock (see Figure 3). The per-PTE (or per-PMD) page lock allows for parallel updates across different PTE (or PMD) pages without locking the whole page table. This significantly improves the performance of operations on the last level (or PMD-level) of the page table in a multi-core system [11, 14].

As explained in Section 4.4, we obtain the PMD lock prior to updating the PMD entries. This is required to avoid a race condition where a parallel migration on another CPU updates the PMD entry. However, locking the PMD serializes the migration of data pages mapped within the PMD with the migration of the corresponding PTE pages. This delays the completion of a data page migration, which in turn increases the page fault latency as the Linux kernel's fault handler has to wait for the completion of the migration. To mitigate the latency overheads, we try to lock the PMD using `try_lock()` prior to migrating a PTE page. If we cannot get the lock, we skip the PTE page migration. As a PTE page points to 512 data pages, it is possible that we will get many more opportunities to migrate the PTE page.

6 Evaluation

In this section, we evaluate the performance of Radiant on a suite of real-world applications and synthetic benchmarks, and compare it with the Linux kernel's memory allocation policies and management techniques. Table 1 provides details on the experiment setup. Support for transparent huge page (THP) is disabled unless otherwise stated. We use an unmodified Linux kernel 5.6 for all our baseline evaluations and enhance it to implement Radiant. Table 2 lists the workloads and Table 3 lists the conventions used for the evaluation.

Table 1. System configuration

Hardware			
CPUs (2×24×2=96)		**Memory (2 TB)**	
Model	Intel-Xeon Gold 6252N	DRAM	384 GB
CPUs	2 Socket, 24 Cores, 2 HT	Optane	1.6 TB (Flat Mode)
System settings			
Linux Kernel: 5.6	DVFS: Performance		ASLR: Off
NUMA: 4 Nodes			
Node 0/Node 1		**Node 2/Node 3**	
CPUs	48	CPUs	0
Memory	DRAM 192 GB	Memory	Optane 800 GB

Table 2. Workloads used to evaluate the performance of Radiant. RSS (resident set size) and PT (page table) size shown.

Name	**Description**	**RSS** (PT size)
Memcached [29]	A commercial in-memory object caching system. Setting: YCSB [9]: 2 M objects. Read using a Zipfian distribution [30].	1 TB (1.9 GB)
Redis [23]	A commercial in-memory key-value store. Setting: Same as Memcached.	1 TB (1.9 GB)
BTree [1]	A benchmark for index look-ups used in database and other large applications. Setting: 7.3 B elements with 40 M look-ups.	666 GB (1.2 GB)
HashJoin [2]	A benchmark for hash-table probing used in database applications and other large memory footprint applications. Setting: 6 B elements.	838 GB (1.6 GB)
XSBench [38]	A key computational kernel of the Monte Carlo neutron transport algorithm [38] Setting: 2 M grid points.	1 TB (1.9 GB)
BFS [37]	A graph traversal algorithm. Setting: rMat order 30 graph [37]	600 GB (1.1 GB)

Table 3. Conventions used in the paper for discussion

Radiant techniques	
BHi	Bind high-level (PGD, PUD and PMD) page table pages in DRAM
Mig	Enable migration of last-level (PTE) page table pages
BHi+Mig	Enabling binding of high-level page table pages along with migration for the last-level of a page table.

6.1 Evaluation Strategy

We compare the performance of Radiant techniques with two memory allocation policies in the default Linux kernel.

❶ First is the default first-touch policy [3, 15]. In this case, the NUMA node for the page table pages is selected based on the data page allocation policy, i.e., a page table page is allocated on the same NUMA node where the data page is allocated. This policy allocates a data page on a NUMA node that is close to the CPU where the application is running – a local NUMA node [15]. However, the allocations can spill over to remote NUMA nodes when an allocation request cannot be served from the local NUMA node.

❷ Second is the interleaved policy where the Linux kernel distributes the data uniformly across all the NUMA nodes in a round-robin order to improve memory bandwidth utilization.

