2018 IEEE International Symposium on High Performance Computer Architecture

Adaptive Memory Fusion: Towards Transparent, Agile Integration of Persistent Memory

Dongliang Xue[1] Chao Li[1] Linpeng Huang[1] Chentao Wu[1] Tianyou Li[2]
[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[2]*Intel Asia Pacific R&D Co., LTD*
xuedongliang010@sjtu.edu.cn, {lichao, huang-lp, wuct} @cs.sjtu.edu.cn, tianyou.li@intel.com

ABSTRACT

The great promise of in-memory computing inspires engineers to scale their main memory subsystems in a timely and efficient manner. Offering greatly expanded capacity at near-DRAM speed, today's new-generation persistent memory (PM) module is no doubt an ideal candidate for system upgrade. However, integrating DRAM-comparable PMs in current enterprise systems faces big barriers in terms of huge system modifications for software compatibility and complex runtime support. In addition, the very large PM capacity unavoidably results in massive metadata, which introduces significant performance and energy overhead. The inefficiency issue becomes even acute when the memory system reaches its capacity limit or the application requires large memory space allocation.

In this paper we propose adaptive memory fusion (AMF), a novel PM integration scheme that jointly solves the above issues. Rather than struggle to adapt to the persistence property of PM through modifying the full software stack, we focus on exploiting the high capacity feature of emerging PM modules. AMF is designed to be totally transparent to user applications by carefully hiding PM devices and managing the available PM space in a DRAM-like way. To further improve the performance, we devise holistic optimization scheme that allows the system to efficiently utilize system resources. Specifically, AMF is able to adaptively release PM based on memory pressure status, smartly reclaim PM pages, and enable fast space expansion with direct PM pass-through. We implement AMF as a kernel subsystem in Linux. Compared to traditional approaches, AMF could decrease the page faults number of high-resident-set benchmarks by up to 67.8% with an average of 46.1%. Using realistic in-memory database, we show that AMF outperforms existing solutions by 57.7% on SQLite and 21.8% on Redis. Overall, AMF represents a more lightweight design approach and it would greatly encourage rapid and flexible adoption of PM in the near future.

1. INTRODUCTION

The industry and academia alike have realized that large memory capacity is necessary for big data applications to conduct in-memory computing today. For example, SAP HANA [1], a commercial database vendor, installs very-large amount of memory (from 128GB to 4TB) on a single node to sustain fast query and real-time analysis in database processing. To keep large data sets in DRAM and ensure low latency, researchers from Stanford propose RAMCloud [2], which forms a 1 petabyte (PB) memory subsystem by aggregating ten thousands servers.

Unfortunately, keeping scaling up/out the DRAM can no longer create a sustainable business advantage today. Existing DRAM pooling strategies result in several issues such as greatly increased DIMM cost, significant board space overhead, and escalating energy needs. Considering the increasing gap between the processor computation capability and the available memory capacity [3], a timely memory capacity expansion strategy combing new hardware technology and efficient integration method is highly desired.

In this study we explore an efficient way to expand the main memory subsystem for high-performance in-memory data processing. Recently, the emergence of large-capacity PM modules presents a great opportunity to achieve this goal. A variety of new memory technologies such as ReRAM, STT-MRAM, and 3D XPoint could provide a performance almost comparable to DRAM (see Table 1). The byte-addressable property of PM further allows it to be directly accessed by load/store instructions just like a traditional DRAM. More importantly, PM allows for much larger hardware capacity than DRAM (e.g., terabytes instead of gigabytes) at highly competitive cost [5].

Although PM hardware highlights non-volatility of stored bits, we do not intend to introduce the persistence property into existing system at this stage. This is primarily because there are many issues that can greatly complicate the integration of PM. For example, exposing the unique feature of PM device to the application requires the revision of the whole software stack [9], which is a formidable task. In fact, traditional OS mechanisms such as page fault handling, memory swapping and address space management should all be tailored to fit the new access interfaces of PM. The procedures for booting and shutting down an operating system also need to be reworked since the PM has preserved a former execution state since last operation. Without appropriate management, selectively rejuvenating partial OS component (micro-boot) on PM can easily lead to unpredictable consequences. Further, many security issues need to be considered as well. Without privacy and security aware garbage collection policies, encryption keys and decrypted data in the durable cells of PM can be easily leaked [10][11][12].

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00036

Gracefully integrating PM into existing computer system is a non-trivial task, even if we only focus on the memory-like property of PM. For enterprise-level in memory computing, it is crucial to ensure compatibility and efficiency. There have been prior work investigate the byte-addressable attribute of persistent memory [13][14][15]. They focus on persistent objects that support inflexible programming interfaces (e.g., Pmalloc, Pfree, NVheap, NVHOpen, and NVobject). It is difficult for them to be widely adopted in existing commercial software. Meanwhile, some proposals such as PMFS [16], HiNFS [17] and Nova [18] provide virtual file system interfaces that are widely used in block devices. Although these works employ PM's memory-like property, they hurt system performance due to the overhead of I/O software stack.

In the era of "big" and "fast" data, persistent memory will play a more dominant role. However, from the perspective of a system designer, the new-generation PM modules are more likely to collaborate with DRAMs rather than replace them quickly. A key driving principle behind our design is to ***make the best use of the capacity benefits of PM while minimizing unnecessary changes to applications and the corresponding running environment.*** To this end, we propose adaptive memory fusion (AMF), an adaptive memory fusion strategy that smartly integrates high-performance PM into existing main memory subsystem.

AMF is a lightweight, application-transparent PM integration scheme. The transparent management of PM denotes that the user does not have to manually activate explicit interface, such as using Pmalloc to allocate space and Pfree to reclaim space. AMF is able to automatically configure all the PM-equipped NUMA nodes to make sure that the PM is correctly initialized. Afterwards, the PM devices are managed in a DRAM-like way in the kernel mode. AMF just employs several mature management mechanisms (e.g., buddy system for contiguous multi-page allocations) that are widely used in today's operating system.

We develop optimization techniques to make AMF an agile scheme that can efficiently handling large-capacity PM management. For example, our PM space allocation technique avoids metadata explosion by only allocating the necessary PM resources. Meanwhile, our system can dynamically append appropriate amount of PM space into a running system based on the memory pressure. This can greatly reduce the performance degradation due to expensive virtual memory swapping. Further, we also offer a compatible programming interface which can conveniently assist programmers to directly use the large capacity of physical PM at the user level.

We believe AMF is an important step in the evolution of systems towards ones with large persistent memories. In this paper we show limitations in the design of current systems that ultimately must be overcome. We also explore a reasonable set of compromises for the design of systems in the near future to take advantage of large-capacity PMs. Our design is orthogonal to techniques that aim to improve the performance and efficiency of DRAM and PM device.

