Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters

Nathan Pemberton
John D. Kubiatowicz, Ed.
Randy H. Katz, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-154
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-154.html

December 1, 2019

Acknowledgement

I would like to first thank Emmanuel Amaro who was integral to the design, motivation, and hardware implementation of the PFA. Howard Mao contributed significantly to general infrastructure and implemented the memory blade. The FireSim simulator was the work of many in the Berkeley architecture research group (BAR), especially Sagar Karandikar. I would also like to thank those in the BAR group, and the RISC-V Foundation for supplying the underlying infrastructure for this research. Finally, I acknowledge my advisers Prof. Randy Katz and Prof. John Kubiatowicz, as well as Prof. Krste Asanovic, for their guidance throughout this project.

Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters

by Nathan Pemberton

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Randy Katz
Research Advisor

(12/15/2017)

* * * * * * *

Professor John Kubiatowitz
Research Advisor

(12/15/2017)

Abstract

Researchers from industry and academia have recently proposed to disaggregate memory in warehouse-scale computers, motivated by the increasing performance of networks, and a proliferation of novel memory technologies. In a system with memory disaggregation, each compute node contains a modest amount of fast memory (e.g. high-bandwidth DRAM integrated on-package), while large capacity memory or non-volatile memory is made available across the network through dedicated memory nodes. One common proposal to harness the fast local memory is to use it as a large cache for the remote bulk memory. This cache could be implemented purely in hardware, which could minimize latency, but may involve complicated architectural changes and would lack OS insights into memory usage. An alternative is to manage the cache purely in software with traditional paging mechanisms. This approach requires no additional hardware, can use sophisticated algorithms, and has insight into memory usage patterns. However, our experiments show that even when paging to local memory, applications can be slowed significantly due to the overhead of handling page faults, which can take several microseconds and pollute the caches. In this thesis, I propose a hybrid HW/SW cache using a new hardware device called the "page fault accelerator" (PFA), with a special focus on the impact on operating system design and performance. With the PFA, applications spend 2.5x less time managing paging, and run 40% faster end-to-end.

Contents

1 Introduction

Traditional data center design aggregates all necessary resources (e.g., disk, memory, power supply, etc.) into many self contained server chassis. This design was motivated by the ability to leverage commodity PC components and networks[1]. Additionally, an aggregated design was desirable because in-chassis interconnects were significantly faster than networks. However, data center-side compute has grown into an important independent market, leading to specialized server platforms and networks (often called warehouse-scale computers (WSCs)). Furthermore, networking technology has seen a rapid increase in performance, with 40 Gbit/s Ethernet becoming commonplace, and 100+ Gbit/s networks readily available, narrowing the gap between off-package DRAM and remote memory. Workloads have also changed; applications are fundamentally distributed (e.g., service-oriented architecture, map-reduce, etc.), use larger and rapidly changing datasets ("Big Data"), and demand latencies that can only be delivered by in-memory processing. Finally, a number of promising new memory technologies are becoming available. New non-volatile memory (NVM) devices are being introduced that promise low idle power, high density, and near-DRAM performance (e.g., fast NAND, phase-change, memristor). On the high-performance side, improvements in packaging technology have led to fast on-package DRAM (e.g., HBM) that offers hundreds of GB/s of bandwidth with capacities in the tens of GB.

These hardware and software trends have lead to proposals from both academia[2][3] and industry[4][5][6][7] for a new style of WSC where resources are disaggregated . In a disaggregated WSC, resources like disk and memory become first-class citizens over a high-performance network. A compute node couples CPUs, network interfaces, and a small amount of high-speed memory into a self-contained system in package (SiP). This design allows data center operators to scale memory capacity, while allocating it more flexibly (avoiding stranded resources and complex resource allocation policies). However, the memory access latency will be higher than traditional off-package DRAM, and bandwidth may be limited or subjected to congestion. The small on-package memory allows us to mitigate some of this performance gap, but the question remains: how best to use it?

One way to harness the on-package DRAM is to use it as a large cache for remote bulk memory. Operating systems have traditionally provided this through virtual memory paging which uses virtual memory to treat local physical memory as a software-managed cache (typically for disk). Indeed, several recent academic research projects have proposed using paging over remote direct memory access (RDMA) as a way of disaggregating memory[8][9]. Paging has traditionally been backed by slow disks with access latencies in the milliseconds. This lead to sophisticated algorithms that can take several microseconds for every cache miss. An alternative is to have fully hardware managed DRAM caches[10][11]. These eliminate much of the overhead, but lack the sophistication and application-level insight of OS-based approaches. For example, operating systems often use significant memory for optimistic pre-fetching and caching of disk blocks. A hardware-managed cache may choose to store these in remote memory, while the OS would simply delete them.

