Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management

Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase*, Onur Mutlu,
Phillip B. Gibbons[†], Michael A. Kozuch[†], Todd C. Mowry, Trishul Chilimbi*

Carnegie Mellon University **[†]Intel Labs Pittsburgh** ***Microsoft Research**

{vseshadr,gpekhime,tcm}@cs.cmu.edu, onur@cmu.edu
{phillip.b.gibbons,michael.a.kozuch}@intel.com, {olruwase,trishulc}@microsoft.com

Abstract

Many recent works propose mechanisms demonstrating the potential advantages of managing memory at a fine (e.g., cache line) granularity—e.g., fine-grained deduplication and fine-grained memory protection. Unfortunately, existing virtual memory systems track memory at a larger granularity (e.g., 4 KB pages), inhibiting efficient implementation of such techniques. Simply reducing the page size results in an unacceptable increase in page table overhead and TLB pressure.

We propose a new virtual memory framework that enables efficient implementation of a variety of fine-grained memory management techniques. In our framework, each virtual page can be mapped to a structure called a page overlay, in addition to a regular physical page. An overlay contains a subset of cache lines from the virtual page. Cache lines that are present in the overlay are accessed from there and all other cache lines are accessed from the regular physical page. Our page-overlay framework enables cache-line-granularity memory management without significantly altering the existing virtual memory framework or introducing high overheads.

We show that our framework can enable simple and efficient implementations of seven memory management techniques, each of which has a wide variety of applications. We quantitatively evaluate the potential benefits of two of these techniques: overlay-on-write and sparse-data-structure computation. Our evaluations show that overlay-on-write, when applied to fork, can improve performance by 15% and reduce memory capacity requirements by 53% on average compared to traditional copy-on-write. For sparse data computation, our framework can outperform a state-of-the-art software-based sparse representation on a number of real-world sparse matrices. Our framework is general, powerful, and effective in enabling fine-grained memory management at low cost.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
©2015 ACM. ISBN 978-1-4503-3402-0/15/06 $15.00
DOI: http://dx.doi.org/10.1145/2749469.2750379

1. Introduction

Virtual memory [18, 22, 31] is one of the most significant inventions in the field of computer architecture. In addition to supporting several core operating system functions such as memory capacity management, inter-process protection, and data sharing, virtual memory also enables simple implementation of several techniques that significantly improve performance—e.g., copy-on-write [23] and page flipping [4, 8]. Due to its simplicity and power, virtual memory is used in almost all modern high performance systems.

Despite its success, the fact that virtual memory, as it is implemented today, tracks memory at a coarse (page) granularity is a major shortcoming. Tracking memory at a page granularity 1) introduces significant inefficiency in many existing techniques (e.g., copy-on-write), and 2) makes it difficult for software to implement previously-proposed techniques such as fine-grained deduplication [12, 24], fine-granularity data protection [61, 62], cache-line-level compression [21, 30, 38], and metadata management [36, 63]. For example, consider the copy-on-write technique, where multiple virtual pages that contain the *same* data are mapped to a single physical page: When even just a single byte within any of the virtual pages is modified, the operating system creates a copy of the *entire* physical page. This operation not only wastes memory space, but also incurs high latency, bandwidth, and energy [44].

Managing memory at a finer granularity than pages enables several techniques that can significantly boost system performance and efficiency. However, simply reducing the page size results in an unacceptable increase in virtual-to-physical mapping table overhead and TLB pressure. Prior works to address this problem either rely on software techniques [24] (high performance overhead), propose hardware support specific to a particular application [36, 47, 63] (low value for cost), or significantly modify the structure of existing virtual memory [12, 62] (high cost for adoption).

We ask the question, *can we architect a generalized framework that can enable a wide variety of fine-grain management techniques, without significantly altering the existing virtual memory framework?* In response, we present a new virtual memory (VM) framework that augments the existing VM framework with a new concept called *page overlays.*

Figure 1 pictorially compares the existing virtual memory framework with our proposed framework. In the existing

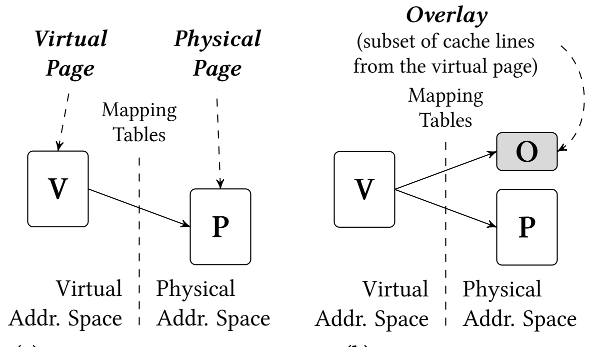


Figure 1: Overview of our proposed framework

framework (Figure 1a), each virtual page of a process can be mapped to *at most one* physical page. The system uses a set of mapping tables to determine the mapping between virtual pages and physical pages. In our proposed framework (Figure 1b), each virtual page may additionally be mapped to a structure called an *overlay*. At a high level, an overlay of a virtual page contains *only* a subset of cache lines from the virtual page. When a virtual page has both a physical page and an overlay mapping, the access semantics is as follows: any cache line that is present in the overlay is accessed from there; all other cache lines are accessed from the physical page.

Our new framework with the simple access semantics has two main advantages over previous approaches to enable fine-grained memory management.

First, **our framework is general and powerful**. It seamlessly enables efficient implementation of a variety of techniques to improve overall system performance, reduce memory capacity requirements, and enhance system security. As an illustration, in the copy-on-write technique, when one of the virtual pages mapped to a single physical page receives a write, instead of creating a full copy of the physical page, our mechanism simply creates an overlay for the virtual page with just the modified cache line. We refer to this mechanism as *overlay-on-write*. In comparison to copy-on-write, overlay-on-write 1) significantly reduces the amount of redundancy in memory and 2) removes the copy operation from the critical path of execution. In all, we describe seven different techniques that can be efficiently implemented on top of our framework. Table 1 summarizes these techniques, illustrating the benefits of our framework over previously proposed mechanisms. Section 5 describes these techniques in more detail.

The second main advantage of our framework is that **it largely retains the structure of the existing virtual memory framework**. This is very important as it not only enables a low overhead implementation of our framework, but also enables the system to treat overlays as an inexpensive feature that can be turned on or off depending on how much the specific system benefits from overlays (backward-compatibility).

Implementing the semantics of our framework presents us with a number of design questions—e.g., how and when are overlays created, how are cache lines within overlays located, how are the overlays stored compactly in main memory? We show that the naïve answers to these questions result in significant performance overhead. In this work, we make several observations that result in a simple and low-overhead design of our framework. Section 3 discusses the shortcomings of naïve approaches and presents an overview of our proposed design. Section 4 describes our final design and the associated implementation cost in detail.

Table 1: Summary of techniques that can be efficiently implemented on top of our proposed page-overlay framework

Technique (Section)	**Mechanism (high-level)**	**Benefits Over the State-of-the-art**
Overlay-on-Write (§2.2, §5.1)	Create an overlay on write and store modified cache line(s) in overlay.	Compared to copy-on-write [23], 1) reduces memory redundancy, 2) removes copy from critical path.
Sparse Data Structures (§5.2)	Use overlays to store non-zero cache lines. Enable computation over overlays (i.e., non-zero data).	Compared to software representations (e.g., [20]), 1) more efficient when sparsity is low, 2) faster dynamic updates. Compared to compression [38], more efficient computation.
Fine-grained Deduplication (§5.3.1)	For pages with similar data, use single base physical page and store differences (deltas) in overlays.	Enables efficient hardware support for difference engine [24]. In contrast to HICAMP [12], avoids significant changes to both existing virtual memory structure and programming model.
Checkpointing [19, 59] (§5.3.2)	Collect memory updates in overlays. Only back up the overlays.	Compared to backing up pages, reduces write bandwidth to backing store, enabling faster, more frequent checkpointing.
Virtualizing Speculation (§5.3.3)	Store speculative memory updates in overlay. Commit/discard overlay if speculation succeeds/fails.	1) Supports potentially unbounded speculation (e.g., [3]), 2) no software handlers needed to handle eviction of any speculatively modified cache line [15].
Fine-grained Metadata Management (§5.3.4)	Use overlays to store fine-grained metadata (e.g., protection bits) for each virtual page.	Compared to prior works [36, 62, 63], 1) easier integration with existing virtual memory framework, 2) no metadata-specific hardware changes required.
Flexible Super-pages (§5.3.5)	Use overlays at higher-level page tables to remap different parts of super-pages.	Enables 1) more flexible super-page mappings, 2) page or cache-line-granularity copy-on-write, 3) multiple protection domains within a super-page.