To enable PTE migrations, we rely on the Linux kernel's memory management technique called AutoNUMA to get data page migration hints. By default, AutoNUMA dynamically migrates data pages only (not page table pages) across NUMA nodes to improve local NUMA accesses from a CPU. We run the experiments with AutoNUMA enabled unless otherwise mentioned.

We are unable to evaluate page table binding technique [40] because of out-of-memory issues mentioned in §3.5. For example, we are unable to fully populate the Memcached in-memory database as the server is killed due to such issues.

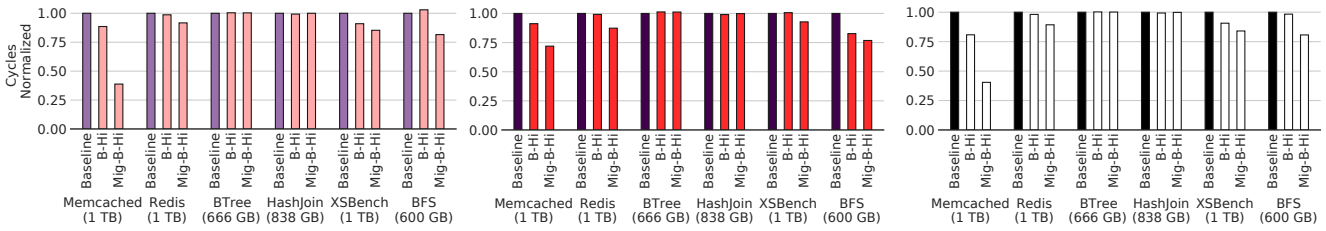
Our evaluation strategy is as follows:

- **Full-system run:** Run the workloads with full system capacity utilizing maximum possible resources, which reflects a typical real-world data center scenario. We compare the performance of Radiant (BHi and BHi+Mig) with Linux kernel's first-touch policy.
- **Multi-tenant scenario:** Evaluate the performance benefits of Radiant in a multi-tenant environment (a typical cloud setting), where different applications can start and exit at any point in time.
- **Interleaved setting:** Compare the performance of Radiant (BHi) with the interleaved memory allocation policy, with AutoNUMA disabled. We show that differentiating between allocation of data and page table pages improves the performance.
- **Start up time:** At the startup of a large memory footprint application, a significant portion of high-level (PGD, PUD, and PMD) page table pages are initialized. We evaluate the performance benefits of BHi in such scenarios.
- **Huge page impact:** Evaluate the performance benefits of Radiant when huge pages are enabled.

6.2 Full-System Run

We evaluate the performance of workloads with the memory footprint size as specified in Table 2 utilizing maximum possible system resources. We compare the performance of the Linux kernel's first-touch policy (baseline) with Radiant (BHi and BHi+Mig) techniques (see Figure 9).

BHi: The high-level page table pages are frequently accessed during a page table walk. Binding them to DRAM ensures a low-latency access during a page table walk and reduces the walk cycles by up to 17.31%. Placement on DRAM also reduces the stall cycles by up to 19.18%. This translates into



(a) Total cycles **(b)** Walk cycles **(c)** Stall cycles

Figure 9. Performance comparison of first-touch policy with Radiant, for the run phase (data loading phase is not shown).

Table 4. Radiant performance improvement summary (geometric-mean across all the workloads). A higher value indicates better performance improvement with Radiant.

	Run Time	Cycles	Walk Cycles	Stall Cycles
Full system run: First-touch policy				
BHi	2.79%	3.32%	4.56%	5.68%
BHi+Mig	20.39%	20.71%	12.38%	20.9%
Multi-tenant scenario: First-touch policy				
BHi+Mig	17.95%	19.85%	32.62%	23.25%
Interleaved: AutoNUMA disabled, Interleaved policy				
BHi	10.41%	10.02%	10.53%	9.01%
Huge page impact: AutoNUMA disabled with THP enabled				
BHi	52.96%	51.82%	36.37%	38.63%
Start up time improvement: AutoNUMA disabled (Redis)				
	Time	**Avg Lat.**	**Max Lat.**	**95th%ile Lat.**	**99th%ile Lat.**
BHi	22.81%	22.82%	17.35%	25.56%	20.70%

a reduction of total cycles by up to 11.43% and a run-time improvement of up to 9.08% (see Table 4).