This thesis introduces a hardware accelerator for OS-managed caching called the page fault accelerator (PFA). The PFA works by handling latency-critical page faults (cache-miss) in hardware, while allowing the OS to manage latency-insensitive (but algorithmically complex) evictions asynchronously. We achieve this decoupling with a queue of free page frames (freeQ)

to be used by the PFA for fetched pages, and a queue of new page descriptors (newQ) that the OS can use to manage new page meta-data. Execution then proceeds as follows:

- The OS allocates several page frames and pushes their addresses onto the *freeQ*.
- The OS experiences memory pressure and selects pages to evict to remote memory. It marks them as "remote" in the page tables and then provides them to the PFA for eviction.
- The application attempts to access a remote page, triggering the PFA to request the page from remote memory and place it in the next available free frame. The application is then resumed.
- Some time later (either through a background daemon, or through an interrupt due to full queues) the OS pops all new page descriptors off the *newQ* and records the (now local) pages in its meta-data. The OS typically provides more free frames at this time.

2 Motivation and Background

2.1 FireBox

While there are several proposals for a disaggregated warehouse-scale computer, we will use the Firebox[2] project as an example throughout this thesis (see Figure 2.1). The Firebox proposal incorporates ideas from several academic and industrial projects and is representative of disaggregated WSCs in general.

Firebox proposes using high-bandwidth, high-radix, networks to connect compute, memory, and storage nodes in a WSC. Firebox does not dictate any particular networking technology, but is motivated by emerging integrated silicon photonic network interfaces because they can be integrated directly on-die (or on package, e.g., SiP), minimizing pin-out and providing high bandwidth at low power. By multiplexing many wavelengths onto a single fiber (wave-division multiplexing), it is possible to create a very high fanout while using few physical interfaces. This fanout may enable very high radix switches that can connect many SiPs within a single hop. Current estimates indicate these photonic networks can achieve aggregate bandwidths exceeding 1 Tbit/s over 128 channels on a single fiber[12][13]. Link latencies (including photonic transceiver crossings) are on the order of 10s of ns, but final latencies will be determined by protocol decisions. We believe round-trip latencies (across a single switch) of approximately 1 µs to be a conservative prediction. For reference, current infiniband EDR networks provide approximately 24 Gbit/s per link (with up to 12 links per NIC), and have round-trip latencies of approximately 2 µs[14].

Compute nodes can be CPUs or special-purpose accelerators. In either case, they will include some amount of high-bandwidth on-package memory (HBM), we typically assume densities of approximately 2 GB per core. This on-package memory will have very high speed (up to 1 TB/s, at much lower power than traditional off-package memories. A relatively large amount of on-package memory, coupled with very fast networks allows most memory in the system to consist of network-attached memory blades which are optimized for cost, power, and density. The total available memory may be as much as 1 PB. Other resources may

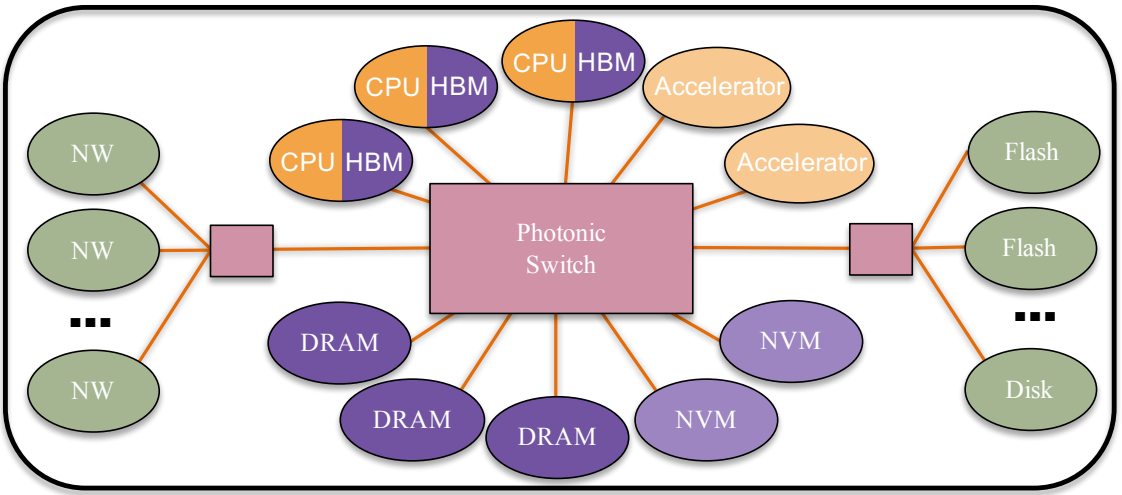


Figure 1: The Firebox WSC proposal. Compute and memory resources become first-class citizens over a high-radix network, allowing flexible resource allocation and scaling. Lower-speed peripherals (such as disk or wide-area networking
can be connected farther away in the network topology.

also be used, such as high-performance NAND flash, disks, and external network bridges. Ultimately, the scale of such a system will be limited by network capacity, but a Firebox would include at least thousands of cores and hundreds of TB of bulk memory.

