Dapper: An Adaptive Manager for Large-Capacity Persistent Memory

Dongliang Xue, Linpeng Huang*, *Member, IEEE,* Chao Li*, *Member, IEEE,* Chentao Wu, *Member, IEEE,*

Abstract—In-memory computing has inspired researchers to consider integrating large-capacity persistent memory (PM) into the main memory subsystem. However, several challenges still remain for providing an integration approach for DRAM-comparable PM on existing enterprise servers. Current commercial servers tend to feature multiple sockets with shared-memory NUMA organizations. Simply constructing a hybrid main memory architecture for these NUMA organizations requires considerable modifications of the system software. Another significant problem in these designs is the high latency of accessing PM on a remote socket, which results in performance degradation. To address these problems, we integrated PM as a memory-based model and as a storage-based model simultaneously on one commercial server, which offers a short-cut approach for enterprises to build commercial NUMA machines with large-capacity PM. In the memory-based model, rather than focusing on the persistence attribute, we propose an architecture that benefits managing the integrated PM and DRAM space in a unified manner and that facilitates bypassing vast modifications to the system software. We also present an adaptive mechanism that can automatically introduce a moderate amount of PM into the local socket to hinder access of a remote socket by the degree of memory pressure. In the storage-based model, under the condition of taking full advantage of the PM's persistence, we abstract a PM volume device and overcome the torn sector problem. To demonstrate the effectiveness of the proposed scheme, we design and implement ***Dapper***, an a***dap***tive ***per***sistent memory manager prototype. The experimental results show that, compared to typical memory management approaches, Dapper achieves performance improvements of 13.1% to 34.0% on average on Graph500 BFS_SSSP benchmarks and SPEC CPU2006 floating point workloads, respectively. Moreover, when deploying F2FS on our PM volume, we find that Dapper outperforms existing methods by 5.8% on ***tar*** and by 11.9% on ***untar***.

Index Terms—Persistent Memory, NUMA, Main Memory Management.

1 Introduction

The NUMA-based commercial server is a shared-memory architecture that pervasively uses either 2 or 4 processor sockets. The memory on these servers is physically distributed through the sockets. Such distributed memory is considered to be a uniform address space platform by programmers; meanwhile, these memories are divided into **local** and **remote** memories according to the distance between the processor and the memory channel. Remote memory access is generally more expensive than local memory access from a latency standpoint, and it can negatively impact application performance if memory is not allocated to local core(s). Therefore, it is important to schedule application running on local memory by optimizing the operating system (OS).

A feasible solution to alleviate remote memory access is to expand the capacity of local memory. Sufficiently expanding local memory capacity can decrease the frequency of remote memory access. The emergence of large-capacity persistent memory (PM) provides a promising opportunity to mitigate the performance degradation problem due to a limited amount of local memory.

However, the existing main memory subsystem is extremely inadequate for managing the persistence property of PM; even integrating persistence into the current main memory subsystem requires revising the entire software stack [1]. On the one hand, the booting and shutting down of the OS are completely different from the conventional procedure because PM has preserved a former execution state since the last operation. Consequently, selectively restarting partial OS components and opportunistically controlling the degree of shutdown are apparently available due to PM's persistence, but some unpredictable harmful consequences should be avoided, which requires extensive evaluations in practice. On the other hand, the system is secure in the absence of sufficient privacy and confidentiality designs on traditional OSs using volatile memory, whereas encryption keys and decrypted data in the durable cells of PM can easily be leaked [2] [3] [4], which results from current OSs being unable to provide sufficient garbage collection policies. More seriously, traditional OS mechanisms, such as page faults, swapping and the process address space, all face new design and implementation challenges.

Meanwhile, the integration of large-capacity PM requires a novel memory organization to support NUMA-based servers. AMF [5] proposes a hidden memory approach that dynamically tunes the memory provisions following the memory demand. However, this work constrains the first socket with a small amount of DRAM and configures the other sockets with a large-capacity PM. Thus, its memory organization can easily lead to remote memory access when the memory pressure reaches the limit. Another important issue for this approach is abandoning the exploitation of the persistence attribute of the PM.

To achieve both a lower frequency of remote memory access and fewer modifications to the OS, we propose the idea to ***adaptively and transparently integrate PM into a local NUMA node***. In this scenario, PM is managed by a DRAM-like approach in the kernel mode, and conventional management mechanisms,

*This work is supported by The National Key Research and Development Program of China under grant (**NO. 2018YFB1003302**), the National Natural Science Foundation of China (NO.61472241,NO.61628208) and the Natural Science Foundation of Shanghai (No.18ZR1418500).*
Dongliang Xue is with Shanghai Jiao Tong University, 200240, Shanghai, China. [E-mail: xuedongliang010@sjtu.edu.cn]
Linpeng Huang* *is the corresponding author, Shanghai Jiao Tong University*
Chao Li* *is the corresponding author, Shanghai Jiao Tong University*
An earlier version of this work has been published in the Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2018) [5]. The new manuscript presents a new architecture and software implementation, and new experimental verifications.

such as NUMA node, normal zone, and buddy system, are all maintained, which minimizes the modifications to the OS. To support the ***adaptive*** management of PM, we dynamically append an appropriate PM space into a local NUMA socket based on the memory pressure. ***Transparent*** management of PM assumes that it is not necessary for a programmer to manually activate an explicit programming interface [6] [7] [8]; thus, this compatible approach is quite popular for massive mature commercial software in practice. Finally, to efficiently utilize the persistence of PM, we design a compatible PM volume device that is similar to a traditional storage device on the premise of guaranteeing sector consistency.

To validate our idea, we present a novel memory organization architecture based on a 4-socket (node) NUMA server. In the first three sockets (nodes), local memory is composed of DRAM space and PM space from both physical and logical perspectives, and the last socket (node) is also physically configured with DRAM space and PM space. However, from the perspective of a logical organization, the last socket (node) only targets DRAM space as its local memory, and PM space is treated as a volume device to achieve proper utilization of PM's persistence attribute. By combining the new memory organization architecture with a state-of-the-art hidden memory mechanism [5], our design can greatly reduce the overhead of remote memory access and minimize the modifications to the OS.

This paper makes the following contributions:

- We propose a memory organization architecture to facilitate bypassing the numerous modifications to the OS resulting from integrating PM's persistence attribute into the main memory subsystem. This architecture 1) supports a hybrid memory (DRAM+PM) as local memory based on a NUMA machine offering a uniform address space, and it 2) is identical to a traditional two-layer framework (memory+storage) on each NUMA socket such that the traditional OS can work smoothly with minor modifications.
- We introduce a mechanism to adaptively manage large-capacity PM. This mechanism 1) keeps PM detectable but inaccessible on each local NUMA socket under the conditions of the OS boot stage and high local memory support time; 2) is a gradual procedure to avoid frequent remote memory access and guarantees an appropriate integration ratio for managing PM as a local memory space; 3) is compatible with the existing main memory subsystem and does not require extra programming interfaces; and 4) is a transparent procedure such that memory-intensive applications can automatically benefit from it.
- We employ two kernel modules and a kernel patch to implement a prototype called Dapper in the Linux 4.5.0 kernel. One kernel module and the kernel patch are responsible for adaptive large-capacity PM management, and the other kernel module is responsible for the volume device on the last NUMA node.
- We extensively evaluate our proposed flexible mechanism from multiple perspectives. Compared to typical memory management approaches, Dapper achieves performance improvements of 13.1% to 34.0% on average on the BFS_SSSP benchmarks in Graph500 and floating point workloads in SPEC CPU2006, respectively. Using our persistency-enabled PM volume, Dapper also outperforms existing methods by 5.8% on ***tar*** and by 11.9% on ***untar***.

The remainder of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 analyzes different memory architectures. Section 4 provides the design and implementation. We present the experimental details and performance in Sections 5 and 6, respectively; present a discussion in Section 7; cover related works in Section 8; and conclude in Section 9.

2 Background and Motivations

PM can be an excellent main memory device for the in-memory computing era, and some memory-intensive workloads and large-memory-footprint applications could benefit from PM. In this section, we mainly present the challenges of managing PM in current NUMA-based commercial servers.

2.1 Bottleneck of Remote Memory Access

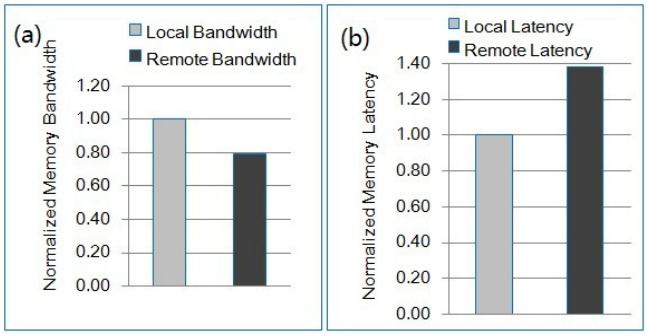


Fig. 1: Local vs remote memory bandwidth and latency

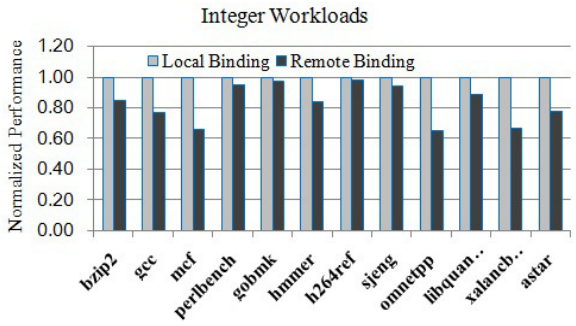


Fig. 2: Local vs remote memory binding on integer workloads

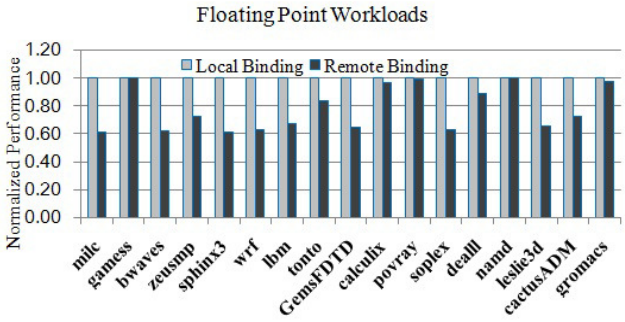


Fig. 3: Local vs remote memory binding on floating point workloads

NUMA systems have dominated the x86 server market for several years with their sustained growth. The NUMA architecture follows the concept that subsets of memory are divided into ***local*** and ***remote*** nodes for systems with multiple physical sockets.

