Kindle: A Comprehensive Framework for Exploring OS-Architecture Interplay in Hybrid Memory Systems

Arun KP
Computer Science and Engineering
Indian Institute of Technology Kanpur
kparun@cse.iitk.ac.in

Debadatta Mishra
Computer Science and Engineering
Indian Institute of Technology Kanpur
deba@cse.iitk.ac.in

***Abstract*—Computers with hybrid memory can offer the benefits of both DRAM and NVM technologies, where NVM provides higher capacity and data persistence, while DRAM offers better performance. One of the primary objectives in hybrid memory systems is to design mechanisms and policies to take advantage of the underlying memory technologies to improve overall system performance and explore new usage scenarios.**

This paper introduces an open-source framework, Kindle, based on gem5 and gemOS, to explore and prototype research ideas in hybrid memory systems crossing the hardware-software boundaries, and perform comprehensive empirical evaluation. Kindle provides mechanisms to realize process persistence in hybrid memory systems while facilitating analysis of different design challenges and alternatives. As an illustration, we compare two design choices to maintain the virtual to physical address translation information (in the page table) for achieving process persistence in Kindle. Moreover, we demonstrate the utility of Kindle by prototyping state-of-the-art research ideas for hybrid memory systems to show that new insights can be derived using the facilities provided in the proposed framework.

***Index Terms*—Non-volatile memory, NVM, hybrid memory, process persistence**

I. Introduction

Non-volatile memory (NVM) provides high memory capacity with reduced energy cost compared to volatile memory (DRAM). However, higher read/write latency of NVM [21] raises performance concerns when used as a drop-in replacement for DRAM. Therefore, a more attractive memory organization is a hybrid memory system with DRAM and NVM [23], designed to complement each other to reduce energy cost, achieve persistence, and provide improved performance. A hybrid memory system allows placing data in NVM and/or DRAM, enabling OS memory managers to coordinate allocations for high memory capacity with low access latency. For example, a typical OS policy can place frequently accessed hot memory pages in DRAM and migrate cold pages to NVM to increase the overall system performance. Several existing works on hybrid memory systems propose efficient access for large workloads by intelligently placing data across the NVM and DRAM [24], [42]–[44], reducing energy consumption by migrating data across the DRAM and NVM tiers [22], [32], [34], [48]. Another usage of NVM is as an alternative to external storage hardware (HDD or SSD) for data persistence. In this usage scenario, application developers or file system designers need to incorporate consistency and durability semantics into their design. Existing research works attempt to address the consistency issues by designing solutions such as persistent object stores [15], lock-based failure atomicity for multi-threaded programs [7], [14], [47], and hardware and/or software memory consistency mechanisms [2], [8], [11], [18], [20], [31].

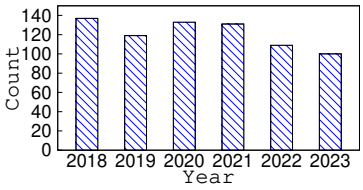


Fig. 1: Number of publications on hybrid memory systems with NVM, listed in Google Scholar.

Hybrid memory provides extensive opportunities in system design, Figure 1 shows the number of publications on hybrid memory systems with NVM for the last six years based on data extracted from Google Scholar [1]. The number of publications shows an average of 120 research papers annually, demonstrating the wide range of research opportunities in hybrid memory systems. The nature of problems and proposed solutions for hybrid memory systems require an infrastructure allowing *extension, validation, and evaluation* of the complete system stack.

Architecture simulators capable of full-system simulation, such as gem5 [25], provide platforms for prototyping ideas crossing the hardware-software boundaries. However, using gem5-Linux full-system simulation setup to explore new ideas in hybrid memory systems has the following shortcomings. *First*, while gem5 models the NVM controller and the Linux kernel can detect NVM on real hardware (such as Intel DCPMM) [17], [45], their integration is non-trivial (in gem5), especially considering constantly evolving hardware and OS design. *Second*, Linux kernel is heavy with features whereby the OS functions and services can consume a significant part of a simulation which may not be desirable for quick prototyping

of design ideas. *Third*, designing proof of concepts (PoCs) in Linux requires significant understanding and changes in the Linux memory management subsystem, which has non-trivial complexity.

In this paper, we propose *Kindle*, a comprehensive infrastructure for quick prototyping and evaluation of novel mechanisms and policies *crossing architecture and operating system boundaries* in hybrid memory systems. In the core of *Kindle*, support for full process persistence is provided, whereby a process can be restarted after a crash in a consistent manner. There are several design alternatives, challenges, and performance insights in achieving process persistence in hybrid/persistent memory systems. As one specific illustration, we compare two design choices for maintaining the page table in a consistent manner across system restarts—*(i)* hosting the page tables directly on NVM, *(ii)* hosting the page tables in DRAM while performing periodic checkpoints into NVM. Next, we present a complete evaluation framework for hybrid memory systems by combining the process persistence support with other tracing and simulation techniques where end-to-end modelling of real-world application benchmarks can be performed in a generic manner (Section II). Finally, we showcase the utility of *Kindle* in gaining new insights by using prototype implementations of two well-established research ideas proposing optimizations for hybrid memory systems in OS and/or hardware layers (Section III).

We design and implement *Kindle* support for full process persistence by modifying a lightweight OS (i.e., *gemOS*). We modify the gemOS system call APIs to allow user applications to allocate memory in DRAM and/or NVM using memory allocation API, enabling exploration of memory usage and access behavior of standard applications on hybrid memory systems. At a high level, *Kindle* consists of two components—*(i)* a preparation component for transforming and extracting required information from the application (and its interaction with the OS) to prepare the stage for the simulation run, and, *(ii)* a simulation component for running the applications in full persistence mode with the configurations provided by the user.