We quantitatively evaluate the benefits of our framework using two of the seven techniques: 1) overlay-on-write, and 2) an efficient mechanism for sparse data structure computations. Our evaluations show that overlay-on-write (when applied to `fork` [1]) improves performance by 15% and reduces memory capacity requirements by 53% compared to the traditional copy-on-write mechanism. For sparse-matrix-vector multiplication, our framework consistently outperforms a baseline dense representation and can even outperform CSR [27], a state-of-the-art software-based sparse representation, on over a third of the large real-world sparse matrices from [17]. More importantly, unlike many software representations, our implementation allows the sparse data structure to be *dynamically* updated, which is typically a costly and complex operation with a software-based representation. Section 5 discusses these results in more detail.

In summary, this paper makes the following contributions.

- We propose a new virtual memory framework that augments the existing VM framework with a new concept called *page overlays*.
- We show that our framework provides powerful access semantics, seamlessly enabling efficient implementation of seven fine-grained memory management techniques.
- We present in detail a simple, low-overhead design of our proposed framework. Our design adds negligible logic on the critical path of regular memory accesses.
- We quantitatively evaluate our framework with two fine-grained memory management techniques, showing that it significantly improves overall system performance and reduces memory consumption.

2. Page Overlays: Motivation

We first present a detailed overview of the semantics of our proposed virtual memory framework. While our proposed framework enables efficient implementation of each of the techniques in Table 1, we will use one such technique—overlay-on-write—to illustrate the power of our framework.

2.1. Overview of Semantics of Our Framework

As illustrated in Figure 1b, in our proposed framework, each virtual page can be mapped to two entities: a regular physical page and a *page overlay*. There are two aspects to a page overlay. First, unlike the physical page, which has the same size as the virtual page, the overlay of a virtual page contains only a *subset of cache lines* from the page, and hence is smaller in size than the virtual page. Second, when a virtual page has both a physical page and an overlay mapping, we define the access semantics such that any cache line that is present in the overlay is accessed from there. Only cache lines that are *not* present in the overlay are accessed from the physical page, as shown in Figure 2 for the simplified case of a page with four cache lines. In the figure, the virtual page is mapped to both a physical page and an overlay, but the overlay contains only cache lines C1 and C3. Based on our semantics, accesses to C1 and C3 are mapped to the overlay, and the remaining cache lines are mapped to the physical page.

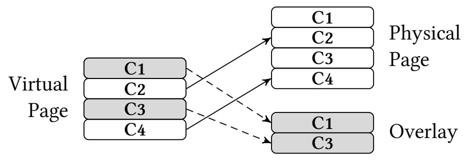
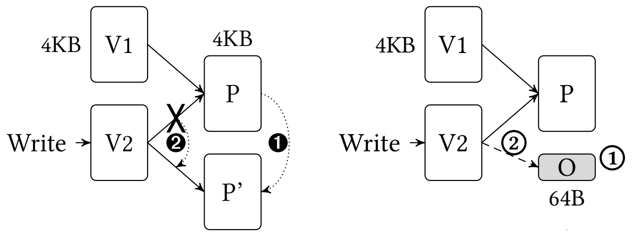


Figure 2: Semantics of our proposed framework

2.2. Overlay-on-write: A More Efficient Copy-on-write

Copy-on-write is a widely-used primitive in many different applications including process forking [1], virtual machine cloning [33], memory deduplication [58], OS speculation [11, 37, 60], and memory checkpointing [19, 52, 59]. Figure 3a shows how the copy-on-write technique works. Initially, two (or more) virtual pages that contain the same data are mapped to the same physical page (P). When one of the pages receives a write (resulting in a data divergence), the OS handles it in two steps. It first identifies a new physical page (P') and *copies* the contents of the original physical page (P) to the new page ❶. Second, it *remaps* the virtual page that received the write (V2) to the new physical page ❷. This remapping operation typically requires a TLB shootdown [7, 55].



(a) Copy-on-Write **(b) Overlay-on-Write**

Figure 3: Copy-on-Write vs. Overlay-on-Write

Despite its wide applications, copy-on-write is expensive and inefficient for two reasons. First, both the copy operation and the remap operation are on the critical path of the write. Both operations incur high latency [7, 41, 44, 55, 57]. The copy operation consumes high memory bandwidth, potentially degrading the performance of other applications sharing the memory bandwidth [44]. Second, even if only a single cache line is modified within the virtual page, the system needs to create a full copy of the *entire* physical page, leading to inefficient use of memory capacity.

Our framework enables a faster, more efficient mechanism (as shown in Figure 3b). When multiple virtual pages share the same physical page, the OS explicitly indicates to the hardware, through the page tables, that the pages should be copied-on-write. When one of the pages receives a write, our framework first creates an overlay that contains *only the modified cache line* ①. It then maps the overlay to the virtual page that received the write ②. We refer to this mechanism as *overlay-on-write*. Overlay-on-write has many benefits over copy-on-write. First, it avoids the need to copy the entire physical page before the write operation, thereby significantly reducing the latency on the critical path of execution (as well as the associated increase in memory bandwidth and energy). Second, it allows the system to eliminate significant redundancy in the data stored in main memory because only the overlay lines need to be stored, compared to a full page with copy-on-write. Finally, as we describe in Section 4.3.3, our design exploits the fact that only a single cache line is remapped from the source physical page to the overlay to significantly reduce the latency of the remapping operation.

As mentioned before, copy-on-write has a wide variety of applications [1, 11, 19, 33, 37, 52, 58, 59, 60]. Overlay-on-write, being a faster and more efficient alternative to copy-on-write, can significantly benefit all these applications.

2.3. Benefits of the Overlay Semantics

For most of the techniques in Table 1, our framework offers two distinct benefits over the existing VM framework. First, our framework **reduces the amount of work that the system has to do**, thereby improving system performance. For instance, in the overlay-on-write and sparse data structure (Section 5.2) techniques, our framework reduces the amount of data that needs to be copied/accessed. Second, our framework **enables significant reduction in memory capacity requirements**. Each overlay contains only a subset of cache lines from the virtual page, so the system can reduce overall memory consumption by compactly storing the overlays in main memory—i.e., for each overlay, store only the cache lines that are actually present in the overlay. We quantitatively evaluate these benefits in Section 5 using two techniques and show that our framework is effective.

3. Page Overlays: Design Overview

While our framework imposes simple access semantics, there are several key challenges to efficiently implement the proposed semantics. In this section, we first discuss these challenges with an overview of how we address them. We then provide a full overview of our proposed mechanism that addresses these challenges, thereby enabling a simple, efficient, and low-overhead design of our framework.

3.1. Challenges in Implementing Page Overlays

Challenge 1: *Checking if a cache line is part of the overlay.* When the processor needs to access a virtual address, it must first check if the accessed cache line is part of the overlay. Since most modern processors use a physically-tagged L1 cache, this check is on the critical path of the L1 access. To address this challenge, we associate each virtual page with a bit vector that represents which cache lines from the virtual page are part of the overlay. We call this bit vector the *overlay bit vector* (`OBitVector`). We cache the `OBitVector` in the processor TLB, thereby enabling the processor to quickly check if the accessed cache line is part of the overlay.

