memif: Towards Programming Heterogeneous Memory Asynchronously

Felix Xiaozhu Lin
Purdue ECE
xzl@purdue.edu

Xu Liu
College of William & Mary
xl10@cs.wm.edu

Abstract

To harness a heterogeneous memory hierarchy, it is advantageous to integrate application knowledge in guiding frequent memory move, i.e., replicating or migrating virtual memory regions. To this end, we present memif, a protected OS service for asynchronous, hardware-accelerated memory move. Compared to the state of the art – page migration in Linux, memif incurs low overhead and low latency; in order to do so, it not only redefines the semantics of kernel interface but also overhauls the underlying mechanisms, including request/completion management, race handling, and DMA engine configuration.

We implement memif in Linux for a server-class system-on-chip that features heterogeneous memories. Compared to the current Linux page migration, memif reduces CPU usage by up to 15% for small pages and by up to 38× for large pages; in continuously serving requests, memif has no need for request batching and reduces latency by up to 63%. By crafting a small runtime atop memif, we improve the throughputs for a set of streaming workloads by up to 33%. Overall, memif has opened the door to software management of heterogeneous memory.

Categories and Subject Descriptors D.4.7 [*OPERATING SYSTEMS*]: Organization and Design

Keywords Operating Systems; Heterogeneous Memory; Data-intensive Computing

1. Introduction

Memory accesses have become a major performance bottleneck in modern data-intensive applications, e.g. big data and high performance computing (HPC) workloads. The reasons are large working sets and cache/prefetcher-unfriendly access patterns. These applications often incur substantial last-level cache misses, suffering from long memory latency [19, 61].

To bridge the speed gap between CPU and memory, modern computers have already employed complex memory hierarchies, including multi-level caches and non-uniform memory access (NUMA) designs. Moreover, emerging architectures are embracing heterogeneous memory, a mixture of fast and slow memory banks that also have significantly different capacities. Fast memories are often built as on-chip SRAM [4], embedded DRAM [23], or die-stacked memory [1, 7]; slow memories are often DRAM or NVRAM.

Unsurprisingly, heterogeneous memory raises new challenges to the hardware-software stack. Most prior work seeks to make memory heterogeneity transparent to applications: this includes hardware mechanisms that manage fast memory as a large cache [40] or part of main memory [54] and OS support that monitors memory access patterns and moves memory data accordingly [45].

Despite the efficacy of the transparent approaches, we believe that they miss a key opportunity: user knowledge. To this end, our driving vision is to empower user, i.e., a synergy among programmer, runtime, and compiler, to explicitly control a heterogeneous memory hierarchy. Towards this vision, we first lay the software foundation by abstracting heterogeneous memories as individual *pseudo* NUMA nodes. The NUMA abstraction for heterogeneous memory, despite its dismissal in prior work [45], enables us to reuse mature OS facilities with great ease. To the best of our knowledge, our work is the first implementation of NUMA abstraction on real heterogeneous memory hardware.

Based on the abstraction, our top design challenge is to support moving *virtual* memory regions – either replicating or migrating them – as requested by applications. Such an OS service is both desirable and feasible: on one hand, impromptu, frequent memory move is proven vital to the use of heterogeneous memory [20, 52], of which faster memories usually have limited capacity and thus have to be used efficiently; on the other hand, modern hardware has already provided good support for memory move, including ample

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA..

ACM 978-1-4503-4091-5/16/04…$15.00.
DOI: http://dx.doi.org/10.1145/2872362.2872401

off-chip memory bandwidth [19, 29], cache-coherent DMA engines [58], and a variety of page sizes.

However, moving memory regions *efficiently* is challenging. Without a careful design, the overheads of kernel interface, virtual memory management and byte copy can easily overshadow what applications can benefit from fast memory. As we will show in Section 2, the closest service in commodity OSes, page migration for NUMA [33], is intensive on both CPU and virtual memory. Point solutions, such as exposing DMA engines to userspace or reserving large chunks of physical memory, are unsafe or impractical to generic workloads.

To address this challenge, we present memif, an OS service offering asynchronous, DMA-accelerated memory move. Unlike userspace DMA drivers, memif is protected and compatible with commodity OSes. To make memif's interface low overhead and low latency, we apply two primary ideas: *i)* making user and kernel directly share *lock-free* data structures, including a unique red-blue lock-free queue; *ii)* using kernel threads to fetch memory move requests and post completion notifications without synchronizing with applications. Both ideas are rarely seen in existing kernel subsystems, such as GPU or NIC drivers. Accordingly, we overhaul kernel mechanisms that are known expensive: page lookup, race handling, and DMA engine reconfiguration.

To offer a proof of concept, we design and implement memif on the Texas Instruments KeyStone II, a server-class system-on-chip (SoC) featuring heterogeneous memory [59]. Compared to Linux page migration, memif reduces CPU usage by up to 15% for small pages (4KB) and up to 38× for large pages (2MB); in continuously serving user requests, it does not require request batching and reduces the latency by up to 63%. On top of memif, we showcase a simple runtime for supporting streaming workloads and gain up to 33% throughput by tapping into KeyStone II's heterogeneous memory.

Overall, memif lays the foundation for software to directly manage heterogeneous memory at medium (hundreds of KB) or coarse (multiple MB) granularities. Atop memif, we expect that compilers or runtime will decide *when* and *where* to move memory by leveraging their domain knowledge. Together, the OS, runtime, and compiler will enable efficient exploitation of memory heterogeneity without burdening application programmers.

We have made the following contributions:

- A novel user interface that enables applications to submit memory move requests and receive completion notifications asynchronously, with no batching and little user/kernel synchronization;
- A suite of overhauled kernel mechanisms for detecting races (instead of avoiding them), efficiently looking up pages, and quickly reconfiguring the DMA engine;
- A detailed cost breakdown, evaluation, and application case study of memif on real heterogeneous memory hardware.

The full source code of memif is available at:
`http://xsel.rocks/p/memif`

2. Background and Motivation

In this section, we highlight the importance of user-guided memory move, identify the inadequacies of existing support, and present our design considerations.

2.1 A Case for User-guided Memory Move

Memory move, i.e., migration or replication, is a key theme in managing heterogeneous memory [20, 52, 54]: a faster memory often has a much lower capacity and cannot hold the working sets of many data-intensive workloads. For instances, many important HPC applications work on terabytes of data [49]; typical big data applications consume data at high rates, e.g., tens of GB per second, with their relative simple compute [29, 61]. To benefit from heterogeneous memories, one must frequently move data in and out of fast memory based on recent or future memory access.

Underutilized off-chip memory bandwidth Fortunately, the ample main memory bandwidth in modern platforms creates rich opportunities for "background" memory move in parallel to foreground computation. Prior work has discovered that scale-out server workloads often utilize less than 10% of off-chip bandwidth [19]; a recent study [29] shows that offline data analytics has bursty memory demand and often uses less than 15% of bandwidth on average; online data analytics often has less than 50% bandwidth utilization and is described as "good and smooth".

User's knowledge is the key The knowledge of programmer, compiler, and runtime is proven essential to exploiting complex memory hierarchies [31, 38]. Compared to transparent approaches that hide memory move from users, a user-guided approach has three key advantages:

- With a full understanding of program design, users can guide data move for catering to complex memory access patterns.
- Unlike transparent management that often reacts based on recent memory access, users are able to move memory proactively or even speculatively for optimal performance.
- User-guided optimization relieves system software of continuous monitoring of application memory access, a mechanism known to incur non-trivial (>10%) runtime overhead [39].