Kindle aids in quickly realizing complex design goals and providing new insights into existing schemes, as we show through the prototype implementations of two state-of-the-art hardware-software hybrid memory schemes—Shadow Sub-Paging (SSP) [31] and Hardware/Software Cooperative Caching (HSCC) [23]. SSP handles memory consistency requirement of NVM by using shadow sub-paging. HSCC provides memory capacity by arranging DRAM and NVM in a flat address space and managing DRAM as a cache to NVM using a hardware/software cooperative caching mechanism. These prototype implementations show the capability of *Kindle* to quickly validate ideas in hybrid memory systems and gather new insights. Through the SSP prototype, Kindle provides new insights into the impact of consistency interval on the performance overhead of SSP, showing that a wider consistency interval reduces the overhead. HSCC prototype with Kindle provides insights into the migration overhead due to the OS activities, which was not shown in the original work as the evaluation framework did not have an OS component.

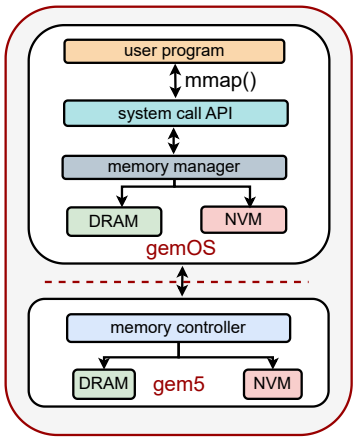


Fig. 2: Interaction between gem5 and gemOS in Kindle

TABLE I: gem5 Memory Configuration

Parameter	Used Setting
DRAM interface	DDR4-2400 16x4
NVM interface	PCM ‡
NVM Write buffer size	48
NVM Read buffer size	64
Memory capacity	3GB DRAM + 2GB NVM
‡PCM timing parameters based on [39]	

The main contributions of this paper are:

- An efficient and lightweight simulation framework with full-system simulation capability for hybrid memory systems.
- Propose an end-to-end design to achieve process persistence in a hybrid memory system and compare design approaches for page-table maintenance.
- Prototype implementation of two state-of-the-art proposals followed by an experimental evaluation to establish new insights and conclusions.

II. Design and Implementation

Kindle provides a hybrid memory system, allowing applications to allocate memory from NVM and DRAM. Figure 2 shows the interactions between gem5 and gemOS in *Kindle* to provision memory based on the application requirements. *Kindle* partitions the physical memory address range between NVM and DRAM, and inserts corresponding entries in the gem5 BIOS implementation of `e280` (BIOS memory map). We configure the NVM memory controller interface in gem5 with specifications mentioned in Table I. *Kindle* provides a hybrid memory system in flat address mode, allowing the OS to expose DRAM and NVM to applications.

The modified gemOS for *Kindle* exposes DRAM and NVM to user-space applications through an extended `mmap` system call API. An application differentiates memory requests for NVM from DRAM by passing an additional flag `MAP_NVM` in `mmap()` system call as shown in the sample code Listing 1.

```c
int main(){
    char* ptr1= (char*)mmap(NULL,4096,PROT_WRITE,MAP_NVM);
      //allocation in NVM
    char* ptr2= (char*)mmap(NULL,4096,PROT_WRITE,0); //
      allocation in DRAM
    ptr1[0] = 'A'; //store to NVM
    ptr2[0] = 'B'; //store to DRAM
    //munmap allocations
    return 0;
}
```

Listing 1: Sample mmap() code to allocate in NVM

Additionally, the virtual memory area (VMA) layout in vanilla gemOS is modified to tag each VMA to be classified as either DRAM or NVM depending on the value of `MAP_NVM` flag passed in the `mmap()` invocations from the user space. The physical memory allocation to the page frames of NVM or DRAM is performed based on this memory type tag.

A. Process Persistence

Process persistence requires saving the state of a process that consists of CPU registers and memory state encompassing the code, heap and stack areas. Process persistence also requires saving the OS meta-data (e.g., address space layout, process relations etc.) to resume execution from a consistent state after a system crash.

We implemented a consistency scheme based on periodic checkpointing of process contexts in the modified gemOS. We maintain per-process *saved state* in NVM, containing two copies of the execution context—one as a consistent copy and another as a working copy. We use redo log (stored in NVM) to capture all modifications to the OS-level process meta-data. As part of the saved state, we also maintain a list of virtual page to NVM physical page frame mappings. At the end of each checkpoint interval, we first log the CPU state and update the consistent copy using all logged entries in that interval. We consider only the consistency of the CPU state and OS-level meta-data, and assume that all heap/stack data pages are consistently maintained in NVM using some existing memory consistency techniques [3], [6], [8], [18], [20], [46]. The list containing the virtual to NVM physical page mapping is maintained to preserve the association from any virtual to physical page, as it is useful for scenarios where we need to rebuild the virtual memory mapping of a process in page table after reboot. We also modify the physical page allocation mechanism in gemOS to persist the page allocation meta-data to ensure correctness after crash and reboot scenarios.

At the end of a checkpoint interval, the *saved state* of a context is updated to reflect changes in the interval, which includes modifying the virtual to NVM physical page mapping list by traversing the page table of the process, getting the working copy of context and applying changes in the redo log. After applying all changes in the redo log, mark the updated context as the latest consistent copy of the context in the *saved state*. The resume procedure reconstructs the execution context using the latest consistent copy of context in the saved state after a crash and reboot. The recovery procedure scans through the list of saved states and creates a new execution context for each saved state. For each process, we copy the latest consistent copy of the context and recreate the virtual memory layout as part of the recovery procedure. Finally, the recovery process sets up the page table mapping for the virtual address space and marks the process state as ready for execution.