WSC Challenges WSCs promise to lower total cost of ownership (TCO) significantly by improving utilization and allowing components to develop and scale independently. These benefits bring with them opportunities and challenges that bear mentioning. The increased flexibility in resource allocation will require sophisticated and scalable resource management algorithms that balance utilization, congestion, and fairness. Encryption and authentication keys will also need to be managed to protect memory traffic that is now exposed to attackers across the network. In addition to being secure, memory-like interfaces must be very high performance and any encryption or authentication mechanisms must match that performance. Finally, this new heterogeneous memory hierarchy presents challenges to applications and operating systems. What is the right interface to expose the gap between local and remote memory? Will applications need to be re-written to take advantage of it? It is these questions that I focus on in the remainder of this thesis.

2.2 Interfaces to Remote Memory

The problem of deep and heterogeneous memory hierarchies is not entirely new; previous mainframe and high-performance computing platforms have previously exposed the concept of remote memory. I will describe some of these approaches in the next two sections.

2.2.1 Low-Level Interfaces

NUMA Non-uniform memory access (NUMA) architectures partition memory resources across several compute nodes such that memory is always local to exactly one compute

resource, but still directly addressable by the others. In this case, all memory has the same interface (loads and stores from CPUs), but some is faster than others (non-uniform). Some NUMA systems include hardware services to aid in page migration to mitigate this effect[15].

NUMA systems are appealing because they appear to software as a single, large memory. They can also offer memory access latencies on the order of 100s of nanoseconds. This performance and tight coupling, however, limit scalability. NUMA the largest NUMA systems can scale to hundreds of nodes and 10s of TB of memory[16], but typical systems support only a few TB and less than 10 nodes (due to poor scaling in cost and power).

RDMA RDMA systems are similar to NUMA in that memory resources are partitioned among several compute nodes (memory is always local to someone)[17][18]. The difference is that while NUMA systems typically expose a cache-coherent load-store interface to both local and remote memory resources, RDMA uses a special put/get interface to access remote memory resources. Typically, this service is provided through the network interface and managed by software. This interface allows RDMA systems to scale beyond what is possible in NUMA systems, at the cost of remote memory access performance and a more complex interface to applications.

RDMA systems can scale to thousands of nodes and petabytes of memory[19]. Performance can vary, and scales with deployment size, but modern Infiniband networks provide round-trip latencies of several microseconds (within a rack) and bandwidths of hundreds of gigabytes per second[20]. These systems have historically been considered costly and only were primarily deployed in supercomputing environments, but recent Ethernet-based implementations have made them increasingly accessible[17].

Memory Semantic Fabrics Finally, a new class of interface has been recently introduced; the memory-semantic fabric. A memory-semantic fabric abstracts memory into a simple load-store interface (rather than technology-specific protocols). These interfaces are tightly coupled with the CPU, often loading memory directly into local caches or even registers. This abstraction enables heterogeneous memory technologies in flexible topologies. Memory thus becomes a first-class citizen (often called a "memory blade") on a memory-optimized interconnect. The hope is that such interfaces will allow for greater scalability and flexibility than NUMA, while providing a more direct interface than RDMA. There are several commercial consortia developing cache-coherent interconnects for integrating accelerators and memories within a rack [21][22]. Some academic projects have focused on scaling NUMA by increasing the level of abstraction (e.g., [23][24]). Finally, an industrial effort called Gen-Z provides a more general interface that can connect memory, accelerators, and storage using memory-oriented operations (like load and store)[25]. While Gen-Z does not include cache-coherence in the core specification, it can be added through custom commands between devices that require it. It remains to be seen how these new interconnects balance performance, scalability, and cost.

2.2.2 Software Interfaces

The low level interfaces listed above do not necessarily mandate a particular software interface. NUMA systems typically expose a virtual memory abstraction to applications. In this case,

the OS manages mappings from virtual to physical addresses while hardware uses those mappings to automatically route loads and stores to the appropriate memory resources. The OS is also responsible for choosing which NUMA domain to allocate memory from. This can be a complex decision and much effort has gone into studying such allocation policies[26].

RDMA systems are further divorced from specific hardware interfaces and enjoy a great diversity of interfaces. Some programming languages use a partitioned global address space to make it appear as if language-level variables are all directly accessible[27][28]. Other systems use RDMA more directly to accelerate applications such as key-value stores[29][30].

Memory-semantic fabrics are newer and it is not clear how their interfaces should be exposed. By coupling tightly with CPUs, it is possible to address them directly using virtual memory. However, it may be desirable to allow applications to choose which memory they access, or have more abstracted interfaces (e.g., disk-like). Furthermore, these fabrics are designed to support highly heterogeneous memory technologies. This has lead to page-migration proposals that try to manage performance and durability requirements either explicitly in the application, or transparently in the OS[31][32].