In summary, we make the following key contributions:

- We introduce a fusion architecture which greatly facilitates memory capacity expansion with emerging PM modules. Existing OS can smoothly work with it with minor modifications.
- We devise a memory space fusion mechanism to adaptively provision PM resources. It is a transparent procedure in that memory-intensive applications can automatically benefit from it.
- We propose a holistic optimization strategy to better manage large-capacity PM. Our agile scheme can minimize metadata overhead, reduce costly I/O operations, and provide fast access to PM devices.
- We present our design as a unified solution called adaptive memory fusion (AMF). We implement AMF as a kernel module in Linux kernel 4.5.0.
- We extensively evaluate our design from different perspectives. Using high-resident-set benchmarks and realistic in-memory database application, we show that AMF has great performance and efficiency potential.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 analyzes different architecture options and depicts our design. Section 4 proposes our adaptive memory fusion mechanism. Section 5 details experiment setup. Section 6 presents evaluation results. Section 7 gives a short discussion and Section 8 covers related work. Finally, Section 9 concludes this paper.

2. BACKGROUND AND MOTIVATIONS

Persistent memory (PM) can be an attractive candidate for database applications and analytic workloads that require intensive memory operation. In this section we briefly introduce the opportunities and challenges related to PM.

2.1 Benefits of Tapping into PM

PM devices typically reside on the high-speed DRAM bus. They can provide very fast DRAM-like access to critical data. Today, new PM device technologies such as phase-change memory (PCM), Resistive RAM (ReRAM), and spin-transfer torque RAM (STT-RAM) provide persistent data storage with access latencies close to DRAM. According to recent studies [44][45][46], some of the high-performance PM mediums such as STT-RAM cloud yield DRAM-comparable performance in terms of read/write latency (Table 1). As technology advances, these PM mediums are expected to provide even higher density, lower power, and shorter latency [47].

Importantly, emerging PM technologies from different vendors allow us to run at memory speed with much larger capacity. In terms of capacity, PM will be roughly an order magnitude larger than DRAM. At the extreme, we can install terabytes rather than gigabytes of memory. For memory-intensive applications, one may be willing to sacrifice a little memory bandwidth performance for much larger capacity. Considering the lower price and reduced power demand of PM, emerging large-capacity PM module is indeed a top candidate for in-memory computing.

2.2 Issues with PM Integration

Integrating PM into current systems faces many challenges. We group the main issues into two categories: system modification and system optimization.

2.2.1 System Modification Challenge

The persistence of PM means that the memory cell retains data during power loss. One needs to devise appropriate abstractions and interfaces for managing the non-volatility property. In other words, introducing PM's persistence [8] could substantially influence operating system design and program development. Many major system components can be affected [9], such as virtual memory system, file system, program execution models, application installation, reliability and security module, etc. For instance, current OS mainly uses a "buddy system" to allocate and reclaim physical memory for every process. PM integration requires the kernel to create a similar system to managing PM space. In addition, the kernel needs to provide extra programming interfaces for upper applications to make use of PM. Besides, the persistent attribute of PM leads to newly allocated/reclaimed space that already includes unreleased persistent data – appropriate mechanisms are required to handle potential security issues.

System modification can be a big barrier to quick, widespread adoption of PM today and tomorrow. Huge modifications made to the corresponding running environment are not acceptable to enterprises. Making drastic changes to commercial databases is not practical as well.

2.2.2 System Optimization Challenge

Large capacity PM consumes huge metadata space. The kernel requires many bytes to describe dynamic properties for each page. The state information about a physical page is resident in a page descriptor (PD). For instance, the kernel uses it to identify the physical page that belongs to kernel code or kernel data. It also indicates that physical page is free or not free. PD can be quite large, requiring 56 bytes space to store in Linux-4.5.0 on the x86-64 architecture. In particular, at the terabyte level, a 1TB PM with 4KB page size requires 14GB space to store all page descriptors (1TB / 4KB × 56 B). Some researchers [19] advise that using a 64-bit miniature page descriptor to avoid the cost of a full page descriptor. However, it leads to huge kernel revision and reduces the compatibility of software. Thus, smarting managing PM space to reduce kernel metadata size is of paramount importance.

There are several other challenges related to large PM management. For example, memory energy consumption can be affected by its capacity in use. In Figure 1 we show the percentage of memory energy consumption on a Dell R920 server by running SPEC CPU2006 benchmarks. We measure six multi-programmed workloads of different memory footprints. As we can see, under high memory footprint, the energy consumption rate can be increased by over 50%. On the other hand, the demand of memory is continually changing even for the same application. In Figure 2 we evaluate the memory footprint for Redis benchmark under different input data sizes. Our results show that the requests of different data size can yield significant memory demand variation. By opportunistically tracking the demand, one can further reduce energy overhead.

Category	Read latency	Write latency	Endurance
DRAM	40-60ns	40-60ns	10^{16}
STT-RAM	10-50ns	10-50ns	10^{15}
ReRAM	50ns	80-100ns	10^{12}

Table 1. A comparison of memory technologies

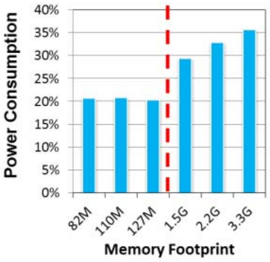


Fig. 1. Impact of capacity on power consumption

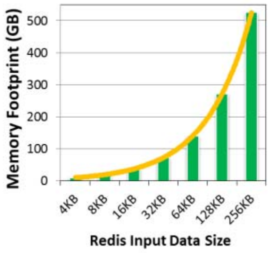


Fig. 2. Memory capacity demand variation

2.3 The Main Goal of This Paper

The intention of this project is to greatly facilitate the integration of PM in existing main memory subsystem. Faced with the system modification and optimization challenges, we aim to create a non-intrusive, light-weight design that allows a shared-memory machine to efficiently expand its memory capacity. We devise a novel memory fusion architecture and system mechanism to bypass the huge modifications faced by the OS; we also propose a holistic optimization strategy to reduce performance/energy overhead from different perspectives (detailed in Sections 3 and 4).

Note that there is no one-for-all ideal design. Heavily instrumented PM systems with additional software support maybe a good direction if one has a firm goal of replacing DRAM with PM and a strong need to unleash the full potential of memory persistency in the future. However, ***if one wants to expand memory capacity with minimum cost and seeks to boost in-memory computing right now, AMF with its lightweight, highly compatible and efficient design would be an attractive alternative.***

3. ARCHITECTURE ANALYSIS AND DESIGN

In this section we first compare different architecture design options for PM integration. We then discuss the key feature of our fusion architecture.

3.1 Architecture Analysis

There are several design options available for integrating PM into the main memory subsystem of NUMA machines. In Figure 3 we illustrate their logical architectures.

Option A1 shows a traditional design in which PM is not used. In this case the native OS works smoothly and no specific modifications are required for existing applications to run on the system. When PM hardware is directly used as a storage device (Option A2), it maintains disk semantics. In this case, the OS just treats the non-volatile device as conventional block storage, and creates a filesystem on it.