Because modern processors are much faster than memory accesses, an increasing amount of time is spent waiting on memory to coordinate these processors. Since local memory is faster than remote memory in multisocket systems, ensuring that local memory is accessed rather than remote memory can be accomplished with a NUMA-aware implementation. The relative locality of a NUMA node depends on which core in a specific processor socket is accessing memory. Compared to access to remote NUMA nodes, operations on a local NUMA node typically yield better performance in terms of latency and bandwidth.

To illustrate the impact of remote access on actual workloads, a set of memory-intensive benchmarks can be chosen to describe binding impacts on performance. SPEC CPU2006 is a primary choice for this task because the benchmarks consist of multi-threaded integer and floating point workloads. We select memory bandwidth and latency as the main metrics to precisely measure the bottleneck of NUMA-based memory access features.

We employ a 2-socket Dell R820 server with 8 real cores per socket and hyperthreading disabled to distinguish between local and remote memory access. Fig. 1(a) shows a memory bandwidth comparison between local and remote NUMA nodes with the STREAM [https://www.cs.virginia.edu/stream/] benchmark. Fig. 1(b) shows the memory latency differences between local and remote NUMA nodes with the LMBENCH [http://www.bitmover.com/lmbench/] benchmark. Fig. 2 demonstrates that the impact of running multithreaded workloads with all remote memory access ranges from a negative impact of 3% with h264ref to a high impact of 34% with the mcf workload. From Fig. 3, except for two workloads experiencing no performance penalty when memory allocation is forced to the remote NUMA node, other workloads are considerably more sensitive and experience up to a 39% reduction in performance.

Based on the evaluated integer and floating point workloads, it can be concluded that a potential performance loss is inevitable when forcing memory to run remotely, and it can also be inferred that local memory allocation is always preferred if available. Therefore, at the moment of ***integrating an emerging PM into a NUMA-based machine, the negative impact resulting from remote access should be eliminated to the greatest extent possible***.

2.2 Some PMs are comparable to DRAM

With the approach of AEP (Intel Apache Pass, DIMM interface) commercialization, increasingly more PMs are now comparable to DRAM. Recent works [9] [10] [11] have found that some PMs (e.g., STT-RAM and RRAM) are even better than DRAM in terms of latency.

Parameter	DRAM*	STT-RAM	RRAM	PCM
Read latency [ns]	40-90	1.2	1.2	60
Write latency [ns]	40-100	11.2	21-100	150
Read energy [pJ]	800-1500	260	193	363
Write energy [pJ]	1000-1500	2337	592	63670
Leakage [mW]	64-534	387	115	153

TABLE 1: Comparison of the parameters of various memories. The values are extracted using a 512 KB capacity. Specifically, the values of the DRAM* column are from IRDS (International Roadmap for Devices and Systems) and [https://www.micron.com/support/tools-and-utilities/power-calc].

As shown in Table 1, for fast PMs (e.g., STT-RAM and RRAM), the order of access latency on a NUMA node abides by the following law: "local DRAM $<$ local PM $<$ remote DRAM $<$ remote PM". In this condition, giving access priority for local PM matches the order of access latency. For slow PMs (e.g., PCM and FeRAM), the access latency abides by the following law: "local DRAM $<$ remote DRAM $<$ local PM $<$ remote PM", and the gap between remote DRAM and local PM is far less than one order of magnitude. For this reason, still giving access priority for local PM apparently does not match the access latency of "remote DRAM $<$ local PM". However, simply obeying this mechanical order will introduce clear drawbacks: remote DRAM has limited capacity, which cannot continually provide sufficient space for severe workloads. Consequently, the whole system is required to switch providing pages between local DRAM and remote DRAM, which inevitably results in system falter/thrashing. More seriously, current OSes do not support providing memory in the order of "remote $<$ local"; thus, large modifications are required to support this function. In brief, ***under the condition that PMs are comparable to DRAM, the performance gap between local and remote access still introduces a degradation of a hybrid memory system***.

2.3 Complexity of Managing PM in OS

Existing works mainly develop a heap-based persistent object to manage PM in the memory subsystem. Mnemosyne [6] provides PM regions that are mapped at fixed virtual addresses to support programmers explicitly using a persistent programming interface. NV-Heaps [7] proposes persistent objects that are mapped to heap regions in virtual address space, and programmers are required to use newly designed primitives to build these persistent objects. HEAPO [8] redefines its own persistent heap layout and further optimizes the persistent object format, name space organization, object sharing, and protection mechanisms. Without loss of generality, the above works need to elaborately design an appropriate heap layout and programming interface for managing the persistence property of PM, which lead to significant kernel revisions and massive redefinitions of the programming interface.

Furthermore, current OS components, such as virtual memory mechanism, file system, programming initialization, application installation and security modules, all need modifications to support PM. For example, existing OSes mainly use the concept of virtual memory to map physical memory, which is restricted/limited to the case where the machine cannot provide sufficient physical memory. However, in an environment with TB-level PM, a single memory address space may be more appropriate. For compatibility, PM integration requires the kernel to use a similar **buddy** algorithm to allocate and reclaim physical memory space. Accordingly, the kernel is required to provide an extra reasonable garbage collection approach to avoid potential security threats because PM's persistence results in newly allocated/reclaimed space that includes unreleased persistent data.

Integrating PM's persistence into the main memory subsystem has a great impact on memory management of the OS and software compatibility. Vast modifications to the OS are likely to cause strong resistance from enterprises because their commodity software requires a stable OS version. Moreover, frequent revisions to commercial databases are also detrimental in practice. In conclusion, maybe ***minimizing changes to the existing OS is key to making PM pervasive***.

2.4 Expense for Managing Large-Capacity PM

Large-capacity PM consumes a considerable amount of energy. Memory energy consumption has a measurable impact of 20-

35% of the total system energy supply. For example, Fig. 4 demonstrates that 1) memory energy consumption can reach up to 35.8%, with an average value of 26.7%, of the total supply on a Dell R920 Server when running the SPEC CPU2006 benchmarks presented in Table 2, and 2) the energy consumed by memory is primarily dependent upon its capacity. Moreover, several works in the literature [12] [13] [14] also illustrate that memory energy consumption is responsible for approximately 25% of the total costs of the system. Worryingly, from Table 1, PMs are highly wasteful with energy during write operations. Therefore, ***a problem exists between increasing memory capacity provision and high energy consumption***.

Workload	Benchmarks	Footprint
S1	gobmk, hmmer, libquantum	110 MB
S2	h264ref, gromacs, namd	127 MB
S3	provary, calculix, tonto	82 MB
S4	zeusmp, cactusADM, xalanc	1548 MB
S5	bzip2, wrf, milc	2214 MB
S6	mcf, bwaves, gemsFDTD	3262 MB

TABLE 2: Sets of Workloads with SPEC

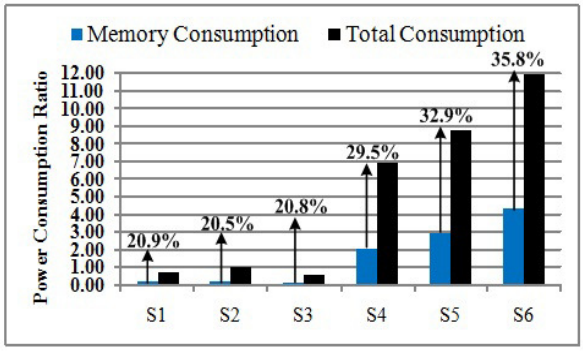


Fig. 4: Memory Energy Consumption for SPEC Benchmarks.

Large-capacity PM consumes an extremely large metadata space. Large-capacity PM uses a massive memory space to describe a page descriptor. Each physical page is associated with a data structure, which is used to keep track of every page. For example, the kernel uses it to identify the physical page that belongs to kernel code or kernel data, and it can also indicate whether the physical page is free or not free. Generally, there are more than 32 flags to describe the status of each page. In the Linux 4.5.0 kernel on an x86-64 architecture, one page descriptor needs 56/64 bytes to store its data structure. For emerging PM with its terabyte density, a 1 TB PM consumes 14/16 GB space to store all page descriptors under the condition of a 4 KB page size (1 TB/4 KB*56 B or 1 TB/4 KB*64 B) [https://www.snia.org/]. Therefore, ***integrating large-capacity PM space into the system can result in a rapid increase in kernel metadata***.

Considering that the phenomenon of "severe workloads occupy the whole memory all the time" is impossible in practice, while a shorter metadata design leads to revisions of the OS and popularization of the PM, a compromise between large PM provisioning and rapid metadata growth is likely an acceptable approach.

In short, from the perspectives of energy consumption and metadata management, ***immediately integrating large-capacity PM into a NUMA-based server is not an optimized option***.

2.5 Our goal

The main goal of this paper is to provide a convenient approach for an enterprise to introduce large-capacity PM into NUMA-based commodity servers. Confronted with the bottleneck of remote memory access, we target maximally decreasing remote memory access by expanding local memory with a moderate amount of PM space. Faced with the complexity of integrating PM into the main memory subsystem, we aim to abandon integrating the persistence property into the main memory subsystem and manage PM with a DRAM-like method, which skillfully bypasses the vast modifications to the memory management subsystem of the OS. However, persistence is an important attribute of the PM, and we infuse this attribute into the storage subsystem of the OS by converting part of the PM into a volume device. Considering the expense of managing large-capacity PM, our objective is to dynamically provide a moderate amount of memory to gradually match the memory demand.

3 Architecture Comparison

AMF [5] has summarized major PM integration architectures, and these architecture analyses inspire us to explore more reasonable and efficient designs. In this section, we primarily discuss the differences between AMF and Dapper from the aspects of hardware organization and system software framework.

3.1 Memory Hardware Organization Analyses

With respect to memory hardware organization, Dapper can be distinguished from AMF in the following three aspects.

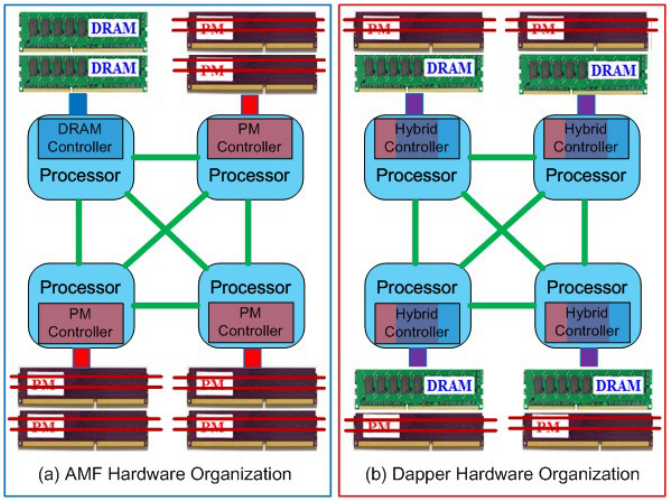


Fig. 5: Memory Hardware Organization Comparisons.

