Thoughts on Merging the File System and Virtual Memory System

Design decisions and their ramifications in developing the Osprey kernel

Bruce Jacob
Cyber Science Department
United States Naval Academy
Annapolis, Maryland, USA
bjacob@usna.edu

ABSTRACT

Near-future computer systems will see the integration of nonvolatile memories into main memory, offering superior density and energy efficiency compared to traditional DRAM. This will prompt a paradigm shift in operating system memory management, driven by the potential merger of the disk and main memory subsystems. This paper discusses possible design directions for a merged memory-management approach in the context of the Osprey kernel, a prototype for exploring the design space. By fusing the file-system and virtual-memory subsystems, Osprey achieves seamless memory mapping, encompassing both application workspace and file-system data, and simplifying kernel code substantially. The design facilitates protection, relocation, memory-mapped files, executable files, and shared memory, with the potential for further enhancements.

1 INTRODUCTION

One of the near-future advances in computer architecture is likely to be the use of nonvolatile memories for main memory, because nonvolatile memories are both denser and, at least in certain aspects, lower-power than DRAM. For instance, one can fit a terabyte into a single nonvolatile package, and the data is retained with zero energy expended, whereas a terabyte of DRAM is a pocket full of DIMMs that costs roughly 10x the nonvolatile storage and needs a constant 10 W to 100 W just to keep its bits alive [4, 5, 2, 8, 9]. Like the transition from SRAM to DRAM, the previous main-memory technology will be available as a cache layer [20], so expect terabyte main memories in the future with last-level DRAM caches in front of them [11, 19].

This warrants a re-thinking of how the operating system manages its physical memory, because the merging of the disk subsystem with the main memory subsystem — the elimination of maintaining separate physical subsystems — would suggest a parallel merging of the kernel's file-system and virtual-memory software subsystems ... i.e., the elimination of what would otherwise be superfluous separate and distinct software entities, both managing the same physical subsystem.

This paper describes several design decisions and their resulting consequences for a merging of memory and file system and lays out some of the advantages gained from the design choices. The proposed design is implemented in a prototype kernel called Osprey, which was built to explore the ideas of recent and future hardware advances, such as merging the different memory systems.

The paper is structured as follows, Section 2 presents related work, Section 3 defines design decisions and how it is implemented in the kernel and Section 4 concludes the paper.

2 RELATED WORK

In the past, persistent memory technologies like PCM [13, 14], STT-MRAM [12], FeRAM [21], and RRAM [1] have been proposed as an alternative for DRAM based memories. Furthermore, hybrid systems, consisting of persistent memory and DRAM have been presented [19, 22, 23]. This is also combined with new memory interface standards like CXL, OpenCAPI, GenZ etc.

Typically, users are presented with the option to either manually control the allocation across diverse memory regions using software or utilize DRAM as a cache for the virtual address range of the persistent memory. Following the initial concept proposals for such hybrid architectures, the actual technology has now achieved commercial availability through products like Intel Optane memory, leveraging the 3D-Xpoint technology. However, in the meanwhile this product has been discontinued.

In order to exploit the advantages of such hybrid memory systems or memory systems that solely consist of persistent memory, some approaches on the software level have already been presented. For example, ecoHMEM [10] performs an offline profiling of the application. The authors of [16] present a kernel-level monitoring module that samples memory patterns and dynamically optimizes the data placement in the memory hierarchy accordingly. A Linux kernel modification that realizes a dynamic page placement is shown in [18, 17].

3 MEMORY MANAGEMENT IN OSPREY

Being permanent, main memory should house both the working space of applications and the long-lived named files and directories; the kernel should need no additional permanent storage beyond main memory, other than backup. This much is clear. However, the quantum leap that merging file system and VM system would provide is the effective memory-mapping of all storage, meaning that a shared namespace enables the access of *all* memory — scratchpad workspace and file-system contents — via load/store operations.

We will show as an example how the Osprey main memory system is in a position to provide the following features/functions:

- **Protection** (just like in existing systems), for instance having RWX values on a per-page level and ACL information at a macro level
- **Relocation** (just like in existing systems), for instance by allowing every process to start executing at address `0x00000000`, and every process's stack starts at address `0xFFFFFFFF`
- **Memory-mapped files** as a default mechanism – for both executables and data files

- **Live executable files**, enabling a simple one-step mechanism to start a process running
- **Shared memory**, either at a page level or a segment level

All other features of modern systems can also be implemented; this is simply a fundamental set, showing the possibilities of a merged system.