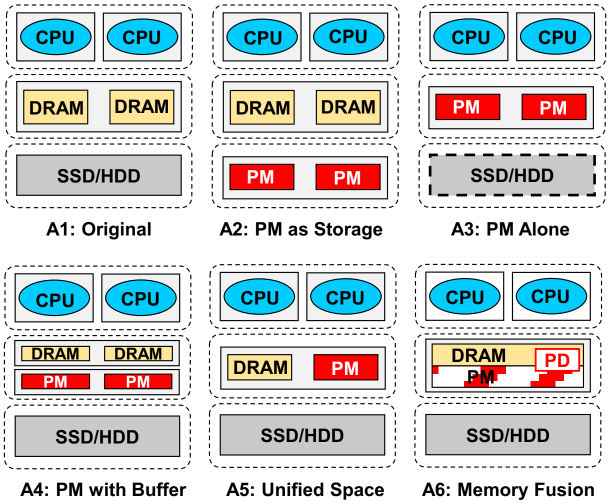


Fig. 3. Schematic diagram of different designs

Due to the block access patterns and the overhead of I/O software stack, the benefits of PM cannot be shown. In this work we focus on constructing the memory subsystem using PM which sits in the system's DIMM slots.

A group of architecture designs require significant modifications to the OS. For example, one can use PM exclusively or use DRAM as a "cache". In Figure 3, A3 replaces the DRAM with PM (A3). The SSD/HDD may also be removed since the data in the PM is persistent. However, significant modifications are required to ensure that the booting and shutting down of the OS works smoothly and correctly. Like A4 shows, one can also insert a DRAM layer as the buffer for PM [31]. For example, Hewlett Packard Enterprise's PM system includes a DRAM layer to accelerate applications, a dedicated flash tier for persistency. Appropriate software modules are required to ensure that the DRAM is transparent to main memory subsystem. The option A5 offers a uniform addressing space [32]. It relies on the memory controller to coordinate the allocation and reclamation of DRAM and PM resources. All the above designs overlook the impact of PM's persistence on OS, which is a troublesome issue in practice.

3.2 Fusion Architecture

We present a fusion architecture that synergistically integrates DRAM and PM, as shown in Figure 3 (A6). It uses both PM and DRAM to get a very large pool of addressable memory without the significant overhead.

The physical architecture of A6 is close to A5. It considers a hybrid system (DRAM+PM) based on a NUMA machine offering uniform addressing space. It uses a traditional two-layer framework (DRAM+Storage) on the first NUMA node. Both DRAM and PM are configured with a DIMM (Dual Inline Memory Modules) interface.

Differently, A6 highlights unique PM organization and management. First, it exposes itself to the above OS as a DRAM by hiding the PM space appropriately. It configures other NUMA nodes with PM and keeps these PM detectable (but not accessible). As a result, the OS is able to directly boot from the DRAM node just like a native OS running on architecture A1. Consequently, there is no need to heavily handle the booting and shutting issues originated from PM's persistence property. Second, A6 intends to hide PM space from the system. It only integrates partial PM space into the system to avoid instantaneous extension of the metadata. The system always stores frequently modified metadata such as page descriptors and page tables on DRAM node. By doing so the system can efficiently decrease the burden of the memory controller and reduce the writing frequency to wear-sensitive PM device.

4. ADAPTIVE MEMORY FUSION

This section presents the design and implementation of AMF, our adaptive memory fusion design. We first give an overview of the system architecture. We then discuss our memory space fusion design that allows existing application to transparently use the DRAM-PM fusion architecture. More importantly, we propose a holistic performance optimization scheme to further improve system efficiency.

4.1 Overview

Figure 4 depicts the basic architecture of AMF which spans across three layers: application, operating system, and hardware. We implement AMF as a prototype in Linux-4.5.0. While the major components of AMF lie in the OS level, our work does not require significant modification to the system software. In fact, our method hides the complexity of PM integration from the user and OS. It provides an efficient way to gracefully utilize PM resources.

In Figure 4, we consider a shared-memory system that supports non-uniform memory access (NUMA). DRAM Node 1 and PM Nodes 2~4 are memory components in a uniform physical address. The system can smoothly manage the PM and DRAM space together. The kernel boots from DRAM Node1 after loading the OS image from a secondary storage. AMF forces all the PM nodes to be hidden from the OS at the beginning. It dynamically allocates and manages PM resources afterwards. Specifically, AMF is composed of three main modules: a kpmemd service, a Hide/Reload Unit, and an On-Demand Mapping Unit.

Kpmemd is our newly inserted kernel service. When the memory footprint reaches its limit, kpmemd can automatically provide moderate amount of PM space (e.g., entire PM Node2 space or partial PM Node3 space) to alleviate the memory pressure. Kpmemd controls a group of managers associated with the physical memory. For example, a unique Manager 0 is created for the DRAM module. It collaborates with our kpmemd service to monitor the memory pressure to determine the activity of the other managers for PM (Managers 1~N). Figure 4 shows two PM managers. Manager 1 can be completely inactive if the whole PM Node2 is hidden; Manager 2 is activated even if only part of the PM is allocated.

The Hide/Reload Unit (HRU) is responsible for hiding and reloading PM resources. On each NUMA node, the

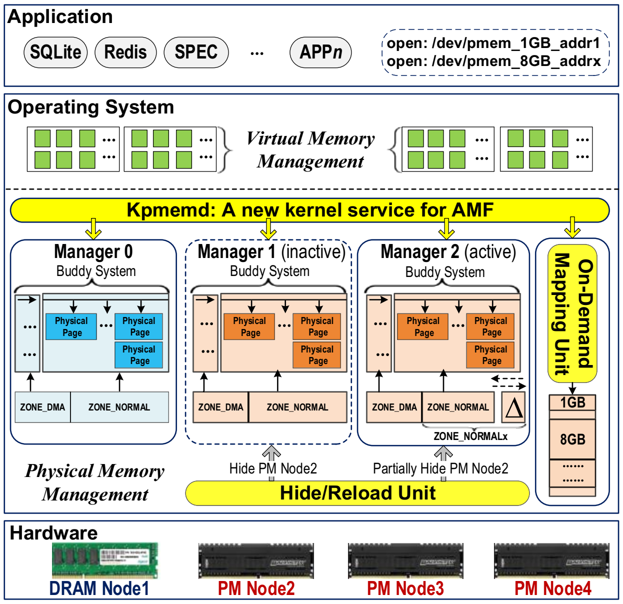


Fig. 4. Adaptive memory fusion (AMF) overview. It assumes a shared-memory NUMA system. Kpmemd, Hide/Reload Unit, and On-Demand Mapping Unit are the key components of our design

memory space consists of ZONE_NORMAL [1] and ZONE_DMA. Additional space can be added on "ZONE_NORMAL" depending on how much PM is hidden. The allocation and reclamation of the integrated PM are both managed by the kernel just like DRAM (i.e., PM Manager is similar to Manager 0). Further, all these integration procedure is transparent to the user.

The AMF architecture also presents a PM access technique in the user mode. This feature is completely compatible with the current programming interface. In Figure 4, the Dynamic On-Demand Mapping Unit works in the kernel mode. It partitions PM into different regions to map different device files. Application can directly access the physical PM space in unprivileged mode by using a traditional "mmap" interface. In this work we re-define "mmap" to enable direct PM space control (detailed in Section 4.3.3)

In the following, we first introduce our memory space fusion mechanism which lays the foundation for PM management. We then discuss detailed optimization schemes that further improve AMF performance.

4.2 Memory Space Fusion Mechanism

The memory space fusion mechanism hides the complexity of PM integration. Its goal is to reserve PM resource for reducing metadata. It features a conservative initialization and dynamic provisioning scheme.