In this thesis, I will focus on a very general interface called demand paging (covered in detail in the next section) that can be implemented under any of the low-level interfaces listed here. We assume a system that allows block reads and writes to remote memory resources. In a NUMA system, this would translate to page migration. For RDMA, we would allocate memory from under-utilized nodes to store pages from oversubscribed nodes (as was done in [8]). In the memory-semantic approach, dedicated memory blades would be used for remote memory and transfers would be initiated directly from the client CPUs (e.g., using *memcpy()*).

2.3 OS Paging Overview

Many architectures expose the abstraction of virtual memory. While the implementation of virtual memory is fairly similar across architectures and operating systems, for concreteness I will use the RISC-V ISA (privileged architecture version 1.10[33]), and Linux version 4.15[34] for most examples in this thesis.

With virtual memory, the addresses issued through load and store instructions do not directly correspond with physical address on the memory bus. Instead, they are translated to physical addresses through a data structure called the page table (see Figure 2). Page tables contain translations for fixed-sized ranges of memory called pages (4 KiB in RISC-V). In addition to translations, each page table entry (PTE) contains meta-data about the page such as read/write permissions, page validity, and whether the page has been read or written to recently (called "accessed" and "dirty" respectively). Throughout this thesis I will refer to the logical group of data as a "page" and the physical region of memory containing that page as a "page frame" or simply "frame".

Figure 3 shows a flow chart for translating a page:

1. The CPU issues a load or store for a particular virtual address
2. The load/store unit queries a small cache of recently used translations called the translation look-aside buffer (TLB).

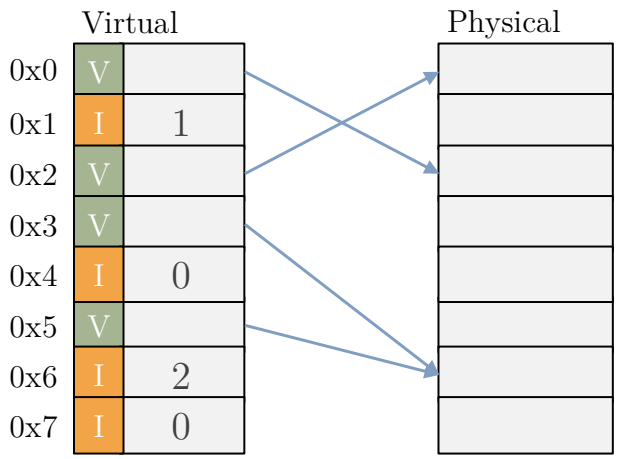


Figure 2: Example page-table. Entries with a "V" are valid (have the valid bit set) while "I" indicates an invalid entry (valid bit clear). Addresses refer to virtual page number (the 52 most significant bits in RISC-V).

   (a) If the translation is found, a physical address is returned to the CPU which then performs the memory access.

3. If the translation is not found, the TLB uses a hardware block called the page table walker (PTW) to find the relevant PTE in main memory.

   (a) If the PTE is marked valid, the physical address is returned to both the TLB (for caching) and the CPU.

4. If the PTE is marked invalid, the PTW issues a page-fault to the CPU which then traps into the OS to handle the invalid memory access.

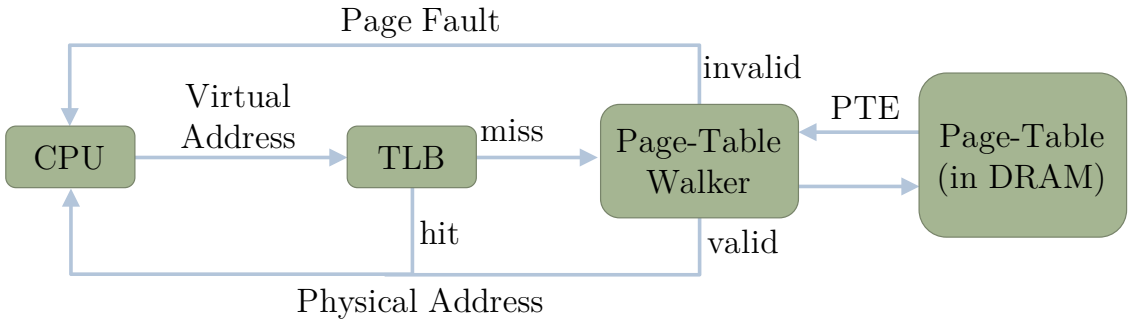


Figure 3: Flow chart for virtual to physical address look up in a typical virtual memory system. The translation look aside buffer (TLB) caches translations. The page-table walker (PTW) fetches mappings from main memory when the TLB misses. Most mappings are valid and can be returned directly to the CPU, but invalid mappings result in a trap to the OS.