3.1 Design Questions & Decisions in Osprey

The following are some of the questions that were asked, and the corresponding answers arrived at, which guided the design decisions in Osprey's memory-management system:

- **How should a file be accessed?** In the traditional operating system, files are effectively character streams, and moving between the memory system and the file system is an act of linearization and de-linearization [3]. This follows from the nature of disk technology. Consequently, if the main memory becomes nonvolatile, and separate disks with their separate namespaces become unnecessary, this linearization requirement disappears. Instead, a more natural means of accessing file data would be by mapping the file directly into the address space and supporting a load/store interface to all file data.
- **How should an executable file be invoked?** In the traditional operating system, executable files are unpacked into main memory before they can be executed. This follows from the nature of disk technology and is similar to the concept of de-linearization mentioned above. In a merged main memory system, given that the storage medium in which the executables are held is "live", meaning that it is part of the main memory system and accessible directly via load/store instructions, a more natural format would be that of raw executable, meaning that if one simply jumps into an executable file (literally jumping, by pointing the program counter to that address), then the code should start running. The issue that needs solving with such an arrangement is how to deal with the BSS segment containing pre-initialized data.
- **How should a file be represented?** In the traditional operating system, a file is a collection of pages, and an inode structure links to those pages. Given the two points above, a file's pages can remain the same as before, but the format of an inode should resemble the kernel's page table or portion thereof, so that a file can seamlessly merge into a process address space. Thus, the kernel's page-table lookup mechanism would be used for both address-space access and file-data access.
- **How should data be shared?** As is done currently, the operating system can always choose to duplicate page-level mappings to share physical pages between address spaces, or even at different locations within the same address space. Given the presupposition of our argument, that files are intended to be incorporated readily into the address space, it makes sense to work out how that can best be accomplished, because file mapping can be thought of as being similar to shared memory, especially when one considers

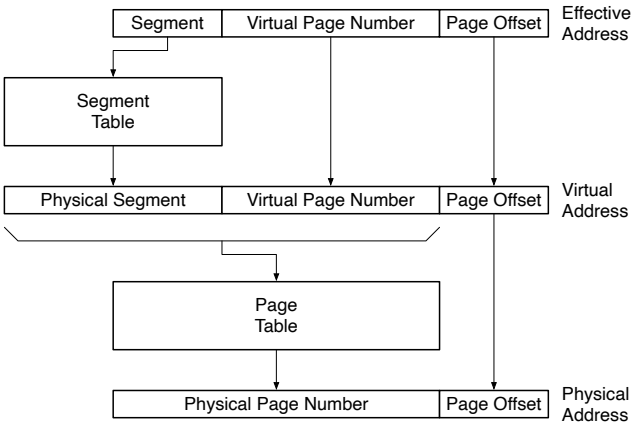

Figure 1: Segmentation creates a large virtual space that can be shared among all processes

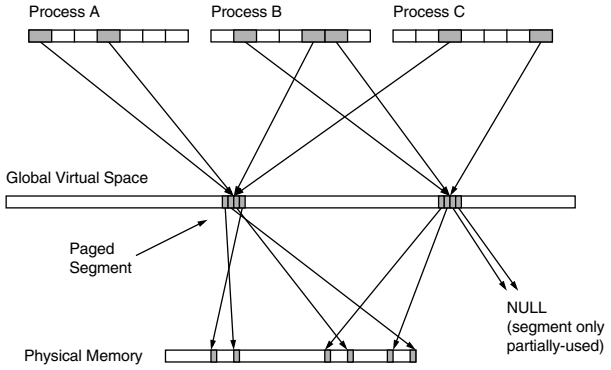

Figure 2: Segmentation solves the aliasing problem [6, pp. 897–901]

segmented address spaces. For instance, the PowerPC segmented address space and others like it provide a mapping mechanism that resembles Figure 1. What this enables is a shared-memory organization that looks like Figure 2. Because of its flexibility, as well as the discussion in the next bullet, Osprey assumes PowerPC-like segmentation at the hardware level — and note that this means that files will be represented as segments in the process address space.

