Towards O(1) Memory

Michael M. Swift
University of Wisconsin–Madison
swift@cs.wisc.edu

Abstract

Since the dawn of computing, memory capacity has been a primary limitation in system design. Forthcoming memory technology such as Intel and Micron's 3D XPoint memory and other technologies may provide far larger memory capacity than ever before. Furthermore, these new memory technologies are inherently persistent and save data across system crashes or power failures.

We conjecture that current operating systems are ill-equipped for an environment where there is ample memory. For example, operating systems do substantial work for every page allocated, which adds unnecessary overhead when dealing with terabytes of memory.

We suggest that now is the time for a complete rethinking of memory management for both operating systems and language runtimes considering excess memory capacity. We propose a new guiding principle: Order(1) operation, so that memory operations have low constant time independent of size. We describe a concrete proposal of this principle with the idea of *file-only memory*, in which most dynamic memory allocation is managed with file-system mechanisms rather than common virtual memory mechanisms.

CCS Concepts

• **Software and its engineering → Virtual memory**; **Main memory**; *File systems management*; *Secondary storage*;

Keywords

persistent memory, single-level storage

ACM Reference format:
Michael M. Swift. 2017. Towards O(1) Memory. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 8 pages.
https://doi.org/10.1145/3102980.3102982

1 Introduction

Forthcoming memory technologies may provide vast amounts of byte-addressable memory at costs comparable to today's systems. For example, projections of phase-change memory (PCM) and other technologies predict large, cheap memories [2, 19]. More recently, 3-dimensional fabrication techniques used in 3D-NAND Flash [20] and Intel and Micron's 3d XPoint memory [25] can create large memories at lower cost beyond the terabytes possible with DRAM today. These technologies are inherently persistent, and thus have low power requirements: power is only needed for reading and writing but not storing stored data, while DRAM requires periodic refresh. The Massive Memory Machine [10] proposed decades ago described many benefits of having so much memory.

While not guaranteed, these technologies may be able to provide memory sizes an order of magnitude (or more) larger than DRAM at similar prices and power. If this happens, current system designs are inadequate. Operating systems, since their origin, devote huge complexity and effort to efficiently allocating and reclaiming scarce memory resources through the virtual memory system and memory allocators. Processors provide page-based fine-grained (4KB) address translation to avoid internal and external fragmentation of scarce memory. As an example of what is possible, with ample memory it may be more efficient to allocate a large page (e.g., 2MB) when only hundreds of kilobytes are needed to improve TLB performance. No current system would choose this, though, because of the wasted space. Persistence of memory provides another opportunity to further simplify systems in many deployments as there is no need to detect which pages must be written back to disk.

Towards this end, we propose *Order(1) memory* as a new organizing principle for memory management: we strive to make all memory management operations constant time, independent of operand or memory size. This ensures that memory management costs do not grow as increasingly larger memories are available. For example, the time to allocate and map a large read-only file should be independent of its size. In many cases this can be accomplished by trading space, in the form of some wasted memory, for time spent managing memory.

As a concrete step towards this goal, we describe *file-only memory*. Recognizing that file systems already manage large, underutilized, persistent data stores, we adopt file-system techniques towards memory management. This enables most operations that currently operate on individual pages to instead operate on large extents or a whole file, and hence provide Order(1) performance.

In addition to this high-capacity persistent memory, future systems will likely have a small amount of high-bandwidth memory, similar to what is used for GPUs today [17, 26]. This memory, while many gigabytes, will likely be a small fraction of total memory, and is used for computations that have high bandwidth needs as compared to capacity needs. In this work, we focus only on high-capacity memory.

While the issue of redesigning operating systems around new memory technologies has been considered in the past [2, 4, 8], there is now more clarity about the future of computer systems, and hence it is possible to propose more concrete designs on the future of memory management.

HotOS '17, May 08-10, 2017, Whistler, BC, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3102982

2 Motivation

Three factors motivate re-examining memory management: (i) emerging technologies allowing cheap, high-capacity memories, (ii) the merging of memory and storage technology, and (iii) current OS memory management practices.

Memory technology: The ability to produce higher-density DRAM has slowed, and analysts predict that other technologies are needed to keep growing capacity [28]. Most candidate technologies, such as Phase-change memory (PCM) and Spin-Transfer Torque MRAM (STT-MRAM), are persistent because they use no energy to store data. This also allows higher three-dimensional structures because there is less need to remove heat. As an example, Intel and Micron's much-delayed 3D XPoint DIMM product promises 6TB of storage in a 2-socket server, up from approximately 768GB today [23]. To enable paging for large physical memories, Intel recently introduced 5-level address translation, which can address 4PB of physical memory but requires up to 35 memory references in virtualized systems [14].