BHi+Mig : With PTE migrations enabled, the percentage of page table pages in DRAM increases (e.g., from 19.6% to 34.0% for Redis). This reduces the walk cycles by up to 28.06% and the stall cycles by up to 59.57%. This causes a reduction in the total cycles by up to 61.19% and improves the run-time by up to 60.88% (see Figure 9).

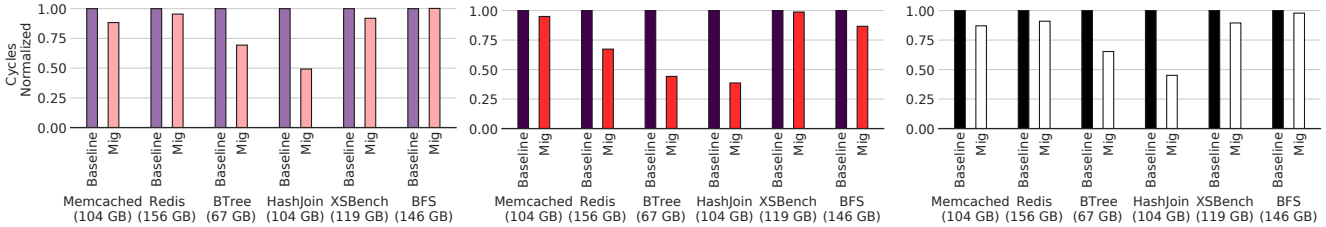
6.3 Multi-Tenant Scenario

In a typical cloud setting, where tiered memory is likely to be deployed, many applications co-exits in parallel in a given period of time. Here, different applications may start or exit at any point in time.

An application (V) started when DRAM is almost full is allocated memory (data and page table pages) on NVMM. At a later point in time when other applications using DRAM exit, DRAM becomes free resulting in the migration of the data pages of V from NVMM to DRAM. However, with the default Linux kernel, the page table pages are not migrated from NVMM, incurring performance overheads even in spite of free memory in DRAM. To capture the benefits of Radiant in such scenarios, we setup a cloud-like environment and compare the performance of Radiant with the default Linux kernel.

To setup the environment, we first launch applications that fill up DRAM. These applications also frequently access the data pages in DRAM. Then we launch our benchmark application. As DRAM memory is full, all the benchmark application's memory is allocated on NVMM. After this, we terminate the applications that filled up DRAM resulting in freeing of significant portion of DRAM memory. This triggers a migration of the benchmark application's data pages from NVMM to DRAM.

For this experiment, the system configurations remain the same as full-system run. However, we run with a smaller input size (see Figure 10). BHi+Mig reduces the walk cycles by up to 61.34% and stall cycles by up to 54.88%. This reduces

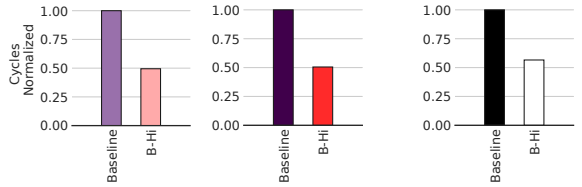


(a) Total cycles **(b)** Walk cycles **(c)** Stall cycles

Figure 10. Performance comparison of Radiant (Mig) in a multi-tenant environment with AutoNUMA (baseline).

Table 5. Number of data page and PTE migrations in multi-tenant environment.

		PTE migrations		
Workload	Data page migrations	Successful migration	Already in destination	Within DRAM
Memcached	66,644,738	50,601	39,272,431	26,763,450
Redis	33,315,590	69,731	27,461,927	5,783,941
BTree	11,820,636	17,061	7,791,351	4,012,020
HashJoin	1,945,151	50,209	1,867,027	27,915
XSBench	371,977	574	285,933	85,470
BFS	6,967,564	20,957	6,942,269	4,338



(a) Total cycles **(b)** Walk cycles **(c)** Stalls

Figure 11. Performance evaluation of BHi for Memcached in an interleaved setting with AutoNUMA disabled.

the total cycles by up to 50.75% and improves the run-time by up to 50.77% (see Figure 10). Table 5 shows the number of data page migrations triggered and the number of successful PTE migrations. We also show the reason for not migrating a PTE page (a PTE page is already in DRAM or in the destination NUMA node). As a PTE page points to 512 data pages, the first data page that is migrated to DRAM triggers a PTE page migration; for the rest 511 data page migrations, PTE migration is not required as it is already in DRAM.