In addition to providing memory protection and dirty/accessed tracking, virtual memory enables a number of operating system techniques to manage physical memory through the

page-fault mechanism. For example, new virtual memory allocations in Linux do not get physical addresses assigned immediately. Instead, the OS protects all reads and writes to the page, and only allocates memory if and when the process actually attempts to use it (virtual address 0x4 in Figure 2 for example). This saves considerable memory allocation overheads when processes allocate more memory than they actually use (a fairly common occurrence). Another technique, called paging, creates the appearance of unlimited memory by transparently moving pages to an external storage device (such as a hard disk) and bringing them back only when accessed. This effectively turns main memory into a software-managed cache for the external storage device. It does this by marking the evicted pages as "invalid", and storing their location on the storage device in the PTE. For example in Figure 2, virtual pages 0x1 and 0x6 are currently stored on an external storage device at index 1 and 2, respectively.

2.4 Paging Limitations

It is important to note that the majority of time in demand paging occurs in operating system software running in the page-fault handler. The OS must walk the page tables, look up per-page meta-data in internal data structures, and allocate a new frame, among other bookkeeping tasks (see Section 4.3 for more details on this process). Our experiments on a modern x86-based platform found that this processing can take anywhere from 2 µs to 5 µs. While this latency is tolerable when accessing a slow disk (with access latencies on the order of 10 ms), it is a significant overhead when using fast remote memory (with 1 µs access latency) as a backing store. To demonstrate this effect, we modified the Linux kernel to swap to pre-allocated DRAM buffers in local memory instead of an external device. This effectively eliminates backing store access times and directly measures the overhead of using paging at all.

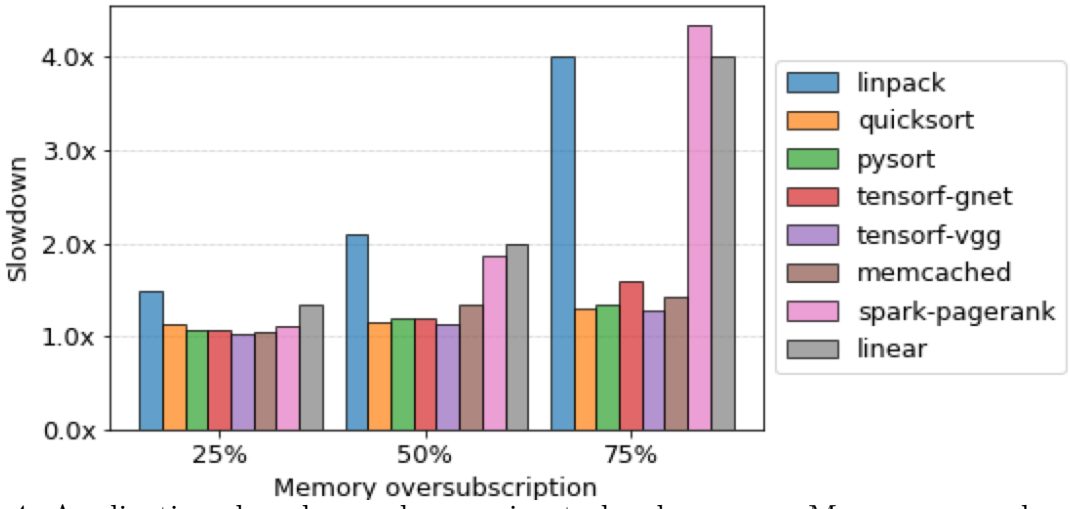


Figure 4: Application slow-down when paging to local memory. Memory oversubscription refers to the percent of peak memory that was stored remotely.

Figure 4 plots the results of this experiment. Some applications (such as quicksort or

Tensorflow) have good locality and therefor experience relatively little impact from paging (although even these applications see slowdowns of as much as 40%). However, applications with poor locality (such as Java or linpack) can experience significant slowdowns (up to 4x in this experiment) due solely to the software overhead of paging.