- **How should a file be extended in size?** In the traditional operating system, files are extendable at will, up to the largest representable size that the file system accommodates. One simply seeks to the address and performs a `read()` or `write()` operation. This is decidedly *not* how virtual memory works, because seeking to a random address and performing a load or store operation at that random address is likely to result in a SIGSEGV and process termination. However, now that we identify files with segments and opening a file as mapping that file to the address space (and this is done at the segment granularity), what we can do is perhaps say that files must be no larger than a segment in

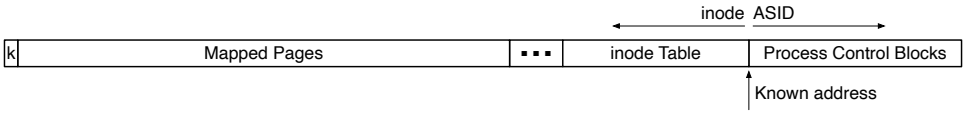

Figure 3: Layout of the large physical nonvolatile memory space in Osprey (regions not to scale)

size, and they can be extended up to that size automatically by using store instructions. Each such write (i.e., *store-word* operation) would cause the operating system to populate the file's page table with a mapping down to the written page, assuming no mapping already existed. Thus, a file in a merged system can indeed be both sparse and automatically extended, just as in a traditional file system.

- **How should the physical memory space be laid out?** In the traditional operating system, the physical address space is roughly partitioned into (a) the kernel code occupying the low memory region, where it provides known locations for important structures such as the interrupt table and handlers, (b) the page table, and (c) pages mapped by the page table. Some OS designs may blur the distinction between items (b) and (c), but this partition more or less describes how most kernels think of physical memory. Given all of the above design points, how can one lay out the memory space to be managed? Everything can be accomplished by simply adding the file system's metadata, the inode table. Figure 3 illustrates such an arrangement. As will be described, the process control blocks represent the highest levels of Osprey's system-wide page table (similar to the global *disjunct* page table of [6]), and the section labeled "mapped pages" includes all real mapped data: user-level main-memory pages, file-system data pages, and PTE pages that make up the bulk of the actual page table. The inode table is just below the table of process control blocks, and can be extended if desired. The mapped pages segment grows upward, and the inode table grows downward, in a heap/stack arrangement.

The following sections present some of the Osprey implementation details.

3.2 The Segmented Process Address Space

As discussed above, the memory-management system tightly integrates both process address spaces and files. A process address space has a fixed number of equal-sized segments. Files are not *opened* in the traditional sense but are instead mapped into the address space at the segment granularity — and this includes the executable binary file of the running process. Figure 4 shows the segments into which a process address space is divided, using 256 segments as an example. The number of segments is a design decision, and this will be discussed in more detail in the next section.

The segment assignments indicate hard limits: in particular, as suggested by the appearance of a singluar *Code* segment at the bottom of the address space, an executable file is not allowed to exceed the size of a single segment, and for symmetry, we extend that to include any file in the file system. Thus, when a file is

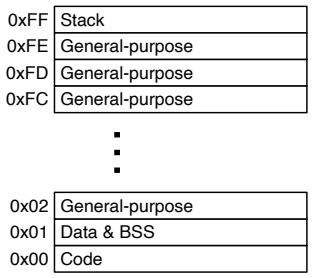

Figure 4: A process address space is a set of fixed-size segments

opened it is actually mapped into one of the unused general-purpose segments of the process's address space.

Software can ask for an unused segment to become mapped, similar to a `malloc()` system call; this causes the kernel to allocate space in physical memory for both data and the mapping information. The data segment and stack segment are allowed to exceed the size of a single segment, because each can extend upward/downward into the general-purpose segments adjacent to them. Both of these segments begin at known locations, to ensure simple code-generation, and they are extended automatically as references are made to adjacent segments.

Due to the extremely large extent of a 64-bit address space, the fixed-file-size limitation is not actually limiting in a significant way. For instance, we have the following:

- A 64-bit address space divided into 256 segments (an 8-bit segment ID) would impose a limit of 64 petabytes per file or code segment.
- A 64-bit address space divided into 4,096 segments (a 12-bit segment ID) would impose a limit of 4 petabytes per file or code segment.
- A 64-bit address space divided into 65,536 segments (a 16-bit segment ID) would impose a limit of 256 terabytes per file or code segment.
- A 64-bit address space divided into 16 million segments (a 24-bit segment ID) would impose a limit of 1 terabyte per file or code segment.
- A 64-bit address space divided into 4 billion segments (a 32-bit segment ID) would impose a limit of 4 gigabytes per file or code segment.