First, AMF has a single memory controller structure on each NUMA node (e.g., Fig. 5(a) shows a DRAM controller on node1 and shows a PM controller on node2, node3 and node4.), whereas Dapper has a hybrid memory controller on each NUMA node (e.g., Fig. 5(b) shows a hybrid memory controller on node1, node2, node3, and node4). Therefore, compared with a single memory controller such as AMF, it is more complicated for industry to develop a hybrid memory controller such as Dapper.

Second, following different design philosophies, the physical memory organization of Dapper can support three categories of memory address space, whereas AMF can only support one type of memory address space. Specifically, these three space categories

correspond to three logical architectures that are separately named DA1, DA2, and DA3. For DA1, DRAM is treated as a buffer layer of the PM, and the size of its physical address space is determined by the PM's capacity. For DA2, where DRAM and PM are on a uniform memory layer, the size of its physical address is determined by the total capacity of DRAM and PM. For DA3, DA1 is converted into DA2, or vice versa, and the size of its physical address is dynamically changed between the PM's capacity and the total capacity of DRAM and PM; thus, considerable difficulties/complexities remain for achieving DA3. Considering that some PMs are comparable to current DRAM, this paper mainly focuses on DA2.

Third, AMF's memory hardware layout is more likely to cause remote memory access than Dapper's. This is because under the conditions of a hidden PM mechanism, AMF has a limited amount of DRAM space on node1, which leads to node2's and node3's PM spaces becoming involved. Although Dapper's local memory is composed of DRAM space and PM space, PM has a high density, and a great amount of PM space can be configured as local memory to alleviate remote memory access.

Note that DA1 is seemingly a perfect logical architecture for slow PMs (e.g., PCM and FeRAM). However, it still needs to capture the approach for DRAM management. As a buffer layer, if DRAM is managed by the memory controller layer, then it is bound to increase the complexity of designing a new memory controller because it 1) requires coordinating a relationship with the cache controller and 2) requires resolving "DRAM-coherence" problems on NUMA-based machine like cache-coherence problems. Analogously, as a buffer layer, if DRAM is managed by the OS layer, it is bound to increase the complexity of designing a hybrid memory management subsystem for PM and DRAM because it 1) needs to ensure that the DRAM's management is transparent to the PM's management and 2) requires to completely alter the procedure of memory detection and initialization. Worse, as a buffer layer, the DRAM management method inevitably incurs the cost of memory persistence irrespective of whether it is hardware management or software management.

3.2 Different System Software for Matching PM

Similar to AMF, Dapper aims to design a lightweight and compatible system software to manage PM. However, Dapper's management method differs from AMF's method in several ways.

First, AMF hides the high address space of the PM occupied on node2, node3, and node4, whereas Dapper digs four holes on each NUMA node, and each hole's range is just equal to the PM's address space in the form of expanded local memory.

Second, both AMF and Dapper manage PM space with a DRAM-like approach on node1, node2 and node3, which is based on the assumption that some PMs (e.g., STT-RAM and ReRAM) are comparable to DRAM in read/write latency. Whereas some PMs, such as FeRAM and PCM exhibit 2/3 times higher read/write latency, under these different latency parameters, AMF's management approach is less sensitive than Dapper's approach because AMF requires bridging not only the latency difference but also the cost of remote memory access.

Third, regarding the management of PM on node4, Dapper devises a volume device by converting byte-addressable PM into a block-addressable storage device. Although AMF utilizes a continuous virtual address space called ***mmap region*** to map physical PM, this technique offers a compatible programming interface to conveniently assist programmers in exploiting large-capacity PM space at the user level.

4 Adaptive Manager for PM

We now present the design and implementation in combination with a new release of the Linux kernel. We first present an overview of the system architecture. We then describe the approach for integrating PM into each NUMA node, which enables existing workloads to transparently benefit from local memory expansion. Finally, to sufficiently use the persistence attribute of PM, we propose a technique that manages PM as a volume device.

4.1 System Overview

Fig. 6 shows overall architecture of Dapper, which includes a hardware hierarchy, kernel level and user level. Based on the deliberate layout of the PM hardware hierarchy, our work can smoothly manage the PM and DRAM space in a unified manner. Under the support of the hidden PM mechanism, Dapper enables the kernel to exactly run on a traditional NUMA-based machine. Therefore, our work only conducts lightweight revisions to the OS.

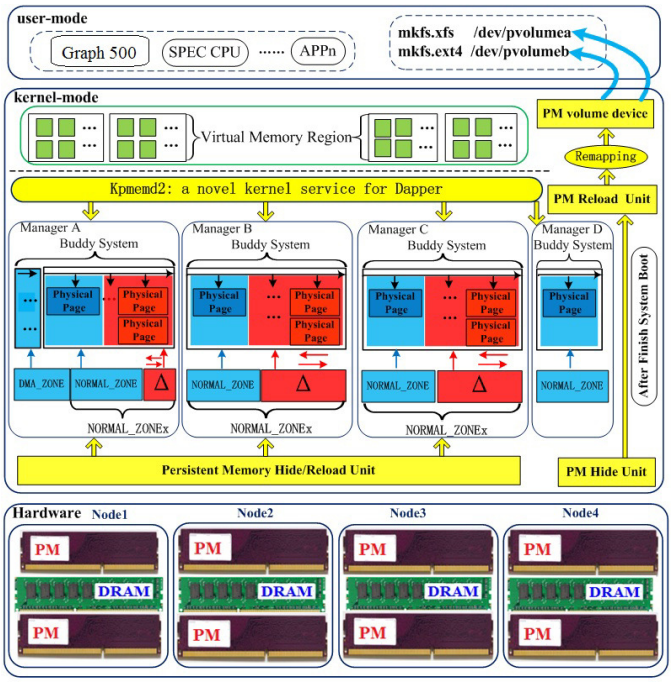


Fig. 6: Dapper Architecture Overview. The architecture is composed of hardware hierarchy, kernel level and user level. This design mainly consists of newly inserted yellow components and PM hardware layout.

Note that Dapper can support symmetric usage of memory resources. In fact, as shown in Fig. 6, we only devise a few components in the kernel by inserting two kernel modules. The first kernel module is responsible for building the memory-based model on node1, node2, and node3, and the second kernel module is responsible for building the storage-based model on node4. To cover more attributes of the PM, we simultaneously integrated PM as a memory-based model and as a storage-based model on one commercial server. However, to decrease potential resource imbalance or to popularize in industry, Dapper can purely support symmetric management for DRAM and PM (both as memory) resources on every node/socket as long as the first kernel module is

independently activated. Similarly, Dapper can also purely support another symmetric management for DRAM and PM (DRAM as memory and PM as volume) on every node/socket as long as the second kernel module is solely enabled.

Our key designs lie at the kernel level. Dapper includes three main components: a kernel service named Kpmemd2, a PM hide/reload unit and a volume management component.

Kpmemd2 is responsible for adaptively scaling local PM. When the consumption of DRAM space reaches a predefined limit, Kpmemd2 can automatically integrate a moderate amount of PM space into the local node to expand the total capacity of the local memory. Kpmemd2 connects with Manager A, Manager B, and Manager C, and it determines whether each manager controls a single DRAM space or manipulates hybrid memory space (DRAM+PM) in a unified memory management method called **buddy system**. Kpmemd2 also monitors the total memory footprint of the applications to facilitate these three managers manipulating the amount of PM space.

The PM hide/reload unit is responsible for hiding and reloading PM space on node1, node2, and node3. A NUMA-based machine is a shared-memory architecture, and from the perspective of physical memory management at the kernel level, memory space is managed at large granularity called a zone and small granularity called a page. In general, a memory zone consists of **ZONE_DMA** and **ZONE_NORMAL**. Dapper mainly tunes the size of ZONE_NORMAL to achieve expansion and shrinkage of the PM. Once the PM space has finished the integration or removal at the granularity of ZONE, the allocation and reclamation of the PM are similar to the conventional DRAM management method. Clearly, all these processes are transparent to the user layer.

To exploit the persistence property of PM, Dapper proposes a technique that manages PM as a volume device. This characteristic allows the kernel to treat PM on node4 as a single logical entity, which describes another behavior for OS components (such as file systems, but not the main memory management part) accessing PM. In other words, Dapper anticipates providing a software abstraction for PM hardware in a fully compatible way, and this way only requires employing a mature synchronization mechanism to handle the torn sector problem [https://pmem.io/] resulting from persistence. As shown in Fig. 6, at the OS boot stage, PM space on node4 is hidden. After finishing this stage, PM space is reloaded in the kernel. We then bind the corresponding PM space to a predefined volume device. Finally, the volume is partitioned into traditional block devices, and the existing file systems, such as ext4 and xfs, can mount on these two block devices without any revisions.

4.2 Deliberate PM Hiding

Hiding PM space can alleviate metadata explosion and diminish unnecessary energy consumption. More importantly, however, it enables the kernel to boot from a DRAM-only architecture as from a traditional NUMA machine, which eliminates PM's persistence property intervention and decreases the OS revisions.

Dapper can handle degrees of memory initialization. Existing OSes initialize all main memory spaces on current NUMA machines at the boot stage, while the concept of hiding PM compels the OS to only initialize part of the memory space. Dapper overcomes this obstacle by following elaborate procedures. The major procedure consists of four phases, as shown in Fig. 7, which shows the basic method to perform the PM space hiding on each NUMA node.