Challenge 2: *Identifying the physical address of an overlay cache line.* If the accessed cache line is part of the overlay (i.e., it is an *overlay cache line*), the processor must quickly determine the physical address of the overlay cache line, as this address is required to access the L1 cache. The simple approach to address this challenge is to store in the TLB the base address of the region where the overlay is stored in main memory (we refer to this region as the *overlay store*). While this may enable the processor to identify each overlay cache line with a unique physical address, this approach has three shortcomings when overlays are stored compactly in main memory.

First, the overlay store (in main memory) does *not* contain all the cache lines from the virtual page. Therefore, the processor must perform some computation to determine the address of the accessed overlay cache line. This will delay the L1 access. Second, most modern processors use a virtually-indexed physically-tagged L1 cache to partially overlap the L1 cache access with the TLB access. This technique requires the virtual index and the physical index of the cache line to be the same. However, since the overlay is smaller than the virtual page, the overlay physical index of a cache line will likely not be the same as the cache line's virtual index. As a result, the cache access will have to be delayed until the TLB access is complete. Finally, inserting a new cache line into an overlay is a relatively complex operation. Depending on how the overlay is represented in main memory, inserting a new cache line into an overlay can potentially change the addresses of other cache lines in the overlay. Handling this scenario requires a likely complex mechanism to ensure the consistency of these other cache lines.

In our design, we address this challenge by using two different addresses for each overlay—one to address the processor caches, called the *Overlay Address*, and another to address main memory, called the *Overlay Memory Store Address*. As we will describe shortly, this *dual-address design* enables the system to manage the overlay in main memory independently of how overlay cache lines are addressed in the processor caches, thereby overcoming the above three shortcomings.

Challenge 3: *Ensuring the consistency of the TLBs.* In our design, since the TLBs cache the `OBitVector`, when a cache line is moved from the physical page to the overlay or vice versa, any TLB that has cached the mapping for the corresponding virtual page should update its mapping to reflect the cache line remapping. The naïve approach to addressing this challenge is to use a TLB shootdown [7, 55], which is expensive [41, 57]. Fortunately, in the above scenario, the TLB mapping is updated only for a *single cache line* (rather than an entire virtual page). We propose a simple mechanism that exploits this fact and uses the cache coherence protocol to keep the TLBs coherent (Section 4.3.3).

3.2. Overview of Our Design

A key aspect of our dual-address design, mentioned above, is that the address to access the cache (the *Overlay Address*) is taken from an *address space* where the size of each overlay is the *same* as that of a regular physical page. This enables our design to seamlessly address Challenge 2 (overlay cache line address computation), without incurring the drawbacks of the naïve approach to address the challenge (described in Section 3.1). The question is, *from what address space is the Overlay Address taken?*

Towards answering this question, we observe that only a small fraction of the physical address space is backed by main memory (DRAM) and a large portion of the physical address space is *unused*, even after a portion is consumed for memory-mapped I/O [40] and other system constructs. We propose to use this unused physical address space for the overlay cache address and refer to this space as the *Overlay Address Space*.[1]

Figure 4 shows the overview of our design. There are three address spaces: the virtual address space, the physical address space, and the main memory address space. The main memory address space is split between regular physical pages and the *Overlay Memory Store* (OMS), a region where the overlays are stored compactly. In our design, to associate a virtual page with an overlay, the virtual page is first mapped to a full size page in the overlay address space using a direct mapping without any translation or indirection (Section 4.1). The overlay page is in turn mapped to a location in the OMS using a mapping table stored in the memory controller (Section 4.2). We will describe the figure in more detail in Section 4.

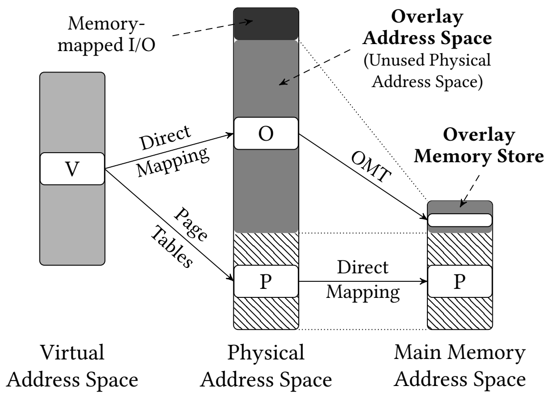


Figure 4: Overview of our design. "Direct mapping" indicates that the corresponding mapping is implicit in the source address. OMT = Overlay Mapping Table (Section 4.2).

3.3. Benefits of Our Design

There are three main benefits of our high-level design. First, our approach makes no changes to the way the existing VM framework maps virtual pages to physical pages. This is very important as the system can treat overlays as an inexpensive feature that can be turned on only when the application benefits from it. Second, as mentioned before, by using two distinct addresses for each overlay, our implementation decouples the way the caches are addressed from the way overlays are stored in main memory. This enables the system to treat overlay cache accesses very similarly to regular cache accesses, and consequently requires very few changes to the existing hardware structures (e.g., it works seamlessly with virtually-indexed physically-tagged caches). Third, as we will describe in the next section, in our design, the Overlay Memory Store (in main memory) is accessed only when an access completely misses in the cache hierarchy. This 1) greatly reduces the number of operations related to managing the OMS, 2) reduces the amount of information that needs to be cached in the processor TLBs, and 3) more importantly, enables the memory controller to completely manage the OMS with *minimal* interaction with the OS.

[1]In fact, a prior work, the Impulse Memory Controller [9], uses the unused physical address space to communicate gather/scatter access patterns to the memory controller. The goal of Impulse [9] is different from ours, and it is difficult to use the design proposed by Impulse to enable fine-granularity memory management.

4. Page Overlays: Detailed Design

To recap our high-level design (Figure 4), each virtual page in the system is mapped to two entities: 1) a regular physical page, which in turn directly maps to a page in main memory, and 2) an overlay page in the Overlay Address space (which is not directly backed by main memory). Each page in this space is in turn mapped to a region in the Overlay Memory Store, where the overlay is stored compactly. Because our implementation does not modify the way virtual pages are mapped to regular physical pages, we now focus our attention on how virtual pages are mapped to overlays.

4.1. Virtual-to-Overlay Mapping

The virtual-to-overlay mapping maps a virtual page to a page in the Overlay Address space. One simple approach to maintain this mapping information is to store it in the page table and allow the OS to manage the mappings (similar to regular physical pages). However, this increases the overhead of the mapping table and complicates the OS. We make a simple observation and impose a constraint that makes the virtual-to-overlay mapping a direct 1-1 mapping.

Our **observation** is that since the Overlay Address space is part of the *unused* physical address space, it can be *significantly larger* than the amount of main memory. To enable a 1-1 mapping between virtual pages and overlay pages, we impose a simple **constraint** wherein no two virtual pages can be mapped to the same overlay page.

Figure 5 shows how our design maps a virtual address to the corresponding overlay address. Our scheme widens the physical address space such that the overlay address corresponding to the virtual address `vaddr` of a process with ID `PID` is obtained by simply concatenating an overlay bit (set to 1), `PID`, and `vaddr`. Since two virtual pages cannot share an overlay, when data of a virtual page is copied to another virtual page, the overlay cache lines of the source page must be copied into the appropriate locations in the destination page. While this approach requires a slightly wider physical address space than in existing systems, this is a more practical mechanism compared to storing this mapping explicitly in a separate table, which can lead to much higher storage and management overheads than our approach. With a 64-bit physical address space and a 48-bit virtual address space per process, this approach can support 2^{15} different processes.

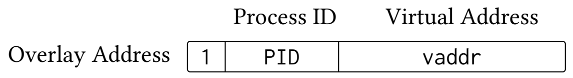


Figure 5: Virtual-to-Overlay Mapping. The MSB indicates if the physical address is part of the Overlay Address space.