3 Page Fault Accelerator

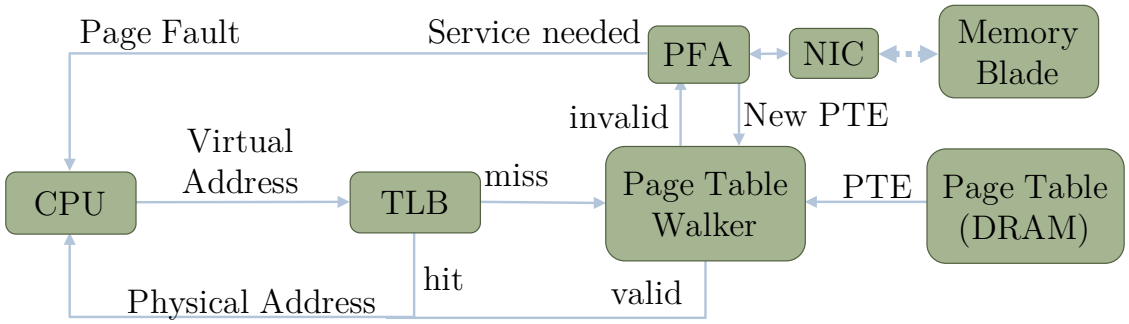


Figure 5: Paging with the PFA. Instead of an invalid PTE causing a trap to the OS (as in figure 2), invalid pages are passed to the PFA to be fetched from remote memory. The PFA may still cause a trap if it cannot handle the request (e.g., full queues).

Much of the work done during a page fault, while important, does not strictly need to occur in order for the application thread to make progress. For example, allocating free frames or updating page meta-data could be performed at any time. Other tasks may be more efficient in hardware than in the OS; the walking of page-tables for example. We propose a hardware accelerator that performs only the bare-minimum of copying a remote page into a pre-allocated frame, updating the relevant PTE, and restarting the application (figure 5).

While this does not eliminate the need for software management of page meta-data (here referred to as bookkeeping), it does provide considerable flexibility to the OS in how such tasks get scheduled. Figure 6 illustrates the difference from the perspective of the OS. One immediate benefit is that the OS can schedule this bookkeeping thread on idle resources, e.g. while the application is blocked on I/O. Another benefit is that bookkeeping tasks can now be batched. Batching improves cache locality and amortizes context switch overheads.

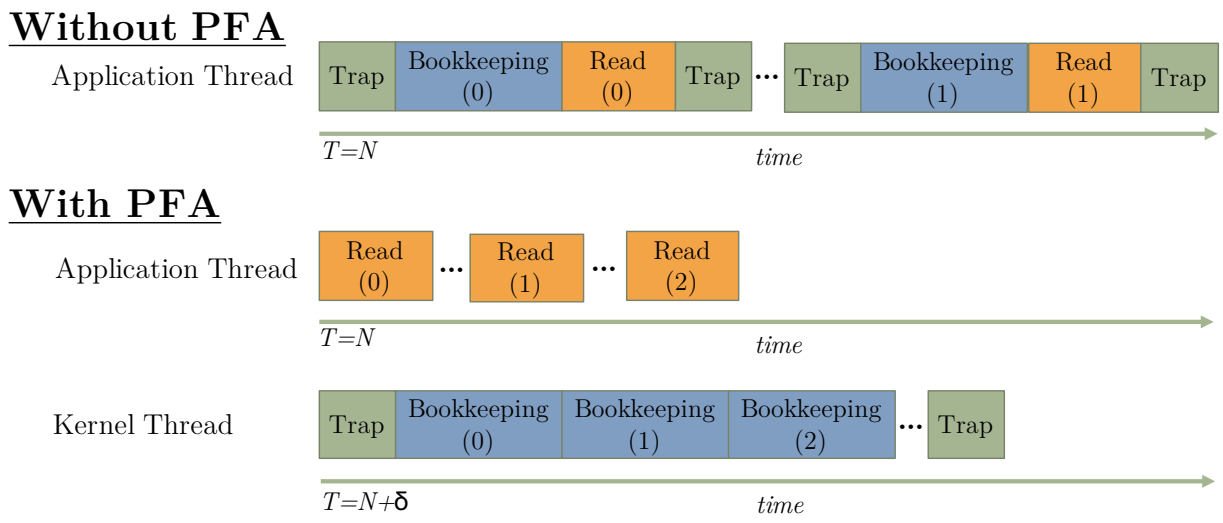


Figure 6: Timeline of page-fault processing with and without the PFA. Without the PFA, the OS must be invoked on every page miss and perform various data structure look ups to decide how to handle the fault. The PFA allows this bookkeeping to occur any time after the fetch in a separate kernel thread. Only the actual page read must occur before the application can be restarted.

3.1 Page Fault Accelerator Design

The primary interface to the PFA is through a number of memory-mapped queues: FreeQ, NewQ, and EvictQ. The FreeQ contains unused page frames that the PFA can use for fetching new pages, the NewQ reports any recently fetched pages to the OS bookkeeping thread, and the EvictQ contains a list of local pages that should be stored in remote memory. Using these queues, execution proceeds as follows:

Eviction The PFA handles all communication with the memory blade. This includes page eviction. The basic procedure is as follows (see figure 7 for a detailed description):

1. The OS identifies pages that should be stored remotely.
2. It evicts them explicitly by writing to the EvictQ.
3. The OS stores a page identifier in the PTE and marks it as remote.