2.2 Existing OS Support Is Inadequate

Ideally, an OS should empower applications to guide memory move efficiently and with ease. Unfortunately, such a service is missing from today's OS.

Page migration for NUMA As we put heterogeneous memory under the NUMA abstraction, existing OS support for migrating pages across NUMA nodes seems a natural fit. However, the mechanism is far from efficient.

To illustrate the problem, we test page migration built in the Linux kernel on both ARM and x86 platforms. In migrating 1500 4KB pages with one `mbind()` syscall, a server-class ARM SoC (see Section 6 for details) shows a throughput of 0.30 GB/sec. On a 2×8 Xeon E5-4650 NUMA machine, the same test shows a throughput of 0.66 GB/sec; even when we migrate 1 million pages in one syscall, the throughput is only 1.41 GB/Sec. All observed throughputs are below 10% of the corresponding memory bandwidths.

Looking deeper into the Linux kernel, we find migrating a page is CPU-bound: in addition to copying bytes from source to destination, CPU performs page table walk, cache flush, page table update with TLB flush, as well as page allocation and free operations. For each page these operations take around 15 μs (of which only 4 μs is for copying bytes) on our ARM-based test platform. The major inefficiencies are twofold: first, CPU is used in copying bytes; second, the kernel mechanism is tailored for synchronous syscalls and repeats some heavy operations for each page with little reuse. Using huge pages will improve the throughput, however, it does not address the inefficiencies and may introduce new performance problems [21]. We will present more overhead details in Section 6.

Direct access of DMA engine Ad-hoc support for memory move often bypasses the kernel infrastructure for efficiency. Userspace DMA drivers enable applications to directly operate DMA engines with no OS overhead. However, this approach has three flaws. First, unless there exists IOMMU for DMA, an application can abuse the unprotected drivers for accessing almost any data in physical memory. Second, lacking a good way of learning memory move completion, applications often have to periodically poll the DMA engine. Third, assigning a DMA engine to a particular application prevents multiple applications from sharing the engine.

Large carveout memory Another common ad-hoc optimization is to amortize the memory move cost over large chunks of physically contiguous memory. This is often done by reserving the chunks during boot time and later linearly mapping them to application's address space. Although effective for certain uses, e.g., embedded multimedia, this approach leads to memory waste and declines all the benefits of kernel memory management. Even worse, it often necessitates an in-application page allocator [60] for managing the large chunks, which fragments the OS features.

2.3 Design considerations

To address these inadequacies, we aim to design an OS service with the following considerations.

Architectural assumptions Our design depends on one common architectural feature: a DMA engine supporting scatter-gather transfer [15], which can move multiple physical memory regions in a single transfer [23, 58]. We consider two optional features greatly beneficial to performance and software simplicity: cache coherence between CPU and DMA engine, which relieves software of expensive cache maintenance; non-aliasing CPU cache, which allows applications and kernel to coherently share data structures from separate address spaces.

Design requirements & goals Our OS service should meet the following basic requirements: *i) asynchronous*, it offloads byte copy to a DMA engine, freeing CPU for user workloads, *ii) protected*, only the kernel code can access the DMA engine and virtual memory states, and *iii) compatible* with commodity monolithic OS kernels such as Linux. Furthermore, it should achieve two performance goals:

- Low overhead. First, the service interface has to be lightweight. Frequent memory move requests, as discussed in Section 2.1, motivate us to minimize user/kernel crossings that are known to significantly interfere with user workloads [55]. Furthermore, the underlying kernel mechanism should incur minimum cost, as every cycle it spends will diminish the benefit from fast memory. This is challenging: although DMA frees CPU from copying bytes, the asynchronous interface introduces new, non-trivial costs, in addition to the existing overhead of physical and virtual memory management.
- Low latency. It is important not to delay application memory requests, because CPU workloads often do not see massive parallelism and are less likely to have enough work to fill the time gap of long wait. Furthermore, quickly pushing application requests to the DMA engine increases the parallelism between CPU and DMA and thus better utilizes the memory bandwidth.

Design decisions At first glance, our two goals appear incompatible: on one hand, conventional wisdom suggests amortizing overhead by delaying and processing requests in batches; on the other hand, individual requests should be serviced with minimum delay. Towards balancing and meeting the two goals, we make the following decisions:

1. Minimize syscalls: make applications and the kernel communicate mostly through shared data structures.
2. Eliminate user-level request batching: make kernel threads proactively "pull" individual requests from application and "post" notifications back – without user/kernel crossings or synchronization. This departs from prior syscall-

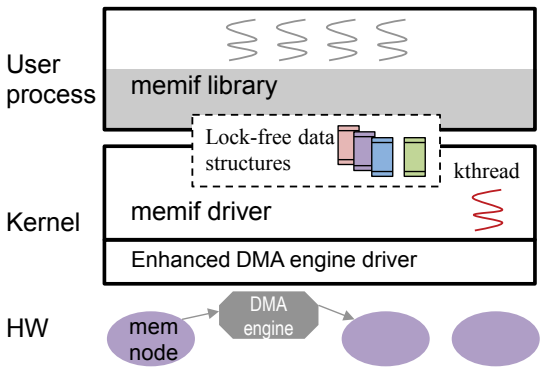


Figure 1. An overview of memif

minimizing approaches [24, 53] that make applications "push" batched requests to the kernel.

3. Exploit the asynchrony of the new interface for optimizing the kernel mechanism.

3. memif in a Nutshell

Implemented as a kernel-mode driver accompanied by a small user library (as shown in Figure 1), memif provides an asynchronous, hardware-accelerated service to replicate and migrate virtual memory regions. It supports two types of move:

- *Replication* implements the semantics of `memcpy()`: an application asks for copying data across two virtual memory regions that are already allocated in its address space. Replication incurs low OS cost: memif does not need to manage virtual memory or physical pages, and is indifferent to potential races between CPU and DMA.
- *Migration* provides the same semantics as page migration for NUMA: an application specifies one of its virtual memory regions and one destination memory node; memif replaces the existing backing pages with new pages allocated from the destination memory; it then fills the new pages with data from the old pages. memif also detects and reports any race between CPU and DMA.

The* memif *interface Simply put, the interface allows an application to submit move requests and later receives the completion notifications without blocking. More specifically, the application first opens a memif device file and owns the corresponding memif instance. The application then communicates with the underlying memif driver via a set of shared, lock-free queues: it submits move requests through enqueueing and receives completion notifications through dequeueing. A syscall is only needed when memif's kernel worker thread is idle and thus needs a kick-start; the memif user library, based on its observation of the queue color, will determine the need for such a syscall and make the syscall automatically (§4.4).

```
int memfd = MemifOpen("/dev/memif0")
struct mov_req *req;
/* Request to move memory regions */
for (int i = 0; i < 10; i++) {
  req = AllocRequest(memfd);
  /* populate all the fields */
  req->src_base = ...
  ...
  SubmitRequest(req); /* non-blocking */
}

/* Do computation ... */

if (req = RetrieveCompleted()) /* Is any move completed? */
  /* Todo: Consume the memory with compute ... */
  ...
/* No other work, sleep until any move is completed. */
poll(fdset);  /* fdset is a set containing memfd. */
...
MemifClose(memfd);
```

Figure 2. A simple example of using the memif user API

Later, if the application becomes idle, it can call `poll()` to sleep while still listening for any notifications from memif, just like a network server waiting for I/O events.