We note that ample memory does not mean that all uses of memory are free. Accessing memory still imposes a cost, and caches, because of their proximity to the processor core, will remain a precious resource. However, capacity in memory itself for data that is not accessed may become cheap.

Implication: There is likely to just be vastly more memory to manage in the near future, even for lower-end systems. Many aspects of memory management are currently linear, such as maintaining per-page metadata. Any operations that are linear in the amount of memory available (physical) or used (virtual) may get relatively slower.

Memory as storage: Large amounts of cheap memory does not mean it can be used freely, as people typically only buy the memory they need. However, non-volatile memory enables the merging of persistent and volatile memory. Thus, there is likely to be excess capacity most of the time: space reserved for future expansion of the file system can be used for volatile memory objects in the meantime [16].

Consider the situation of storage devices such as hard drives and SSDs. An old study from Microsoft [1] showed that the mean and median file system utilization was below 50%, which is explained by disks gradually increasing in utilization until they are replaced by a larger device. When local storage moves into non-volatile memory, there may be similar utilization patterns, implying vast amounts of memory provisioned for future persistent data but currently unused.

Implication: Storage-like utilization patterns for persistent memory indicate there will likely be excess memory capacity that can be used to improve performance.

Cost of memory management: Current memory management techniques are generally optimized to be space efficient at the expense of performance: 4KB pages have high overhead, but low fragmentation. Heap allocators and garbage collectors tend to spend CPU cycles to conserve memory. However, recent efforts such as TCMalloc [11] and log-structured memory [27] that waste space for improved performance show some of the potential available.

The kernel's management of physical memory is also designed around a scarce resource backed by a separate persistent store. As a result, the kernel maintains extensive metadata about pages and devotes substantial effort to tracking the use and status of every physical page. For example, the Linux `PAGE` structure has 25 separate flags to track memory status and 38 fields (many overlapping in unions). This design also leads to expensive per-page operations to ensure data remains in place (i.e., pinning), which may be necessary for DMA transfers. If data is kept in persistent memory, there is no need to track if it is clean or dirty, as it need not be returned to backing storage. Similarly, if memory is ample, there is no need to track what has been recently referenced to reclaim idle pages.

Implication: Much of the information tracked by the memory manager is either unnecessary or can be tracked at much coarser granularity.

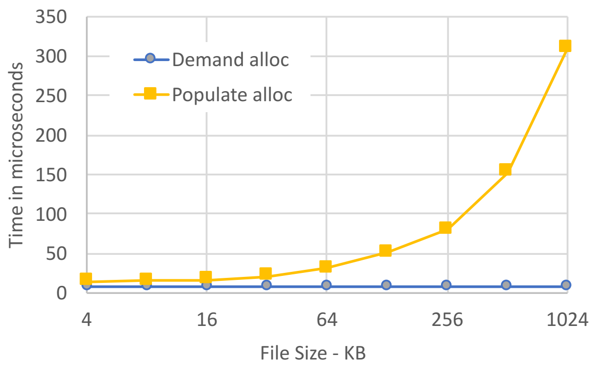
3 Order(1) Memory

Drawing on these observations, we propose *Order(1)* operations as the central design principle for memory management. Our design strives to operate in constant time for any memory-management operation, independent of the operand size. This ensures that with very large memories, common memory operations such as allocation and mapping remain fast. Furthermore, current processors support various page sizes, and O(1) OS management of memory may enable better utilizing this support [18, 24], But this is not enough: Intel and ARM processors support only a few page sizes, and large pages have alignment restrictions so the system must resort to small pages in many cases. For example, on x86-64, large pages are powers of 512 times bigger than 4KB (2MB, or 1GB), so allocations may still use many pages to satisfy a request. When swapping pages in or out, 2MB pages are expensive to swap and Linux instead fragments them into 4KB pages.

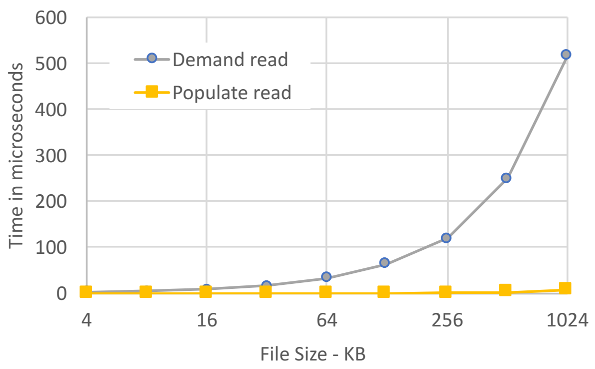
Today's systems memory management operations require per-page activities. When allocating memory (e.g. `mmap(MAP_ANONYMOUS)` in Linux), the operating system separately allocates every page. Similarly, when mapping a file, the OS separately creates a page-table entry for every page of the mapped region. This cost is ameliorated with demand paging, which only performs these operations on reference. However, for sparse access to large data sets, the fundamental linear operation cost remains.