1) ***Changing physical memory address ranges***. Following the early startup of the OS, the information about physical memory address ranges is reported through a BIOS interruption called E820. Specifically, E820 indicates which memory address ranges are usable and which are reserved. These reports include both DRAM's range messages and PM's range messages. Before these messages completely transfer to the next initialization stage, Dapper redefines the memory address ranges (see the first part surrounded by the dashed line in Fig. 7) and transfers it to a predefined area. In this phase, we detect both DRAM and PM address ranges but only allow DRAM space to attend the next initialization stage.
2) ***Revising each node's page frame number***. After reporting the information about memory ranges, the system will parse SRAT (static resource affinity table, which stores topology information and physical locations for all the processors and memory) to check the memory range info. Thus, Dapper alters the result of SRAT and only exposes the DRAM ranges on each NUMA node (see the second part surrounded by the dashed line in Fig. 7). Simultaneously, the last page frame number on each NUMA node should correspond to DRAM's last page frame number on each NUMA node.
3) ***Building zone and sparse memory model***. Based on the changed memory space limitation, we need to adjust the size of **ZONE_NORMAL**. In other words, we build the zone management method only for DRAM space on each NUMA node. Along with this modification, when moving to the initialization of the sparse memory model, Dapper merely divides the DRAM ranges into multiple sections in which page descriptors reside. This action means that the extremely large page descriptors of the PM are not initialized; thus, Dapper does not require occupying a large space to store metadata at this stage.
4) ***Enabling buddy system mechanism***. For the same reason, we activate the buddy system only for DRAM space ranges. This memory management mechanism does not require making extra modifications except for specifying its limitation.

In conclusion, from the above minor alterations, we merely initialize DRAM space (e.g., in Fig. 7, the P3 and P4 phases are covered with blue, which denotes that operations only refer to DRAM space) and leave the PM space detectable but unavailable/inaccessible on each NUMA node.

4.3 Automatic PM Provisioning

After finishing the system boot, Dapper immediately monitors the consumption of memory space. In the event that the DRAM space on the local node cannot satisfy the current memory-intensive applications, Dapper will automatically activate a PM integration procedure to expand the capacity of local memory. Fig. 8 presents the major phases to support the automatic PM provisioning.

In this section, we first target determining the topology, range and physical location information of the hidden PM. Then, we reload PM into a local node and make it accessible to efficiently alleviate memory deficiencies.

4.3.1 Rediscovering PM Space

The technique of rediscovering PM is crucial to this research, which determines whether the PM can be integrated into the local nodes. There are two methods to obtain the topology, range and physical location information of the hidden PM.

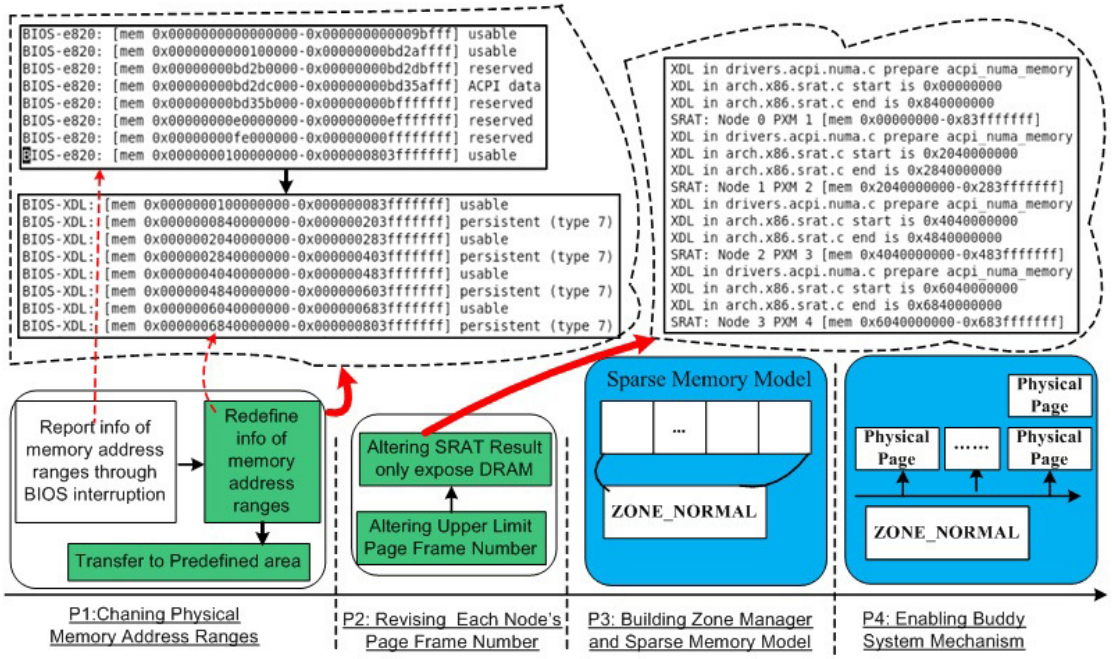


Fig. 7: Procedure of Hiding PM.

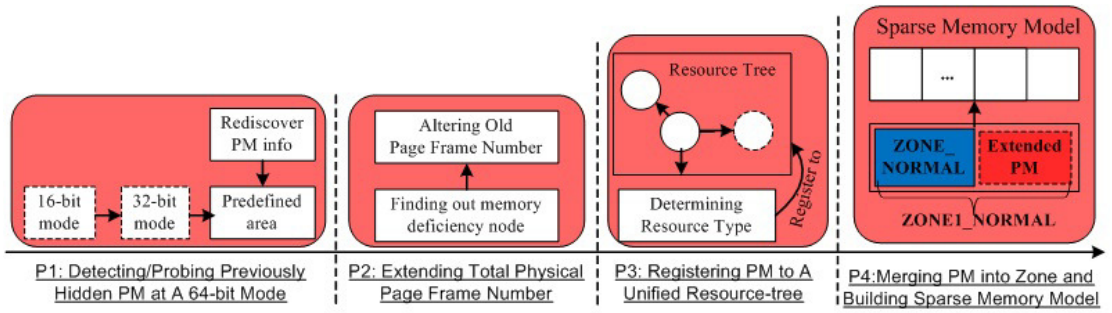


Fig. 8: Procedure of Automatic PM Provisioning.

One method is to rebuild a detection procedure by triggering a BIOS interruption. However, a BIOS interruption is effective only in a real address mode (a 16-bit mode in Fig. 8), but the runtime environment has already been in a 64-bit mode since finishing the startup process. To overcome this conflict, it is theoretically feasible that the system can temporarily return to the real address mode. However, this context switch not only introduces serious system thrashing and unpredictable consequences to the running program but also leads to programming difficulties in practice.

The other method is to transfer the detected information from a real address mode to a 64-bit mode. Dapper employs this method to copy the detected information from a data structure called **boot-parameter-page** to the predefined area, and the detected information contained in the boot-parameter-page just originates from a BIOS interruption in a real address mode. Dapper takes full advantage of the transferring approach, which guarantees that the detected information can be sequentially delivered from a real address mode, then to a protect mode, and finally to a 64-bit mode. Eventually, Dapper acquires topology, range and physical location information of the hidden PM in a 64-bit mode, which is necessary to prepare reloading of the PM.

4.3.2 Reloading PM space

After obtaining the report of topology, range and physical location information of the hidden PM on each NUMA node, the most important task is to convert the detectable PM into accessible PM according to the deficiency of local memory on each NUMA node. To achieve this aim, Dapper requires rebuilding a memory space initialization procedure. To provide a uniform management approach for the detected PM and DRAM, this initialization procedure is similar to DRAM's initialization, except for the following modification (see Fig. 8).

1) ***P2: Selecting a node to extend the total physical page frame number***. All information about the hidden PM has already been exposed to our predefined area; clearly, it is not necessary to make all the detected PM available. We use a round robin approach to check the amount of remaining/free DRAM space on node1, node2 and node3. If the amount is less than a predefined value based on the **memory watermarks**, then we select the corresponding node to integrate moderate PM space (detailed policy in section 4.4). Afterward, we alter the total physical page frame number on the corresponding node based on the size of the integrated space.
2) ***P3: Registering PM to a unified resource tree***. The existing Linux kernel exploits a resource-tree mechanism to remember

the resource that did not register at boot time. To fully utilize the convenience of this mechanism, we only need to determine the root of the resource descriptor. Subsequently, following the basic tree traversal algorithm, the PM can be registered to the suitable position.

3) ***P4: Merging PM into zone manager and building sparse memory model***. After finish registering, the zone range is correspondingly expanded from **ZONE_NORMAL** to **ZONE1_NORMAL** (see Fig. 8). In other words, we build a zone management approach for DRAM space and PM space in a unified manner. Based on this revision, at the stage of the sparse memory model, we store the metadata (page descriptors) in the DRAM space for decreasing write times on PM because this type of metadata is frequently changed in the kernel, which is not suitable for wear-sensitive PM devices.

4.4 Adaptive PM Space Management

Achieving a QoS-aware main memory subsystem is rather difficult for multiple reasons. Considering that the capacity is the major contributor in this work, we propose a capacity-aware PM space allocation/reclamation policy based on **memory watermarks**. Because memory watermarks are fixed once the OS finish the boot on a definite NUMA machine, the comparison between remaining memory pages and memory watermarks can determine the amount of allocation/reclamation. ***Note that*** *different commercial servers have different memory capacity configurations, and thus, the policy parameters in tables 3 and 4 are perhaps not suitable for all capacity configurations. However, the core idea behind these two tables is significant, and other servers can conveniently determine their own policy parameters according to our examples.*

4.4.1 Moderate PM Space Expansion

A set of memory allocation policies exist in recent NUMA-aware Linux distributions. The bind policy requires that all memory allocation must come from the specified NUMA nodes; thus, there are hard limits to its utilization. The preferred policy requires that memory allocation first be directed to the specified single node supplied as an argument. If this node is full and memory allocation fails, then the memory will be allocated to adjacent nodes based on node distance. The interleave policy requires that each page allocation is interleaved between a set of node numbers specified as an argument. This mode is useful in different scenarios where there are not enough available logical processors local to a node or there is not enough memory in a single NUMA node for a large workload to obtain sufficient resources. However, these memory allocation policies are primarily utilized by specifying parameters to enable NUMA-awareness.

The local policy is the default policy for the whole system, which dictates memory allocation for tasks that occur on the NUMA node closest to the logical processor where the task is executing. This policy is a reasonable balance for many general computing environments because it provides good overall support for NUMA memory topologies without the need for manual user intervention. However, if the memory allocation fails in the local node, it also triggers a remote access. Therefore, determining the triggering point of remote memory allocation is vital for designing a moderate PM space expansion approach under the support of this local policy.

The existing kernel implements two linked lists to support remote memory access. The first linked list (Link0) is responsible for organizing all ZONE regions contained in all the nodes, and these ZONE regions are sorted according to the node distances between each other. The second linked list (Link1) only includes ZONE regions constrained in the local node. Generally, if Link1 is not sufficient for a new allocation, Link0 will be activated. Thus, we insert the PM integration before the activation of Link0, and the remote memory will be interrupted. Note that if the PM integration still cannot decrease the memory pressure, Link0 will be activated again to illustrate that our design never disables the compatibility.