All file systems have *de facto* maximum file sizes, and this is no different ... in particular, individual files in the terabyte range would be larger than many physical disk systems in use today.

In contrast, consider a 32-bit address space that is divided into, say, 16 segments (a 4-bit segment ID) ... such a design would impose a maximum file or executable size of 256 MB, which may or may

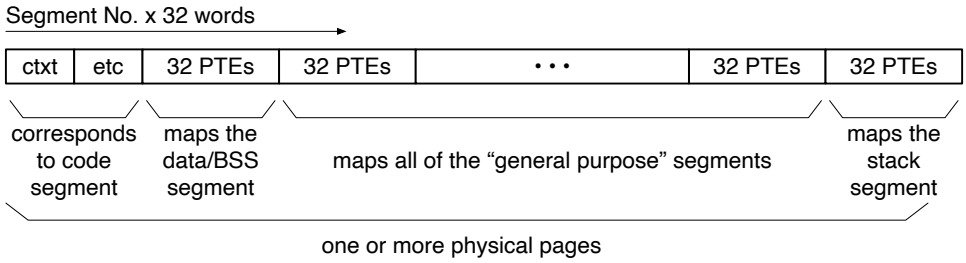

Figure 5: An Osprey process control block is the process context (register set and PC), additional space rounded up to power-of-two, and a number of regions each the size of *context + etc*, totaling one or more physical pages

not be unacceptably limiting, depending on one's target application for the operating system.

3.3 The Page Table and Process Control Block

The *Process Control Block* (PCB) is a data structure that contains information about the process, including the process context, its register-file contents and program counter value. The PCB can be held within the user address space, at a known virtual address as in the BSD **u.** struct [15], but we place it at a known *physical* address to form the topmost level of the global page table. The Osprey page table is formed around the PCB, in a manner that is reminiscent of the Mach page table [7] and the disjunct page table [6], in that a global table is formed from concatenating together the individual user-process page tables … in this case, the collected PCBs of all user processes. Because a process address space comprises a number of segments, and the first segment belongs to a known file within the file system (a raw executable), we structure the PCB as a collection of the most important immediately-accessible data items: the process context, a few pointers, and the top level of the page table, and we place these in a known location so that kernel handlers can easily find them. Other components of a traditional process control block are maintained in a structure homed within the address space.

An Osprey process control block is shown in Figure 5 and assumes that the process's register-file context is sixteen 64-bit words. Following that is another 16 words, which include hint PTEs and other data items that kernel handlers such as *context_switch* might need. Following that is the top level of the user page table … for each segment in the address space, there is a set of 32 8-byte PTEs that ultimately map the segment. The segment number is used to index into this structure, where the segment number is multiplied by 128 bytes (32 times an 8B word). The first segment, corresponding to the executable, is not held in the user page table but is instead mapped by the inode structure, and for references to segment 0 in each address space, the kernel goes to the inode instead of the user page table. An example

The user PCB structure forms the backbone of the mapping structure for all of physical memory. The user PCBs are organized into a linear table that is based at a known location within the physical address space and is indexed by the address-space identifier (ASID). When a PTE needs to be found, the running process's ASID indexes into the table of PCBs, and the segment ID is used to index into the mapping structures held within that process's control block. Note that, while the description focuses on linear page tables, in many cases a more efficient and less sparse design can be obtained with inverted page tables.

Figure 6 illustrates the mapping of a segment, using as an example the process control block segment map of 32 PTEs plus an additional 2 levels of mapping tables, with 16KB pages. This is a design choice for reducing the cost of mapping at the expense of a smaller virtual address space size. Given a 16KB page size, it yields a terabyte-scale segment size and a petabyte-scale per-process address space.

Figure 7 illustrates the mapping of a segment, using as an example the process control block segment map of 32 PTEs plus an additional 3 levels of mapping tables, with 16KB pages. This is a design choice for maximizing the per-process virtual address space size, at the expense of an additional memory lookup in page-table lookup procedure. Given a 16KB page size, it yields a petabyte-scale segment size and an exabyte-scale address space.