It is worth noting that multiple application threads, in accessing memif concurrently, are unable to cause data races. By design, all operations on data structures of the lock-free interface are atomic. This leaves no room for data race regardless of the application access patterns.

Execution of the* memif *driver The memif driver serves requests one by one and moves memory regions accordingly. In making a "kick-start" syscall, an application thread enters the kernel to execute the memif driver for just one request; it exits the kernel as soon as the resultant DMA transfer starts. Once the request is completed, a memif kernel thread takes charge of serving all the queued requests without userspace involvement. Meanwhile, the application may continue to submit new requests asynchronously – no syscall or locking is needed. Upon the completion of each request, the driver posts a notification by enqueueing the completed request for the application. After all submitted requests (including the new ones submitted asynchronously) are completely drained, the kernel thread goes back to sleep.

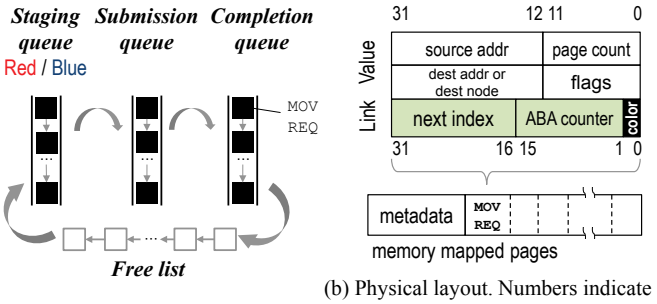
4. The Interface Design

In this section we present the exterior and interior of the memif interface. We first describe the user API (§4.1), then dive in the kernel interface design (§4.2 and 4.3), and last describe how the user API is implemented atop the kernel interface (§4.4).

4.1 The User API

We next describe the memif user API from a programmer's perspective. To make the discussion concrete, we also show a simple code example in Figure 2.

```
int MemifOpen(const char *device_name);
int MemifClose(int memfd);
```



(a) Logic view

(b) Physical layout. Numbers indicate bit offsets. Link is used for queues.

Figure 3. The core of the memif kernel interface: lock-free data structures that are shared between userspace and kernel

Each memif instance exposes a Linux device file. Given a device file name, the above two functions initialize a memif instance and clean it up, respectively.

```
struct mov_req *AllocRequest(int memfd);
void FreeRequest(struct mov_req *req);
```

These two functions allocate and free a `mov_req` structure, respectively. `mov_req` is a hardware-independent representation of a move request, which specifies a virtual memory region consisting of multiple pages in an application's address space. Figure 3(b) shows all fields of a `mov_req`. After a blank `mov_req` is allocated, the programmer is responsible for populating all the fields in `mov_req` with the desired parameters.

```
int SubmitRequest(struct mov_req *req);
```

This function submits a new move request to memif. By calling it, the caller application is oblivious to whether a syscall will be made; instead, the memif library will decide so judiciously. We will discuss the details in Section 4.4. The function is non-blocking; it returns the error code or 0 on success.

```
struct mov_req *RetrieveCompleted(void);
```

This function attempts to retrieve one completion notification from memif. If there is no pending notification, the function returns `NULL` immediately without blocking.

In addition to the non-blocking retrieval, an application can wait for memif notifications while sleeping by calling a POSIX syscall `poll()`, as demonstrated in Figure 2. Supporting `poll()` is useful. First, it allows sleep for saving CPU cycles. Second, `poll()` is generic: through one invocation, applications can blocking wait for memif notifications and other types of I/O events at the same time, such as those of disk and network.

4.2 Kernel Interface

The memif kernel interface, which backs the user API, consists of two components per memif instance:

- One memif device file: just like normal Linux device files, it supports `open()`, `close()`, `mmap()`, and `poll()`. In addition, it supports a new `ioctl` command `MOV_ONE`;
- A set of data structures shared between userspace and kernel: three queues implemented as singly linked lists and one free list. All the shared data structures reside in a set of pinned pages that are first allocated by the memif driver and then mapped into the application's address space. These data structures are lock-free: they can be concurrently accessed by application threads and any kernel contexts without synchronization.

Shared data structures As shown in Figure 3 (b), the internal layout of the memory-mapped area starts with the metadata for the list and queues followed by an array of `mov_req` entries. As shown in Figure 3 (a), the memif initialization procedure chains all the `mov_req` entries in the free list through their `link` fields. Subsequent enqueueing and dequeueing operations update these `link`s and thus logically move `mov_req`s among the following queues:

- *Staging queue* holds the submitted requests that are not yet known to the kernel.
- *Submission queue* holds the requests that are known to the kernel and are currently waiting to be processed.
- *Completion queue* holds the completed requests posted back to the application. In practice, we have implemented the queue as two: one for successful moves and the other for failed ones; we refer to them as one for brevity.

The new `ioctl` command, `MOV_ONE`, only does one thing: entering the kernel, it dequeues a `mov_req` from the *submission* queue and executes the memif driver for the request.

Why lock-free? All the three queues are based on a classic lock-free queue design [47] (the first queue is augmented with a red/blue color as will be discussed later). Lock-free data structures provide a unified, conceptually simple mechanism that allows concurrent access to the memif queues. No single kernel locking mechanism provides this benefit. Synchronization using sleepable locks, e.g., semaphores, prevents the *completion* queue from being accessed by interrupt handlers for notification delivery. Synchronization using non-sleepable spinlocks, on the other hand, exposes kernel to lockup hazards: a misbehaving application failing to release a spinlock will deadlock the kernel.

In theory, any lock-free data structures supporting `set` operations including `add` and `remove` can be used for the memif interface. Having considered other lock-free data structures including singly or doubly linked list [25, 57], we chose lock-free queue for its simplicity and low overhead.

It is worth noting that the use of lock-free data structures is uncommon in user/kernel communication: existing work often makes user and kernel share normal queues (i.e. not lock-free) [24, 53], since user and kernel only access these queues synchronously. With the asynchrony of the memif interface, normal queues no longer work: the integrity of the

shared interface data structures must be guaranteed under any user/kernel access pattern.

Why a red-blue queue? We further extend the *staging* queue to be a novel *red-blue* lock-free queue: unlike a vanilla lock-free queue that only guarantees the atomicity of each queue operation, the new design guarantees the atomicity of each queue operation together with the access of a queue-wide flag, i.e. the queue color.

This feature is critical to memif. To minimize latency, threads from both an application and the kernel share not only the *staging* queue but also a key flag, which indicates which thread is responsible for flushing all the queued requests for execution. While moving requests in/out of the queue, all the threads also pass around the responsibility of flushing the queue. To do so, each thread must manipulate both the queue and the flag exclusively; otherwise a race condition will occur. This raises non-trivial challenge: were memif to employ a vanilla lock-free queue and a normal flag, it would require a lock for protecting the two, breaking lock freedom and incurring lockups.

The red-blue queue realizes the feature with a novel idea. It encodes the flag, dubbed "queue color", in each link inside the queue; it atomically propagates the flag as the queue grows. As a result, the flag can be manipulated as part of an atomic queue operation. Lock freedom is retained.