6.4 Interleaved vs. Radiant

Interleaved memory allocation policy allocates the page table pages and the data pages on DRAM and NVMM in a round robin manner. Radiant still follows the interleave policy for data pages, but binds the high-level page table pages to DRAM (BHi). We compare the performance of BHi with the default kernel allocation (Figure 11). As AutoNUMA is disabled for this experiment, page table pages are not migrated and hence, we do not report BHi+Mig statistics. We can clearly observe that having a different placement and allocation policy for data and page table pages is beneficial.

Binding the high-level pages in DRAM reduces the walk cycles up to 49.48% and stall cycles by up to 43.42%. This reduces the total cycles by up to 50.51% and improves the run-time by up to 51.75%. It can be further observed from Figure 12 that page walk latency decreases by 23% when we bind the high-level page table pages in DRAM as the interleaved allocation policy spreads the high-level page table across the DRAM and NVMM nodes.

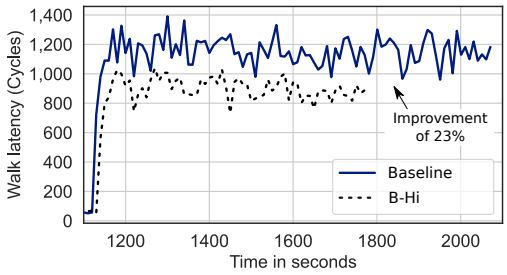
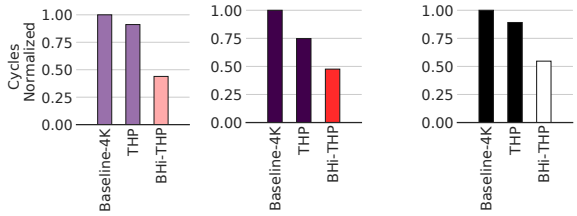


Figure 12. Improvement in the page walk latency with BHi for the interleaved policy (Memcached, RSS 1TB, 100% read). Baseline is interleaved memory allocation policy with AutoNUMA disabled.



(a) Total cycles **(b)** Walk cycles **(c)** Stalls

Figure 13. Performance of BHi with THP enabled. Baseline is AutoNUMA disabled with 4K pages.

6.5 Improving Application Start Up Time

During an application start up there are many data page faults that requires a page table walk. By placing the high-level of a page table pages in DRAM, we reduce the cycles spent on page table walks. While inserting 1 TB of data in Redis, we reduce the total page walk cycles by ≈ 9%. This results in a 21% reduction in total stalls cycles, that corresponds to an improvement of 22% in total start up time, when compared with default first-touch policy (see Figure 1 and Table 4).

6.6 Huge Page Impact

We evaluate the performance of Radiant when transparent huge page (THP) support is enabled.

Figure 13 shows that BHi improves performance when THP is enabled. BHi binds PGD, PUD, and PMD levels of the page table to DRAM. For a huge page as a PMD page is the last or leaf-level page (no PTE page), BHi is effectively binding the entire page table resulting in performance improvement. However, BHi+Mig does not improve performance as there are no PTE-level pages to migrate.

6.7 Discussions

In a modern out-of-order CPU, a page table walk performed by the Page Miss Handler (PMH) in the hardware can overlap with other work [3]. Hence, a reduction in page table walk

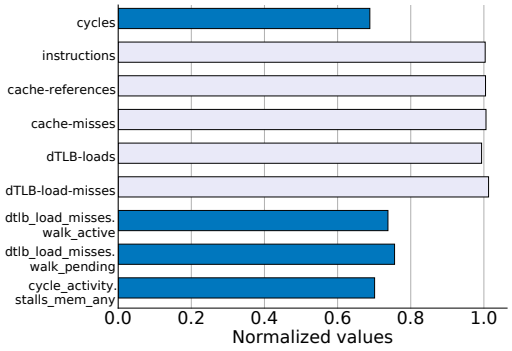


Figure 14. Performance statistics from the `perf` tool for BHi+Mig in full-system run for BFS (normalized to default first-touch policy baseline).

cycles need not always result in the reduction in total execution cycles. On the other hand, we see a reduction in total execution cycles even when there is no significant reduction in walk cycles. We use the hardware performance counters to reason and understand the impact of page walk cycles on total execution cycles.