Remaining free pages [R]	Amount of integration
R ∈ (page_high × 2048, Total memory × 1]	DRAM's capacity × 0
R ∈ (page_high × 1024, page_high × 2048]	DRAM's capacity × 1
R ∈ (page_low × 1024, page_high × 1024]	DRAM's capacity × 2
R ∈ (page_min × 1024, page_low × 1024]	DRAM's capacity × 4
R ∈ (page_high × 1, page_min × 1024]	DRAM's capacity × 8
R ∈ (page_low × 1, page_high × 1]	DRAM's capacity × 16

TABLE 3: PM space integration policy

After determining the trigger point of the remote memory access on each NUMA node (except for node4), we launch a processing (as a partial function of kpmemd2) to monitor the remaining free pages on Link1. We exploit modified memory watermarks to enable the monitor function. Traditional watermarks are highly correlated to the SWAP operation due to its low granularity. We devise a capacity-aware policy in a GB granularity for each node (except for node4), as shown in Table 3.

4.4.2 Lazy PM Space Reclamation

Removing PM space from the eye of the OS's memory management (available/unavailable PM in OS) is beneficial to a QoS-aware main memory subsystem. However, if the PM reclamation is aggressive, it can lead to page thrashing. If the PM reclamation is too conservative, it can lead to meaningless energy consumption originating from unnecessary costs for the maintenance of vast amounts of metadata. Considering that page thrashing can result in overall instability of the system, our reclamation policy tends toward being slightly conservative.

Total free memory [F]	Amount of removal
F ∈ (PM's capacity × 1, Total memory × 1)	PM's capacity × 1
F ∈ (PM's capacity × 3/4, PM's capacity × 1]	PM's capacity × 1/2
F ∈ (PM's capacity × 1/2, PM's capacity × 3/4]	PM's capacity × 1/4
F ∈ (PM's capacity × 1/4, PM's capacity × 1/2]	PM's capacity × 1/8
F ∈ (PM's capacity × 1/8, PM's capacity × 1/4]	PM's capacity × 1/16
F ∈ (PM's capacity × 1/16, PM's capacity × 1/8]	PM's capacity × 0

TABLE 4: PM space reclamation policy

Concretely, our reclamation policy is shown in Table 4. Assume that a server is configured with 1 TB PM and 64 GB DRAM on each NUMA node. If the capacity of the total free memory is greater than the PM's own capacity at a given time, the entire 1 TB PM will be removed from the OS's management; similarly, if the capacity of total free memory on node1 (or node2 or node3) remains at 65 GB, it is not necessary to remove any PM space because the DRAM's capacity is 64 GB.

The core technique for removing PM is to shrink the size of the ZONE_NORMALx on each NUMA node. Kpmemd2 periodically scans the amount of reclaimed space from the free list of the buddy system, and once the condition in Table 4 is met, quantities of contiguous pages will be removed from the control of the buddy system. Correspondingly, the range of the ZONEx_NORMAL (e.g., ZONE1_NORMAL in Fig. 8) is also shrunken. Finally, Dapper resets the space that the PM's page descriptors occupied in DRAM.

4.5 PM Volume Space Abstraction

This section designs a pseudo-volume device to enable traditional file systems to build their functionality by directly accessing PM. To achieve this objective, Dapper provides a volume as the software abstraction for PM hardware on node4. We mainly discuss two points: one is the technique for building a volume space, and the other is to employ the persistence attribute by overcoming the torn sector problem. Fig. 9 present the principle of the volume space software abstraction.

4.5.1 Building a Volume

The PM volume builds memory mapping capability via a memory channel with the help of the CPU's load and store operations; thus, we should provide an approach for the OS context switch for access to the PM volume.

At the kernel mode level, Dapper devises the following three steps:

- Step 1 holds back initializing PM space on node4. The basic procedure is similar to the PM hiding approach; the only additional requirement is to return a pair (start and offset) of physical address ranges, which is used to identify a volume.
- Step 2 allocates a contiguous, logical address range for mapping the physical address range. After finishing booting, Dapper first allocates a contiguous kernel address space related to a single base, and then page tables are created for connecting the kernel address space and physical address range.
- Step 3 constructs a series of volume-related data structures to format the kernel address space. As shown in Fig. 9, Dapper abstracts the volume device as a software data structure, which contains three domains called **Cendisk**, **Request Queue** and **Pdata**. Dapper also compatibly builds other data structures, such as request vector, major device number, and partitions to manage the volume device. Particularly, we use **make request mode** to directly control the request object to avoid enabling the whole block subsystem. More importantly, we define the device object operations in a **Devices-Drivers Model**, which will create two new device files (e.g., /dev/pvolumea and /dev/pvolumeb in Fig. 6) at the user level.

At the user level, Dapper deploys the file systems to PM volumes identified with device files. The device files are a software abstraction for PM hardware and profile functionality for OS components. Therefore, based on these device files, we can smoothly mount existing file systems, such as ext4 and xfs.

4.5.2 Atomic Sector Update

A power exception or a system crash will cause a partial write to a sector, resulting in a mixture of old and new data; this problem is called **torn sectors**. Traditional HDDs typically provide protection against torn sectors in hardware, using stored energy in capacitors to complete in-flight block writes, and SSDs resolve this problem in FTL or at the firmware level. However, a PM-based volume is capable of performing IO at a cache line granularity but fails to exactly provide any atomicity guarantees at a sector granularity. In addition, existing applications may not be prepared to handle such a scenario.

Facing this challenge, Intel proposes block translation table (BTT) [http://pmem.io/], which can provide atomic sector update semantics for PM devices such that applications that depend on sector writes are not torn again. Borrowing the idea from BTT, we also devise an approach to synchronize volume content to a state of consistency and durability.

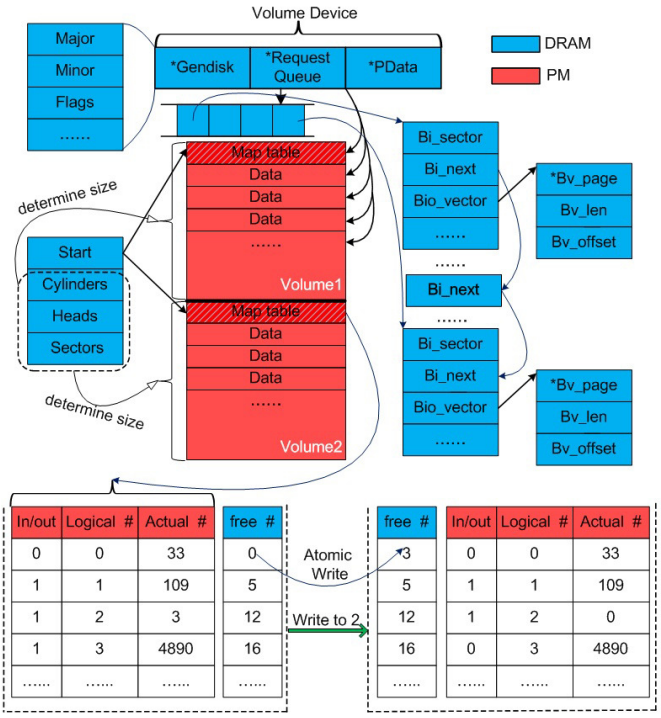


Fig. 9: Atomic Sector Update. 0-free, 1-occupied. **Gendisk** includes some traditional volume semantics. **Request Queue** includes PM sectors that are currently being used, which describes the relationship among these PM sectors. **Pdata** serves as a pointer to denote the real address of data stored in PM.

In contrast to BTT, Dapper rebuilds an indirection table and an extra index vector of the free sector. As shown in Fig. 9, Dapper builds a mapping table to store the relationship between the logical sector index and actual sector index. To facilitate searching the free sector, an in/out flag is also configured in this table. Dapper ***specially*** builds another vector to accelerate the retrieval of the free sectors. In Fig. 9, the size of the map table's element is less than 64 bits; thus, an update of each data structure is constrained in an atomic domain with the help of residual capacitance when facing a crash or power outage. When a logical write request to a new sector is interrupted, its actual sector is not simultaneously updated except for the free sector; therefore, an actual sector is equivalent to an undo log. Once the write to the free sector is finished, the update of metadata in the map table can be immediately finished in an atomic way. At the heart of this design is an indirection table that re-maps all the blocks on the volume. It can be considered as an extremely simple function that only provides atomic sector updates.

5 Experimental Setup

Dapper is deployed on a quadruple-socket Intel Xeon-based server with a new version of CentOS. This platform has a large memory capacity organized in a NUMA architecture and can easily emulate the large capacity of the PM. Table 5 describes the specifications of the platform.

Given that PM technology is still in an active development stage, in this paper, we emulate PM with DRAM. This method has been adopted by many recent studies [15] [16] on PM-related

System component	Specification
Platform	Dell R920 NUMA-based Server
CPU	4 × 2.0 GHz Intel Xeon E7-4820 Processors, 8 Cores per Processor
Last Level Cache	16 MB
Main Memory	512 GB 1066 Mhz
Operating System	CentOS 6.6
Kernel Version	Linux kernel 4.5.0
File System	EXT4

TABLE 5: Specifications of the platform

system designs. The performance of PM is comparable to that of DRAM. In this paper, we mainly discuss exploiting PM's capacity advantage to decrease the remote memory access penalty. Therefore, the evaluation results presented below do not consider the latency difference between PM and DRAM.

Our server has 4 NUMA nodes with a total of 512 GB of memory, and each node is configured with 128 GB of memory. On each node, the first 32 GB of memory is regarded as DRAM, and the remaining 96 GB of memory is treated as PM. The DRAM area is managed by the original kernel and is available to applications at any time. PM regions are managed by the original kernel together with Dapper. Note that PMs on node1, node2 and node3 are accessed in a memory-only mode, and PM on node4 is accessed in a volume-only mode.

To demonstrate that Dapper can effectively diminish remote memory access, we select both integer and floating point workloads from the SPEC CPU2006 suite [http://www.spec.org]. We use htop [https://hisham.hm/htop/], which is an interactive process viewer for Unix, to monitor the memory footprints of these benchmarks. We repeat the experiments three times and average the results to decrease the measurement randomness. We bind the STREAM [https://www.cs.virginia.edu/stream/] benchmark on each NUMA node and find that the bandwidth difference among them is within 2%.

We also employ Graph500 [http://graph500.org/] to analyze Dapper's memory, providing a way to process data-intensive workloads. Graph500 is an important form of analysis of large-scale graphs on shared memory NUMA architectures. Graph500 benchmarks involve three kernel functions: the first function constructs a graph data generator, the second function executes a breadth-first search, and the third performs multiple single-source shortest path computations.