In addition to the three main queues, there are a number of other maintenance registers that are used for querying queue status and initializing the PFA. See appendix B for a complete specification. I will mention one status register here; the EVICT_STAT register. When a page is placed on the evict queue, the PFA begins transferring it to remote memory, but does not block the OS. This allows the OS to perform useful work while the eviction is taking place, potentially hiding some of the write latency. In order to re-use the page frame, however, the OS must poll the EVICT_STAT register to ensure the write has completed.

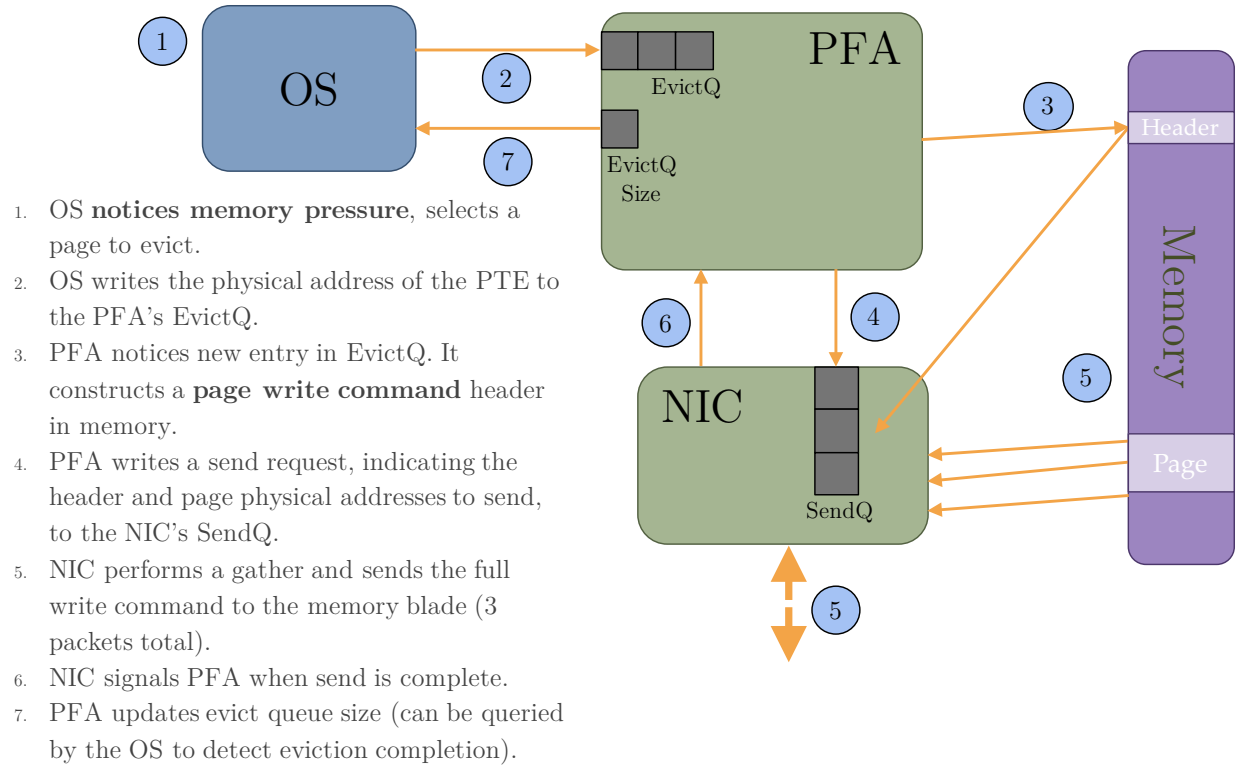
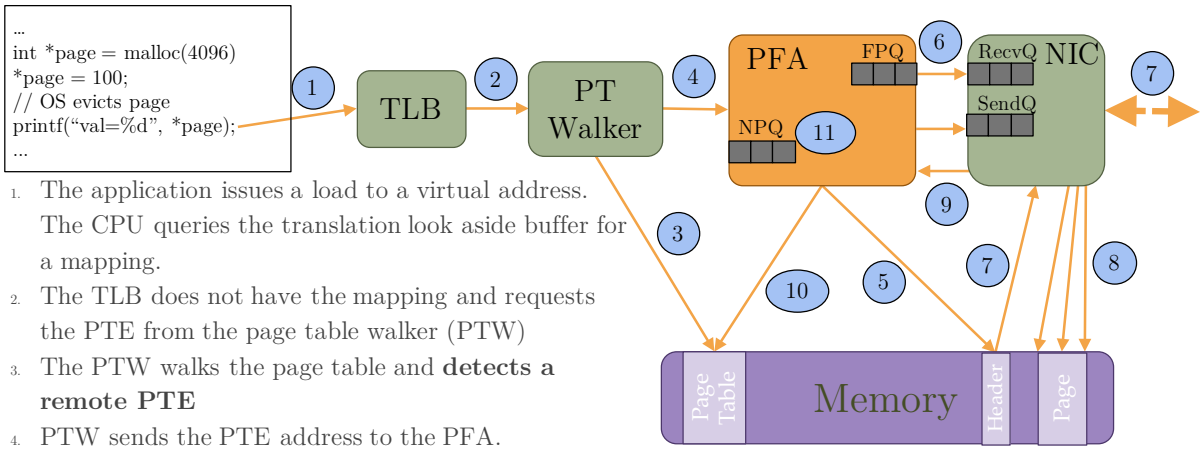