Note that a similar approach *cannot* be used to map virtual pages to physical pages due to the *synonym* problem [10], which results from multiple virtual pages being mapped to the same physical page. However, this problem does not occur with the virtual-to-overlay mapping because of the constraint we impose: no two virtual pages can map to the same overlay page. Even with this constraint, our framework enables many applications that can improve performance and reduce memory capacity requirements (Section 5).

4.2. Overlay Address Mapping

Overlay cache lines tagged in the Overlay Address space must be mapped into an Overlay Memory Store location upon eviction. In our design, since there is a 1-1 mapping between a virtual page and an overlay page, we could potentially store this mapping in the page table along with the physical page mapping. However, since many pages may not have an overlay, we store this mapping information in a separate mapping table similar to the page table. This *Overlay Mapping Table* (OMT) is maintained and controlled fully by the memory controller with minimal interaction with the OS. Section 4.4 describes Overlay Memory Store management in detail.

4.3. Microarchitecture and Memory Access Operations

Figure 6 depicts the microarchitectural details of our design. There are three main changes over the microarchitecture of current systems. First (❶ in the figure), main memory is split into two regions that store 1) regular physical pages and 2) the Overlay Memory Store (OMS). The OMS stores both a compact representation of the overlays and the *Overlay Mapping Table* (OMT), which maps each page from the Overlay Address Space to a location in the Overlay Memory Store. At a high level, each OMT entry contains 1) the `OBitVector`, indicating if each cache line within the corresponding page is present in the overlay, and 2) `OMSaddr`, the location of the overlay in the OMS. Second ❷, we augment the memory controller with a cache called the *OMT Cache*, which caches recently accessed entries from the OMT. Third ❸, because the TLB must determine if an access to a virtual address should be directed to the corresponding overlay, we extend each TLB entry to store the `OBitVector`. While this potentially increases the cost of each TLB miss (as it requires the `OBitVector` to be fetched from the OMT), our evaluations (Section 5) show that the performance benefit of using overlays more than offsets this additional TLB fill latency.

To describe the operation of different memory accesses, we use overlay-on-write (Section 2.2) as an example. Let us assume that two virtual pages (V1 and V2) are mapped to the same physical page in the copy-on-write mode, with a few cache lines of V2 already mapped to the overlay. There are three possible operations on V2: 1) a read, 2) a write to a cache line already in the overlay (*simple write*), and 3) a write to a cache line not present in the overlay (*overlaying write*). We now describe each of these operations in detail.

4.3.1. Memory Read Operation. When the page V2 receives a read request, the processor first accesses the TLB with the corresponding page number (`VPN`) to retrieve the physical mapping (`PPN`) and the `OBitVector`. It generates the overlay page number (`OPN`) by determining the overlay address (as shown in Figure 5) and dividing it by the page size—i.e., concatenating the bit 1, the process ID (`PID`) of the process and the `VPN`. Depending on whether the accessed cache line is present in the overlay (as indicated by the corresponding bit in the `OBitVector`), the processor uses either the `PPN` or the `OPN` to generate the L1 cache `tag`. If the access misses in the entire cache hierarchy (L1 through last-level cache), the request is sent to the memory controller. The controller checks if the requested address is part of the overlay address space by checking the overlay bit in the physical address. If so, it looks up the overlay store address (`OMSaddr`) of the corresponding overlay page from the OMT Cache, and computes the exact location of the requested cache line within main memory (as described later in Section 4.4). It then accesses the cache line from the main memory and returns the data to the cache hierarchy.

4.3.2. Simple Write Operation. When the processor receives a write to a cache line already present in the overlay, it simply has to update the cache line in the overlay. The path of this operation is the same as that of the read operation, except the cache line is updated after it is read into the L1 cache.

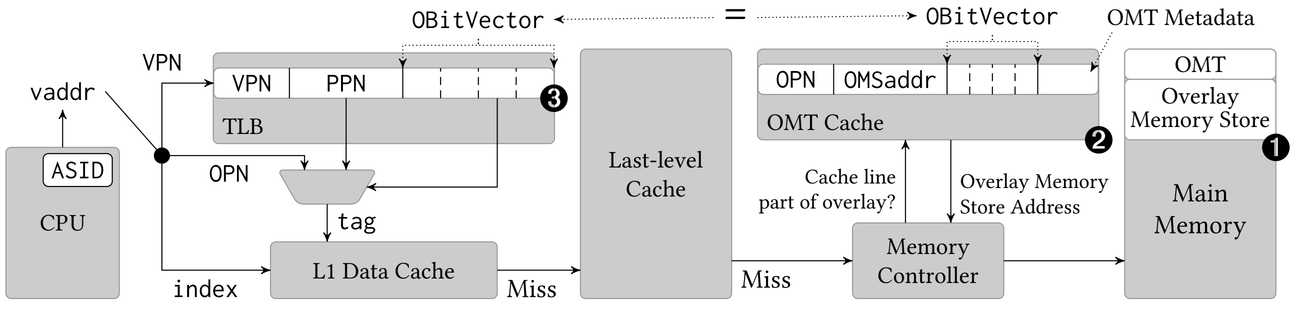


Figure 6: Microarchitectural details of our implementation. The main changes (❶, ❷ and ❸) are described in Section 4.3.

4.3.3. Overlaying Write Operation. An *overlaying write* operation is a write to a cache line that is *not* already present in the overlay. Since the virtual page is mapped to the regular physical page in the copy-on-write mode, the corresponding cache line must be remapped to the overlay (based on our semantics described in Section 2.2). We complete the overlaying write in three steps: 1) copy the data of the cache line in the regular physical page (PPN) to the corresponding cache line in the Overlay Address Space page (OPN), 2) update all the TLBs and the OMT to indicate that the cache line is mapped to the overlay, and 3) process the write operation.

The first step can be completed in hardware by reading the cache line from the regular physical page and simply updating the cache tag to correspond to the overlay page number (or by making an explicit copy of the cache line). Naïvely implementing the second step will involve a TLB shootdown for the corresponding virtual page. However, we exploit three simple facts to use the cache coherence network to keep the TLBs and the OMT coherent: i) the mapping is modified *only* for a single cache line, and not an entire page, ii) the overlay page address can be used to uniquely identify the virtual page since no overlay is shared between virtual pages, and iii) the overlay address is part of the physical address space and hence, part of the cache coherence network. Based on these facts, we propose a new cache coherence message called *overlaying read exclusive*. When a core receives this request, it checks if its TLB has cached the mapping for the virtual page. If so, the core simply sets the bit for the corresponding cache line in the `OBitVector`. The *overlaying read exclusive* request is also sent to the memory controller so that it can update the `OBitVector` of the corresponding overlay page in the OMT (via the OMT Cache). Once the remapping operation is complete, the write operation (the third step) is processed similar to the simple write operation.

Note that after an *overlaying write*, the corresponding cache line (which we will refer to as the *overlay cache line*) is marked dirty. Unlike copy-on-write, which must allocate memory *before* the write operation, our mechanism delays the memory allocation until the eviction of the dirty overlay cache line. This *lazy* allocation of overlay space in main memory significantly improving performance.

4.3.4. Converting an Overlay to a Regular Physical Page. Depending on the technique for which overlays are used, maintaining an overlay for a virtual page may be unnecessary after a point. For example, when using overlay-on-write, if most of the cache lines within a virtual page are modified, maintaining them in an overlay does not provide any advantage. The system may take one of three actions to promote an overlay to a physical page: The *copy-and-commit* action is one where the OS copies the data from the regular physical page to a new physical page and updates the data of the new physical page with the corresponding data from the overlay. The *commit* action updates the data of the regular physical page with the corresponding data from the overlay. The *discard* action simply discards the overlay.