We will sketch the internals of a red-blue lock-free queue in Section 4.3 and show its use in Section 4.4.

Safety Concerns Directly sharing data structures between userspace and kernel naturally raises safety concerns. Fortunately, the memif interface does not jeopardize kernel's integrity. First, the only object references, the link field in `mov_req`, are indices into the array of `mov_req`, which will be validated by the memif driver before use. Second, as discussed above, as applications hold no lock, their use of memif will never lock up the kernel. Last, one memif device is owned by one process. Multiple memif devices maintain separate copies of queues and free lists and are therefore isolated from each other.

However, it is still possible that applications abuse the memif service for excessive data movement. Such problems are not introduced by the memif interface and can be prevented by the driver without safety risks.

4.3 Internals of Red-blue Lock-free Queue

As mentioned in Section 4.2, application or kernel may need exclusive access to both a *staging* queue and its color. To avoid introducing a lock, our key idea is to "entangle" color in the atomic queue operations. We next provide a generic, memif-agnostic description of how this is implemented.

To provide some background, a classic lock-free queue [47] is often implemented as a singly linked list. It uses compare-and-swap (CAS), an atomic CPU instruction, to update the links between its elements. To enqueue or dequeue, a thread checkpoints the link to be modified, prepares a new version of the link on the side, and then tries to "swap in" the new link with a CAS. Guaranteed by hardware, the swap attempt will succeed if and only if the checkpoint is still up-to-date; otherwise, the queue must have just been modified by another concurrent thread, and the whole procedure is retried.

Based on the design, our red-blue lock-free queue encodes one color bit in each element link, as shown in Figure 3 (b). This allows performing a queue operation (i.e., link update) and setting/getting color with a single CAS. Compared to the classic design, the overhead added by coloring is negligible. We next overview the queue interface and its implementation.

```
color_t set_color(queue_t q, color_t new);
```

This function attempts to change one queue's color to `new`, which, as a rule, will only succeed on an empty queue. Inside, the function first checks if the queue is empty, i.e., whether the head's `next` link is `NULL`. If so, it then follows the aforementioned procedure to swap in a `NULL` link encoding the new color. On success, the old color is returned.

```
color_t enqueue(queue_t q, element_t e);
element_t dequeue(queue_t head);
```

A queue's color is maintained and returned as part of the enqueueing and dequeueing functions. The enqueueing function extracts the color from the old tail's `next` link during checkpointing; it then propagates the color to the new tail's `next` link during CAS. On success, it returns the color to the caller. Since the dequeueing function returns the dequeued element, the caller can simply extract the color from the element's `next` link.

From a high-level perspective, our red-blue lock-free queue is a generic design that maintains a queue-wide property (not limited to a binary color value) as part of the atomic queue operations.

4.4 Putting It Together: Implementing the User API

Most memif user functions presented in Section 4.1 are thin wrappers around the kernel interface: opening or closing the device file, getting a `mov_req` from the free list or putting one back, and dequeuing a `mov_req` from the *completion* queue. The only non-trivial one is `SubmitRequest()`, for which simplified pseudo code is listed and explained below.

```
1  SubmitRequest(request) {
2    color = enqueue(staging_queue, request);
3    if (color == BLUE) { /* user should flush */
4  flush:
5      while (request = dequeue(staging_queue))
6        enqueue(submission_queue, request);
7      old_color = set_color(staging_queue, RED);
8      if (old_color == -1) /* queue not empty */
9        goto flush;
10     if (old_color == RED)
11       return;
12     ioctl(MOV_ONE);
13   }
14 }
```

This function first deposits a `mov_req` in the *staging* queue (line 2). As described in Section 4.1, the *staging* queue

has a blue or red color indicating who should flush it: blue for the application and red for the kernel. If the color is red, the function does nothing but return, knowing that an active kernel thread will later flush the entire queue including the just enqueued request; if the color is blue, the function flushes all `mov_req`s from the *staging* queue to the *submission* queue (line 4–6). At the end of flushing, the function attempts to change the the queue's color from blue to red (line 7). If the attempt fails because the queue is no longer empty (line 8) – one another application thread may have just submitted new requests – flushing is retried. On success, the function invokes `ioctl()` to issue a `MOV_ONE` command (line 12). After the queue color is changed to red, any subsequent requests will accumulate in the *staging* queue, waiting to be flushed by the kernel thread.

There is no race among concurrent application threads attempting to flush the *staging* queue: only the thread that successfully changes the queue color from blue to red will call `ioctl()`; other threads, in attempting to change the queue color, will see the color is already red and quit (line 10).

5. The memif Driver

As mentioned in Section 3, the memif driver serves `mov_req` requests. To break down the challenges faced by the driver, we list the major operations in fulfilling each `mov_req` in Table 1. For these operations, the table shows baseline designs that are closely modelled after the Linux page migration – except DMA and notification delivery. In this section, we discuss key optimizations applied to individual operations (also summarized in Table 1) and then present a complete execution workflow with the optimized operations.

5.1 Gang Page Lookup

To move a virtual memory region, the memif driver needs to locate all the physical page descriptors. Conceptually, given the virtual address of each page, the driver should walk the page table to locate the page table entry (PTE) that leads to the corresponding physical page descriptor. Since all pages targeted by a request are virtually contiguous, we leverage the fact that most of their PTEs are adjacent. Thus, only for the first page in the region, the driver descends "vertically" from the page table root down to the PTE; for each of the remaining pages, the driver walks "horizontally" to traverse neighboring PTEs without starting over from the root.

5.2 Lightweight Race Detection

In migrating each page, the baseline workflow modifies the user address space twice (in Remap and Release) as summarized in Table 1 and illustrated in Figure 4(a). The implications are two. First, changing PTE and TLB has significant direct cost, e.g., up to a couple of μs based on our measurement; in addition, flushing TLB also has indirect cost [2, 55]. Second, since modifying user address space is a heavy action, any code trying to do so must first obtain a sleepable

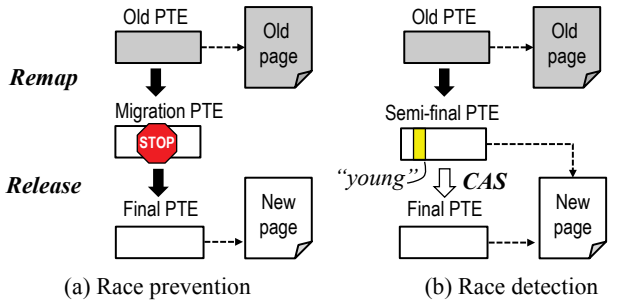


Figure 4. A comparison of (a) race prevention as in Linux page migration for NUMA and (b) race detection in memif

lock as required by the Linux kernel. This prevents Release from being done in non-sleepable contexts such as interrupt handlers.

In retrospect, modifying user address space in Remap used to be warranted: the installed migration PTE *prevent*s races between CPU and DMA by blocking any user thread attempting to modify pages being migrated. With the legacy synchronous interface, blocking applications is the only way to handle race: applications have no other way to learn migration completion and therefore cannot actively avoid races.

Fortunately, since the memif interface explicitly delivers completion notifications to applications, the memif driver can be relieved of race *prevention* and instead simply performs race *detection*, therefore skipping installing migration PTEs. Based on this rationale, we have built the following mechanism.