4.2.1 Conservative Initialization

The basic idea of conservative initialization is to avoid initializing all the hardware resource at the boot stage. Our system can control the degree of initialization. Figure 5 shows the basic procedure in our system to perform conservative initialization. It mainly consists of four phases: profiling, redefining, preparing, and launching.

In the profiling phase, the system needs to detect and probe the physical memory regions and convert the detectable information into a useable form. We can obtain basic memory information through BIOS in the real mode (16-bit mode) in the early stage of booting. These important data will be passed to a predefined area that can be detected by the system after booting.

In the redefining phase our system changes the upper limit of memory to hide the PM space. To initialize partial PM space, we need to modify the last frame number as a predefined value. The last/highest frame number of the whole memory should be replaced by the DRAM's last frame number on DRAM Node1.

Phases 3 and 4 are necessary to execute corresponding modifications based on the value of the last frame number. We need to initialize the sparse memory model. In this model, the memory space is divided into multiple sections, and the page descriptors are just initialized at the head of each section.

Finally, our system starts the buddy system. The above phases set up the basic mechanism for managing the physical memory. Our design ensures that the system only initializes part of the PM space. It leaves the remaining PM space detectable but inaccessible.

4.2.2 Dynamic PM Provisioning

During runtime, AMF keeps a watchful eye on system memory footprint. If the DRAM has inadequate space to satisfy an application's memory request, AMF will trigger PM space integration. Figure 6 shows the basic procedures that support dynamic PM provisioning.

In Figure 6, the first phase aims to obtain the distribution information of physical regions and its capacity range in 64-bit mode at runtime. Afterwards, the other three phases are responsible for reloading the hidden PM space and make it accessible to the operating system.

Information Detection:

It is the first and most important step (called probing phase) for dynamic PM provisioning. Obtaining the distribution information can be tricky. The conventional method for obtaining the information is to rebuild a detecting procedure by triggering a BIOS interruption. However, it is effective only in a real mode (a.k.a. real address mode), not in a 64-bit mode as OS finishes booting.

In this work we choose to transfer the detected information from a real address mode to a 64-bit mode. AMF employs this method to copy the detected information from the boot-parameter-page to the predefined probe area; the information contained in the boot-parameter-page is originated from BIOS interruption in the real address mode. AMF takes advantage of a sequential transferring approach, which guarantees that the detected information is delivered from the real address mode to the protect mode and then to 64-bit mode. Finally, AMF can get the transferred data from the predefined probe area.

[1] ZONE_NORMAL and ZONE_DMA are all Linux concepts for memory management

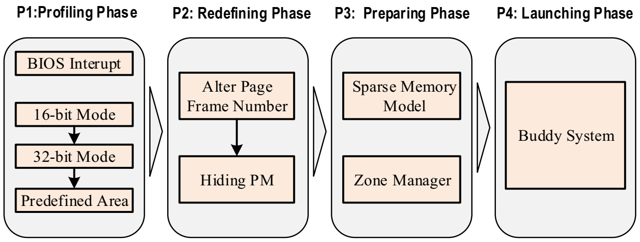


Fig. 5. Conservative initialization procedure

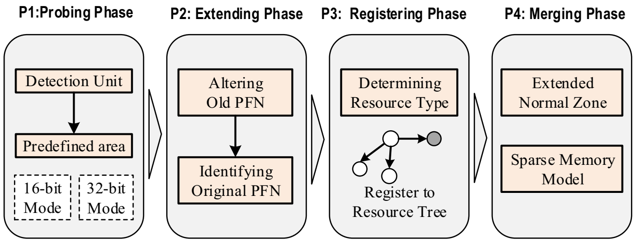


Fig. 6. Dynamic PM provisioning procedure

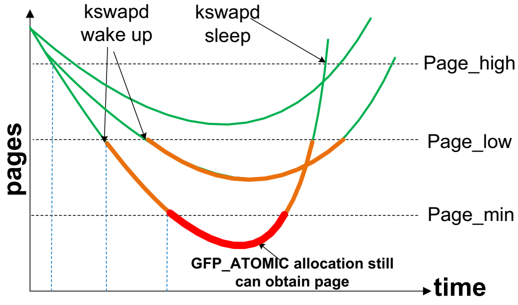


Fig. 7. Memory watermarks

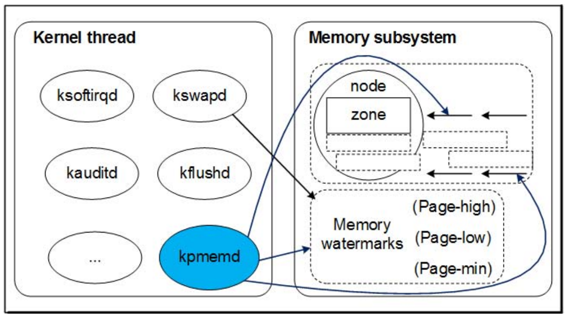


Fig. 8. AMF kernel thread

Reloading PM Space:

After information detection, the remainder/hidden PM space is already detectable in the 64-bit mode on a running system. The next task is to dynamically release the hidden PM space. To achieve this end, we need to force the partial PM space to operate in a DRAM-alike way. From the perspective of system implementation, it is necessary to build a series of functions and interfaces.

AMF extends the total physical page frame number in the extending phase. To extend the original last page frame number, we need to calculate an offset value. Here the offset denotes the total page frames of the newly added PM space (detailed in Section 4.3).

In the registering phase, the system registers the newly added PM space to a unified resource tree. The resource tree is a special data structure for managing resources in Linux. It facilitates manage of device resources.

In the merging phase, AMF merges the newly added PM space into existing system. A new ZONE_NORMAL on the corresponding node is formed based on the memory distribution information coming from the probe area. When building the sparse memory model, the newly added PM has to be spitted into multiple sections.

Finally, the newly added PM space becomes manageable. It is under the control of a unified buddy system.

4.3 Agile Memory Space Management

Note that it is not enough to just routinely hide and reload the PM. If the PM resource allocation is aggressive, more active pages and kernel metadata can be generated. If the allocation strategy is too conservative (i.e., freeing up inadequate amount of memory), it can trigger costly SWAP operations. Therefore, an agile memory space management is needed to achieve a better design tradeoff.

4.3.1 Relaxed PM Allocation

AMF leverages memory watermarks to enable memory pressure-aware allocation. Memory watermarks represent current memory pressure on a running system. In Figure 7, Page_min, Page_low, Page_high are memory watermarks. Their values are fixed once the kernel obtains the amount of present pages. Page_min identifies the minimum memory space that must remain free for critical allocations. Page_low is a warning line: once the remaining free pages drop below it, a kernel thread called kswapd will be activated to trigger memory SWAP operation. Page_high is a threshold: the kswapd will sleep if the observed number of free pages is larger than it.

To reduce or prevent the costly SWAP operations, it is necessary to provide relatively larger amount of PM space (relaxed allocation). During the runtime, our kernel services dynamically monitors the memory watermarks to obtain a reasonable estimation of the amount of PM that we need to integrate. We then extend the ZONE_NORMAL and increase the number of present pages.