Figure 14 shows the counters for BFS from the full-system run (§6.2). Here, the instructions executed, cache misses, and data TLB loads/load-misses remain the same, as expected. However, we can observe a significant reduction in `walk_active` and `walk_pending` cycles (i.e., cycles when PMH is performing a page walk). This contributes to the reduction in stall cycles `stalls_mem_any`, (execution stalls either due to an outstanding load/store or due to an address translation). It can be thus observed that reduction in total execution cycles is proportional to reduction in the stall cycles.

However, for few benchmarks, a reduction in the walk cycles does not result in a proportional reduction in the stall cycles. Because most of the stalls are due to an outstanding load/store and not due to address translation (Redis and BTree in Figure 10c). As a result we do not see significant improvement in total execution cycles.

7 Related Works

7.1 Mitosis

Mitosis [3] proposes to reduce the page table overheads in a multi-socket NUMA systems by transparently replicating the page table pages on all the NUMA nodes. Mitosis shows that accessing page table pages from a remote NUMA node increases the page-fault latency. The basic assumption is that all sockets are equipped with low-latency DRAM memory. However, in a tiered-memory system with high latency NVMMs, replicating page table pages has several disadvantages. First, replicating a page table and ensuring its consistency on NVMMs incurs high overheads. Second, accesses to a page table on local NVMM-backed NUMA nodes are costly due to 3× higher access latency. Hence, replication of page table may not be helpful for large memory footprint applications running on tiered memory systems.

Table 6. Comparison of Radiant with Mitosis [3]

	Radiant	Mitosis
Tiered Memory Support	Yes	No
Migration Support	Direct	Via replication
Migration b/w DRAM and NVMM	Yes	No
Migration Granularity	L4 pages only	Full page table
Page table DRAM binding	L1, L2, L3	None
Replication	No	Yes
Page table sync. overheads	No	Yes
Hot L4 page identification	Yes	No

Even though Mitosis supports migration of page table pages, it is achieved via replication, i.e., replicate the page table on the destination node and then lazily free the replica on the local node. Radiant binds critical parts of the page table in DRAM and dynamically migrates the L4 pages pages between DRAM and NVMM; thus avoiding a full page table migration (Table 6).

Finally, Radiant employs the novel data-page-migration triggered page table page migration to identify and migrate page table pages between DRAM and NVMM. Mitosis neither identifies nor migrates relevant page table pages.

7.2 Linux Kernel Community

Linux kernel patches [40] posted in the Linux Kernel Mailing List (LKML) propose to bind all the page table pages in DRAM to avoid accessing it from NVMM (this patch is not a part of the Linux kernel). However, such an approach results in pathological behaviours mentioned in §3.5. Radiant proposes to bind only 0.18% of the page table pages in DRAM (i.e., L1, L2 and L3 pages) and dynamically migrates L4 pages between DRAM and NVMM.

8 Conclusion

In this paper, we show that explicit and efficient management of page table on tiered memory systems with terabytes of memory is important. We study the performance impact of page table placement and argue that different placement and migration policies are required for data and page table pages. We demonstrate that binding a small but critical page table pages to DRAM and dynamically managing the rest of the page table pages by enabling migration results in significant performance improvement on systems with terabytes of NVMM memory.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Haikun Liu, for their insightful comments.

References

[1] Reto Achermann. 2020. mitosis-project/mitosis-workload-btree: The BTree workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-btree. (Accessed on 10/03/2020).

[2] Reto Achermann. 2020. mitosis-project/mitosis-workload-hashjoin: The HashJoin workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-hashjoin. (Accessed on 10/03/2020).