Because many, if not most process address spaces use very little memory — for example, a few KB of stack and a few MB of data — our process control block has "shortcut" PTEs that link directly to the top of the stack segment and the bottom of the data segment. Depicted in Figures 6 and 7 is the stack's shortcut PTE. If the data and/or stack are no larger than 32MB (assumes a 16KB page size), then this is all that is needed to map the data and stack regions: two words in the *etc* portion of the PCB, and two PTE pages. If the process requests any address outside this range, the kernel must go through the full mapping step of reading the 1st-level table to get to the 2nd-level table and so on, ultimately reaching the page in the desired segment.

As mentioned earlier, requests to the code segment go directly to the inode structure: because the executable is a known file and has its own mapping, the code segment needs no mapping at the process level and is instead mapped in the inode structure. The inode structure maps the address space in the same way as shown in Figures 6 and 7.

As shown in Figure 3, the table of process control blocks is at a known location in the top of the physical address space. Starting at this known location, the kernel indexes upward into the PCB table using the address space identifier (equivalent to a process ID). This takes the kernel to the process's control block, which has shortcut PTEs and links to the full page table if necessary. As an example, say we have the following:

- A physical space of 256 TB (48 bits)

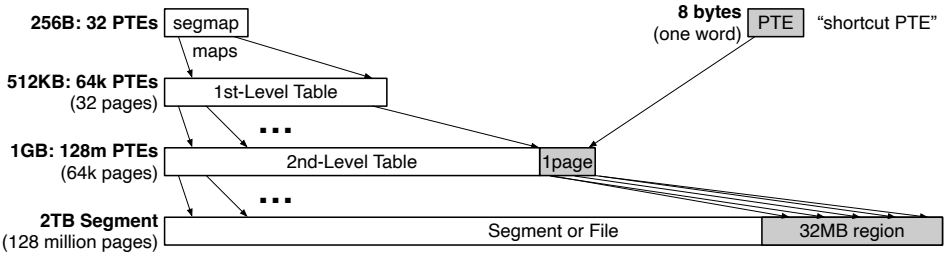

Figure 6: Example segment mapping with 16KB page size and three levels of mapping

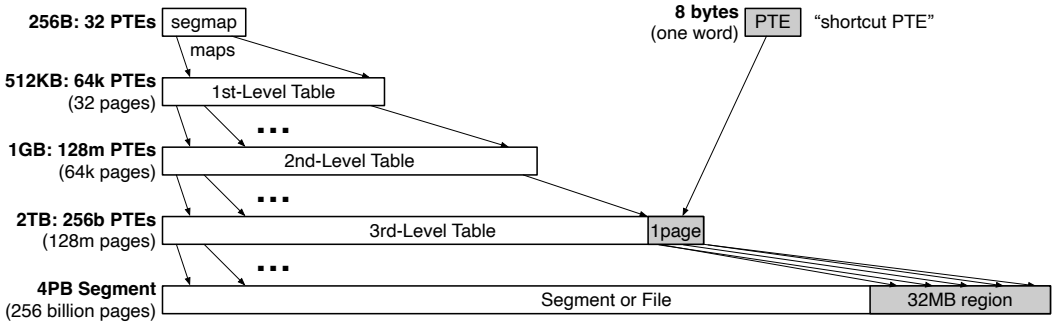

Figure 7: Example segment mapping with 16KB page size and four levels of mapping

- An ASID of size 18 bits (256K processes)
- A PCB of size 16KB (14 bits)

Given the above assumptions, then the PCB array would require four gigabytes (18 bits + 14 bits), and the known address at which the array begins would be `0xFFFF 0000 0000` at the top of the physical space.

At this address, and indexing downward is found the inode array. Each of the inodes contains mapping information just like the process control block. If each inode is 1KB, then one can support four million files in a table of four gigabytes, putting the combined size of the two fixed tables at a total of 8GB. One can adjust the size of each of these tables, resulting in moving the fixed address point.

3.4 Process Invocation from a Raw Executable

To start a process, one need only specify the desired inode number. The operating system assigns an ASID and creates the necessary data structures. The pseudocode shown in Listing 1 gives an idea of the process.

```
[assume a process has 1024 segments]
[assume executable is file number inode_no]
choose an address space ID, named ASID
allocate page for data
at PTE_array[ASID] set up mapping for data page
allocate page for stack
at PTE_array[ASID] set up mapping for stack page
put ASID and inode_no into hardware registers
set PC=0
```

Listing 1: Pseudocode which explains the idea

That is all that is required to invoke a process: one need only jump into an executable file to start running it. What such an arrangement requires is that either the BSS segment is handled by init code in the `_start()` routine, or a second data segment is maintained in parallel with the code segment for each executable, which then could be mapped into the address space in a *copy-on-write* form. So far, we have done the former, out of simplicty.