For example, Figure 6b shows the cost to memory-map a file on `tmpfs`. If the page table is pre-populated (MAP_POPULATE), the cost linearly increases with file size. In contrast, Figure 1b shows the cost of accessing one byte from each page of a mapped file. Here, the cost of demand faulting in the file (MAP_PRIVATE) for large files is more than 50x that of pre-populating page tables. As file sizes increase, the overhead of generating mappings increases linearly.

With O(1) operations, allocating memory could be done in constant time, at the cost of wasting memory that is unused. Similarly, mapping a file could also happen in constant time rather than in small pieces as the file is accessed. While mapping files currently is expensive and hence rare, if made fast it could be much more efficient than standard file system APIs. Given that data is already



(a) Time of a `mmap()` operation on a file on `tmpfs` populating the mapping and using demand paging.



(b) Total time to access one byte of each page of a file on `tmpfs` using a pre-populated mapping compared to on-demand mapping.

Figure 1: Cost of memory-mapping operations.

in memory, it is natural to simply expose that data to programs directly rather than forcing the kernel to interpose on every access.

To demonstrate the power of O(1) memory management techniques, we demonstrate a design leveraging this principle. *File-only memory* performs most memory operations on the granularity of a whole file, rather than on individual pages. We follow with discussions of how processor hardware can further reduce the cost of memory operations.

3.1 File-only Memory

Our work draws on the observation that operating systems already know how to manage large quantities of persistent data efficiently through the file system. Compared to the memory system, files maintain metadata at coarse granularity (e.g., permission is granted for the whole file and not individual blocks). Modern file systems, when possible, translate addresses in long extents (e.g., Ext4, NTFS) rather than individual blocks. Furthermore, unused blocks are represented by a single bit in a bitmap, as compared to the complex per-page metadata memory maintained by memory management systems. Finally, most storage devices operate in a regime where they are far from full, so file systems are optimized for situations where space is available. As a result, they contain many of the mechanisms needed to efficiently manage large quantities of persistent memory. Unlike past efforts to back all memory in files (e.g., Opal [4]), we propose to remove the memory layer *above* files.

Within the operating system, we propose that all user-mode memory be allocated as files, backed by a memory file system such as Linux's `tmpfs`. When launching a process, code segments, heap segments, and stack segments can all be represented as separate files, so there is no need to allocate individual pages. Creating a thread stack becomes allocating a file with a single extent containing a region of memory and mapping it into the address space. There is no need to update metadata about the individual pages of the stack. When a process allocates memory, it maps a file into its address space. For persistent data, such as program code or data, this may be a named file in the file system. For volatile data, this may be a temporary file. Memory permissions can be managed at the granularity of the whole file, and the file system is responsible

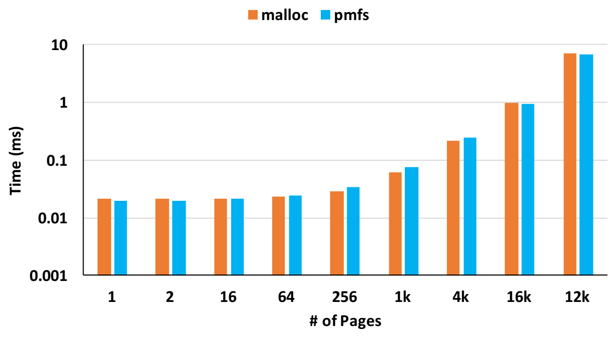


Figure 2: Time to allocate memory pages using anonymous memory (`malloc`) and memory mapping a file in PMFS.

for allocating physical memory for the file. Rather than reference counting pages, we propose to only count references to files.

This approach (allocating memory as files) is already used in select cases: one current use of `tmpfs` is to provide file-system controls over memory allocation, such as quotas or recovery of large in-memory data sets after a process crash.

To demonstrate that using the file system instead of memory system to manage memory, we compared the cost of allocating data using anonymous memory (`MAP_ANON`) against allocating data through a file in the PMFS file system for persistent memory [7]. Figure 7 shows that across a range of sizes, using the file system to allocate memory has little extra cost.

Just moving allocation to the file system is not enough achieve O(1) operations. However, file systems make some of that easier.