While the *copy-and-commit* action is used with overlay-on-write, the *commit* and *discard* actions are used, for example, in the context of speculation, where our mechanism stores speculative updates in the overlays (Section 5.3.3). After any of these actions, the system clears the `OBitVector` of the corresponding virtual page, and frees the overlay memory store space allocated for the overlay (discussed next in Section 4.4).

4.4. Managing the Overlay Memory Store

The *Overlay Memory Store* (OMS) is the region in main memory where all the overlays are stored. As described in Section 4.3.1, the OMS is accessed *only* when an overlay access completely misses in the cache hierarchy. As a result, there are many simple ways to manage the OMS. One way is to have a small embedded core on the memory controller that can run a software routine that manages the OMS (similar mechanisms are supported in existing systems, e.g., Intel Active Management Technology [26]). Another approach is to let the memory controller manage the OMS by using a full physical page to store each overlay. While this approach will forgo the memory capacity benefit of our framework, it will still obtain the benefit of reducing overall work (Section 2.3).

In this section, we describe a hardware mechanism that obtains both the work reduction and the memory capacity reduction benefits of using overlays. In our mechanism, the memory controller fully manages the OMS with minimal interaction with the OS. Managing the OMS has two key aspects. First, because each overlay contains only a subset of cache lines from the virtual page, we need a *compact representation for the overlay*, such that the OMS contains only cache lines that are actually present in the overlay. Second, the memory controller must manage multiple *overlays of different sizes*. We need a simple mechanism to handle such different sizes and the associated free space fragmentation issues. Although operations that allocate new overlays or relocate existing overlays are slightly more complex, they are triggered only when a dirty overlay cache line is written back to main memory. Therefore, these operations are rare and are not on the critical path of execution.

4.4.1. Compact Overlay Representation. One simple approach to compactly maintain the overlays is to store the cache lines in an overlay in the order in which they appear in the virtual page. While this representation is simple, if a new cache line is inserted into the overlay before other overlay cache lines, then the memory controller must *move* such cache lines to create a slot for the inserted line. This is a read-modify-write operation, which results in significant performance overhead.

We propose an alternative mechanism, in which each overlay is assigned a *segment* in the OMS. The overlay is associated with an array of pointers—one pointer for each cache line in the virtual page. Each pointer either points to the slot within the overlay segment that contains the cache line or is invalid if the cache line is not present in the overlay. We store this metadata in a single cache line at the head of the segment. For segments less than 4KB size, we use 64 5-bit slot pointers and a 32-bit vector indicating the free slots within a segment—total of 352 bits. For a 4KB segment, we do not store any metadata and simply store each overlay cache line at an offset which is the same as the offset of the cache line within the virtual page. Figure 7 shows an overlay segment of size 256B, with only the first and the fourth cache lines of the virtual page mapped to the overlay.

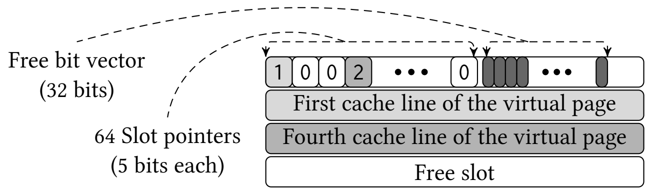


Figure 7: A 256B overlay segment (can store up to three overlay cache lines from the virtual page). The first line stores the metadata (array of pointers and the free bit vector).

4.4.2. Managing Multiple Overlay Sizes. Different virtual pages may contain overlays of different sizes. The memory controller must store them efficiently in the available space. To simplify this management, our mechanism splits the available overlay space into segments of 5 fixed sizes: 256B, 512B, 1KB, 2KB, and 4KB. Each overlay is stored in the smallest segment that is large enough to store the overlay cache lines. When the memory controller requires a segment for a new overlay or when it wants to migrate an existing overlay to a larger segment, the controller identifies a free segment of the required size and updates the OMSaddr of the corresponding overlay page with the base address of the new segment. Individual cache lines are allocated their slots within the segment as and when they are written back to main memory.

4.4.3. Free Space Management. To manage the free segments within the Overlay Memory Store, we use a simple linked-list based approach. For each segment size, the memory controller maintains a memory location or register that points to a free segment of that size. Each free segment in turn stores a pointer to another free segment of the same size or an invalid pointer denoting the end of the list. If the controller runs out of free segments of a particular size, it obtains a free segment of the next higher size and splits it into two. If the controller runs out of free 4KB segments, it requests the OS for an additional set of 4KB pages. During system startup, the OS proactively allocates a chunk of free pages to the memory controller. To reduce the number of memory operations needed to manage free segments, we use a *grouped-linked-list* mechanism, similar to the one used by some file systems [48].

4.4.4. The Overlay Mapping Table (OMT) and the OMT Cache. The OMT maps pages from the Overlay Address Space to a specific segment in the Overlay Memory Store. For each page in the Overlay Address Space (i.e., for each OPN), the OMT contains an entry with the following pieces of information: 1) the OBitVector, and 2) the Overlay Memory Store Address (OMSaddr), pointing to the segment that stores the overlay. To reduce the storage cost of the OMT, we store it hierarchically, similar to the virtual-to-physical mapping tables. The memory controller maintains the root address of the hierarchical table in a register.

The OMT Cache stores the following details regarding recently-accessed overlays: the OBitVector, the OMSaddr, and the overlay segment metadata (stored at the beginning of the segment). To access a cache line from an overlay, the memory controller consults the OMT Cache with the overlay page number (OPN). In case of a hit, the controller acquires the necessary information to locate the cache line in the overlay memory store using the overlay segment metadata. In case of a miss, the controller performs an OMT walk (similar to a page table walk) to look up the corresponding OMT entry, and inserts it in the OMT Cache. It also reads the overlay segment metadata and caches it in the OMT cache entry. The controller may modify entries of the OMT, as and when overlays are updated. When such a modified entry is evicted from the OMT Cache, the memory controller updates the corresponding OMT entry in memory.

4.5. Hardware Cost and OS Changes

There are three sources of hardware overhead in our design: 1) the OMT Cache, 2) wider TLB entries (to store the OBitVector), and 3) wider cache tags (due to the wider physical address space). Each OMT Cache entry stores the overlay page number (48 bits), the Overlay Memory Store address (48 bits), the overlay bit vector (64 bits), the array of pointers for each cache line in the overlay page (64*5 = 320 bits), and free list bit vector for the overlay segment (32 bits). In all, each entry consumes 512 bits. The size of a 64-entry OMT Cache is therefore 4KB. Each TLB entry is extended with the 64-bit OBitVector. Across the 64-entry L1 TLB and 1024-entry L2 TLB, the overall cost extending the TLB is 8.5KB. Finally, assuming each cache tag entry requires an additional 16 bits for each tag, across a 64KB L1 cache, 512KB L2 cache and a 2MB L3 cache, the cost of extending the cache tags to accommodate a wider physical address is 82KB. Thus, the overall hardware storage cost is 94.5KB. Note that the additional bits required for each tag entry can be reduced by restricting the portion of the virtual address space that can have overlays, resulting in even lower overall hardware cost.

The main change to the OS in our implementation is related to the Overlay Memory Store management. The OS interacts with the memory controller to dynamically partition the available main memory space between regular physical pages and the OMS. As mentioned in Section 4.4, this interaction is triggered only when a dirty overlay cache line is written back to main memory and the memory controller is out of Overlay Memory Store space. This operation is rare and is off the critical path. In addition, the operation can be run on a spare core and *does not* have to stall any actively running hardware thread. Note that some of the applications of page overlays may require additional OS or application support, as noted in the sections describing the applications (under Section 5).