AMF's relaxed PM allocation is supported by our kernel service kpmemd. Importantly, kpmemd can collaborate very well with existing kernel services, as shown in Figure 8. To detect the memory pressure, kpmemd inserts itself before kswapd. If kpmemd effectively alleviates the problem, kswapd maintains the sleep state. Otherwise, kswapd and kpmemd jointly handle the memory pressure issue.

Conventionally, these watermarks are at the granularity of MBs. For example, the watermarks value on our platform is: Page_min–16MB (4097 pages), Page_low–20MB (5121 pages), Page_high–24MB (6145 pages). To effectively handle the applications whose memory footprint are large (e.g., at GB levels), we devise a pressure-aware ca-

Remainder free pages	Amount of integrating
>page_high× 1024	DRAM's capacity× 0
(page_low× 1024, page_high× 1024]	DRAM's capacity× 1
(page_min× 1024, page_low× 1024]	DRAM's capacity× 2
(page_high, page_min× 1024]	DRAM's capacity× 3
[page_low, page_high]	DRAM's capacity× 5

Table 2. Policy of integrating amount

Descriptions: Example of application directly using physical PM space. Our "mmap" (row 3) completely compatible with traditional "mmap" (row 4).

Input: an opened device file represents a huge PM space and a huge ISO image file on disk.
Output: huge image file is moved to PM space.

```
1. fd1 = open("/dev/pmem1_8GB_0x30000000000",
O_RDWR);
2. fd2 = open ("/media/CentOS7.iso", O_RDWR);
3. pdata1 = (char*)mmap(NULL, 0x200000000, ROT_READ |
PROT_WRITE, MAP_SHARED, fd1, 0x30000000000);
4. pdata2 = (char*)mmap(NULL, 0x200000000, ROT_READ |
PROT_WRITE, MAP_SHARED, fd2, 0);
5. memcpy(pdata1, pdata2, 0x200000000);
6. close(fd1);
7. munmap(pdata1, 0x200000000);
8. close(fd2);
9. munmap(pdata2, 0x200000000);
```

Fig. 9. PM pass-through usage example

pacity expansion policy as shown in Table 2. By increasing PM to reach a high level of memory watermarks, AMF is able to prevent or postpone the costly swapping.

4.3.2 Lazy PM Reclamation

The space of PM's page descriptors occupied must to be reclaimed; otherwise it will nibble away the precious DRAM space. Releasing the space of PM's page descriptors occupied denotes that it is necessary to firstly remove the reclaimed PM pages from the buddy system. This process must be very careful since immediate reclamation can result in page thrashing.

Our idea is to dynamically assess the benefits of PM reclamation. If the expected DRAM space saving is higher than a predefined threshold value (e.g., 3% of the installed DRAM space in our system), our kernel service will remove the selected PM space from the system. AMF mainly removes the reclaimed space from the free list of the buddy system and to shrink the size of the ZONE_NORMALx. Finally, we reset the space that PM's page descriptors occupied. Particularly, the whole removal is also automatically handled as a partial function of the kpmemd. Our kernel service periodically scans the amount of the reclaimed PM space to remove multiple sections from the system.

4.3.3 Direct PM Pass-Through

Existing kernel provides an efficient file access mechanism through the mmap system call. The main feature of the mmap is that the file content can be directly mapped to a continuous virtual address space (referred to as MMAP Region), and the physical space can map this MMAP Region under automatic control of the Kernel. Hence, read and write operations can be converted into memory accessing of this MMAP Region in the user mode.

The mmap approach implies that partial physical PM space can be conveniently accessed in the user mode. We can allocate different amount of PM space by constructing different device file (e.g., /dev/pmem_1GB_addr1). We call this PM usage approach as direct PM pass-through. The benefit of this approach is two-fold. First, the device file can be easily registered to Devices-Drivers-Model which employs existing functions and interfaces. Second, different sizes of PM space are explicitly organized in user-mode so that programmer can conveniently access them by the file system interface (e.g., open/close).

In the environment of Linux-64, the virtual MMAP Region has reached TB level. It is sufficient for managing the huge physical PM space. In this paper we directly allocate virtual memory area which belongs to MMAP Region in the Kernel. Our system dynamically builds page table for mapping between device file and the virtual memory region (Figure 4). Our customized mmap only borrows open and close interfaces from the Virtual File System (VFS). It can effectively avoid the overhead of IO software stack and maximizes the byte-addressable property of PM. In Figure 9 we provide an instance of using our mmap.

5. EXPERIMENTALSETUP

We run AMF on a quadruple-socket Intel Xeon-based system and a fresh Linux kernel. This platform has a large memory capacity organized in NUMA architecture and is easy to emulate large capacity of the persistent memory. Table 3 describes the specification of the platform.

Since PM technology is still in an active developing phase now, in this study we emulate PM with DRAM. This approach has been adopted by many prior studies on PM based system designs [42][45]. The performance of PM is comparable to DRAM. In this paper we mainly focus on exploiting the capacity benefits of integration PM in existing main memory subsystems. Therefore, the evaluation results presented below does not take into account the difference of accessing latencies between PM and DRAM.

The total amount of memory is 512GB on our platform. On Node1, the first 64GB is regarded as DRAM and the second 64GB is regard as PM. The remainder 384GB on Node2, Node3 and Nnode4 are all treated as PM. DRAM area is managed by original kernel and is available to applications at any time. PM regions are managed by the original kernel together with AMF to monitor the memory pressure. The PM region on Node4 is managed by the original kernel and AMF as well, and notably, applications can also utilize this region to map/munmap AMF's device file.

To demonstrate that AMF can effectively eliminate or decrease the memory deficits, we select nine benchmarks from the SPEC CPU2006 suite (http://www.spec.org). The memory footprint of the benchmarks is large enough to evoke memory deficiency. We use htop [33] which is an

Component	Specification
Platform	Dell R920 shared-memory Server
CPU	Xeon E7-4820 8-core Processor ×4, 16M LLC
Main Memory	512GB, 1066Mhz
OS	Centos 6.6
Kernel Version	Linux Kernel 4.5.0
File system	Ext4

Table 3. Specification of our platform

# of Instance		Unified (static PM)	AMF[dynamic PM]
Exp. 1	129	64G DRAM+(64G PM)	64G DRAM+[64G PM]
Exp. 2	193	64G DRAM+(128G PM)	64G DRAM+[128G PM]
Exp. 3	277	64G DRAM+(192G PM)	64G DRAM+[192G PM]
Exp. 4	385	64G DRAM+(320G PM)	64G DRAM+[320G PM]

Table 4. Evaluated baseline configurations

vm.overcommit_memory = 1	port = 6379
rdbcompression = yes	save = disable
appendonly = no	appendfsync = no
timeout = 300	maxclinets = 60000
hostname = local	clients = 500
requests = 30 million	pipeline = 512
random keys= 400k	data size = 4kB

Table 5. Major parameters used for Redis

interactive process viewer for Unix to monitor the memory footprint of these benchmarks. Since the DRAM capacity of the platform is large (reaches 64GB), our experiment requires executing multiple instances of the benchmarks to cause large quantities of memory access to activate the function of AMF. We repeat experiments five times and average the results to reduce the randomness of measurement. We run STREAM benchmark [35] on our system to ensure that the bandwidth difference between our emulating platform and PM is within 5%.