Memory allocation: Further reductions in the cost of memory operations are possible moving more work to the file system. For example, all management of virtual memory can be done at the granularity of files. File systems can efficiently allocate large contiguous extents, which reduces the per-page cost of allocation. Furthermore, metadata that is currently maintained about individual pages (clean, dirty, referenced, locked) will instead be managed

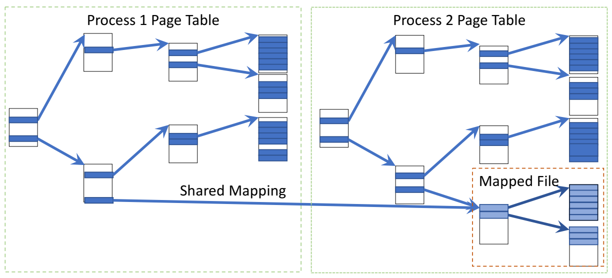


Figure 3: Efficient shared mappings.

for the entire file. This simplifies data structures and reduces the CPU overhead of memory management.

We propose that memory is only reclaimed in the unit of a file, such as when a process unmaps a file (e.g., `munmap()`), or when the process terminates. The operating system does not scan for idle pages to reclaim, scan dirty page to swap out, and the heap need not identify unused pages to release with `madvise()`. Another potential benefit of managing memory with files is reclamation under (rare) memory pressure: if applications use a file API to access non-critical data (i.e., discardable data such as caches), the OS can reclaim the memory by deleting non-critical files. This provides many of the benefits of transcendent memory [21].

O(1) operation is only possible if most memory can be allocated contiguously. We observe that heap allocators address the same problem: how to allocate contiguous memory with very little overhead. We propose using techniques from heaps, such as slab allocators [3], to manage physical memory.

Memory mapping: Working with entire files can speed memory mapping, but creating a page-table entry for each page is inherently a linear cost. However, the constant overhead of mapping can be reduced with whole files. In addition, as files are stored in memory, it is possible to pre-create page tables, so that mapping becomes changing a single pointer in a page table to refer to existing page tables [13]. This is complicated in that it requires mapping files at the natural granularities of page table structures (e.g., 2MB, 1GB). Another mechanism for efficiency is to share mappings across processes, which is easy if they are aligned on page-table boundaries (again, 2MB, 1GB, etc.). These can be accomplished by creating a pointer from one process's page table to an internal page-table node of another process sharing the file. Figure 3 shows how aligned mappings can be efficiently shared. To make mapping even more efficient, pre-created page tables can be stored persistently, so that even when mapping a file the first time, an existing page table can be re-used for O(1) operations.

Persistence management: As memory is large and persistent, we assume there will generally be no swapping to disk either for capacity or for writing back dirty file data. Thus, there is no need to track the clean/dirty/referenced status of most memory, which avoids the need for page reclamation algorithms (e.g., clock, 2-queue). Those applications that need swapping could implement it themselves using techniques service such as `userfaultd` [5].

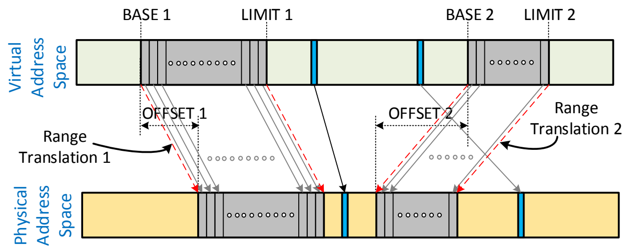


Figure 4: Range translations.

The use of files for memory also makes it simple to separate out memory management from persistence [12], in that all data lives in files that can be marked at any time as volatile or persistent to indicate whether they should survive process terminations and system restarts.

We note that storing volatile data in persistent memory introduces its own complexities and overhead. In particular, for security purposes memory must be zeroed out before being reused. For volatile data, the OS explicitly erase memory before reusing it following a failure. This is currently a linear-time operation and suggests the need for new techniques to efficiently erase memory in constant time.

Memory locking: Currently letting a device access memory often requires locking the page in memory; even devices that support page faults through an IOMMU incur high penalties [15]. With file-only memory, data is implicitly pinned in memory, as pages are never reclaimed or relocated until the file is explicitly unmapped.

We note that there are some optimizations that become more difficult when moving memory management to the file system. Specifically, Linux merges adjacent memory regions when possible (i.e., same flags). This reduces the size of internal metadata, and provides the benefits of growing regions (decreased overhead) without the costs of guaranteeing they can. In addition, some operations that depend on page-level mappings, such as guard pages or copy-on-write, cannot easily be supported.