Processor	2.67 GHz, single issue, out-of-order, 64 entry instruction window, 64B cache lines
TLB	4K pages, 64-entry 4-way associative L1 (1 cycle), 1024-entry L2 (10 cycles), TLB miss = 1000 cycles
L1 Cache	64KB, 4-way associative, tag/data latency = 1/2 cycles, parallel tag/data lookup, LRU policy
L2 Cache	512KB, 8-way associative, tag/data latency = 2/8 cycles, parallel tag/data lookup, LRU policy
Prefetcher	Stream prefetcher [34, 51], monitor L2 misses and prefetch into L3, 16 entries, degree = 4, distance = 24
L3 Cache	2MB, 16-way associative, tag/data latency = 10/24 cycles, serial tag/data lookup, DRRIP [28] policy
DRAM Controller	Open row, FR-FCFS drain when full [35], 64-entry write buffer, 64-entry OMT cache, miss latency = 1000 cycles
DRAM and Bus	DDR3-1066 MHz [29], 1 channel, 1 rank, 8 banks, 8B-wide data bus, burst length = 8, 8KB row buffer

Table 2: Main parameters of our simulated system

5. Applications and Evaluations

In this section, we describe seven techniques enabled by our framework, and quantitatively evaluate two of them. For our performance evaluations, we use a modified version of Memsim [2, 45], a multi-core simulator that models out-of-order cores coupled with a DDR3-1066 [29] DRAM simulator. All systems use a three-level cache hierarchy with a uniform 64B cache line size. We do not enforce inclusion in any level of the hierarchy. We use the state-of-the-art DRRIP cache replacement policy [28] for the last-level cache. All our evaluated systems use an aggressive multi-stream prefetcher [51] similar to the one implemented in IBM Power 6 [34]. Table 2 lists the main configuration parameters in detail.

5.1. Overlay-on-write

As discussed in Section 2.2, overlay-on-write is a more efficient version of copy-on-write [23]: when multiple virtual pages share the same physical page in the copy-on-write mode and one of them receives a write, overlay-on-write simply moves the corresponding cache line to the overlay and updates the cache line in the overlay.

We compare the performance of overlay-on-write with that of copy-on-write using the `fork` [1] system call. `fork` is a widely-used system call with a number of different applications including creating new processes, creating stateful threads in multi-threaded applications, process testing/debugging [13, 14, 52], and OS speculation [11, 37, 60]. Despite its wide applications, `fork` is one of the most expensive system calls [42]. When invoked, `fork` creates a child process with an identical virtual address space as the calling process. `fork` marks all the pages of both processes as copy-on-write. As a result, when any such page receives a write, the copy-on-write mechanism must copy the whole page and remap the virtual page before it can proceed with the write.

Our evaluation models a scenario where a process is checkpointed at regular intervals using the `fork` system call. While we can test the performance of fork with any application, we use a subset of benchmarks from the SPEC CPU2006 benchmark suite [16]. Because the number of pages copied depends on the *write working set* of the application, we pick benchmarks with three different types of write working sets: 1) benchmarks with low write working set size, 2) benchmarks for which almost all cache lines within each modified page are updated, and 3) benchmarks for which only a few cache line within each modified page are updated. We pick five benchmarks for each type. For each benchmark, we fast forward the execution to its representative portion (determined using Simpoint [46]), run the benchmark for 200 million instructions (to warm up the caches), and execute a `fork`. After the `fork`, we run the parent process for another 300 million instructions, while the child process idles.[2]

Figure 8 plots the amount of additional memory consumed by the parent process using copy-on-write and overlay-on-write for the 300 million instructions after the `fork`. Figure 9 plots the performance (cycles per instruction) of the two mechanisms during the same period. We group benchmarks based on their type. We draw three conclusions.

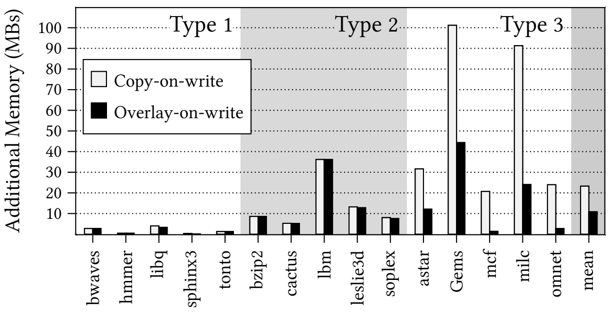


Figure 8: Additional memory consumed after a `fork`

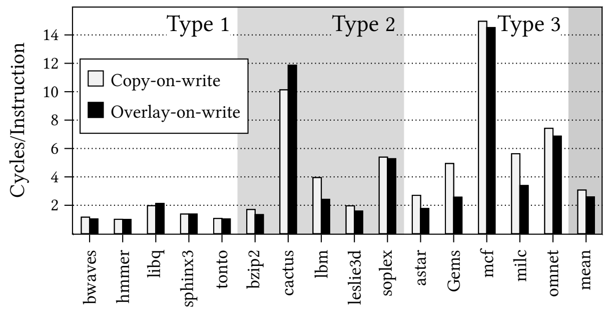


Figure 9: Performance after a `fork` (lower is better)

First, benchmarks with low write working set (Type 1) consume very little additional memory after forking (Figure 8). As a result, there is not much difference in the performance of copy-on-write and that of overlay-on-write (Figure 9).

[2]While 300 million instructions might seem low, several prior works (e.g., [13, 14]) argue for even shorter checkpoint intervals (10-100 million instructions).

Second, for benchmarks of Type 2, both mechanisms consume almost the same amount of additional memory. This is because for these benchmarks, almost all cache lines within every modified page are updated. However, with the exception of *cactus*, overlay-on-write significantly improves performance for this type of applications. Our analysis shows that the performance trends can be explained by the distance in time when cache lines of each page are updated by the application. When writes to different cache lines within a page are close in time, copy-on-write performs better than overlay-on-write. This is because copy-on-write fetches *all* the blocks of a page with high memory-level parallelism. On the other hand, when writes to different cache lines within a page are well separated in time, copy-on-write may 1) unnecessarily pollute the L1 cache with all the cache lines of the copied page, and 2) increase write bandwidth by generating two writes for each updated cache line (once when it is copied and again when the application updates the cache line). Overlay-on-write has neither of these drawbacks, and hence significantly improves performance over copy-on-write.

Third, for benchmarks of Type 3, overlay-on-write significantly reduces the amount of additional memory consumed compared to copy-on-write. This is because the write working set of these applications are spread out in the virtual address space, and copy-on-write unnecessarily copies cache lines that are actually *not* updated by the application. Consequently, overlay-on-write significantly improves performance compared to copy-on-write for this type of applications.

In summary, across all the benchmarks, overlay-on-write reduces additional memory capacity requirements by 53% and improves performance by 15% compared to copy-on-write. Given the wide applicability of the `fork` system call, and the copy-on-write technique in general, we believe overlay-on-write can significantly benefit a variety of such applications.

5.2. Representing Sparse Data Structures

A *sparse* data structure is one with a significant fraction of zero values, e.g., a sparse matrix. Since only non-zero values typically contribute to computation, prior work developed many software representations for sparse data structures (e.g., [20, 27]). One popular software representation of a sparse matrix is the Compressed Sparse Row (CSR) format [27]. To represent a sparse matrix, CSR stores only the non-zero values in an array, and uses two arrays of index pointers to identify the location of each non-zero value within the matrix.

While CSR efficiently stores sparse matrices, the additional index pointers maintained by CSR can result in inefficiency. First, the index pointers lead to significant additional memory capacity overhead (roughly 1.5 times the number of non-zero values in our evaluation—each value is 8 bytes, and each index pointer is 4 bytes). Second, any computation on the sparse matrix requires additional memory accesses to fetch the index pointers, which degrades performance.

Our framework enables a very efficient hardware-based representation for a sparse data structure: all virtual pages of the data structure map to a zero physical page and each virtual page is mapped to an overlay that contains only the *non-zero cache lines* from that page. To avoid computation over zero cache lines, we propose a new computation model that enables the software to *perform computation only on overlays*. When overlays are used to represent sparse data structures, this model enables the hardware to efficiently perform a computation only on non-zero cache lines. Because the hardware is aware of the overlay organization, it can efficiently prefetch the overlay cache lines and hide the latency of memory accesses significantly.