The C code shown in Listing 3 provides slightly more detail.

```
// kernel runs in physical-address mode
PCB_t *PCB_table = 0xFFFF 0000 0000;
int ASID = new_ASID();
char *page = allocate_page();
int *PTEpage = allocate_page();
PTEpage[0] = VALID | page;  // set PTE valid bit
PCB_table[ASID].data_shortcut = PTEpage;
page = allocate_page();
PTEpage = allocate_page();
PTEpage[1023] = VALID | page;  // set valid bit
PCB_table[ASID].stack_shortcut = PTEpage;
PCB_table[ASID].executable = inode;
init_context(ASID);
```

Listing 2: Actual code which demonstrates the idea

The call to `init_context()` zeroes out the register values of the context data structure, inits hardware registers to support the process and its code mapping, sets the stack pointer to `0xFFF…F`, and sets the program counter to zero. After this, the operating system performs a return-from-exception, which reads from the PCB context identified, filling the register file with those newly-initialized

values, and when the program counter gets the value 0, the process jumps to the referenced inode and begins execution.

In our experimental version of the operating system, we have kept the data segments zero by convention, initializing them from the code, as opposed to storing a populated BSS section. However, one could certainly maintain two separate segments for a given executable: one for the code, and one for the BSS.

3.5 File Access

As mentioned, file access is memory-mapped. Thus, when opening a file, the operating system maps the file into an unused segment in the process address space and returns a pointer to the segment mapped. The mapping information is not copied into the user's page table, but instead, at the segment level, the PCB indicates that the segment is being provided by an inode.

In Osprey, the UNIX `copy()` utility could be implemented as shown in Listing 3.

```
char *in, *out;
int i, size;
in = open_file(file1);
size = file_size(file1)
out = malloc_segment();
for (i=0; i<size; i++) {
    out[i] = in[i];
}
unlink(in);
permanentify(out, size, filename, perms);
```

Listing 3: Implementation of copy

Opening a file maps the file's contents into the process's address space and returns a pointer to the mapped region, which is aligned on a segment boundary.

Allocating a region of memory returns a pointer to a segment-aligned region of memory, and it can be extended at will by software.

Making a segment-sized region of memory into a permanent file is done with the `permanentify()` system call, which creates a page table for the specified extent of memory ... note that the mapping information is taken directly from the user's page table, and so it may be sparse (may contain empty spaces in between valid data). This is perfectly acceptable.

At the end of the call to `permanentify()`, the operating system removes the mapping information from the process's page table and points the process's segment table at the newly created inode.

Thus, the process still maintains access to the file contents, but when the process exits and its pages are reclaimed, the file contents are not reclaimed and garbage-collected.

4 CONCLUSION

The main ideas explored in the Osprey experiment are those of code simplicity and code reduction. Making kernel code simpler and making the kernel smaller are both reasonable goals for operating systems design: as the current experiment shows, future trends in memory-system design will enable both these goals.

Combining the main memory system with the nonvolatile disk system enables a direct load/store access paradigm for all of a system's data, which simplifies things tremendously:

- The need for separate virtual memory and file-system kernel subsystems goes away
- The need to linearize and de-linearize file contents, during file writing and reading, respectively, goes away
- The need to access files one buffer-full at a time goes away
- Raw executables become possible — meaning that one can simply *jump* into an executable file directly without the need to first unpack it into main memory

While it is admittedly an objective measure, this simplification is significant, reducing both lines of code and code complexity in a substantial way. The 16-bit version of Osprey is operational now, and the 64-bit version will be released to the public domain next year.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98, 12, 2237–2251. DOI: 10.1109/JPROC.2010.2070830.

[2] Paolo Cappelletti. 2015. Non volatile memory evolution and revolution. In *2015 IEEE International Electron Devices Meeting (IEDM)*, 10.1.1–10.1.4. DOI: 10.1109/IEDM.2015.7409666.