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

[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* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[5] 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* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[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* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[7] Abhishek Bhattacharjee. 2013. Large-Reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[8] Filipe Brandenburger. 2020. numactl/numactl: NUMA support for Linux. https://github.com/numactl/numactl. (Accessed on 10/03/2020).

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[10] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling [LWN.net]. https://lwn.net/Articles/488709/. (Accessed on 10/04/2020).

[11] Jonathan Corbet. 2013. Split PMD locks [LWN.net]. https://lwn.net/Articles/568076/. (Accessed on 09/30/2020).

[12] 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. *SIGPLAN Not.* 48, 4 (March 2013), 381–394. https://doi.org/10.1145/2499368.2451157

[13] DAXCTL. 2020. DAXCTL Man Pages - NDCTL User Guide. https://docs.pmem.io/ndctl-user-guide/daxctl-man-pages. (Accessed on 10/05/2020).

[14] Linux Kernel documentation. 2020. Split page table lock — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/vm/split_page_table_lock.html. (Accessed on 09/30/2020).

[15] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (Nov. 2015), 59–66. https://doi.org/10.1145/2814328

[16] G. Gill, Roshan Dathathri, Loc Hoang, R. Peri, and K. Pingali. 2020. Single machine graph analytics on massive datasets using Intel optane DC persistent memory. *Proceedings of the VLDB Endowment* 13 (2020), 1304 – 1318.

[17] Dave Hansen. 2019. Allow persistent memory to be used like normal RAM. https://patchwork.kernel.org/cover/10829019/.

[18] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 875–890. https://doi.org/10.1145/3373376.3378465

[19] Sungjoo Hong. 2010. Memory technology trend and future challenges. In *2010 International Electron Devices Meeting*. https://doi.org/10.1109/IEDM.2010.5703348

[20] Intel. 2020. Intel Optane DC Persistent Memory. https://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[21] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 521–534. https://doi.org/10.1145/3140659.3080245

[22] Kinam Kim. 2008. Future memory technology: challenges and opportunities. In *2008 International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*. 5–9. https://doi.org/10.1109/VTSA.2008.4530774

[23] Redis Labs. 2020. Redis. https://redis.io/. (Accessed on 10/03/2020).

[24] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[25] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 277–289.

[26] Linux MAN pages 2020. move_pages - Linux man page. https://linux.die.net/man/2/move_pages.

[27] L. Liu, S. Yang, L. Peng, and X. Li. 2019. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2223–2236.

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

[29] Memcached. 2020. memcached - a distributed memory object caching system. https://memcached.org/. (Accessed on 10/03/2020).

[30] N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan. 2018. Chapter 8 - Multivariate Lifetime Models. In *Reliability Modelling and Analysis in Discrete Time*, N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan (Eds.). Academic Press, Boston, 387 – 428. https://doi.org/10.1016/B978-0-12-801913-9.00008-7

[31] NDCTL. 2020. NDCTL Man Pages - NDCTL User Guide. https://docs.pmem.io/ndctl-user-guide/ndctl-man-pages. (Accessed on 10/05/2020).

[32] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th*

Annual International Symposium on Computer Architecture (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[33] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 288–303. https://doi.org/10.1145/3357526.3357541

[34] PMEM.IO. 2020. Volatile use of persistent memory as a hotplugged memory region. https://pmem.io/2020/01/20/memkind-dax-kmem.html.

[35] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-Bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3329785.3329917

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

[37] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[38] 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"

[39] Tao Wang. 2020. Baidu Feed Stream Services Restructures Its In-Memory Database with Intel Optane Technology. https://newsroom.intel.com/wp-content/uploads/sites/11/2019/08/baidu-feed-case-study.pdf.

[40] Fengguang Wu. 2018. x86/pgtable: allocate page table pages from DRAM. https://lkml.org/lkml/2018/12/26/145.

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

[42] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[43] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Antibes Juan-les-Pins, France) *(SIGMETRICS '16)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/2896377.2901456

[44] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 183–193. https://doi.org/10.1145/2688500.2688507