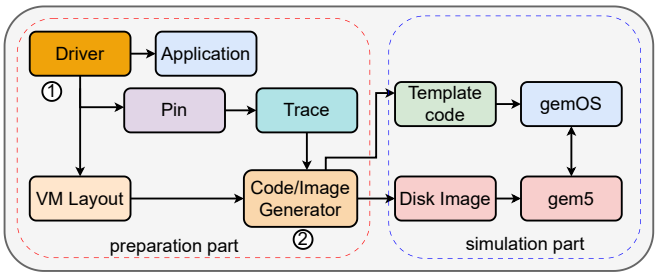


Fig. 3: Schematic diagram of Kindle

B. Kindle Framework

Figure 3 shows the complete *Kindle* framework and interaction between its different components. *Kindle* consists of two major components—a simulation sub-system and a preparation sub-system. The simulation part holds cycle-accurate architecture simulator gem5 [5], [25] and the extended lightweight operating system gemOS [27], specialized for running on gem5.

The preparation component creates the disk image for gem5 and the benchmark template code for gemOS. The vanilla gemOS can not run standard workloads as it is primarily designed for OS education and has limited user-space libraries. The preparation component overcomes this limitation by setting up an environment for tracing and replaying the memory operations on gemOS for running standard applications. The preparation sub-system consists of a driver program (①) to trace the instructions executed by the application of interest using Intel's dynamic binary instrumentation tool Pin [26]. The driver program (using fork and exec) coordinates an application's execution and memory access tracing with Pin while saving the virtual memory layout by reading the `/proc/pid/maps` pseudo file exposed by the Linux kernel. In case of multi-threaded applications, *Kindle* can use SniP [19] along with the *maps* file to capture address layout of application. SniP is a framework capable of capturing the stack area of threads. The trace generated by Pin contains the type of memory access (read/write), address and size of memory access, and time of access.

The code/image generator component (②) makes the disk image required by gem5 in the simulation part of *Kindle*. It processes the trace file to generate a tuple containing *(period, offset, operation, size, area)* for each memory access. The *period* shows the time of memory access, *offset* shows the address of memory access within a heap/stack area, *operation* indicates the type of memory access, whether it is a read/write and *size* denotes size of memory read/write and *area* denotes name of the memory area, i.e., which heap and stack area is

TABLE II: Benchmark Details

Benchmark	Total Ops	read %	write %
Gapbs_pr [4]	10,000,000	77	23
G500_sssp [29]	10,000,000	68	32
Ycsb_mem [10]	10,000,000	71	29

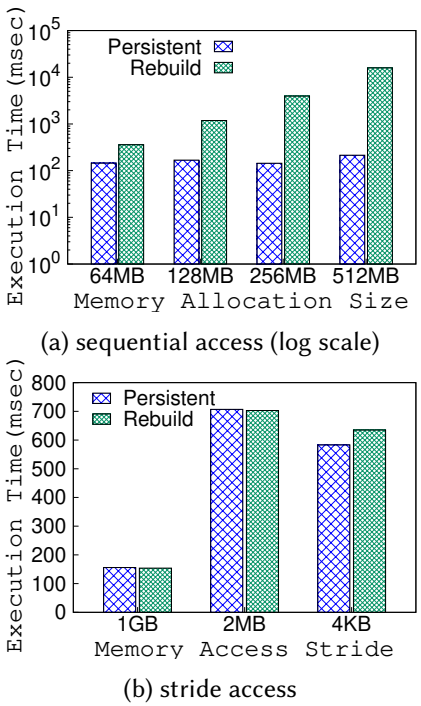
read/written. The *image generator* labels each memory area in the virtual memory layout information captured using *maps* pseudo file and then associates memory accesses in trace to an *area* name by checking whether access lies within the address range of that area. The prepared disk image contains *(period, offset, operation, size, area)* information of all memory accesses in the trace. The *code generator* prepares a template gemOS code containing heap and stack allocations matching the number and size of allocations in the application. The generated code also contains routines to access *(period, offset, operation, size, area)* tuple from the disk image for mimicking the memory access in the application. Users of the Kindle framework can update this template code to include additional functionality before launching the `init` process (the first user process in gemOS) with required arguments [27].

III. Empirical Evaluation

In this section, we show a consistency scheme based on checkpointing for execution context to achieve process persistence. We study trade-offs in checkpoint performance while using two different approaches to keep the page table consistent. We also demonstrate the capability of *Kindle* in performing an initial evaluation of existing or new ideas on a hybrid memory system. To demonstrate the utility of *Kindle* in hybrid memory research, we implemented two research ideas—*(i)* shadow sub-paging (SSP) [31] to ensure consistent memory state of an application in NVM by employing shadow paging [9], [31] at sub-page cache line granularity, *(ii)* a hardware-software co-operative caching mechanism, HSCC [23], for managing DRAM as cache to NVM. In these two studies, we used standard applications in Table II and configured gem5 with Intel 64-bit in-order CPU at 3GHz with 32KB L1, 512KB L2 and 2MB/core LLC.

A. Process Persistence

As the end-to-end performance of checkpointing the execution context depends upon virtual address space management, we studied the performance trade-offs of maintaining the page table using two approaches—*(i)* rebuild the page table on reboot from virtual to NVM page mapping maintained in the saved state (*rebuild scheme*). *(ii)* maintain the complete page table in NVM and wrap page table modifications inside NVM consistency mechanism [2], this only requires setting the PTBR (Page Table Base Register) to point to the first level of page table after a reboot (*persistent scheme*). While the *rebuild* scheme allows hosting page table in DRAM it may suffer from checkpoint overheads due to additional maintenance of virtual to physical mapping information. On the other hand, while the *persistent* scheme simplifies the checkpoint process, it adds additional overheads to host and maintain the page table in NVM in a consistent manner.



(a) sequential access (log scale)

(b) stride access

Fig. 4: Influence of memory access size and stride length on execution time with periodic checkpointing for context while using different page table consistency schemes.

We looked into the impact of *address space size* and *page table size* on the end-to-end performance of periodic checkpointing of execution context while using *rebuild* and *persistent* schemes for page table consistency. Figure 4 shows the end-to-end execution time (in msec) for consistently maintaining context using periodic checkpointing under *rebuild* and *persistent* page table maintenance schemes. The periodic checkpoint interval is fixed to 10 msec (based on Aurora [40]).