[3] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12, 4, (Nov. 1994), 271–307. DOI: 10.1145/195792.195795.

[4] Yangyin Chen and Chris Petti. 2016. Reram technology evolution for storage class memory application. In *2016 46th European Solid-State Device Research Conference (ESSDERC)*, 432–435. DOI: 10.1109/ESSDERC.2016.7599678.

[5] Scott W. Fong, Christopher M. Neumann, and H.-S. Philip Wong. 2017. Phase-change memory—towards a storage-class memory. *IEEE Transactions on Electron Devices*, 64, 11, 4374–4385. DOI: 10.1109/TED.2017.2746342.

[6] Bruce Jacob, S. Ng, and D. Wang. 2010. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science. ISBN: 9780080553849.

[7] Bruce L. Jacob and Trevor N. Mudge. 1998. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VIII). Association for Computing Machinery, San Jose, California, USA, 295–306. ISBN: 1581131070. DOI: 10.1145/291069.291065.

[8] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. 2020. Tileable monolithic reram memory design. In *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, 1–3. DOI: 10.1109/COOLCHIPS49199.2020.9097632.

[9] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Donald Yeung, and Bruce Jacob. 2019. Design for reram-based main-memory architectures. In *Proceedings of the International Symposium on Memory Systems* (MEMSYS '19). Association for Computing Machinery, Washington, District of Columbia, USA, 342–350. ISBN: 9781450372060. DOI: 10.1145/3357526.3357561.

[10] Marc Jordà, Siddharth Rai, Eduard Ayguadé, Jesús Labarta, and Antonio J. Peña. 2022. Ecohmem: improving object placement methodology for hybrid memory systems in hpc. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 278–288. DOI: 10.1109/CLUSTER51413.2022.00040.

[11] Minjae Kim, Bryan S. Kim, Eunji Lee, and Sungjin Lee. 2022. A case study of a dram-nvm hybrid memory allocator for key-value stores. *IEEE Computer Architecture Letters*, 21, 2, 81–84. DOI: 10.1109/LCA.2022.3197654.

[12] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 256–267. DOI: 10.1109/ISPASS.2013.6557176.

[13] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (ISCA '09). Association for Computing Machinery, Austin, TX, USA, 2–13. ISBN: 9781605585260. DOI: 10.1145/1555754.1555758.

[14] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE Micro*, 30, 1, 143–143. DOI: 10.1109/MM.2010.24.

[15] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley Publishing Company, USA. ISBN: 0-201-06196-1.

[16] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30, 10, 2223–2236. DOI: 10.1109/TPDS.2019.2908175.

[17] Miguel Marques. 2021. *Ambix: Rethinking Linux's Page Management to Support the New Intel Optane DC Persistent Memory*.

[18] Miguel Marques, Ilia Kuzmin, João Barreto, José Monteiro, and Rodrigo Rodrigues. 2021. Dynamic page placement on real persistent memory systems. (2021). arXiv: 2112.12685 [cs.DC].

[19] Deepak M. Mathew, Felipe S. Prado, Éder. F. Zulian, Christian Weis, Muhammad Mohsin Ghaffar, Matthias Jung, and Norbert Wehn. 2020. An Energy Efficient 3D-Heterogeneous Main Memory Architecture for Mobile Devices. In *International Symposium on Memory Systems (MEMSYS 2020)*. ACM/IEEE, (Oct. 2020).

[20] Sparsh Mittal and Jeffrey S. Vetter. 2016. A survey of techniques for architecting dram caches. *IEEE Transactions on Parallel and Distributed Systems*, 27, 6, 1852–1863. DOI: 10.1109/TPDS.2015.2461155.

[21] J. Müller et al. 2013. Ferroelectric hafnium oxide: a cmos-compatible and highly scalable approach to future ferroelectric memories. In *2013 IEEE International Electron Devices Meeting*, 10.8.1–10.8.4. DOI: 10.1109/IEDM.2013.6724605.

[22] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. Exploring the performance benefit of hybrid memory system on hpc environments. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 683–692. DOI: 10.1109/IPDPSW.2017.115.

[23] Subisha V, Varun Gohil, Nisarg Ujjainkar, and Manu Awasthi. 2020. Prefetching in hybrid main memory systems. In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems* (HotStorage'20) Article 11. USENIX Association, USA, 1 pages.