Proceed and fail The driver detects races and treats them as program errors, as shown in Figure 4(b). In Remap, the driver installs a semi-final PTE, in which all bits are identical to the final PTE (the one installed by the baseline Release) except that the "young" bit is set. With the Linux design, the young bit will be cleared by the kernel automatically when the page is accessed for the first time or when the page is migrated. When it comes to Release, the memif driver tries to clear the young bit: it uses a CAS instruction to swap in the final PTE in which young bit is cleared as it should be. The CAS will fail if any code has modified the semi-final PTE or any page reference has occurred causing its young bit to be cleared. In such a case, the driver sends a `SEGFAULT` signal to the application. On success, no TLB flush is needed since the semi-final PTE never enters TLB.

Alternative: proceed and recover We have also considered handling races more transparently at the price of higher complexity and overhead: upon a race, memif aborts the migration, restores the original mapping, and gracefully notifies the application of the race. This could be useful in case one application decides to stop waiting for an outstanding memory move and demands immediate memory access.

To implement this feature, the driver provides a custom page fault handler for aborting migration when race is de-

	Baseline Operations	Optimized Operations	Major Costs
1. Prep → ***R & M***	For each page: based on the user-provided virtual address, look up the corresponding physical page descriptor.	For the whole request: given the virtual address range, find all physical page descriptors with gang page lookup. **See Section 5.1.**	**++Per-request:** **++Page lookup** **--Per-page:** **--Page lookup**
2. Remap → ***M***	For each page: allocate a new page on the destination memory. Replace the page table entry (PTE) with a special migration PTE, so that any process trying to access the page will be blocked until the migration ends. Flush TLB.	For each page: allocate a new page on the destination memory. Replace the page table entry (PTE) with one pointing to the new page. Set the "young" bit in the PTE. Flush TLB. **See Section 5.2.**	Per-page: Page allocation Replace PTE TLB flush
3. DMA/cfg → ***R & M***	For the whole request: assemble a scatter-gather list and pass the list to the DMA engine driver. The DMA engine driver programs the chain of transfer descriptors and triggers the transfer.	For the whole request: assemble a scatter-gather list and pass the list to the DMA engine driver. The DMA engine driver reuses a chain of transfer descriptors and triggers the transfer. **See Section 5.3.**	Per-request: Initialize the list Per-page: **++Reuse descriptor** **--Write descriptor**
(… DMA transfer …)			
4. Release → ***M***	For each page: replace the migration PTE with the final one pointing to the new page. Flush TLB. Free the old page.	For each page: use a CAS to clear the "young" bit in the PTE and report any race happened during the DMA transfer. **See Section 5.2.** Free the old page.	Per-page: **++CAS** **--Replace PTE** **--TLB flush** Page freeing
5. Notify → ***R & M***	For the request: enqueue the completed `mov_req` to the *completion* queue and wake up any waiting application threads.		Per-request: Enqueue notif.

Time ***R: operation needed in replication*** ***M: operation needed in migration*** ***-- Cost in baseline only*** ***++ Cost in optimized only***

Table 1. The major operations of the memif driver in serving one move request

tected. Remap installs a special PTE so that any write to a migrating page will be trapped as a page fault and handled by the custom fault handler. This handler restores the old PTE, drops the outstanding DMA transfer, and enqueues the aborted `mov_req` to notify the application. The CPU's new write that causes the race will thus be preserved.

5.3 Minimal Reconfiguration of DMA Engine

Perhaps a bit surprisingly, the overhead of configuring DMA engine is non-trivial as compared to the saved CPU cycles. A modern DMA engine often offers sophisticated mechanisms to copy data. For instance, the TI EDMA3 engine [58] exposes an array of transfer descriptors. Consisting of 12 parameters, one transfer descriptor commands the engine to copy a chunk of bytes as if the bytes are organized in a three-dimensional array. To implement scatter-gather transfers [15], CPU can chain descriptors by writing to a specific field in each descriptor to indicate the next descriptor that the engine should execute. According to our knowledge, the configurable descriptors and chaining interface are popular in other DMA engines such as Intel's Crystal Beach 3 [28].

To move a memory region which spans multiple pages, the DMA engine driver composes a chain of descriptors. Without assuming IOMMU, DMA requires one descriptor to cover a physically contiguous area. As a result, the driver dedicates each descriptor to one page, the largest physically contiguous memory area that applications are guaranteed to get.

The major overheads of DMA engine configuration are thus two: *i)* calculating the parameters for each descriptor; *ii)* writing to each descriptor which is in unbuffered, uncached I/O memory. It sometimes takes 4-5 μs to configure one descriptor according to our measurement.

To minimize the first overhead, we have done an obvious optimization to cache calculated parameters since each descriptor always copies a page. To minimize the second overhead, we reuse descriptor chains that have been configured for previous transfers. In order to do so, we enhance the DMA engine driver so that it maintains the knowledge of existing descriptor chains. In a simple example, since the enhanced driver knows that "starting from descriptor 42, there exists a chain of 32 descriptors, each configured for a 4KB transfer", it can reuse part of or the whole chain in the next transfer. For each reused descriptor, the driver only needs to overwrite the source and destination fields, reducing the second overhead by 4×.

5.4 Putting It Together: Driver Execution

With the above optimization applied, the refined operations and their costs are shown in Table 1. Thanks to the lock-free data structures (§4.2), these operations are performed in three different kernel contexts to achieve the smallest latency

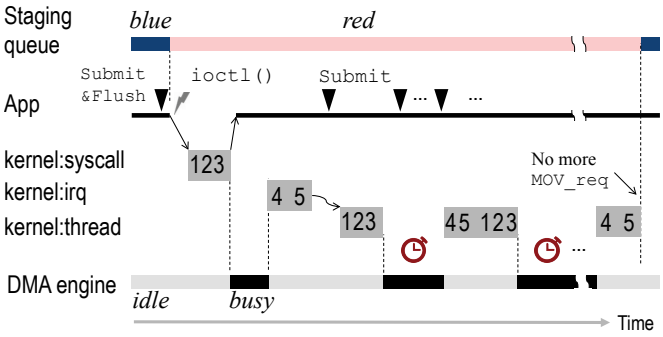


Figure 5. An example execution of memif driver. The numbers are referring to the driver operations summarized in Table 1. Time on x-axis is not drawn to scale.

possible: the caller's process, interrupt handlers, and kernel threads. We next describe the three execution paths in the memif driver. An example execution timeline is shown in Figure 5.

Syscall path As mentioned in Section 4.4, an application invokes `ioctl(MOV_ONE)` to enter the memif driver. Executed in the context of the application process, the driver does operation 1–3 for one queued request. Right after activating the DMA engine, the execution returns to userspace.

Interrupt path When the DMA engine completes data copy for a request, an interrupt is fired. The interrupt handler performs Release (4) and Notif (5) immediately. Note that this is made possible through our lightweight race detection described in Section 5.2; otherwise, modifying address space in Release (4) is forbidden in interrupt handlers as it may lock up the kernel. Last, the interrupt handler wakes up a kernel thread.

Kernel thread path Once woken up, the kernel thread issues all queued requests from the *submission* queue and the *staging* queue. Inspired by network interface drivers [48], the kernel thread switches between the interrupt-driven mode and polling mode depending on the amount of data to move. For small requests (less than 512KB in our implementation) for which completion time is short and predictable, the kernel thread turns off the DMA interrupt and sleeps shortly before performing Release(4) and Notify(5). This is shown in Figure 5. Otherwise, the kernel thread leaves the interrupt on and lets the interrupt handler perform Release and Notify. Finally, when queues are emptied, the kernel thread colors the *staging* queue as blue, indicating that the application is responsible for flushing the queue from now on.