In the sequential memory allocation and access experiment (Figure 4a), using a micro-benchmark we allocate virtual memory of different sizes using *mmap* system call with `MAP_NVM` flag and sequentially access all pages in the allocated space. In the stride access experiment (Figure 4b), micro-benchmark performs a fixed number of 4KB page allocations with a predefined gap (1GB, 2MB or 4KB) in the virtual address space to ensure different page table levels are mapped to result in larger page table size. For example, with 1GB gap, the micro-benchmark allocates ten 4KB pages at a gap of 1GB to make entries at page-directory-pointer-table (Level-3), page-directory-table (Level-2), and page-table (Level-1) in Intel x86-64 system page table [13].

In the sequential access experiment (Figure 4a), *rebuild* scheme resulted in higher execution time for all allocation sizes with overhead ranging from ∼2.4× (64MB) to ∼74.2× (512MB) compared to *persistent* scheme. The overhead in

TABLE III: Execution time with periodic checkpointing of execution context for different VMA modification size

Alloc/Free Size	Persistent (msec)	Rebuild (msec)
64MB	325	19377
128MB	389	23438
256MB	517	29376

rebuild scheme comes from the need to maintain a list containing virtual to NVM physical page mapping for re-constructing the page table after reboot, and the overhead to maintain this list increases with increase in mapped virtual memory area size, $\sim$44$\times$ increase in execution time from 64MB allocation size to 512MB. On the other hand, for stride access experiment (Figure 4b), *persistent* scheme resulted in slightly more execution time compared to *rebuild* for 1GB and 2MB access stride scenarios, as more page table levels are updated for 1GB and 2MB. For the 4KB case, page table modifications are minimal, and *persistent* scheme performed better than the *rebuild* scheme. The overhead in *rebuild* scheme can be attributed to maintaining virtual to NVM physical page mapping, similar to memory allocation size experiment. In short, *persistent* scheme results in more overhead if the virtual address space is sparsely populated. However, the *presistent* scheme performs better than *rebuild* for scenarios with minimum page table modifications as shown in the memory allocation size experiment where page table is updated only on the first access.

Next, we looked into a *scenario with more page table updates* using a sequence of *mmap* and *munmap* operations. The micro-benchmark allocates a virtual address space of 512MB and writes to all pages in 512MB to create valid page table entries. The benchmark then frees a fixed size memory (i.e., 256MB, 128MB, and 64MB) from the start of 512MB space by calling *munmap* and reallocated the same fixed size memory by calling *mmap*. Similarly, *munmap* and *mmap* of the same fixed size are performed on 512MB space for one more time. The newly allocated fixed size regions are then accessed for reading, and finally, the entire area is unallocated by called *munmap*. Table III shows end-to-end execution time with execution context checkpointing and maintaining page table using *persistent* or *rebuild* method. The table shows execution time while performing *munmap* and *mmap* sequences of different allocation/free fixed sizes. The *persistent* scheme overhead increases with an increase in allocation/free size as more page table changes are required with increased *munmap* and *mmap* size, showing $\sim$1.6$\times$ increase in execution time from 64MB to 256MB. The execution overhead increases for *rebuild* scheme as well since the virtual to NVM physical page mapping maintained for page table rebuilding requires constant change due to changes in virtual memory area, showing $\sim$1.5$\times$ increase in execution time from 64MB to 256MB.

Finally, we looked into the scenario which shows the benefit of keeping the page table in DRAM. Page table is accessed in two cases, to update mapping entries as a result of virtual memory area changes due to *mmap*, *munmap*, *mremap*, *mprotect* calls or to translate virtual address by page table walker hardware after missing in TLBs and other intermediate caches. Keeping the page table in DRAM benefits with an increase in page table accesses as DRAM provides better read and write latency than NVM. We used a micro-benchmark similar to the previous case to study the benefit of keeping page table in DRAM. Micro-benchmark allocates a virtual address space of 512MB and accesses pages to make page table entries and then frees a fixed size memory (i.e., 256MB, 128MB, and 64MB) from the start of 512MB space by calling *munmap* and reallocated the same fixed size memory by calling *mmap*. After allocation, all pages in the virtual memory area are accessed multiple times to cause TLB misses. The benchmark does one more round of deallocation and allocation of the same fixed size and multiple rounds of accesses to allocated space. Finally, the entire area is unallocated by called *munmap*.

TABLE IV: Influence of checkpoint interval on execution time for periodic checkpointing of execution context

Alloc/Free Size	Interval	Persistent (msec)	Rebuild (msec)
64MB	10 msec	445	55270
	100 msec	445	10580
	1 sec	444	430
128MB	10 msec	534	68078
	100 msec	532	13103
	1 sec	532	515
256MB	10 msec	710	88193
	100 msec	708	15775
	1 sec	708	685

Table IV shows the end-to-end execution time to checkpoint process context with different checkpoint intervals while using *persistent* and *rebuild* schemes for page table maintenance. The execution time for *persistent* scheme remains similar across different checkpoint intervals for a virtual memory area size of alloc/free operations. This is because the overhead of *persistent* scheme comes from the NVM consistency mechanism and the number of page table modifications remains the same for a virtual memory area modification size irrespective of checkpoint intervals. Whereas, the execution time for *rebuild* scheme increases with the frequency of checkpoints. This is because of the overhead to check and update virtual to physical address mapping during each checkpoint. Table IV shows that increasing the checkpoint interval from 10 to 100 milliseconds reduces execution time by $\sim$5$\times$ on average for the *rebuild* scheme across all virtual memory area size of alloc/free operations. When we further increase the interval to one second, beyond the execution time of benchmark, the benefit of keeping the page table in DRAM appears for the *rebuild* scheme as the execution time is lower than *persistent* scheme, highlighting the reason for performance overhead in *rebuild* scheme as maintenance of virtual to physical address mapping. In summary, keeping the page table in NVM with *persistent* scheme is beneficial for applications with minimal

virtual address modifications. Access to page table entries for address translation gets the benefit of multiple levels of TLBs and intermediate caches, thus hiding NVM read latency while accessing page table entries for address translation in *persistent* scheme.