Generally, the **scale** parameter determines the memory consumption of Graph500. For example, scale 26 means 2^{26} * (2*16+1) * 8 bytes $\approx$ 17 GB when the **edgefactor** (i.e., half the average degree of a vertex in the graph) is calculated with 16 under the condition of a vertex consumed 8 bytes using CSR (compressed sparse row). In the Graph500 benchmark, scale 26 is the ***toy*** class defined in their input size, and ***toy***, ***mini***, ***small***, ***medium***, ***large*** and ***huge*** classes exist, denoting different scale input sizes. However, the experiment finds that even the ***toy*** class requires a large amount of time to finish a complete evaluation in our platform. For an acceptable degree of time consumption, we configured the **scale** parameter as 22, and we defined it as the ***tiny*** class in our experiment.

We devise four group evaluations that are configured with different memory allocation capacities. Table 6 shows the comparison between Dapper and the baseline. Dapper ***dynamically*** provides a different amount of local PM (e.g., labeled with LP in table 6) and fixed local DRAM (e.g., labeled with LD in table 6). The baseline ***statically*** provides a different amount of remote PM (e.g., labeled with RP in table 6) and fixed local DRAM.

index	# of mcf	# of tiny	CPU: 0, 1 Memory: LD + LP **(Dapper)**	CPU: 0, 1 Memory: LD + RP **[Baseline]**
ex.1	62	25	32G LD + 32G LP	32G LD + 32G RP
ex.2	78	33	32G LD + 48G LP	32G LD + 48G RP
ex.3	94	36	32G LD + 64G LP	32G LD + 64G RP
ex.4	126	50	32G LD + 96G LP	32G LD + 96G RP

TABLE 6: Baseline configuration

In the evaluated setup, Dapper activated local DRAM space, and with increasingly more memory demand, it dynamically integrated a moderate amount of local PM space into the system. Note that in the evaluated setup, the baseline only activated the local DRAM space, and with increasingly more memory demand, remote PM space is allocated to serve these workloads by the native main memory subsystem. To perform a fair and reasonable comparison, consider that the experimental platform has 4 NUMA nodes, and we limit that remote PM space should be from node 1, which is to eliminate the latency differences resulting from node distances.

To demonstrate the effectiveness of the abstracted PM volume devices, we select a representative and optimized implementation named PMBD [17] as our comparison target. PMBD implements a RAMDISK-like management approach by modifying the virtual memory management of the OS. More importantly, PMBD is not subject to interferences from the memory-based model of the OS and easily forms a fair comparison.

We select five file systems to evaluate the impact of running a PM volume and PMBD. Among these five systems, NOVA [18], Ext4-DAX and PMFS [19] are in-memory file systems; F2FS [20] is a log-structured file system designed for SSD-based devices; and Btrfs [21] is a highly scalable file system that uses copy-on-write, checksum and snapshot techniques to guarantee the data integrity. In addition, we also compare to ext4 in data journal (ext4-journal) mode that provides data atomicity.

6 Evaluation Results

We evaluate the advantage of employing Dapper to deploy different benchmarks in this section. To demonstrate that Dapper can eliminate or decrease the remote memory access, from the SPEC CPU2006 suite, we select ***mcf***, the memory footprint of which is convenient to activate the function of Dapper. The experiment finds that the memory footprint of mcf has reached 1682 MB, which is the upper limit in the whole SPEC CPU2006 benchmarks. Considering that the DRAM capacity of each node reaches 32 GB, it is necessary to execute multiple instances of this benchmark to cause large quantities of memory demand to enable both local and remote memory accesses.

6.1 Performance Improvement on SPEC CPU2006

In this section, we mainly use SPEC CPU2006 benchmarks to examine the differences between Dapper and the baseline.

6.1.1 Single workload with multiple instances

Fig. 10 presents the normalized performance of using multiple mcf instances. The results show that Dapper is indeed able to decrease the remote memory access. Interestingly, when increasing the amount of instances, Dapper's performance gradually increased from 16% to 33%. This result is because the baseline has

more remote memory access: the more instances there are, the more frequent is the remote memory access, and the more the performance degrades. When facing a considerable memory deficit, compared with the baseline, Dapper immediately provides sufficient local memory, which decreases the number of remote memory accesses to the greatest extent possible.

6.1.2 Multiple workloads with multiple instances

To study the impact of multiple workloads on our work, we separately divided the whole SPEC CPU2006 benchmarks into integer and float groups. In the integer group, we mixed 12 different categories of benchmarks according to the memory configuration in Table 6. Thus, under the condition of four different memory provisions, each benchmark is executed at 15, 19, 23 and 31 instances, respectively. Fig. 11 shows the results of the performance on integer workloads. We find that Dapper can also improve the performance of the multiple workloads in comparison to the baseline. Specifically, Dapper's performance improved from an average of 20.0% to 31.0% when the total instances increased from 180 to 372, respectively. This result also demonstrates that higher memory demand is more likely to benefit from Dapper's local memory provisioning.

In Fig. 12, we examine the performance improvement on float workloads. Similarly, Dapper's performance improved from an average of 26.0% to 34.0% when the total instances increased from 180 to 375, respectively, under the condition that each float benchmark is executed with 12, 16, 19 and 25 instances. This result demonstrates that float workloads can also achieve some improvements with the help of Dapper. However, a few workloads such as ***calculix***, ***povray*** and ***namd*** are not sensitive to the memory provisioning approach.

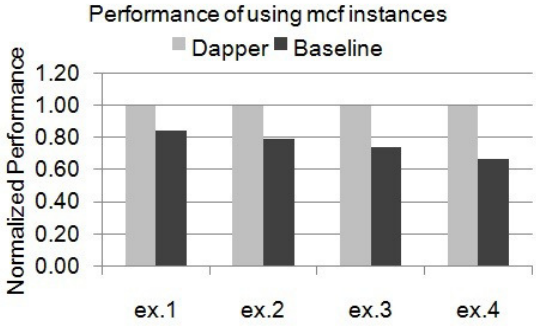


Fig. 10: Dapper vs Baseline: Normalized Performance on Mcf Workload.

6.2 Performance Improvement on Graph500

In this section, we mainly use the TEPS (traversed edges per second) metric to examine the differences between Dapper and the baseline. In addition, we use ***graph generation time***, ***graph construction time*** and ***graph validation time*** [http://www.graph500.org] as auxiliary metrics. We measured both ***BFS*** (breadth first search of the graph) and ***SSSP*** (single-source shortest path computation on the graph) problems.

Fig. 13(a) and Fig. 13(b) present the TEPS performance for BFS and SSSP, respectively. Compared with the baseline, we find that Dapper achieves average 5.0% and 13.1% performance improvements in the BFS and SSSP benchmarks, respectively. This result means that the SSSP benchmark is more sensitive to local memory provisioning than the BFS benchmark. The reason is that SSSP benchmark needs to continually handle the vertices originally

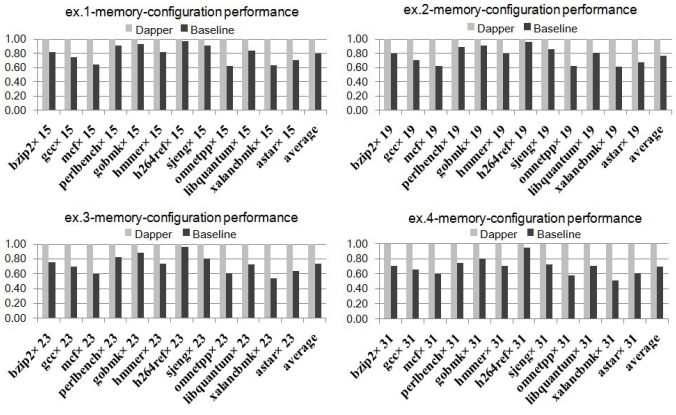


Fig. 11: Dapper vs Baseline: Normalized Performance on Integer Workloads.

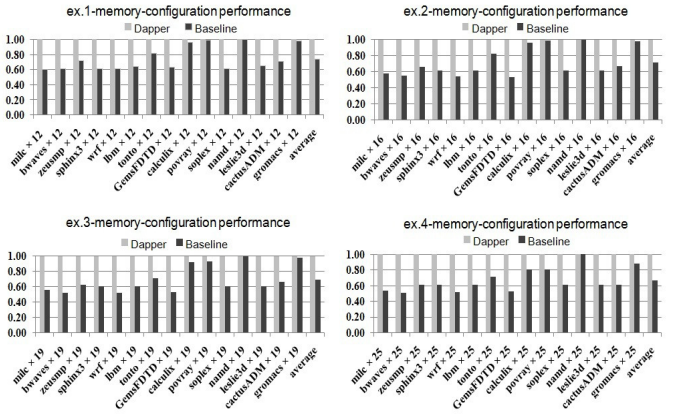


Fig. 12: Dapper vs Baseline: Normalized Performance on Float Workloads.

stored in the remote memory, Dapper just stores these kinds of vertices in local memory. Another reason, however, is that BFS benchmark always processes the relationship among the adjacent vertices, while the adjacent vertices are originally stored in one NUMA node, accordingly, BFS can get less benefit from Dapper's memory provisioning approach compared with SSSP.

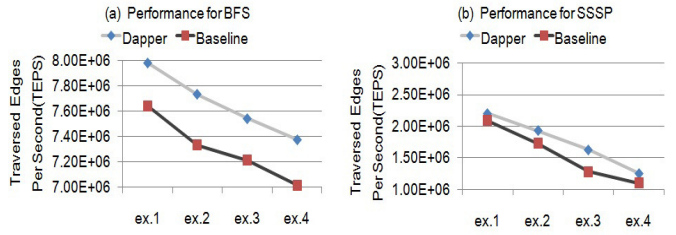


Fig. 13: Dapper vs Baseline: Graph500 TEPS Metric.

Interestingly, the total TEPS in BFS is considerably more than that of SSSP, which clearly illustrates that SSSP requires more computational and memory resources. Furthermore, in both BFS and SSSP, TEPS decreases along with the number of ***tiny*** instances. This result occurs because plenty of ***tiny*** instances lead to a great amount of competition for memory and computational resources. However, both BFS and SSSP can benefit from Dapper's local memory provisioning approach.