3.2 Hardware Optimizations

Enabling O(1) memory operations in software can be helped by improved hardware support for memory. With current page-base virtual memory, processors fundamentally require per-page operations to create page tables or handle TLB misses, and hence O(1) memory is not completely achievable. For example, in our experiments we observed that it was faster to make a `read()` system call to read 16KB than to access data *already mapped into a process* if it would cause TLB misses.

But, changes to how a processor performs address translation can dramatically reduce the cost of accessing large memory, and can benefit from OS support for O(1) memory. In particular, proposed *range translations* [9] allow mapping an arbitrary length of contiguous physical memory to contiguous virtual memory using a fixed size structure comprising a base, limit, and offset, as shown in Figure 4. With this structure, any contiguous physical range of memory can be added to the address space by populating a single entry in a *range table* (analogous to a page table, but a different

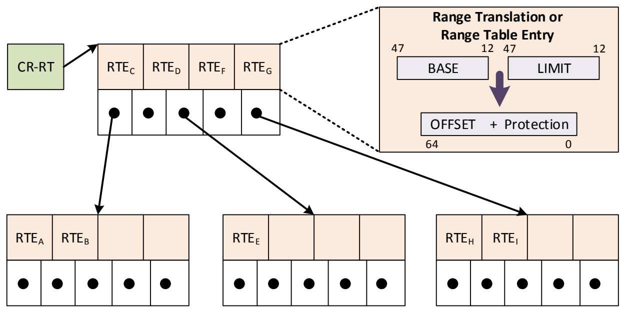


Figure 5: Range table and entries

data structure). The processor adds this entry to a *range TLB* on a reference to any address in the range. Figure 9 shows how the table efficiently represents translations.

We observe that range translations complement our proposed file-only memory: memory managed as extents in a file can be efficiently mapped by assigning one virtual memory range to each extent. Similarly, unmapping a file can be a single operation to update the range table and shoot down the entry in the TLB. Thus, we believe that in concert with advances on the operating system for O(1) memory operations, there must be concurrent advances in processor memory management to support efficient software designs.

4 Order(1) Memory

Drawing on these observations, we propose *order(1)* operations as the central design principle for memory management. This design strives to achieve constant-time for any memory-management operation, independent of the operand size. This ensures that with very large memories, common memory operations such as allocation and mapping, remain fast. Furthermore, current processors support various page sizes, and O(1) OS management of memory may enable better utilizing this support [18, 24], But this is not enough: Intel and ARM processor support only a few page sizes, and large pages have alignment restrictions so the system must resort to small pages in many cases.

Today's systems memory management operations require per-page activities. When allocating memory (e.g. `mmap(MAP_ANONYMOUS)` in Linux), the operating system separately allocates every page. Similarly, when mapping a file, the OS separately creates a page-table entry for every page of the mapped region. This cost is ameliorated with demand paging, which only performs these operations on reference. However, for sparse access to large data sets, the fundamental linear operation cost remains.

For example, Figure 6a shows the cost to memory-map a file on `tmpfs`. If the page table is pre-populated (MAP_POPULATE), the cost linearly increases with file size. In contrast, Figure 6b shows the cost of accessing one byte from each page of a mapped file. Here, the cost of demand faulting in the file (MAP_PRIVATE) for large files is more than 50x that of pre-populating page tables. As file sizes increase, the overhead of generating mappings increases linearly.

With O(1) operations, allocating memory could be done in constant time, at the cost of wasting memory that is unused. Similarly, mapping a file could also happen in constant time rather than in small pieces as the file is accessed. While mapping files currently is expensive and hence rare, if made fast it could be much more efficient than standard file system APIs. Given that data is already in memory, it is natural to simply expose that data to programs directly rather than forcing the kernel to interpose on every access.

To demonstrate the power of O(1) memory management techniques, we demonstrate two concrete designs leveraging this principle. First, *file-only memory* performs most memory operations on the granularity of a whole file, rather than on individual pages. Second, *physically based mappings* exploit physical addresses to allow efficient mapping operations. We follow with discussions of how processor hardware can further reduce the cost of memory operations.