In the next two sections, we show the prototype implementation of two research ideas to demonstrate the benefit of using Kindle in exploring ideas and providing the initial results below.

B. SSP using Kindle

Shadow Sub-Paging (SSP) [31] provides memory consistency in NVM. It ensures consistency of memory modifications by maintaining a copy of unmodified data at cache line granularity. SSP allocates two physical pages for each virtual page and uses a remapping scheme at the cache controller hardware to route modifications at cache line granularity to alternate physical pages. SSP also extends the TLB by adding extra fields per entry to capture the supplementary physical page mapping and bitmaps (updated, current) to track the page containing the latest modification. SSP proposes a background OS thread to consolidate two physical pages but leaves out the detailed implementation and evaluation of the *the consolidation* aspects in the paper.

In *Kindle*, we allocate the additional physical page in the page allocation routine call in gemOS. The original and the extra page addresses and the bitmap values (commit, current) are recorded in a metadata area (i.e., SSP cache). We extend the page table walker hardware in gem5 to fill fields in the TLB during an address translation on TLB miss. TLB may contain translations for DRAM and NVM pages in a hybrid memory system, and the memory consistency requirement applies only to NVM pages. Therefore, in the prototype design, we use Model Specific Registers (MSRs) to communicate the virtual address range corresponding to NVM allocation to hardware. We also use MSR to pass the base address of SSP cache to translation hardware in gem5. The address translation hardware checks the address range and sets the corresponding bit in the *updated* bitmap in TLB if a write happens to the NVM address range. The translation hardware generates a memory request to modify metadata in SSP cache when a consistency interval ends, or a TLB entry eviction happens. We mark the entry as *TLB evicted* in the SSP cache.

We use a programming model in which the user demarcates the failure atomic section (FASE) in code using `checkpoint_start` and `checkpoint_end` calls. `checkpoint_start` enables custom hardware components in the address translation and cache controller hardware in gem5. A consistency interval of choice is set in gemOS. For example, setting consistency intervals as 5 msec ensures that at every 5 msec interval ends, and activities associated with `checkpoint_end` are performed, i.e., gemOS kernel instructs the address translation hardware to initiate a memory request to send all modified bitmap in TLBs to the metadata region. The gemOS kernel then calls `clwb` write back instructions to flush all data and metadata updates in hardware caches to NVM. Physical page consolidation happens asynchronously; a thread periodically calls page consolidation routine to merge pages corresponding to evicted TLB entries by inspecting the SSP cache entries in gemOS.

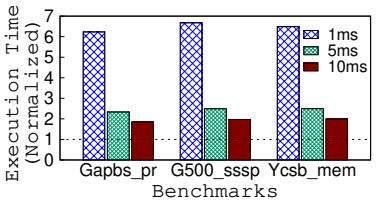


Fig. 5: Influence of memory consistency interval on performance. Y-axis shows normalized execution time with no memory consistency.

Figure 5 shows the overhead introduced by SSP in making the memory state of applications consistent. This study used consistency intervals of 1, 5, and 10 msec. The page consolidation thread interval is fixed to 1 msec as a lower interval would result in higher consolidation overhead. Figure 5 shows the execution time of applications normalized to the execution time with no memory consistency applied. Having a wider consistency interval (10 msec) reduces the consistency overhead as the number of metadata inspections and `clwb` calls to writeback cache lines reduce with a wider consistency interval. All applications in Figure 5 show an average $\sim 3\times$ reduction in memory consistency overhead with 10 ms compared to a 1 ms consistency interval.

Kindle provides an easy way to extend studies such as the influence of consistency interval on the application performance, and it also allows carrying out additional studies on the influence of page consolidation thread invocation frequency on an application by varying the thread time interval, which is not explored in original SSP proposal.

C. HSCC using Kindle

Hardware/Software Cooperative Caching (HSCC) [23] aims to utilize high NVM capacity in a hybrid memory system for improved system performance. HSCC maintains NVM and DRAM in a flat address space and uses DRAM as a cache managed by OS in a hardware/software cooperative manner. HSCC tracks access count of NVM pages to select candidate pages for migration to DRAM and maintains an NVM-to-DRAM page mapping after migration. HSCC extends the page table and TLB for handling NVM to DRAM remapping and tracking the access count of NVM pages. NVM pages with an access count exceeding a specific fetch threshold value in a migration interval are selected for migrating to DRAM.

HSCC extends page table entry (PTE) to record DRAM and NVM page frame numbers, using 96 bits (12 bytes) for PTE as opposed to 64 bits in conventional x86-64 systems. In this case, the last level page table in HSCC can only map 341 pages (i.e., 4KB/12B), leaving 171 pages unmapped in a 2MB address region. In our implementation, we have designed NVM to DRAM mapping in a lookup table to avoid the previously

TABLE V: Number of Pages Migrated

Benchmark	Th-5	Th-25	Th-50
Gapbs_pr	354	273	132
G500_sssp	4489	1475	1346
Ycsb_mem	23093	1661	221

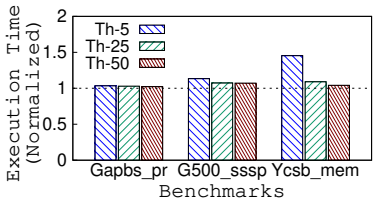


Fig. 6: Influence of OS migration activities on application performance under DRAM fetch threshold 5, 25, and 50.

mentioned PTE size issue. The mapping table entries can be looked up using both DRAM and NVM page frame numbers as an offset. We also maintain a pool of DRAM pages (512 pages), categorized as lists of free, clean, and dirty pages, updated at the start of each migration interval of 31.25 msec (equivalent to 10^8 cycles mentioned in the HSCC paper). The migration activity inspects the *page access count* maintained in PTEs corresponding to NVM pages and migrates the pages to DRAM cache if the count exceeds the fetch threshold. The page access count is also maintained in TLB and is incremented if the data access misses in the LLC. The access count in TLB is written out to PTE on TLB eviction or once during the translation in a migration interval. We have not incorporated dynamic fetch threshold adjustment in our implementation and have fixed the threshold to static values. The page access count in PTEs is reset in each migration interval to ensure that NVM pages from the most recent interval are considered for migration.