Our representation stores non-zero data at a cache line granularity. Hence, the performance and memory capacity benefits of our representation over CSR depends on the spatial locality of non-zero values within a cache line. To aid our analysis, we define a metric called *non-zero value locality* ($\mathcal{L}$), as the average number of non-zero values in each non-zero cache line. On the one hand, when non-zero values have poor locality ($\mathcal{L} \approx 1$), our representation will have to store a significant number of zero values and perform redundant computation over such values, degrading both memory capacity and performance over CSR, which stores and performs computation on only non-zero values. On the other hand, when non-zero values have high locality ($\mathcal{L} \approx 8$—e.g., each cache line stores 8 double-precision floating point values), our representation is significantly more efficient than CSR as it stores significantly less metadata about non-zero values than CSR. As a result, it outperforms CSR both in terms of memory capacity and performance.

We analyzed this trade-off using real-world sparse matrices of double-precision floating point values obtained from the UF Sparse Matrix Collection [17]. We considered all matrices with at least 1.5 million non-zero values (87 in total). Figure 10 plots the memory capacity and performance of one iteration of Sparse-Matrix Vector (SpMV) multiplication of our mechanism normalized to CSR for each of these matrices. The x-axis is sorted in the increasing order of the $\mathcal{L}$-value of the matrices.

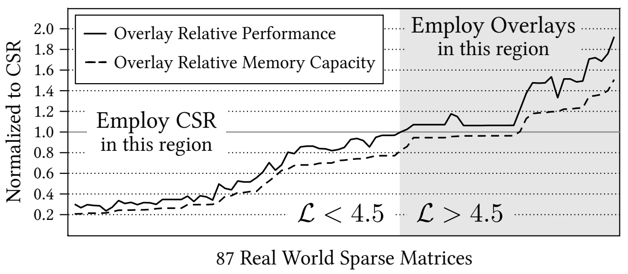


Figure 10: SpMV multiplication: Performance of page overlays vs. CSR. $\mathcal{L}$ (non-zero value locality): Average number of non-zero values in each non-zero cache line.

The trends can be explained by looking at the extreme points. On the left extreme, we have a matrix with $\mathcal{L} = 1.09$ (*poisson3Db*), i.e., most non-zero cache lines have only one non-zero value. As a result, our representation consumes 4.83 times more memory capacity and degrades performance by

70% compared to CSR. On the other extreme is a matrix with $\mathcal{L} = 8$ (*raefsky4*), i.e., *none* of the non-zero cache lines have any zero value. As a result, our representation is more efficient, reducing memory capacity by 34%, and improving performance by 92% compared to CSR.

Our results indicate that even when a little more than half of the values in each non-zero cache line are non-zero ($\mathcal{L} > 4.5$), overlays outperform CSR. For 34 of the 87 real-world matrices, overlays reduce memory capacity by 8% and improve performance by 27% on average compared to CSR.

In addition to the performance and memory capacity benefits, our representation has several **other major advantages over CSR** (or any other software format). First, CSR is typically helpful *only* when the data structure is very sparse. In contrast, our representation exploits a wider degree of sparsity in the data structure. In fact, our simulations using randomly-generated sparse matrices with varying levels of sparsity (0% to 100%) show that our representation outperforms the dense-matrix representation for all sparsity levels—the performance gap increases linearly with the fraction of zero cache lines in the matrix. Second, in our framework, dynamically inserting non-zero values into a sparse matrix is as simple as moving a cache line to the overlay. In contrast, CSR incur a high cost to insert non-zero values. Finally, our computation model enables the system to seamlessly use optimized dense matrix codes on top of our representation. CSR, on the other hand, requires programmers to rewrite algorithms to suit CSR.

Sensitivity to Overlay Cache Line Size. So far, we have shown the benefits of using overlays with 64B overlay cache lines. However, one can imagine employing our approach at a 4KB page granularity (i.e., storing only non-zero pages as opposed to non-zero cache lines). To illustrate the benefits of fine-grained management, we compare the memory overhead of storing the sparse matrices using different overlay cache line sizes (from 16B to 4KB). Figure 11 shows the results. The memory overhead for each size is normalized to the ideal mechanism which stores only the non-zero values. The matrices are sorted in the same order as in Figure 10. We draw two conclusions from the figure. First, while storing only non-zero (4KB) pages may be a practical system to implement using today's hardware, it increases the memory overhead by 53X on average. It would also increase the amount of computation, resulting in significant performance degradation. Hence, there is significant benefit to the fine-grained memory management enabled by overlays. Second, the results show that a mechanism using a finer granularity than 64B can outperform CSR on more matrices, indicating a direction for future research on sub-line based overlay management (e.g., [32]).

In summary, our overlay-based sparse matrix representation outperforms the state-of-the-art software representation on many real-world matrices, and consistently better than page-granularity management. We believe our approach has much wider applicability than existing representations.

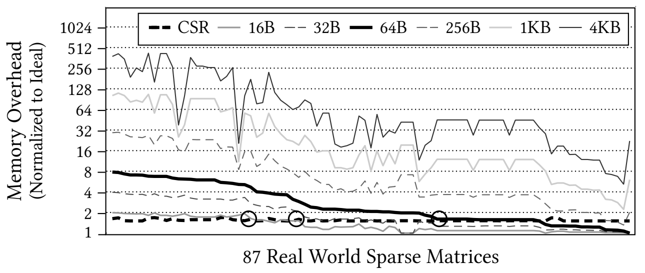


Figure 11: Memory overhead of different overlay cache line sizes over "Ideal" (stores only non-zero values). Circles indicate points where fine-grained management begins to outperform CSR.

5.3. Other Applications of Our Framework

We now describe five other applications from Table 1 that can be efficiently implemented on top of our framework. While prior works have already proposed mechanisms for some of these applications, our framework either enables a simpler mechanism or enables efficient hardware support for mechanisms proposed by prior work. Due to lack of space, we describe these mechanisms only at a high level, and defer more detailed explanations to future work.

5.3.1. Fine-grained Deduplication. Gupta et al. [24] observe that in a system running multiple virtual machines with the same guest operating system, there are a number of pages that contain *mostly same* data. Their analysis shows that exploiting this redundancy can reduce memory capacity requirements by 50%. They propose the *Difference Engine*, which stores such similar pages using small patches over a common page. However, accessing such patched pages incurs significant overhead because the OS must apply the patch before retrieving the required data. Our framework enables a more efficient implementation of the Difference Engine wherein cache lines that are different from the base page can be stored in overlays, thereby enabling seamless access to patched pages, while also reducing the overall memory consumption. Compared to HICAMP [12], a cache line level deduplication mechanism that locates cache lines based on their content, our framework avoids significant changes to both the existing virtual memory framework and programming model.

5.3.2. Efficient Checkpointing. Checkpointing is an important primitive in high performance computing applications where data structures are checkpointed at regular intervals to avoid restarting long-running applications from the beginning [6, 19, 50, 59]. However, the frequency and latency of checkpoints are often limited by the amount of memory data that needs to be written to the backing store. With our framework, overlays could be used to capture all the updates between two checkpoints. Only these overlays need to be written to the backing store to take a new checkpoint, reducing the latency and bandwidth of checkpointing. The overlays are then committed (Section 4.3.4), so that each checkpoint captures precisely the delta since the last checkpoint. In contrast to prior works on efficient checkpointing such as INDRA [47], ReVive [39], and Sheaved Memory [53], our framework is more flexible than INDRA and ReVive (which are tied to recovery from remote attacks) and avoids the considerable write amplification of Sheaved Memory (which can significantly degrade overall system performance).