4.1 File-only Memory

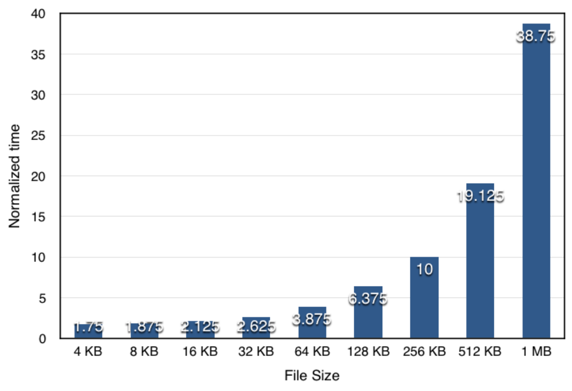
Our work draws on the observation that operating systems already know how to manage large quantities of persistent data efficiently through the file system. Compared to the memory system, files maintain metadata at coarse granularity (e.g., permission is granted for the whole file and not individual blocks). Modern file systems, when possible, translate addresses in long extents (e.g., Ext4, NTFS) rather than individual blocks. Furthermore, unused blocks are represented by a single bit in a bitmap, as compared to the complex per-page metadata memory maintained by memory management systems. Finally, most storage devices operate in a regime where they are far from full, so file systems are optimized for situations where space is available. As a result, they contain many of the mechanisms needed to efficiently manage large quantities of persistent memory.

Within the operating system, we propose that all user-mode memory be allocated as files, backed by a memory file system such as Linux's `tmpfs`. When launching a process, code segments, heap segments, and stack segments can all be represented as separate files, so that noneed to allocate each individual page. Creating a thread stack becomes allocating a file with a single extent containing a region of memory and mapping it into the address space. There is no need to update metadata about the individual pages of the stack.

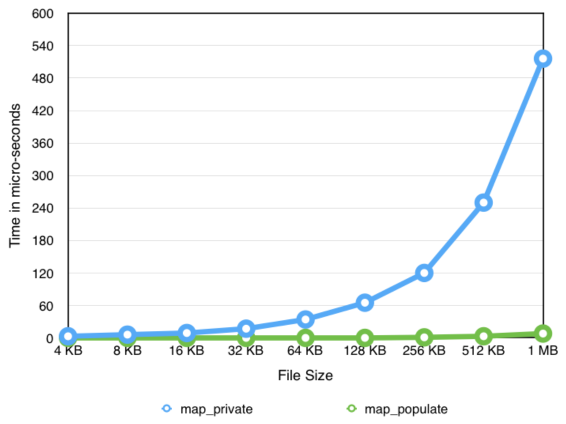
When a process allocates memory, it maps a file into its address space. For persistent data, such as program code or data, this may be a named file in the file system. For volatile data, this may be a temporary file. Memory permissions can be managed at the granularity of the whole file, and the file system is responsible for allocating physical memory for the file. Rather than reference counting pages, we propose to do reference counting for reclamation of whole files.

This approach (allocating memory as files) is already used for in select cases: one current use of `tmpfs` is to provide file-system controls over memory allocation, such as quotas or recovery of large in-memory data sets after a process crash.

As a preliminary experiment, we compared the cost of allocating data using anonymous memory (MAP_ANON) against allocating data through a file in the PMFS file system for persistent memory [7]. Figure 7 shows that across a range of sizes, using the file system to



(a) Time to of a `mmap()` operation a file on `tmpfs` populating the mapping (MAP_POPULATE) relative to the time to use demand paging (MAP_PRIVATE).



(b) Total time to access one byte of each page of a file on `tmpfs` using a pre-populated mapping (MAP_POPULATE) compared to on-demand mapping (MAP_PRIVATE).

Figure 6: Cost of memory-mapping operations.

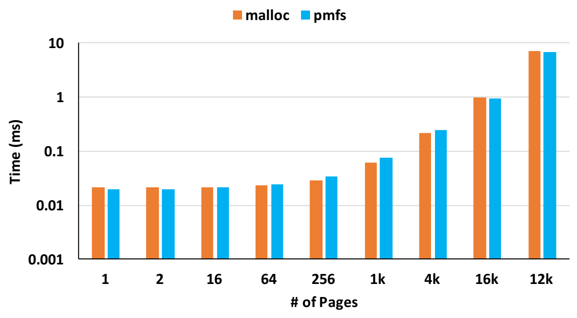


Figure 7: Time to allocate memory pages using anonymous memory (malloc) and memory mapping a file in PMFS.

allocate memory has little extra cost.

Just moving allocation to the file system is not enough achieve O(1) operations. However, file systems make some of that easier.

Memory allocation: Further reductions in the cost of memory operations are possible moving more work to the file system. For example, all management of virtual memory can be done at the granularity of files. File systems can efficiently allocate large contiguous extents, which reduces the per-page cost of allocation. Furthermore, metadata that is currently maintained about individual pages (clean, dirty, referenced, locked) will instead be managed for the entire file. This simplifies data structures and reduces the CPU overhead of memory management. It is necessary to better manage memory for contiguity. Linux manages pages using a buddy allocator, but does not aggressively merge pages, so there may be contiguity present that is not available for use. Alternate designs, such as using slab allocator for page allocations, may be better.