We investigated the migration-related processing overheads in the OS-mode and its impact on the execution time of applications. The candidate pages for migration are identified by inspecting the *page access count* maintained in the PTEs (by performing a software page table walk) corresponding to the pages mapped to NVM. Migrating a page to DRAM consists of two steps—*(i) page selection*, selecting the destination DRAM page, and *(ii) page copy*, copying the page from NVM to DRAM. Page selection includes allocating the destination DRAM page from the free pool or from the clean or dirty list of DRAM pages. If any page is selected from the dirty list, then we copy back the page from DRAM to NVM before use. Page copy step includes flushing cache lines corresponding to the NVM page under migration before copying data from NVM to DRAM and then copying data to DRAM. The corresponding PTE entry is updated with DRAM page address, access count in PTE is reset and the corresponding TLB entry is invalidated. The page access count in all PTEs is reset, and corresponding TLB entries are invalidated in a migration activity to ensure that page accesses for the most recent interval are considered for migrations.

Figure 6 shows the overhead of migration activities performed by OS. The figure shows the execution time of applications with migration (i.e., performing hardware and OS migration activities) normalized to the execution time without OS migration activities (i.e., performing only hardware migration activities) under different fetch thresholds; the fetch threshold decides the number of candidate NVM pages for migration. Two important factors influencing the execution time of an application with migration are—the overhead of activities performed by OS as part of the migration and the benefit in memory access time after migrating pages to DRAM. All applications in Figure 6 show migration overhead due to OS activities, and a higher value indicates that the overhead of activities performed by OS as part of the migration overshadows the benefit in memory access time after migrating pages to DRAM. Gapbs_pr shows the minimum overhead, indicating that Gapbs_pr has the maximum benefit in memory access time after migrating pages to DRAM. For all applications, the migration overhead reduces with an increase in the fetch threshold as the number of candidate pages migrated reduces with an increase in the threshold as shown in Table V; hence, the overhead of OS activity reduces. For example, Ycsb_mem showed ∼13× and ∼101× reduction in the number of pages migrated for Th-25 and Th-50 compared to Th-5 respectively.

TABLE VI: Percentage of execution time spent for page selection and page copy in OS migration activity.

Benchmark	Fetch Threshold	Page Selection (%)	Page Copy (%)
Gapbs_pr	Th-5	1.74	98.26
	Th-25	1.92	98.08
	Th-50	2.06	97.94
G500_sssp	Th-5	37.35	62.65
	Th-25	1.37	98.63
	Th-50	1.39	98.61
Ycsb_mem	Th-5	21.71	78.29
	Th-25	19.16	80.84
	Th-50	1.86	98.14

Table VI shows the percentage of execution time spent for activities associated with selecting a destination DRAM page (referred to as Page Selection) and copying the page from NVM to DRAM (referred to as Page Copy) as part of OS migration activity. Two factors contributing to page the selection time are—*(i)* number of pages migrated, and *(ii)* availability of pages in free and clean list of pages. The second component is relevant because, if the page is not available in the free list, selecting a dirty page requires copying back data to NVM before using that page. In case of Gapbs_pr, page selection time is less than ∼2% across all DRAM fetch thresholds as the number of pages migrated for Gapbs_pr (Table V) is lower than total number of pages in the DRAM pool. Thus, majority of requests for pages are satisfied from the free or clean list of pages, requiring no copying from DRAM to NVM before use. G500_sssp with fetch threshold five spends ∼37% of execution time in for page selection, owing to large number of pages migrated with fetch threshold five (Table V), similar is the case with Ycsb_mem with fetch threshold five. Even when relatively less number of pages migrated, page

selection can consume significant portion of execution time in OS migration activities due to lack of free or clean pages. For example, Ycsb_mem with fetch threshold 25 takes ∼19% of time in page selection even with 1475 pages migrated (refer Table V). Across all benchmarks and fetch thresholds, page copy occupies a significant portion of execution time with contribution varying from 98.63% to 62.65% of execution time in OS migration activities.

HSCC in original work used ZSim [37], a user-level simulator, and Zsim does not support OS-level simulation [23]; hence, it can not account for the overhead of OS migration activities, such as copying pages from NVM to DRAM. As *Kindle* provides a full-system simulation, it allows studying the overhead of OS activities on page migration, the influence of other OS activities such as context switches, and the effect of cache pollution due to OS activities on migration. on the contrary, user-level simulators like ZSim miss out on such insights about OS interactions in hybrid memory systems. Kindle also allows separately investigating performance overhead due to hardware and OS activities, as shown in Figure 6 for OS migration activities.

Kindle allows researchers to carry out quick evaluation of ideas crossing hardware-software layers on hybrid memory systems, as we show in the two prototype implementations.

IV. Related Work

The ever-increasing demand for data processing and the capacity scaling limitation of DRAM [30] has motivated new byte addressable Non-Volatile Memory technology. NVM allows capacity expansion and data persistence at memory access latency.

Hybrid memory for capacity: When used for capacity, NVM complements DRAM in a hybrid memory setup to provide better read-write latency along with capacity as NVM demonstrates higher read-write latency than DRAM [21], [35]. A common strategy to attain memory performance is maintaining frequently accessed memory pages in DRAM and others in NVM [36], [49]. For calculating page access frequency, hardware scheme using the memory controller to monitor access patterns and categorize pages as hot and cold for migration [36]. For using DRAM as a cache for NVM, DRAM and NVM are arranged in a flat address and mapped the physical address from NVM to DRAM through page table and TLB extension [23].