Why use a kernel thread? As schedulable entities, kernel threads are able to acquire sleepable locks such as semaphores, thus capable of time-consuming operations such as Remap(2). More importantly, kernel threads can be executed on cores other than the ones where data-intensive application runs [55], shielding the latter from frequent context switches and CPU exceptions, both known costly.

Table 2. A summary of the test platform, the TI KeyStone II SoC

Hardware	Specs
CPU	4x Cortex-A15, each @ 1.2GHz
Memory	*Fast*: 6 MB on-chip SRAM Measured bandwidth: 24.0 GB/sec *Slow*: 8 GB DDR3 1600MHz Measured bandwidth: 6.2 GB/sec
DMA Engine	TI EDMA3: coherent; 6 transfer controllers; 512 entries for transfer descriptors.

Table 3. A summary of the memif source code

	Library	Driver	DMA	Test	Total
KSLoC	0.8	3.3	0.8	1.7	6.6

6. Prototype & Evaluation

In this section, we describe our test platform and prototype (§6.1), present the evaluation methodology (§6.2) and results (§6.3 – 6.5), and discuss our limitations (§6.7).

6.1 Test Platform & Prototype

We have prototyped memif atop the Texas Instruments KeyStone II SoC [59]. A server-class SoC, KeyStone II powers microservers such as the HP Proliant m800 [26]. As summarized in Table 2, the SoC embraces two types of memory, both cached. An on-chip controller manages two memories and implements coherency between CPU and other bus masters including the DMA engine.

Applying the NUMA abstraction As mentioned in Section 1, we have abstracted the heterogeneous memories as pseudo NUMA nodes. In order to do so, we have ported an existing ARM NUMA patchset [37] to KeyStone II. One challenge is the physical address of SRAM is lower than any DDR bank, luring the kernel to use the capacity-limited SRAM for booting and then crash due to out of memory. To prevent this, we have patched the Linux boot memory allocator so that the SRAM bank only becomes visible after the booting. By that time, all kernel subsystems and the userspace, e.g., the `numactl` utility, can see and use two NUMA nodes: the CPU cores and the DRAM (slow) are on the same node, while the SRAM (fast) is on the other node.

***Implementing* memif** Table 3 summarizes the new implementation source code introduced by our memif prototype. The user library and driver share the implementation of lock-free data structures. Based on Linux 3.10, the memif driver is compiled as a standalone kernel module. Only minor changes are made to the rest of the kernel, e.g., reverse memory mapping, to export a handful of internal functions to the module.

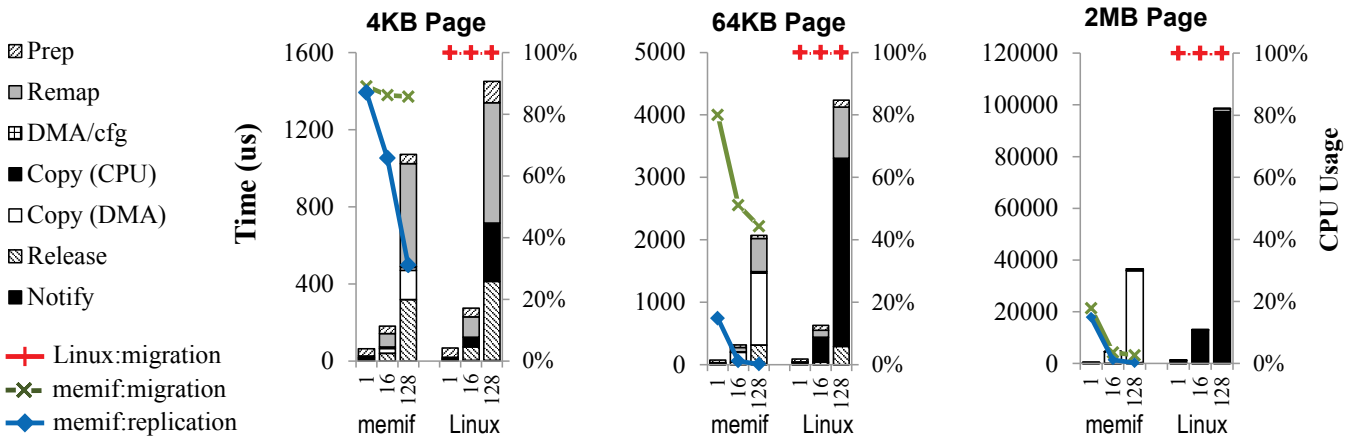


Figure 6. The time breakdown (columns; left y-axis) and CPU usage (lines; right y-axis) in fulfilling a single `mov_req`. Each subfigure is for a different page size as marked on top. Numbers on the x-axis indicate the number of pages in each `mov_req`.

We have applied the DMA enhancement described in Section 5.3 to EDMA3 on KeyStone II. Because EDMA3 is unfortunately not supported by TI's official kernel tree, we have to port a similar driver from TI's DaVinci SoC. In bringing up EDMA3, we have also implemented memory-to-memory transfer for it and added a couple of locks for thread-safety.

6.2 Methodology

We test whether memif has achieved the goals of low cost and low latency as set in Section 2. We benchmark memif replication and migration by moving memory from the slow node to the fast node, and compare memif migration with Linux page migration. In addition to testing the default 4KB page, we further investigate how memif will be impacted by typical large pages. Given that large page support is not yet mature on our ARM-based test platform, we emulate the impact by modifying the kernel so it moves extra bytes while keeping other operations unchanged. By doing so, we have slightly overestimated the cost of page table walk as using large pages often reduces the page table depth by one or two, which, however, is negligible as compared to the total cost.

6.3 Cost Analysis

As has been shown in Figure 6, memif provides memory move service at low cost. Compared to the Linux page migration (labelled as "Linux"), memif *i)* moves the same amount of pages with much lower CPU usage and *ii)* does so in much shorter time for most cases.

By comparing the three subfigures for different page granularities, we can see how memif gains the edge. When small (4KB) pages are in use, the overheads of managing virtual memory and physical pages are the majority. memif manages to offset the overhead with its interface and mechanism optimizations described in Section 4 and 5; memif loses its advantage over Linux only in the extreme case where each request only targets one page. When page size becomes medium (64KB) or large (2MB), byte copy cost starts to dominate and the DMA benefit gradually eclipses every other cost, giving memif a clear win.

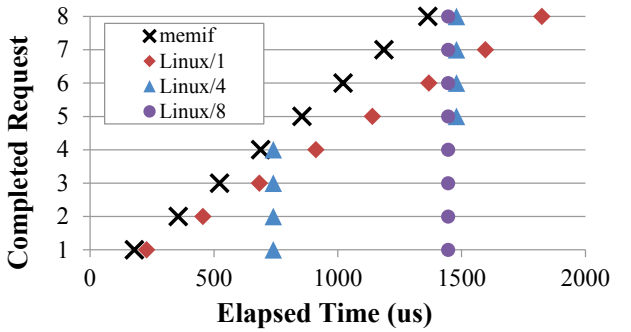


Figure 7. The latency in completing a sequence of eight migration requests, each covering sixteen 4KB pages. Numbers in legend indicate the count of requests batched in each syscall.