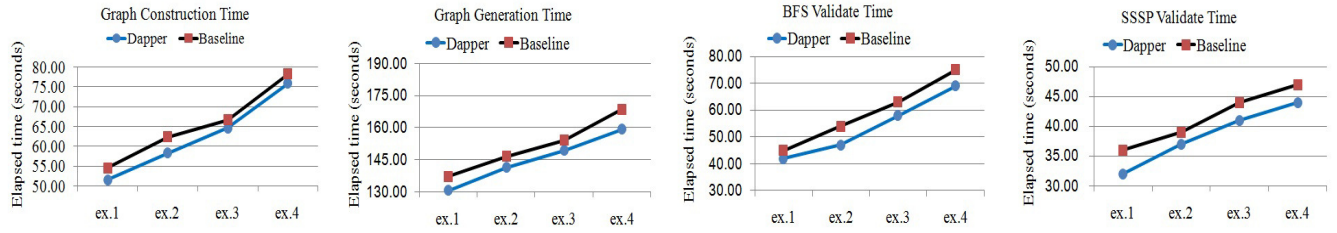


Fig. 14: Dapper vs Baseline: Graph500 different time metrics.

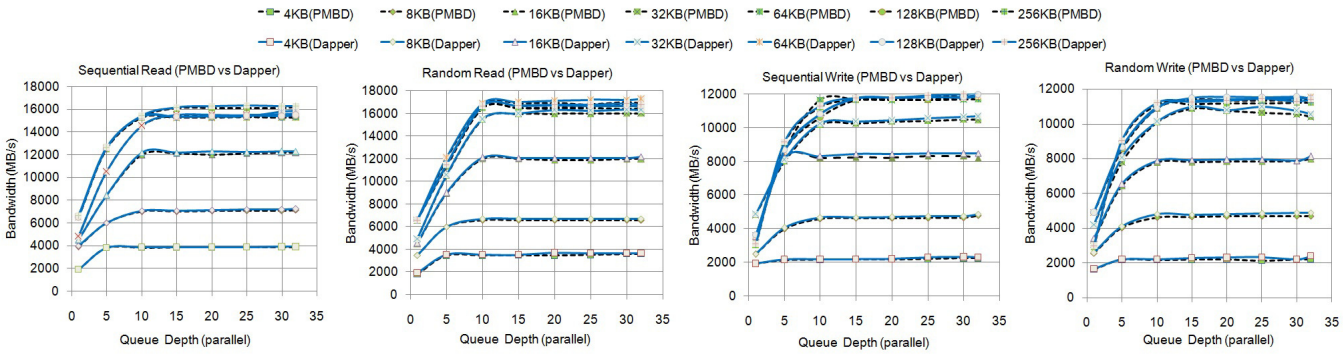


Fig. 15: Dapper vs PMBD: with different input parameters.

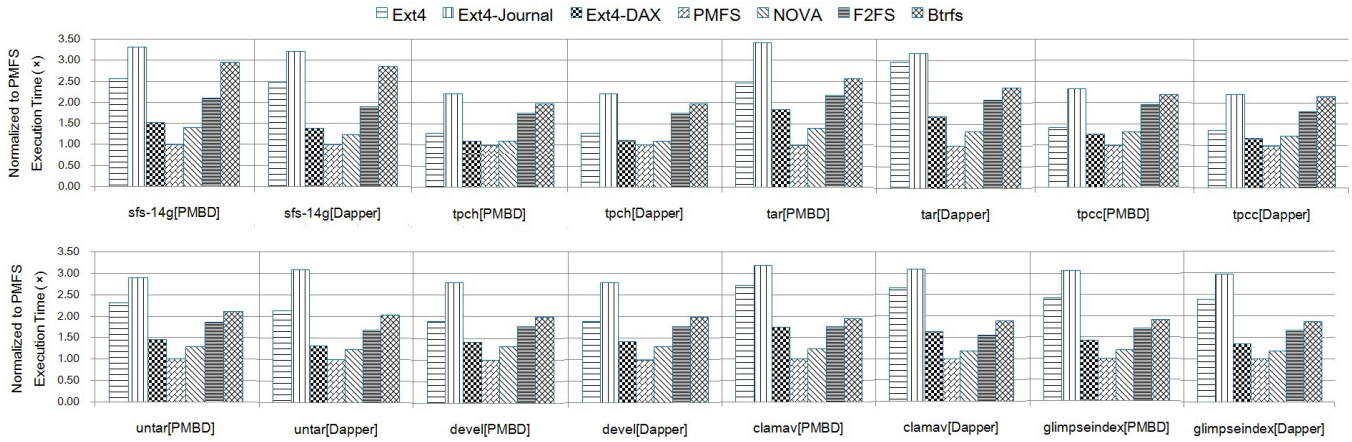


Fig. 16: Dapper vs PMBD: normalized to execution time.

As shown in Fig. 14, Dapper presents clear superiority in terms of graph construction time, graph generation time and validation time. These time metrics consistently demonstrate that Dapper's sufficient local memory provisioning approach is beneficial to Graph500 benchmarks.

6.3 Impact of Running on PM Volume

To construct a fair and reasonable comparison, we use the Intel Open Storage Toolkit [https://sourceforge.net/projects/intel-iscsi/] to form various types of microbenchmark workloads. These workloads print bandwidth, IOPS, and latency under different input parameters, such as random/sequential ratio, read/write ratio, request size and queue depth.

Fig. 15 presents a performance comparison between PMBD and Dapper. PMBD uses private mappings, nontemporal stores, and write barriers, while Dapper uses ***atomic sector update***. As shown in this result, Dapper can achieve 16.3 GB/sec read bandwidth and 11.4 GB/sec write bandwidth, which is slightly better than PMBD's corresponding performance. This result is because Dapper's design provides a stable mapping once the OS finish the initialization of the PM volume; in other words, Dapper finishes the whole mapping at once between the physical PM space on node4 and the logical volume block. Therefore, Dapper directly operates on the sector index, which is not referred to more transformation between page granularity and sector granularity.

From the perspective of an essential storage component in a general-purpose system, the PM volume should be no different than any other physical storage device. Enterprise users can create partitions and file systems on the PM volume such that traditional applications can be compatibly executed on it. For this reason, we deploy different general file systems to estimate whether Dapper's volume design generates disruptive performance consequences.

We also use 8 different workloads, which is the same as PMBD. Among these workloads, ***sfs***, ***tpch***, ***glimpseindex*** and ***clamav*** are read intensive, while ***tar*** and ***untar*** have similar amounts of reads and writes. We use execution time as the performance metric, and all the results are normalized to PMFS's running time. Note that we configured PMFS with ***pmfs-new*** [https://github.com/Andiry/pmfs-new], which does not support memory protection. Thus, ***pmfs-new*** is fit for use as a baseline. Considering that ***tpcc*** is a database online transaction processing workload with 63.7% writes, we convert the number of transactions into execution time. However, note that we configured the PM space on node4 as a memory mode when the PMFS and NOVA are deployed because these two file systems are in-memory file systems.

Fig. 16 provides the results of multiple workloads running on multiple file systems. We find that in-memory file systems undoubtedly achieve higher performance than other block-based file systems. However, we are amazed that F2FS and Btrfs obtain encouraging performance in all workloads except for ***sfs***. These results demonstrate that for most workloads, read and write operations are not the sole determining factor; appropriate adoption of a disk-based approach can achieve acceptable performance without sacrificing compatibility to exploit immature and unstable in-memory file systems. Therefore, some workloads with a critical real-time requirement can use in-memory file systems to achieve a good cost-effectiveness.

As shown in Fig. 16, Dapper is slightly faster than PMBD in most workloads. For example, compared to PMBD, I/O-intensive workloads ***tar*** and ***untar*** running on F2FS can achieve 5.8% and 11.9% performance improvement, respectively. Write-intensive workload ***tpcc*** running on F2FS can also achieve an 8.3% performance improvement. This result indicates that Dapper finishes all the mapping at the PM volume initialization stage, and when the workload is running on this device, our approach does not spend too much time handling mapping relationship. Another important feature of this figure is that ext4 and ext4-journal show unsatisfactory results. Notably, ext4-journal consumes a large amount of time to perform journal work regardless of workload types.

7 Discussion

Storing metadata in PM. Storing the metadata in PM will bring two unbearable drawbacks. On the one hand, it requires complex OS revisions. Traditionally, the initialization of page descriptors must be done in the process of the OS boot stage, and in this time-limited and space-limited stage, inserting a migration of moving metadata from DRAM space to PM space is a complex task. On the other hand, it decreases PM's endurance. Page descriptors themselves are continually changed with page allocation and reclamation, and considering that PM is a wear-sensitive device, storing this frequently updated data structure in PM definitely reduces the mediums endurance.

Devising huge page for PM. A huge page size also brings some drawbacks. First, inappropriate huge page sizes (e.g., 16 KB and 64 KB) result in vast modifications to the OS and incompatibilities with applications. Second, huge page creates preallocated contiguous memory space, whereas some NoSQL databases such as Couchbase [https://www.couchbase.com] generally need sparse memory access patterns and rarely have contiguous access patterns. Third, huge page produces a mass of memory inner holes that waste considerable memory space. For example, on a running Redis [https://redis.io/] server, many allocations are smaller than 2 MB. In addition, huge page is not swappable; thus, it easily incurs data loss for some security-sensitive applications.

8 Related Work

Introducing PM into a current commercial server is a challenging task because it is necessary to consider multiple factors, such as consistency, performance, compatibility and balancing of these factors. Fortunately, scholars and engineers have performed a great amount of important work in this interesting field. There are mainly two classes of approaches to manage PM devices: one focused on a memory-based model and the other based on a storage-based model. In the memory-based model, researchers consider the consistency problem [22] [23] [24] [25], maximization byte-addressable property [6] [7] [8] and customized file systems [18] [19] [26] [27] in PM. In the storage-based model, researchers focus on PM's storage abstraction approach [17] [28] and optimization [29]. Compared to previous works, Dapper mixes the memory-based model and storage-based model on one commercial server. Meanwhile, this paper exploits the expansion of local PM to decrease the cost of remote memory access in terms of memory organization and system software design. More remarkably, the following works are highly related to our work.

8.1 Optimizing Memory Affinity in Real NUMA Servers

Berkeley National Laboratory optimizes sparse matrix-vector multiplication on a multicore NUMA architecture [30], and this work handles specific scientific computations by memory-bound numerical algorithms. Grenoble University provides a user-level memory affinity interface [31] to control the NUMA platform, which requires programmers to manually modify some parallel benchmarks for geophysics applications; in fact, it mainly uses the Linux NUMA system calls. NUMA-ICTM [32] is a parallel solution that uses a memory affinity interface, which improves the subdivision work in geophysics. Some studies [33] [34] [38] [36] [37] depend on instrumented data from previous time steps to balance load for future time steps; thus, these measurement-based approaches lead to performance fluctuations and costly memory migration. Some works use cache miss rate [38] [39] [40] and intercache access [41] rate to measure memory resource contention and sharing, respectively. In addition, NUMA-aware lock [42] and energy consumption [43] in NUMA are also discussed in previous works. In contrast to these works, Dapper adaptively monitors the degree of memory deficit and automatically expands the local memory capacity to decrease remote memory access. Therefore, our work is a universal optimization that is fit for a large proportion of workloads.