In this design, memory is only reclaimed in the unit of a file: when a process unmaps a file (e.g., `munmap()`), or when the process terminates. The operating system does not scan for idle pages to reclaim, scan dirty page to swap out, and the heap need not identify unused pages to release with `madvise()`. Another potential benefit of managing memory with files is reclamation under (rare) memory pressure. If applications cache objects in files and only open the file when using it, access patterns can be tracked at coarse granularity (an entire file), and data can be reclaimed the same granularity. This provides many of the benefits of transcendent memory [21].

Memory mapping: Working with entire files can speed memory mapping, but it is inherently a linear cost to create a page-table entry for each page given hardware. However, the constant overhead of mapping can be reduced with whole files. In addition, as files are stored in memory, it is possible to pre-create page tables, so that mapping becomes changing a single pointer in a page table to refer to existing page tables [13]. However, this is complicated in that it requires mapping files at the natural granularities of page table structures (e.g., 2MB, 1GB). Another possibility is to reserve a ranges of page-table aligned virtual addresses (again, 2MB, 1GB, etc.) for temproary shared mappings. After mapping a file once, this alignment allows the mapping's page table structures to be directly used by other processes, and when the file is close the range can be re-used for another file.

Persistence management: As memory is persistent, we assume there will be no swapping to disk. Thus, there is no need to track the clean/dirty/referenced status of most memory, which avoids the need for page reclamation algorithms (e.g., clock, 2-queue). The use of files for memory also makes it simple to separate out memory management from persistence [12], in that all data lives in files that can be marked at any time as volatile or persistent, indicating whether they should survive process terminations and system restarts.

Memory locking: Currently letting a device access memory often requires locking the page in memory; even devices that support page faults through an IOMMU incur high penalties [15]. With file-only memory, data is implicitly pinned in memory, as pages are never reclaimed or relocated until the file is explicitly unmapped.

4.2 Physically Based Mappings

We observe that when files are mapped into a process, they generate unique page table entries for every process. While there has been some effort to share page tables entries across processes [6, 22], it is not widespread, partially because it cannot ensure that files are mapped at the same address in every process. This hinders sharing page tables in general.

We observe that processors already manage a single address space shared by all processes: the physical address space. If programs referred to memory using physical addresses (which we do *not* advocate), those addresses would be guaranteed to be common to all processes.

We build on this observation with *physically based mappings*, which are virtual addresses generated algorithmically, such as by adding an offset to the physical address of memory. If the algorithm is the same for all processes, then the virtual address for this memory can be guaranteed to be (a) the same in all processes, and (b) have no collisions as physical memory has no colliding addresses.

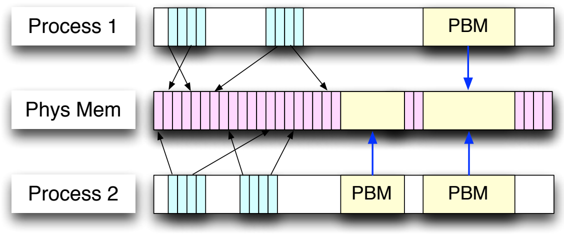


Figure 8: Physically Based Mappings.

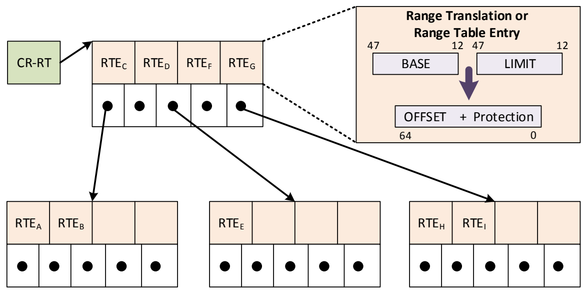


Figure 9: Range table and entries

Having common addresses simplifies sharing page tables because the virtual addresses are the same. Two processes with the same accesses to memory, such as a mapped file, can point to the same sub-tree of a page table as they are guaranteed to map it at the same location. It may be necessary to maintain two sets of page tables to allow different permissions (read vs read/write).

Using physically based mapping requires allocating memory objects contiguously in physical memory. However, we already observe that most programs do not allocate their entire data set in one large contiguous chunk, but instead call an allocator repeatedly to allocate small regions, and internally the allocator repeatedly call the OS to allocate ranges of memory. These allocations are candidates for physically based mappings.

Physically based mappings provide some benefits even on existing hardware. However, their support for simple mapping can provide much better benefits with additional hardware support.