It is worth noting that the time breakdown in Figure 6 is not always accurate. Since memif is asynchronous, its operations do not always have counterparts in the Linux page migration code. In such cases, we identify rough equivalences with best effort.

6.4 Request Latency

In continuously serving requests, the asynchronous interface of memif incurs minimum latency at the cost of only one user/kernel crossing. To demonstrate this, we write an application to submit a sequence of eight requests (each targeting 16 pages) to memif; in comparison, we use a second application to perform the same task by using NUMA migration syscalls, and vary the number of requests batched in each syscall from one, to four, and to eight. As shown in Figure 7, the memif-based application receives each notification soon after the corresponding request is completed. Through the course, the application only makes one syscall – `ioctl()`

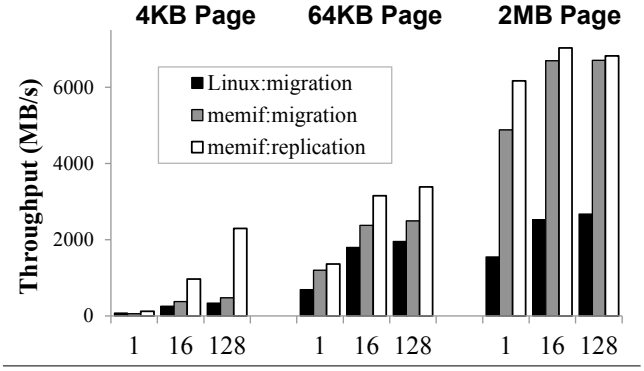


Figure 8. The memory move throughputs across different page granularities. Numbers on x-axis indicate the number of pages in each move request.

Table 4. The throughputs of streaming workloads on top of a mini runtime that uses memif. Numbers are in MB/Sec.

	StreamCluster.pgain	STREAM.triad	STREAM.add
Linux	1440.1	2384.1	2390.1
Memif	1778.4 (+23.5%)	3184.4 (+33.6%)	3186.9 (+33.3%)

for the first request. In contrast, the second application is unable to reduce latency and overhead at the same time: it either has to send each request in a separate syscall to favor latency and suffers from high syscall overhead, or it has to batch multiple requests in one syscall and thus suffers from high latency.

6.5 Throughput

Thanks to its low cost, memif is able to achieve a significantly higher throughput in memory move. We compare an application that streams migration requests to memif with `migspeed`, a Linux NUMA utility released with the standard `numactl` library. As shown in Figure 8, except for the extreme case where each request covers only one 4KB page (the leftmost columns), the memif-based application outperforms `migspeed` by at least 40% (small pages) and up to 3× (large pages). The throughput of memif replication is even higher as compared to the memif migration: the simpler semantics of replication eliminates the need for virtual memory management.

6.6 Case study: A Mini Runtime for Streaming Workloads

To study how memif could be used by real applications, we craft a mini runtime library that uses memif for parallel streaming workloads. The runtime is based on a simple idea: using the fast memory as an array of prefetch buffers and managing outstanding moves just like asynchronous I/O requests. As soon as one application starts, the runtime fills all buffers by replicating data from the slow memory asynchronously. During execution, once a buffer is ready, the runtime invokes the workload's compute function to consume the buffer with all available CPU cores; immediately after any buffer is consumed, the runtime requests to fill the buffer with fresh data again. If all prefetched data are consumed when memory move is still in flight, the runtime invokes compute function to consume data in the slow memory. We implement the runtime with around 400 SLoC.

We have ported three compute kernels from two well-known benchmarks, STREAM [43] and StreamCluster [6]. The throughputs measured on our test platform are shown in Table 4. Our experience suggests that *i)* memif is practical and easy to use and *ii)* a synergy between memif and runtime/compiler offers a promising framework to boost application performance.

6.7 Limitation

Memory move at fine granularity Despite the great efficiency improvement over the state of the art, the cost of memif is still non-trivial when it moves small memory regions – a few to tens of small pages, as has been shown in Figure 6. This is a fundamental limitation rooted in our basic design requirements (§2.3): user/kernel crossings cannot be completely eliminated since virtual memory mappings and DMA engine have to be protected by the kernel; the management of virtual and physical memory cannot be made free due to their software cost. As the granularity of memory region shrinks, all these are emerging as the major cost.

Without an overhaul of the hardware/software stack, user-guided approaches may never be as efficient as transparent, hardware-only approaches for very fine-grained memory move, e.g., at 2 KB [54]. Yet, to data-intensive workloads with large working sets, moving memory at a medium (hundreds of KB) or large (multiple MB) granularity creates a highly valuable balance between efficiency and flexibility.

Platform limitations In testing a variety of data-intensive applications, e.g., wordcount [61] and psearchy [9], we find many of them see little performance gain from memif. Our investigation shows the causes as two limitations of the KeyStone II platform. First, the 6 MB fast memory is not significantly larger than the 1 MB per-core last-level cache. As a result, applications whose working sets fit in the fast memory are also likely cache-friendly. Second, as applications can only use 4 KB pages, memif has to move memory at a relative high cost as discussed above.

We expect the limitations to disappear from emerging platforms as large fast memory and medium/large pages become pervasive. For instance, fast memory is expected to be as large as 1/8 of the main memory [45]. With them, memif will substantially benefit a much wider range of applications.

Implementation limitations As a research prototype, the current memif cannot automatically swap out fast memory; it can only move anonymous pages but not pages backed by files; its support for moving pages shared among processes is primitive. Furthermore, although by design memif is capable of serving multiple concurrent applications, we have not evaluated the feature. Down the road, we plan to coevolve

the memif prototype with emerging heterogeneous memory hardware and examine the implementation more thoroughly.

7. Related Work

Heterogeneous Memory Architecture Memory heterogeneity is common. Small on-chip SRAM, often known as scratchpad memory, is widely used in embedded systems for its low power and high predictability [4]. However, ranging from tens of KB [27] to a few hundreds of KB [42], it is too limited for today's data-intensive workloads. Such small capacities often warrant pure user-level memory management via compilers [18, 44, 56] or runtime [3]. Lacking kernel support, these solutions are unsafe in multiprogrammed environments.

GPGPU often embraces a variety of memory types in one memory hierarchy, making proper placement of program data critical to good performance. Unlike memif, most software work on GPU memory [13, 36] focuses on static data placement rather than dynamic data move.

Memory move Memory move is a key facility in managing complex memory hierarchies. Most software approaches move memory without application's awareness. Treating SRAM as the main memory and DRAM as a paging device, RAMPage [41] transparently copies data between the two. Targeting heterogeneous memory, Meswani *et al.* propose a hardware counter that assists OS to track memory hotness and to migrate pages accordingly [45]. Memory move is also widely used for NUMA. To increase threads and data affinity, kMAF [17] monitors memory access by injecting page faults and migrates pages dynamically across NUMA nodes. To alleviate memory traffic congestion, Carrefour [16] dynamically replicates pages among NUMA nodes. Shoal [31] transparently replicates program arrays among NUMA nodes using DMA. In moving memory, most of the prior work relies on Linux page migration which we have shown as inefficient. Unlike any of them, memif exposes memory move as an OS service.