8.2 Optimizing Memory Affinity in Virtual Machine

Colorado University finds that inaccurate abstract mapping between virtual hardware and physical hardware is always inefficient within virtual machines [44]. Based on this fact, they introduce NUMA awareness into the virtual machine scheduling via a bias random virtual CPU migration algorithm. More importantly, this work considers the memory locality factor by optimizing vCPU-to-core assignment. Xen guest NUMA supports [https://www.xenproject.org/] and VMware Resources management guide [https://www.vmware.com/] limit the virtual memory resources on a single NUMA node. Hyper-V virtual NUMA overview [https://docs.microsoft.com/]

transfer the NUMA hardware organization information to the virtual hardware, but these types of methods lead to a tight coupling to the virtual machine and physical resources, which limits the flexibility of the scheduling. Compared with these optimizations in virtual NUMA machine, Dapper focuses on optimizing memory locality on the physical NUMA machine, and Dapper's design philosophy can also be applied for virtual NUMA machines.

8.3 Memory Hotplug

The differences between memory hotplug and our work can be summarized as follows. First, memory hotplug has a wider application scope than Dapper because memory hotplug is a physical memory reconfiguration mechanism that can handle hardware errors, balance the workload, support physical memory extension, and so forth. In contrast, Dapper mainly provides an adaptive integration mechanism for decreasing remote memory access and automatically managing the PM space. Second, memory hotplug adjusts memory utilization by adding/deleting a real memory device directly, and the total physical memory spaces can be changed along with the physical memory reconfiguration. However, Dapper adds the detected PM space gradually and automatically makes them available for applications (support allocation and reclamation). The total physical PM space is fixed in advance. Third, memory hotplug consists of a physical memory hotplug phase and logical memory hotplug phase. In contrast, Dapper does not need to identify these two phases. Finally, memory hotplug needs to modify the entire memory subsystem, and our work only adds a kernel module in the local OS with minimal revisions.

9 Conclusion

In this paper, we present a hybrid memory organization architecture that is convenient for the OS to manage DRAM and PM space in a unified manner. Rather than access remote memory at the memory deficit stage, Dapper automatically scales the local memory's capacity according to memory demand. Our design avoids instant expansion of kernel metadata and unnecessary energy consumption through an adaptive memory hiding and provisioning mechanism. We also abstract a compatible PM volume device and guarantee the atomic sector update. We have implemented two kernel modules on a commercial server based on the Linux 4.5.0 kernel. Extensive evaluations show that Dapper can effectively decrease remote memory access and strongly decrease the performance degradation. We believe our idea also can be applicable to hyperscale datacenters.

References

[1] K. Bailey, L. Ceze, et al.. *Operating sys-tem implications of fast, cheap, non-volatile memory*. In HotOS, 2011

[2] A. Awad, P. Manadhata, et al.. *Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers*. In ASPLOS, 2016

[3] V. Young, P. J. Nair, et al.. *Deuce: Write-efficient encryption for non-volatile memories*. In ASPLOS, 2015

[4] S. Chhabra and Y. Solihin. *i-nvmm: a secure non-volatile main memory system with incremental encryption*. In ISCA, 2011

[5] X. Dongliang, L. Chao, et al. *Adaptive Memory Fusion: Towards Transparent, Agile Integration of Persistent Memory*. In HPCA, 2018

[6] H. Volos, A. J. Tack, et al. *Mnemosyne: Lightweight persistent memory*. In ASPLOS, 2011

[7] J. Coburn, A. M. Caulfield, A. et al. *Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories*. In ASPLOS, 2011

[8] T. Hwang, J. Jung, et al. *Heapo: Heap-based persistent object store*. In TOS, 2015

[9] Thomas Luc. *Basic Principles, Challenges and Opportunities of STT-MRAM for Embedded Memory Applications*. In MSST, 2017

[10] Bishnoi, Rajendra, et al. *Architectural aspects in design and analysis of SOT-based memories*. In Design Automation Conference (ASP-DAC), 2014

[11] Mittal Sparsh, Rujia Wang, and Jeffrey Vetter. *DESTINY: a comprehensive tool with 3D and multi-level cell memory modeling capability*. In Journal of Low Power Electronics and Applications 23, 2017

[12] K. Lim, P. Ranganathan, et al. *Understanding and designing new server architectures for emerging warehouse-computing environments*. In ISCA, 2008

[13] I. Paul, W. Huang, et al. *Harmonia: Balancing compute and memory power in high-performance gpus*. In ISCA, 2015

[14] H. Huang, K. G. Shin, et al. *Cooperative software-hardware power management for main memory*. In HotPower, 2004

[15] N. Agarwal and T. Wenisch. *Thermostat: Application-transparent page management for two-tiered main memory*. In ASPLOS, 2017

[16] Luc Thomas. *Basic Principles, Challenges and Opportunities of STT-MRAM for Embedded Memory Applications*. In MSST, 2017

[17] C. Feng, M. P. Mesnier, et al. *A protected block device for persistent memory*. In MSST, 2014

[18] J. Xu and S. Swanson, et al. *Nova: a log-structured file system for hybrid volatile/non-volatile main memories*. In FAST, 2016

[19] Dulloor, Subramanya R., et al. *System software for persistent memory*. In EuroSys, 2014

[20] Lee C, Sim D, et al. *F2FS: A New File System for Flash Storage*. In FAST, 2015

[21] RRodeh O, Bacik J, et al. *BTRFS: The Linux B-tree filesystem*. In TOS, 2013

[22] W.-H. Kim, J. Kim, et al. *Nvwal: exploiting nvram in write ahead logging*. In ASPLOS, 2016

[23] J. Seo, W.-H. Kim, et al. *Failure-atomic slotted paging for persistent memory*. In ASPLOS, 2017

[24] S. Nalli, S. Haria, et al. *An analysis of persistent memory use with whisper*. In ASPLOS, 2017

[25] Q. Hu, J Ren *Log-Structured Non-Volatile Main Memory*. In ATC, 2017

[26] J. Ou, J. Shu, et al. *A high performance file system for non-volatile main memory*. In EuroSys, 2016

[27] Y. Lu, J. Shu, et al. *Octopus: an RDMA-enabled Distributed Persistent Memory File System*. In ATC, 2017

[28] Kumar P, Huang H, et al. *SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection*. In IEEE International Congress on Big Data, 2017

[29] Liu. W, Wu. K, et al. *Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory*. In NAS, 2017

[30] Williams. S, Oliker. L, et al. *Optimization of sparse matrixvector multiplication on emerging multicore platforms*. In SC, 2007

[31] Ribeiro C P, Mehaut J F, et al. *Memory affinity for hierarchical shared memory multiprocessors*. In SBAC-PAD, 2009

[32] Castro M, Fernandes L G, et al. *NUMA-ICTM: A parallel version of ICTM exploiting memory placement strategies for NUMA machines*. In IPDPS, 2009

[33] Liu X, Mellor-Crummey J, et al. *A tool to analyze the performance of multithreaded programs on NUMA architectures*. In PPoPP, 2014

[34] Majo Z, Gross T R, et al. *Matching memory access patterns and data placement for NUMA systems*. In CGO, 2012

[35] Blagodurov S, Zhuravlev S, et al. *A case for NUMA-aware contention management on multicore systems*. In PACT, 2010

[36] Dashti M, Fedorova A, et al. *Traffic management: a holistic approach to memory placement on NUMA systems*. In ASPLOS, 2013

[37] Drebes A, Pop A, et al. *NUMA-aware Scheduling and Memory Allocation for data-flow task-parallel Applications*. In PPoPP, 2016

[38] S. Blagodurov, S. Zhuravlev, et al. *Contention-aware scheduling on multicore systems*. In TOC, 2010

[39] S. Blagodurov, S. Zhuravlev, et al. *A case for NUMA-aware contention management on multicore systems*. In ATC, 2011

[40] F. Guo and Y. Solihin. *A framework for providing quality of service in chip multi-processors*. In MICRO, 2007

[41] D. Tam, R. Azimi, et al. *Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors*. In EuroSys, 2007

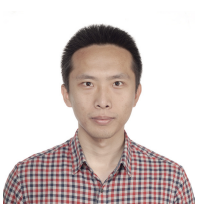
[42] Zhang M, Lau F, et al. *Scalable adaptive NUMA-aware lock: combining local locking and remote locking for efficient concurrency*. In PPoPP, 2016

[43] Francesquini E, Castro M, et al. *On the energy efficiency and performance of irregular application executions on multicore, NUMA and manycore platforms*. In Journal of Parallel Distributed Computing, 2015

[44] Rao J, Wang K, et al. *Optimizing virtual machine scheduling in NUMA multicore systems*. In HPCA, 2013

Dongliang Xue is a Ph.D. student in Department of Computer Science and Engineering at Shanghai Jiao Tong University since July 2015. He was a system software engineer from July 2013 to June 2015 at Shanghai Dianji University. Between Jan 2013 and July 2013, he was a system software engineer at ICT, Beijing, China. He received his masters degree in 2013 from USTC, China. His research is in the areas of in-memory computing, cloud computing, and non-volatile memory.

Linpeng Huang received his MS and PhD degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests lie in the areas of distributed systems and service oriented computing. He is a chief scientist for in-memory computing in the area of persistent memory.



Chao Li received his BS from Zhejiang University in 2009 and his PhD from the University of Florida in 2014. He is currently a Tenure-Track Assistant Professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include computer architecture, green data center, and emerging computing platforms. He received the Best Paper Award from IEEE HPCA in 2011 and IEEE Computer Architecture Letter in 2015.

Chentao Wu Chentao Wu received the BS, ME and Ph.D. degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2004, 2006, and 2010, respectively. He also received the PhD degree in electrical and computer engineering from Virginia Commonwealth University, Richmond, in 2012. He is currently an associate professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China. His research interests include computer architecture and data storage systems, and he has published more than 40 papers in prestigious international conferences and journals, such as IEEE TPDS, HPCA, DSN, and IPDPS. He is a member of the IEEE and China Computer Federation.