Hybrid memory for data persistence: Using NVM for data persistence addresses challenges different from using it for capacity. The challenges appear due to the volatile nature of caches and the order of writebacks from caches [2]. These challenges necessitate a framework such as memory persistency [33] that allows application developers to reason about the order of writes to NVM and mechanisms to ensure consistency of memory updates [2], either by enclosing updates inside an atomic failure section (FASE) [3], [8], [28], by using specialized memory allocation routines [38], [50], [51] or through ISA and architecture extensions [50]. In FASE schemes, logging is a common approach for consistency, undo, redo logging [41], or justdo logging [16] provides required FASE guarantees by logging either at the hardware level [8] or at software [7].

A hybrid memory framework like *Kindle* allows researchers to explore hybrid memory capacity and data persistence usage by quick prototyping of state-of-the-art works.

V. Discussion

A. Validation of Kindle

We have validated the process persistence feature of *Kindle* by crashing and restarting the application multiple times. Validating the working of *Kindle* can be associated with the fidelity of individual components such as Intel Pin and gem5. Additionally, *Kindle* does not impact the simulation time of gem5.

B. Usage of gemOS for Kindle

We use gemOS as the operating system component of *Kindle* since the primary aim is to provide a framework for quick prototyping and gemOS reduces simulation time compared to Linux. GemOS while providing most of the POSIX-compliant APIs, does not include most of the background processes present in production OSes as they interfere with the application under study and hide its actual behavior in the statistics collected. Thus, gemOS benefits in providing cleaner statistics.

Currently, gemOS also has limited support for user space libraries, therefore requiring the preparation part of *Kindle* to trace and simulate standard applications. *Kindle* can help gain fast and accurate comparative insights across design and policy alternatives in this research space.

C. Capabilities and limitations of Kindle

Kindle enables a hybrid memory with NVM and DRAM in a flat addressing mode, allowing users to study the memory behavior of applications with required hardware-software changes. While *Kindle* can provide process persistence, it also has limitations originating from the trace-based approach used in its design to run applications, similar to any other trace-based simulators such as ChampSim [12], as trace file only captures the non-speculative path of application and loses possible thread interleaving in multi-threaded applications, etc. We targeted memory system study using *Kindle* and hence focused on tracing memory operations.

D. Studying NVM memory technologies beyond PCM

We configured the NVM interface in gem5 with PCM configuration (a widely used NVM technology) to showcase the utility of Kindle and the process persistence feature. However, we can use *Kindle* to study other NVM technologies by changing NVM interface parameters in gem5. The scope for such studies increases the value of *Kindle* in hybrid memory research.

E. Availability of Kindle

Kindle is open-sourced and available for download at https://github.com/arunkp1986/Kindle. Users can explore new hardware-software designs in hybrid memory systems by changing the simulation part, implementing software level changes in gemOS and hardware level changes in gem5, and then running applications of interest by following the README documentation.

VI. Conclusion

The hybrid memory system provides the benefit of both DRAM and NVM technology. NVM offers high capacity and data persistence, and DRAM delivers better performance. The existing infrastructure for hybrid memory exploration crossing architecture-OS boundary using simulators such as gem5 is limited by the complexity of integrating NVM support in Linux for gem5 and the simulation overhead of Linux due to OS services and functions running.

In this paper, we introduced an open-source framework, Kindle, based on gem5 and gemOS for hybrid memory exploration in architecture and operating systems. Using Kindle, We studied end-to-end overhead in maintaining execution context using periodic checkpointing for achieving process persistence under two schemes to keep the page table in a consistent manner. Kindle also provides a quick way to study and prototype solutions for hybrid memory systems. We showed prototype implementation of state-of-the-art hardware-software hybrid memory schemes, SSP [31] and HSCC [23], using Kindle to demonstrate its efficacy in realizing complex design goals and analyzing new insights.

Acknowledgment

We thank all the anonymous reviewers for their valuable feedback. We also thank the members of the CDOS research group at the CSE department (IIT Kanpur) for their valuable feedback and support. This research work was partially supported by Research-I Foundation of IIT Kanpur.

References

[1] "Google scholar," https://scholar.google.com/, accessed: 2024-05-30.

[2] K. Arun, D. Mishra, and B. Panda, "Empirical analysis of architectural primitives for nvram consistency," in *Proceedings of HiPC*. IEEE, 2021, pp. 172–181.

[3] A. Baldassin, J. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–37, 2021.

[4] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[6] M. Cai, C. C. Coats, and J. Huang, "Hoop: efficient hardware-assisted out-of-place update for non-volatile memory," in *Proceedings of ISCA*. IEEE, 2020, pp. 584–596.

[7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[8] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of ASPLOS*, 2019, pp. 441–454.

[9] 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 *Proceedings of the Symposium on Operating systems principles*, 2009, pp. 133–146.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[11] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of SPAA*, 2018, pp. 271–282.

[12] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[13] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, pp. 1–64, 2011.

[14] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 468–482.

[15] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, pp. 1–21, 2014.

[16] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[17] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[18] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 520–532.

[19] A. KP, S. Kumar, D. Mishra, and B. Panda, "Snip: an efficient stack tracing framework for multi-threaded programs," in *Proceedings of MSR*, 2022, pp. 408–412.

[20] A. KP, D. Mishra, and B. Panda, "Prosper: Program stack persistence in hybrid memory systems," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1168–1183.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of ISCA*, 2009, pp. 2–13.

[22] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[23] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–10.

[24] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[25] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[27] D. Mishra, "Gemos: Bridging the gap between architecture and operating system in computer system education," in *Proceedings of the Workshop on Computer Architecture Education*, 2019, pp. 1–8.

[28] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of Timely Results in Operating Systems*, 2013, pp. 1–17.

[29] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[30] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IEEE International Memory Workshop*. IEEE, 2013, pp. 21–25.

[31] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of MICRO*, 2019, pp. 836–848.

[32] H. Park, S. Yoo, and S. Lee, "Power management of hybrid dram/pram-based main memory," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 59–64.