We mainly compare our design with conventional design that tries to build a unified space of DRAM and PM (i.e., architecture design A5). Table 4 shows the configuration of our experiment. We implement a benchmark to evaluate our direct PM pass-through technique. It allocates/reclaims the PM space using AMF's self-defined but compatible "mmap/munmap" interface to replace traditional array space based on STREAM.

We also evaluate two representative in-memory computing applications. SQLite has been widely used in datacenters as the underlying storage engine for application-specific database servers. We measure the throughput of SQLite on servers which configured with large capacity PM space. Then, we implemented a benchmark which creates a database purely in memory and performs random insert, update, select and delete transactions. Redis [36] is a popular in-memory data structure store, widely used as a database, cache and message broker. Twitter uses Redis to scale exploding growth of cache service because of its in-memory nature. Table 5 shows the parameters we use for evaluating Redis. In test AMF's capability of handling memory pressure, we push nearly 30 million requests which are sufficient to trigger huge memory footprint.

6. EVALUATION RESULTS

In this section we discuss the benefits of applying AMF to PM-DRAM hybrid systems.

6.1 Performance Implications

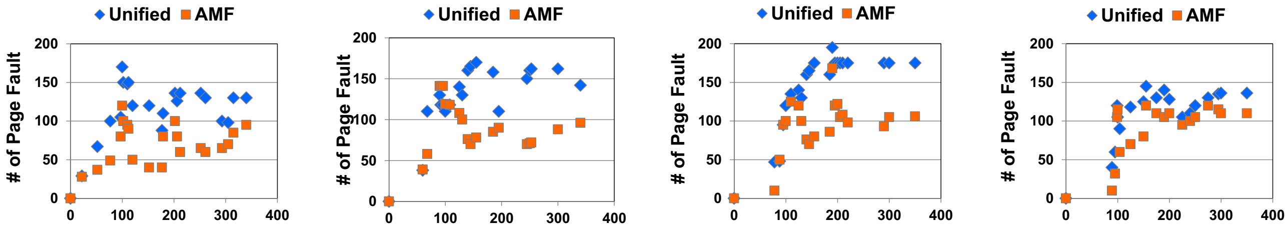
We first evaluate the impact of AMF on system performance. We measure and collect different system statistics that are closely related to system performance.

Average number of page faults: Figure.10 presents the average page fault number of AMF and Unified at different timestamps. The results show that AMF is indeed able to effectively alleviate page fault with different sizes of memory footprint. In fact, excessive memory allocation can lead to memory deficit/pressure, which often evokes the inherent page fault mechanism of the OS kernel. Differently, our system just provides moderate amount of PM space to expand the memory space and satisfy more memory allocation needs. Thus, the total number of page faults declines, comparing with Unified.

Occupied SWAP partition size: In Figure.11 we compare the occupied SWAP partition size of AMF and our baseline Unified over the time. The plot shows that AMF can decrease the consumption of SWAP partition space. This is because our system dynamically provides appropriate amount of PM space. The newly introduced PM space prevents the OS kernel from frequently activating kswapd. AS a result the kernel does not have to swap the memory space to the slow HDD/SSD. In fact, SSDs can quick wear out if we frequently use it for swap.

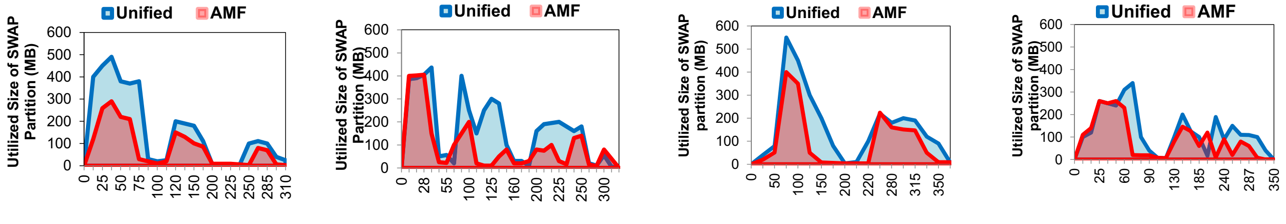
Percentage of kernel/user mode: Figure.12 further presents the percentage of CPU time spent in kernel mode and CPU time spent in user mode. It is evident that AMF's CPU time in user mode is significantly higher than that of Unified, while our CPU time in kernel mode is slightly lower than that of our baseline. The higher CPU time in user mode denotes that CPU cycles spent more useful time to execute user-level instructions. The system does not have to frequently trap into the kernel mode to handle page fault. We can also see that both baseline and AMF have a particularly huge memory demand. Thus, they spend a similar portion of their execution time in kernel mode. The curve moves sharply at intervals signifies a new batch of instances are launched in user-mode every once in a while, because the total number of instances is far greater than the number of cores in the system, all the instances cannot completely finish executing at once. At the end of every batch, part of the core resources have been gradually released, which leads to a dithering for CPU time.

Performance of multiple benchmarks: We also constructed total 675 instances (a group of SPEC CPU2006 benchmarks) to examine the improvement of our work in term of total page faults and total occupied size of SWAP partition. Figure 13 depicts the normalized total page faults. Our results show that the total page fault number is dropped by up to 67.8% with an average of 46.1%. Figure 14 further depicts the total occupied size of SWAP partition using AMF. The plot shows that the total occupied size of SWAP partition is dropped by up to 72.0% with an average of



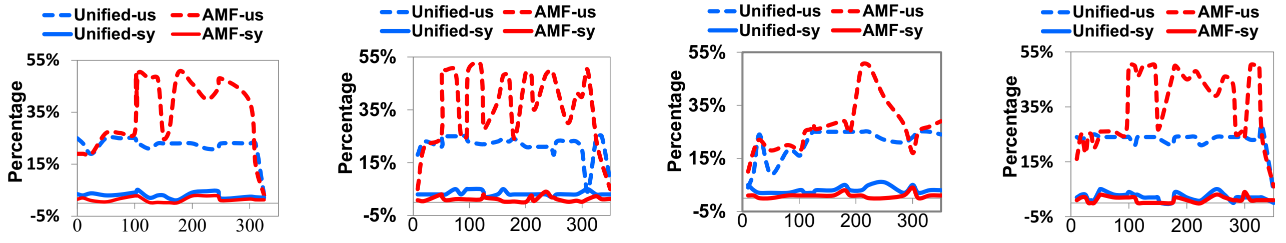
(a) Exp.1 (b) Exp.2 (c) Exp.3 (d) Exp.4

Fig. 10. Average page fault number using mcf application. Horizontal axis shows time in minutes



(a) Exp.1 (b) Exp.2 (c) Exp.3 (d) Exp.4

Fig. 11. Utilized size of SWAP partition over the time. Horizontal axis shows time in minutes



(a) Exp.1 (b) Exp.2 (c) Exp.3 (d) Exp.4

Fig. 12. CPU time in system (sy) and user (us) mode. The horizontal axis shows time in minutes

29.5%. Therefore, by smartly integration PM into existing memory systems one can greatly improve the performance.