4.3 Hardware Optimizations

Enabling O(1) memory operations in software can be helped by improved hardware support for memory. With current page-base virtual memory, processors fundamentally require per-page operations to create page tables or handle TLB misses, and hence O(1) memory is not completely achievable. For example, in our experiments we observed that it was faster to make a `read()` system call to read 16KB than to access data *already mapped into a process* if it would cause TLB misses.

But, changes to how a processor performs address translation can dramatically reduce the cost of accessing large memory, and can benefit from OS support for O(1) memory. In particular, proposed *range translations* [9] allow mapping an arbitrary length of contiguous physical memory to contiguous virtual memory using a fixed size structure comprising a base, limit, and offset. With this structure, any contiguous physical range of memory can be added to the address space by populating a single entry in a *range table* (analogous to a page table, but a different data structure). The processor adds this entry to a *range TLB* on a reference to any address in the range. Figure 9 shows how the table efficiently represents translations.

We observe that range translations complement our proposed file-only memory: memory managed as extents in a file can be efficiently mapped by assigning one virtual memory range to each extent. Similarly, unmapping a file can be a single operation to update the range table and shoot down the entry in the TLB. Thus, we believe that in concert with advances on the operating system for O(1) memory operations, there must be concurrent advances in processor memory management to support efficient software designs.

5 Conclusions

Forthcoming memory technologies allow, for the first time, systems with more than enough memory. Having grown up in a time of scarcity, current operating systems are not prepared to deal with this new bounty. We conjecture that how systems manage memory should be reinvestigated and rethought to achieve O(1) operations, from processors, through the operating system, and up to language runtimes and applications. There is a new opportunity to rethink how memory is managed and used, for trading space for time, and to shape how memory is considered.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3), Oct. 2007.

[2] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[3] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, June 1994.

[4] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4), Nov. 1994.

[5] J. Corbet. Page faults in user space: Madv_userfault, remap_anon_range(), and userfaultfd(), Oct. 2014.

[6] X. Dong, S. Dwarkadas, and A. L. Cox. Shared address translation revisited. In *Proceedings of the 11th European Conference on Computer Systems*, 2016.

[7] S. R. Dulloor, S. K. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of EuroSys*, 2014.

[8] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojici. Beyond processor-centric operating systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2015.

[9] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Unsal. Range translations for fast virtual memory. *IEEE Micro Special Issue: Micro's Top Picks from Architecture Conferences*, 3(3), May/June 2016.

[10] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. In A. R. Hurson, L. L. Miller, and S. H. Pakzad, editors, *Parallel Architectures for Database Systems*, pages 408–416. 1989.

[11] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[12] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *Proceedings of the Usenix Annual Technical Conference*, June 2012.

[13] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 580–591, 2015.

[14] Intel Corp. 5-level paging and 5-level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, Dec. 2016.

[15] Intel Corp. Intel virtualization technology for directed i/o, rev 2.4. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, June 2016.

[16] S. Kannan, A. Gavrilovska, and K. Schwan. pvm: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the 11th European Conference on Computer Systems*, 2016.

[17] J. Kim and Y. Kim. Hbm: Memory solution for bandwidth-hungry processors. In *Hot Chips Tutorials*, 2016.

[18] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[19] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Z hao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1), 2010.

[20] H. T. Lue, S. H. Chen, Y. H. Shih, K. Y. Hsieh, and C. Y. Lu. Overview of 3d nand flash and progress of vertical gate (vg) architecture. In *Proeedings of the 11th International Conference on Solid-State and Integrated Circuit Technology*, Oct 2012.

[21] D. Magenheimer, C. Mason, D. Mccracken, and K. Hackel. Transcendent memory and linux. In *Ottawa Linux Symposium*, 2009.

[22] D. McCracken. Shared page tables in the linux kernel. In *Proceedings of the Ottawa Linux Symposium*, 2003.

[23] T. P. Morgan. Intel shows off 3d xpoint memory performance. https://www.nextplatform.com/2015/10/28/intel-shows-off-3d-xpoint-memory-performance/, Oct. 2015.

[24] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation*, 2002.

[25] I. Newsroom. Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015.

[26] J. T. Pawlowski. Memory as we approach a new horizon. In *Hot Chips Tutorials*, 2016.

[27] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the Usenix Conference on File and Storage Technologies*, 2014.

[28] TechInsights. Technology roadmap of dram for three major manufacturers. https://www.techinsights.com/uploadedFiles/Public_Website/Content_-_Primary/Marketing/2013/DRAM_Roadmap/Report/TechInsights-DRAM-ROADMAP-052013-LONG-version.pdf, 2013.