5.3.3. Virtualizing Speculation. Several hardware-based speculative techniques (e.g., thread-level speculation [49, 54], transactional memory [15, 25]) have been proposed to improve system performance. Such techniques maintain speculative updates to memory in the cache. As a result, when a speculatively-updated cache line is evicted from the cache, these techniques must necessarily declare the speculation as unsuccessful, resulting in a potentially wasted opportunity. In our framework, these techniques can store speculative updates to a virtual page in the corresponding overlay. The overlay can be *committed* or *discarded* based on whether the speculation succeeds or fails. This approach is not limited by cache capacity and enables potentially unbounded speculation [3].

5.3.4. Fine-grained Metadata Management. Storing fine-grained (e.g., word granularity) metadata about data has several applications (e.g., memcheck, taintcheck [56], fine-grained protection [62], detecting lock violations [43]). Prior works (e.g., [36, 56, 62, 63]) have proposed frameworks to efficiently store and manipulate such metadata. However, these mechanisms require hardware support *specific* to storing and maintaining metadata. In contrast, with our framework, the system can potentially use an overlay for each virtual page to store metadata for the virtual page instead of an alternate version of the data. In other words, *the Overlay Address Space serves as shadow memory* for the virtual address space. To access some piece of data, the application uses the regular load and store instructions. The system would need new *metadata load* and *metadata store* instructions to enable the application to access the metadata from the overlays.

5.3.5. Flexible Super-pages. Many modern architectures support super-pages to reduce the number of TLB misses. In fact, a recent prior work [5] suggests that a single arbitrarily large super-page (direct segment) can significantly reduce TLB misses for large servers. Unfortunately, using super-pages reduces the flexibility for the operating system to manage memory and implement techniques like copy-on-write. For example, to our knowledge, there is no system that shares a super-page across two processes in the copy-on-write mode. This lack of flexibility introduces a trade-off between the benefit of using super-pages to reduce TLB misses and the benefit of using copy-on-write to reduce memory capacity requirements. Fortunately, with our framework, we can apply overlays at higher-level page table entries to enable the OS to manage super-pages at a finer granularity. In short, we envision a mechanism that divides a super-page into smaller segments (based on the number of bits available in the `OBitVector`), and allows the system to potentially remap a segment of the super-page to the overlays. For example, when a super-page shared between two processes receives a write, only the corresponding segment is copied and the corresponding bit in the `OBitVector` is set. This approach can similarly be used to have multiple protection domains within a super-page. Assuming only a few segments within a super-page will require overlays, this approach can still ensure low TLB misses while enabling more flexibility for the OS.

6. Conclusion

We introduced a new, simple framework that enables fine-grained memory management. Our framework augments virtual memory with a concept called *overlays*. Each virtual page can be mapped to both a physical page and an overlay. The overlay contains only a subset of cache lines from the virtual page, and cache lines that are present in the overlay are accessed from there. We show that our proposed framework, with its simple access semantics, enables several fine-grained memory management techniques, without significantly altering the existing VM framework. We quantitatively demonstrate the benefits of our framework with two applications: 1) *overlay-on-write*, an efficient alternative to copy-on-write, and 2) an efficient hardware representation of sparse data structures. Our evaluations show that our framework significantly improves performance and reduces memory capacity requirements for both applications (e.g., 15% performance improvement and 53% memory capacity reduction, on average, for `fork` over traditional copy-on-write). Based on our results, we conclude that our proposed framework for page overlays is an elegant and effective way of enabling many fine-grained memory management mechanisms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We acknowledge members of the SAFARI group for their feedback. We acknowledge the generous support of Intel, Facebook, Google, and Samsung. This work is supported in part by NSF grants 0953246, 1212962, 1320531, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

[1] fork(2) - Linux manual page. `http://man7.org/linux/man-pages/man2/fork.2.html`.
[2] Memsim. `http://safari.ece.cmu.edu/tools.html`, 2012.
[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, 2005.
[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.
[5] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *ISCA*, 2013.
[6] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *SC*, 2009.
[7] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: A software approach. In *ASPLOS*, 1989.
[8] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *INFOCOM*, volume 2, 1999.
[9] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and

T. Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA*, 1999.

[10] M. Cekleov and M. Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 17(5), 1997.

[11] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *OSDI*, 1999.

[12] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and Omid A. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *ASPLOS*, 2012.

[13] K. Constantinides, O. Mutlu, and T. Austin. Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *MICRO*, 2008.

[14] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-Based Online Detection of Hardware Defects Mechanisms, Architectural Support, and Evaluation. In *MICRO*, 2007.

[15] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, chapter 8. Intel Transactional Synchronization Extensions. Sep 2012.

[16] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmark Suite. `www.spec.org/cpu2006`, 2006.

[17] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *TOMS*, 38(1), 2011.

[18] P. J. Denning. Virtual Memory. *ACM Computer Survey*, 2(3), 1970.

[19] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems. *Journal of Supercomputing*, 2013.

[20] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale Sparse Matrix Package I: The Symmetric Codes. *IJNME*, 18(8), 1982.

[21] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA*, 2005.

[22] J. Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Commun. ACM*, 1961.

[23] M. Gorman. *Understanding the Linux Virtual Memory Manager*, chapter 4, page 57. Prentice Hall, 2004.

[24] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI*, 2008.

[25] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*, 1993.

[26] Intel. Architecture Guide: Intel Active Management Technology. `https://software.intel.com/en-us/articles/architecture-guide-intel-active-management-technology/`.

[27] Intel. Sparse Matrix Storage Formats, Intel Math Kernel Library. `https://software.intel.com/en-us/node/471374`.

[28] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ISCA*, 2010.

[29] JEDEC. DDR3 SDRAM, JESD79-3F, 2012.

[30] L. Jiang, Y. Zhang, and J. Yang. Mitigating Write Disturbance in Super-Dense Phase Change Memories. In *DSN*, 2014.

[31] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-Level Storage System. *IRE Transactions on Electronic Computers*, 11(2), 1962.

[32] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *MICRO*, 2012.

[33] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *EuroSys*, 2009.

[34] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM JRD*, 51(6), 2007.

[35] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-2, University of Texas at Austin, 2010.

[36] V. Nagarajan and R. Gupta. Architectural Support for Shadow Memory in Multiprocessors. In *VEE*, 2009.

[37] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.

[38] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *MICRO*, 2013.

[39] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA*, 2002.

[40] E. D. Reilly. Memory-mapped I/O. In *Encyclopedia of Computer Science*, page 1152. John Wiley and Sons Ltd., Chichester, UK.

[41] B. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *HPCA*, 2010.

[42] R. F. Sauers, C. P. Ruemmler, and P. S. Weygant. *HP-UX 11i Tuning and Performance*, chapter 8. Memory Bottlenecks. Prentice Hall, 2004.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *TOCS*, 15(4), November 1997.

[44] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[45] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *PACT*, 2012.

[46] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[47] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. In *ISCA*, 2006.

[48] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*, chapter 11. File-System Implementation. Wiley, 2012.

[49] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, 1995.

[50] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, 2002.

[51] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[52] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[53] M. E. Staknis. Sheaved Memory: Architectural Support for State Saving and Restoration in Pages Systems. In *ASPLOS*, 1989.

[54] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-level Speculation. In *ISCA*, 2000.

[55] P. J. Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6), 1990.

[56] G. Venkataramani, I. Doudalis, D. Solihin, and M. Prvulovic. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *HPCA*, 2008.

[57] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *PACT*, 2011.

[58] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *OSDI*, 2002.

[59] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS*, 1995.

[60] B. Wester, P. M. Chen, and J. Flinn. Operating system support for application-specific speculation. In *EuroSys*, 2011.

[61] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser. Legba: Fast Hardware Support for Fine-Grained Protection. In Amos Omondi and Stanislav Sedukhin, editors, *Advances in Computer Systems Architecture*, volume 2823 of *Lecture Notes in Computer Science*, 2003.

[62] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *ASPLOS*, 2002.

[63] Q. Zhao, D. Bruening, and S. Amarasinghe. Efficient Memory Shadowing for 64-bit Architectures. In *ISMM*, 2010.