A few studies improve the mechanism of memory move. Goglin *et al.* [22] augment Linux page migration with a lazy option, migrating a page upon its first access on the new memory node. Compared to memif, they defer migration without addressing the major inefficiency. Bock *et al.* [8] propose a new hardware feature for pinning pages in CPU cache so that the pages remain accessible while being migrated. Compared to memif, they do not reduce the OS overhead of migration. Lepers *et al.* [35] speed up Linux page migration by mainly reducing the use of locks. Compared to memif, they freeze applications during migration, and do not exploit DMA for byte copy.

For heterogeneous memory, memif opens the door to a classic approach – codesigning task scheduling and data move, as has been applied to CPU cache [14] and GPU [30].

User's knowledge User's knowledge has been long known important in managing complex memory hierarchies. For CPU cache, software prefetching [11] is a classic technique to hide latency. Targeting heterogeneous memories, memkind [12] is a user-level heap manager that exposes a variety of memory properties to applications. Similarly, TLM malloc [46] allows programmers to allocate buffers on fast or slow memory under the guidance of profiling. Pena *et al.* [50] propose a profiler for optimally placing program objects in heterogeneous memories. NUMA systems also see rich support for compiler [10, 51] or programmer [32, 38, 39] to guide page replication or migration. These programming abstractions and profiling techniques are complementary to memif and can be retrofitted to guide memory move.

Efficient user/kernel interface High-performance kernel subsystems, such as GPU driver and network stack [24, 53], often directly share buffers with userspace. To use their services, applications push batched requests to kernel using syscall, thus removing the need for explicit synchronization. This is reasonable in handling a stable stream of requests, e.g. in packet I/O applications. The kernel thread in memif is inspired by FlexSC [55], in which all syscalls are greedily deferred until the caller process becomes idle and then get executed in dedicated kernel threads. Similar to them, memif minimizes the use of syscalls; unlike them, memif minimizes the latency in serving memory move requests, as has been discussed in Section 2. Thus, memif eliminates user-level batching and enables user and kernel to asynchronously access the shared data structures in the interface.

Event delivery mechanisms in modern OSes, such as `epoll` [5] and `kqueue` [34], hold pending notifications in kernel buffers. Since the buffers are invisible to userspace, applications have to use syscalls to retrieve the notifications. Unlike them, the memif driver delivers notifications to applications asynchronously without requiring any syscall.

8. Conclusions

We present memif, a lightweight OS service for moving memory regions across heterogeneous memories. memif is asynchronous, hardware-accelerated, and safe to use. memif has achieved low latency and low overhead through a lightweight interface built around a set of shared lock-free data structures and a suite of key optimizations for its kernel mechanisms. Through a prototype atop heterogeneous memory hardware, we have demonstrated that memif enables applications to leverage their knowledge in exploiting heterogeneous memories with ease.

Acknowledgments

The authors thank Texas Instruments for donating the Keystone II evaluation modules and the anonymous reviewers for their useful feedback.

References

[1] S. Anthony. Intel unveils 72-core x86 knights landing cpu for exascale supercomputing. ExtremeTech, 2013.

[2] ARM. ARM architecture reference manual: Armv7-a and armv7-r edition, 2014.

[3] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. Obrien, and K. Obrien. A novel asynchronous software cache implementation for the cell-be processor. In *Languages and Compilers for Parallel Computing*, pages 125–140. Springer, 2008.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[5] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 19–19, 1999.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[7] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 469–479, Dec 2006.

[8] S. Bock, B. R. Childers, R. Melhem, and D. Mossé. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 31:1–31:10, 2014.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, 2010.

[10] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *Intl. Journal of Parallel Programming*, 38(5-6):418–439, 2010.

[11] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):40–52, Apr. 1991.

[12] C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, and S. Hammond. User extensible heap manager for heterogeneous memory platforms and mixed memory policies. Architecture document, 2015.

[13] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100, 2014.

[14] D. Chiou, S. Devadas, J. Jacobs, P. Jain, V. Lee, E. Peserico, P. Portante, L. Rudolph, G. E. Suh, and D. Willenson. Scheduler-based prefetching for multilevel memories. *Lab. Comput. Sci., MIT, Boston, MA, Group Memo*, 444, 2001.

[15] J. Corbet. The chained scatterlist api. `https://lwn.net/Articles/256368/`, 2007.

[16] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–394, 2013.

[17] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kmaf: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 277–288, 2014.

[18] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540, 2005.

[19] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 37–48, 2012.

[20] Y. Gao, F. Zhang, and J. Bakos. Sparse matrix-vector multiply on the keystone ii digital signal processor. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014.

[21] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma. Large pages may be harmful on numa systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, June 2014.

[22] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9, May 2009.

[23] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 135–148, 2012.

[25] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[26] HP. Data sheet: Hp proliant m800 server cartridge, 2014.

[27] Intel. Product brief: Intel ixp425 network processor. `ftp://download.intel.com/design/network/ProdBrf/27905105.pdf`, 2006.

[28] Intel. Intel xeon processor e5-1600/e5-2600/e5-4600 v2 product families, 2014.

[29] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. Mckee, Z. Jia, and N. Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 22–30, Oct 2014.

[30] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 1:1–1:8, 2014.

[31] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 263–276, July 2015.

[32] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 53–64, 2012.

[33] C. Lameter. Local and remote memory: Memory in a linux/numa system. In *Linux Symposium*, 2006.

[34] J. Lemon. Kqueue-a generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[35] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, July 2015.

[36] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into gpu on-chip memory resources. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 23–33, Feb 2015.

[37] Linaro. Numa support for arm. `https://wiki.linaro.org/LEG/Engineering/Kernel/NUMA`, 2013.

[38] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 259–272, 2014.

[39] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, 2013.

[40] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 454–464, 2011.

[41] P. Machanick, P. Salverda, and L. Pompe. Hardware-software trade-offs in a direct rambus implementation of the rampage memory hierarchy. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, 1998.

[42] T. Maeurer and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604, 2005.

[43] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[44] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*, pages 31–40, 2008.

[45] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136, Feb 2015.

[46] M. Meswani, G. Loh, S. Blagodurov, D. Roberts, J. Slice, and M. Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16, Nov 2014.

[47] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[48] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, Aug. 1997.

[49] D. D. Neteworks. Ddn solution brief – accelerate seismic processing. `http://www.ddn.com/pdfs/SeismicProcessing_SolutionBrief.pdf`, 2013.

[50] A. Pena and P. Balaji. Toward the efficient use of multiple explicitly managed memory subsystems. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 123–131, Sept 2014.

[51] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in numa architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 369–380, 2014.

[52] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, pages 85–95, 2011.

[53] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[54] J. Sim, A. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. Transparent hardware management of stacked dram as part of memory. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 13–24, Dec 2014.

[55] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 1–8, 2010.

[56] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415, 2002.

[57] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Principles of Distributed Systems*, volume 3544 of

Lecture Notes in Computer Science, pages 240–255. Springer Berlin Heidelberg, 2005.

[58] Texas Instruments. Enhanced dma (edma3) controller. literature no.: Spruel2b, 2009.

[59] Texas Instruments. Multicore DSP+ARM KeyStone II System-on-Chip (SoC), 2013.

[60] Texas Instruments. Cmem overview. `http://processors.wiki.ti.com/index.php/CMEM_Overview`, 2014.

[61] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499, Feb 2014.