Primary metrics, including the number of page faults, the size of occupied SWAP capacity and the ratio of user mode running time consistently demonstrate that AMF is superior to our baseline. This is mainly because AMF is able to maximally utilize the available DRAM space. At the application launch state, AMF has more available DRAM space than Unified because it avoids excessive Page Descriptors. Thus, AMF can run more instances from the beginning. Once memory deficit became visible, AMF immediately integrates appropriate PM space into the system, which seldom restricts from running more instances.

6.2 Power Efficiency Analysis

We then estimated the potential energy saving of AMF using the actual system log collected from our system and analytical models. Similar to prior work [4][6][7], we ignore other memory states and calculate power demand based on Micron's methodology [34], In idle states the system consumes about 0.23W/GB while in the active states consumes about 1.34W/GB. The transition from idle to active states consumes about 0.76 W/GB. Our estimation considers the actually number of workload instances.

Compared to the conventional design, AMF shows significant energy savings due to the lower memory capacity overhead and relatively faster execution time. Our estimation is conservative since we primarily rely on DRAM parameter for calculation. With actual PM devices that are typically more energy-efficient than DRAM, the overall power demand of our system can be even lower.

6.3 Impact of PM Pass-Through

We further evaluate the execution time of using device file identified PM space (e.g., the On-Demand Mapping Unit in AMF) Figure.16 presents the execution time of the STREAM operations, normalized to native running.

Our results show that the execution time of each operation using AMF provided interface is comparable to original array interface. The largest gap is less than 1%. This demonstrates that our designed mapping mechanism does not incur significant performance degradation, although it requires a real-time mapping operation.

6.4 Case Studies

To understand the performance of commercial software using AMF, we evaluate two different types of database software: SQLit and Redis.

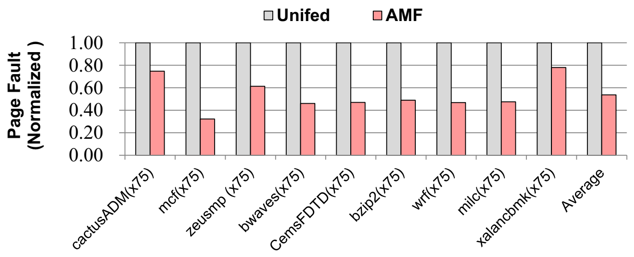


Fig. 13. Page faults with mixed benchmarks

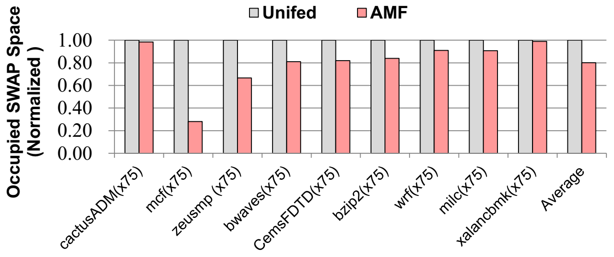


Fig. 14. Occupied size of SWAP partition

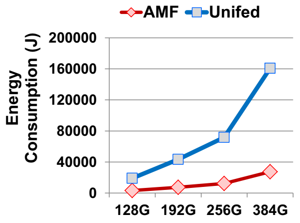


Fig. 15. Energy benefits from adaptive memory fusion

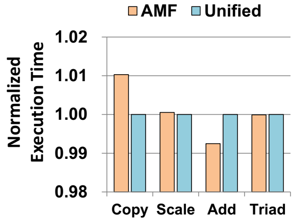


Fig. 16. Impact of direct PM pass-through on performance

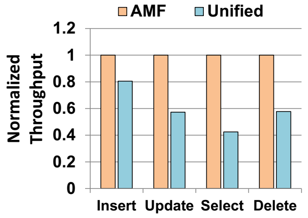


Fig. 17. Performance impact of AMF on SQLite database

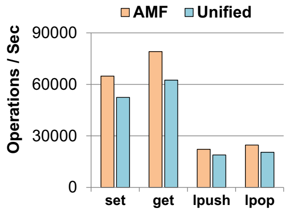


Fig. 18. Performance impact of AMF on Redis key-value store

SQLite Database Transactions: Figure 17 presents the transaction improvements triggered by our work, normalized to native running. We prepare approximately 17 million records for insert transaction and 3 million records for each update, select, delete respectively. The histogram shows that the throughput of in-memory database SQLite is improved by up to 57.7% with an average of 40.6%. In particularly, we define "throughput" as the number of transactions executed per seconds, while in SPEC-CPU-2006 criterion it means the number of instructions executed per seconds. That is, our granularity of "throughput" is larger than traditional ones but does not affect the results.

Redis Key-Value Store Requests: Finally, in Figure 18 we present the results of several popular several Redis operations (set, get, lpush, lpop). AMF shows that it can achieve average 25.1% requests improvement of set/get, and average 18.5% requests improvement of lpush/lpop compared to the conventional design. The reason for this lies in providing more available space by AMF's adaptive PM provisioning mechanism just matches Redis's in-memory computing demand.

7. DISCUSSIONS

Tapping into Huge Pages. Linux supports huge page and transparent huge page [42]. Huge Pages create pre-allocated contiguous memory space designed to assist application performance. However, this mechanism requires the system software to implement its own support in user space to take advantage of these potential performance benefits. Huge page uses fewer pages to cover the physical address space. As a result it requires fewer TLB entries and incurs fewer TLB misses. In this study, huge pages allow AMF to easily integrate more PMs.

Note that some NoSQL databases such as Couchbase [43] usually need sparse memory access patterns and rarely have contiguous access patterns. In addition, huge pages are not swappable. Thus, it easily incurs data loss for some security-sensitive applications. Huge pages require large areas of contiguous physical memory, while a rabbit hole (memory inner hole) begins when a Redis-server process is running because of substantially allocations smaller than 2MB.

Storing Page Descriptors in PM. Storing the metadata (i.e., page descriptors) in the PM requires complex OS revision. In addition, migrating page descriptors to PM space to store metadata is not practical. The initialization of page descriptors must be done at the OS booting procedure, which introduces significant overhead. Even worse, page descriptors itself are frequently changed data structures. Storing them in PM significantly increases the burden of the hardware and seriously decreases device lifetime.

Consistency. Previous works such as logging [21] [22], shadow method [23], persistent transactions [24] [25], light-weight method [26] [27] have provided effective methods for guaranteeing the consistency. In this paper, the consistency follows these assumptions: 1) the system supports an atomic write of 64 bits and 2) does not use flight memory operations. We solve the consistency problem at the memory controller layer similar to ThyNVM [28].

Wear Levering. Prior works [29] [31] proposed a few mechanisms to promote PM lifetime at hardware/software layer or from the perspective of hardware-software co-design. As it is believed that wear levering will be resolved at the memory controller layer in the near future, we omit this issue in this study. Our work tries to decrease the burden of hardware by considering wear levering.

8. RELATED WORK

As the release date of Intel Apache Pass DIMM [37] on the Sky-Lake based servers approached, researchers and engineers have made significant progress for introducing PM into current computer systems. These works mainly excavate PM's persistent property [16] [17] [18], byte-addressable property [13] [14] [15], and maximally relieve the consistency problem [22] [25] [27] and wear leveling problem [30] [38] [39] [20]. In contrast to these prior works, this paper focuses on PM's large-capacity property, from the perspective of memory architecture and system software, to integrate PM into current computer system.