[33] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 265–276, 2014.

[34] B. Peng, Y. Dong, J. Yao, F. Wu, and H. Guan, "Flexhm: A practical system for heterogeneous memory with flexible and efficient performance optimizations," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, pp. 1–26, 2022.

[35] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of ISCA*, 2009, pp. 24–33.

[36] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

[37] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[38] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." *Adms@ Vldb*, vol. 15, pp. 61–72, 2015.

[39] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Improving phase change memory performance with data content aware access," in *Proceedings of the Symposium on Memory Management*, 2020, pp. 30–47.

[40] E. Tsalapatis, R. Hancock, T. Barnes, and A. J. Mashtizadeh, "The aurora single level store operating system," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 788–803.

[41] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *Proceedings of NVMSA)*, 2016, pp. 1–6.

[42] B. Wang, J. Tang, R. Zhang, W. Ding, S. Liu, and D. Qi, "Energy-efficient data caching framework for spark in hybrid dram/nvm memory architectures," in *Proceedings of International Conference on High Performance Computing and Communications*. IEEE, 2019, pp. 305–312.

[43] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, "Supporting superpages and lightweight page migration in hybrid memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–26, 2019.

[44] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 163–173.

[45] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons, "An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2019, pp. 1–19.

[46] S. Wu, F. Zhou, X. Gao, H. Jin, and J. Ren, "Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, pp. 1–27, 2019.

[47] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Luján, "Pmthreads: Persistent memory threads harnessing versioned shadow copies," in *Proceedings of PLDI*, 2020, pp. 623–637.

[48] Y. Xie, "Modeling, architecture, and applications for emerging memory technologies," *IEEE design & test of computers*, vol. 28, no. 1, pp. 44–51, 2011.

[49] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 337–344.

[50] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of MICRO*, 2013, pp. 421–432.

[51] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proceedings of OSDI*, 2018, pp. 461–476.

Appendix

A. Abstract

The artifact contains the full source code and other documentation of *Kindle* along with implementation of two design choices to maintain the virtual to physical address translation information (in the page table) for process persistence. Artifact also contains two prototype implementations of state-of-the-art research ideas for hybrid memory systems, SSP and HSCC, to demonstrate the utility of *Kindle*. It includes full-system architecture simulator (gem5), operating system (gemOS) modifications, and scripts to run experiments and generate outputs. The code is suitable to be executed on Linux systems.

B. Artifact check-list (meta-information)

- **Program:** For evaluating *Kindle* framework, bash scripts to run the *preparation* and *simulation* part are provided in `framework` directory. The disk images to use for process persistence, SSP and HSCC evaluation sections are provided in `bench_diskimages` directory at GitHub[1].
- **Compilation:** Scripts for compilation are included, we use GCC version 9.4.0.
- **Binary:** We have provided bash scripts in respective folders to generate gemOS and gem5 binaries on GitHub[1].
- **Run-time environment:** A system with Ubuntu latest version. We provide a Docker image exported with all required dependencies for easy setup at GitHub[1].
- **Hardware:** Intel x86-64.
- **Output:** Python scripts are provided to parse gem5 statistics files and generate output files. Bash scripts are provided to invoke these Python scripts and format output files to generate result plots in respective output folders for easy comparison with expected results.
- **Experiments:** Manual invocation of scripts, which launch corresponding experiments and generate outputs in designated folders.
- **How much disk space required (approximately)?:** 40–50 GB of disk after compilation.
- **How much time is needed to prepare workflow (approximately)?:** 20-30 minutes for gem5 compilation and 1-2 minutes for gemOS compilation.
- **How much time is needed to complete experiments (approximately)?:** Each experiment with gemOS under the "Process Persistence" subsection takes three to four hours. Similarly, Each experiment under "SSP using Kindle" and "HSCC using Kindle" takes three to four hours.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Zenodo[2]

C. Description

1) How to access: All the source code of *Kindle* is available at GitHub[1] and Zenodo[2].

2) Hardware dependencies: Intel x86-64 with at least 16GB RAM.

3) Software dependencies: A Linux system that supports Docker run time. We provide a Docker image export with all required dependencies for easy setup. The link to download Docker export and instructions to use the container are available at GitHub[1].

[1] https://github.com/arunkp1986/Kindle.git
[2] https://doi.org/10.5281/zenodo.13292083

D. Installation

Kindle framework installation consists of building two components—gem5 simulator and operating system (gemOS). The GitHub[1] contains bash scripts to build the gem5 with relevant modifications and compile gemOS to produce gemOS kernels required for running on gem5. The `README` file in GitHub[1] details how to use these scripts to build/run gem5, gemOS, and generate results. We also provide Python scripts to parse and format the output files along with the expected output files for comparison.

We also provide a Docker image export containing dependencies required for building gem5, gemOS. How to set up a Docker container using this export is provided at GitHub[1].

E. Experiment workflow

We provide the source code of our implementation and bash scripts (in GitHub[1]) to build and execute evaluations corresponding to results under the "Process Persistence" subsection (Figure 4, Table-3), results under the "SSP using Kindle" subsection (Figure 5) and results under "HSCC using Kindle" subsection (Figure 6, Table 4). The Workflow involves invoking these scripts to generate outputs.

You can run bash scripts in parallel to reduce the overall execution time of experiments as explained in the `README` file in GitHub[1].

F. Evaluation and expected results

We provide Python scripts to parse results generated by gem5 in respective output folders. We use bash scripts to invoke these Python scripts and format output files generated by Python. We have provided "expected" results under each output folder. Please refer to the `README` file in GitHub[1] for further details.

For evaluating the page table maintenance under "Process Persistence", the expected results are execution time with *rebuild* and *persistent* schemes. The bash script generates plots in the "results" folder under respective output directories.

For evaluating the "SSP using Kindle", the expected result is execution time normalized to execution with no memory consistency (Figure 5) and "HSCC using Kindle", the expected result is execution time normalized to time with no migration (Figure 6).