Besides the above PM related studies, there are also a few works highly related to our work.

Memory Ballooning. Hypervisor employs memory ballooning technique to coordinate the available memory between the Virtual Machines (VMs) [40]. Our work differs from this technique in four aspects. First, ballooning is based on virtualization technology, while our work is based on a native OS. Second, ballooning aims to optimize the utilization of memory between VMs by the balloon driver, while AMF aims at managing the PM space. Third, ballooning adjusts memory utilization in low granularity of pages and often incurs page thrashing. We manage the PM space in high granularity of sections, which is able to alleviate the overall memory pressure and never shrinks memory-resident-set. Forth, all the memory space is detectable and available at initialization stage in the hypervisor. Differently, AMF hides PM space for efficiency.

Disaggregated Memory. Disaggregated memory [3] is a perfect solution to enable memory capacity expansion match the computational scaling. This work differs from our work in the following aspects. First, their work aims to resolve the memory capacity wall problem, while our work targets to intelligently scale large-capacity memory. Second, from the perspective of hardware architecture, their work provides extra memory capacity by provisioning an additional separate physical memory blade. In contrast, we build a new architecture to facilitate the compatible integration of PM on a NUMA-based machine. Third, their solution requires a virtualization layer to provide page-level access to a memory blade, while our solution purely relies on kernel revision to gain DRAM-alike access.

Memory Hotplug. Memory hotplug [41] technology allows users to increase/decrease memory capacity. This technology consists of physical memory hotplug phase and logical memory hotplug phase. The physical memory in DIMM must first be initialized. Afterwards, the initialized memory is turned from offline to online (this logical phase called Memory Online). The differences between memory hotplug and our work can be summarized as follows. First, memory hotplug has wider application domain than AMF. It is because memory hotplug is a physical memory reconfiguration mechanism, which can handle the hardware errors, balance the workload, support memory extension, etc. Differently, AMF mainly provides an adaptive integrating mechanism for the sake of decreasing memory pressure 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 change dynamically. However, AMF adds the detected PM space gradually and automatically makes them available for applications (allocation and reclamation are feasible). The total physical PM space is fixed in advance. Third, memory hotplug requires updating the SRAT table (a table in ACPI which includes memory info.) at its running stage. In contrast, AMF needn't to update the table. Finally, memory hotplug needs to modify the whole memory subsystem, and our work only adds a kernel module in local OS with minute revision.

9. CONCLUSION

In this paper, we propose an architecture which allows the OS to manage DRAM and PM space in a combined manner. Unlike traditional main memory subsystems, which initialize all the physical memory at the boot stage, our adaptive memory fusion (AMF) mechanism only initializes partial PM space to avoid instant expansion of kernel metadata. Our agile system management strategy guarantees low performance overhead. We have implemented a kernel module on a commercial server based on Linux-4.5.0. Extensive evaluation shows that AMF allows memory-intensive workloads to provide much better performance than conventional design.

ACKNOWLEDGMENTS

The authors would like to thank all the anonymous reviewers for providing valuable feedbacks. This work is supported by the National High Technology Research and Development Program of China (No.2015AA015303) and the Natural Science Foundation of China (No. 61472241, No. 61502302, No. 61628208). Chao Li is also funded by a CCF-Intel Young Faculty Research Program Award. Corresponding authors are Chao Li and Linpeng Huang at Shanghai Jiao Tong University.

10. REFERENCES

[1] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAPHANAdatabase: data management for modern business applications," *ACM SIGMOD Record*, vol. 40, issue. 4, 2012

[2] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, issue 3, 2015

[3] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 267–278, ACM, 2009

[4] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *the 35th International Symposium on Computer Architecture (ISCA)*, 2008

[5] SNIA Researcher. "Bringing Persistent Memory Technology to SAP HANA: Opportunities and challenges". https://www.snia.org /sites/defaulat/files/PM-summit/2017 /presentations

[6] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing compute and memory power in high-performance gpus," in *the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2015

[7] H. Huang, K. G. Shin, C. Lefurgy, K. Rajamani, T. Keller, E. V.Hensbergen, and F. Rawson, "Cooperative software-hardware power management for main memory," in *Workshop on Power-Aware Computer Systems (HotPower)*, 2004

[8] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *the 49th*

Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016

[9] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory.," in *HotOS-XIII*, 2011

[10] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016

[11] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015

[12] S. Chhabra and Y. Solihin, "i-nvmm: a secure non-volatile main memory system with incremental encryption," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011

[13] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 91–104, 2011

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011

[15] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, p. 3, 2015

[16] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014

[17] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016

[18] J. Xu and S. Swanson, "Nova: a log-structured file system for hybrid volatile/non-volatile main memories," in *the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016

[19] Dave, "Dax and fsync: the cost of forgoing page structures." https://lwn.net/Articles/676737/

[20] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011

[21] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015

[22] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: exploiting nvram in write-ahead logging," in *the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 385–398, 2016

[23] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pp. 133–146, ACM, 2009.

[24] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016

[25] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017

[26] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *the 32nd IEEE International Conference on Computer Design (ICCD)*, IEEE, 2014.

[27] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[28] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in the 48th *International Symposium on Microarchitecture (MICRO)*, 2015.

[29] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini, "Practical and secure PCM systems by online detection of malicious write streams," in the *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[30] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *the 42nd Annual IEEE/ACM International Symposium on Micro Architecture (MICRO)*, 2009

[31] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *the International Symposium on Computer Architecture (ISCA)*, 2009

[32] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and ram main memory system," in *the 46th ACM/IEEE Design Automation Conference (DAC)*, pp. 664–669, IEEE, 2009

[33] H. Muhammad, "htop - an interactive process viewer for unix." https://hisham.hm/htop/

[34] M. Researcher, "Calculating memory system power for ddr3 introduction." /https://www.micron.com/support/toolsandutilities/power-calc.2017/

[35] J. Mccalpin, "Stream: Sustainable memory bandwidth in high performance computers." https://www.cs.virginia.edu/stream/

[36] Redis, https://redis.io/.

[37] Trends in Storage for 2017, https://www.servethehome.com/trends-in-storage-for-2017/

[38] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture (ISCA)*, 2009

[39] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010

[40] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002

[41] D. Hansen, M. Kravetz, B. Christiansen, and M. Tolentino, "Hotplug memory and the linux vm," in *Linux Symposium*, Citeseer, 2004

[42] N. Agarwal and T. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017

[43] C. Researcher, "Couchbase." https://www.couchbase.com/

[44] SNIA Researcher, Dr. Rajiv Y. Ranjan Co-founder & CTO "STT-MRAM: Emerging NVM ", https://www.snia.org/, 2016

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

[46] Jung, Ju-Yong and Cho, Sangyeun. "Memorage: Emerging persistent RAM based malleable main memory and storage architecture*". in the 27th international ACM conference on International conference on supercomputing,* 2013

[47] Y. Jin, M. Shihab and M. Jung. "Area, Power, and Latency Considerations of STT-MRAM to Substitute for Main Memory", *The Memory Forum,* 2014