Figure 7: Detailed eviction flow

Fetch The primary function of the PFA is to automatically fetch pages from remote memory when an application tries to access it. It does this by detecting page table entries that are marked remote and transparently re-mapping them to the next available free frame. The basic operation is as follows:

1. Application code issues a load/store for the (now remote) page.
2. The PFA automatically and synchronously brings the page from remote memory and stores it in a free frame.
3. The PFA clears the remote bit in the PTE.
4. The PFA pushes the virtual address of the fetched page to the NewQ.
5. The application is restarted.

Figure 8 describes the process in more detail.



1. The application issues a load to a virtual address. The CPU queries the translation look aside buffer for a mapping.
2. The TLB does not have the mapping and requests the PTE from the page table walker (PTW)
3. The PTW walks the page table and **detects a remote PTE**
4. PTW sends the PTE address to the PFA.
5. PFA assembles a **load command** in memory.
6. PFA signals NIC to send the command and provides the first free page (from the freeQ) as a receive buffer.
7. NIC reads request from memory, sends request to memory blade.
8. NIC gets new page, DMAs page to destination frame.
9. NIC signals PFA done.
10. PFA updates page table.
11. PFA adds **pageID to NewQ**.
12. The application is restarted with the new mapping.

Figure 8: Detailed fetch flow

Metadata Management The OS should ensure that there are sufficient free frames in the FreeQ to ensure smooth operation. If a remote page is requested and there are no free frames, the PFA will trap to the OS with a conventional page-fault. The OS must enqueue one or more free-frames before returning from the interrupt. This may involve evicting pages synchronously in the page-fault handler.

Similarly, the OS needs to drain the new page queue periodically to ensure it does not overflow. This will also trap to the OS with a conventional page fault.

3.1.1 Page Table Entry

The PFA uses a special PTE format for remote pages (Figure 9). The fields are as follows:

- **pageID**: This acts as an address in remote memory for the remote page. It is used by the PFA to look up pages in remote memory, and for the OS to identify each page during bookkeeping.
- **Prot**: This sets the protection bits that the PFA will use when fetching a page. These bits include things like read/write permissions, as well as other page metadata (see the RISC-V privileged architecture manual for more details [33]).
- **R**: This bit indicates that a page is remote (when the valid bit is clear).

- **V**: This indicates whether a page is valid. A valid page is currently in main memory and would not trigger a page-fault. This is also referred to as the "present bit" in Linux.

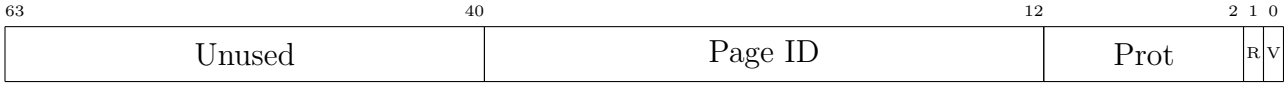


Figure 9: Remote PTE Format. The **Page ID** is a unique identifier of this page and serves as a remote memory address. The **Prot** field contains the permission and metadata bits that should be set after a page is fetched (see the RISC-V specification for details[33]). The **R** bit indicates that this page is remote while the **V** bit indicates that the PTE is not a valid mapping (needed for backward compatibility).

It is worth noting that the RISC-V instruction set manual specifies that all bits other than the valid bit are considered "don't care" when the valid bit is clear. This deviation from the standard may be problematic for some OSs, but is compatible with the Linux kernel. Another interesting feature of this design is the use of pre-defined protection bits. This includes a valid bit which can be cleared by the OS before evicting to trigger a page fault on this page immediately after fetching (a useful debugging feature). Also, bits 8 and 9 are reserved for software by the RISC-V ISA and can aid the OS in bookkeeping and debugging (see Section 4.3).

3.1.2 Remote Memory Protocol

When the PFA receives a request for a remote page from the PTW, it must communicate with a remote memory blade to retrieve the appropriate data. It does this through a custom network protocol. This involves forming network packets and interfacing with the NIC. For this project we assume an Ethernet-based network with a maximum transfer unit (MTU) of 1500 B and a page size of 4 kB, this is not fundamental to the design. Figures 10 and 11 show the basic operation of the protocol. See appendix C for a detailed description of these packet types. Note that due to the MTU, each page takes 3 packets to transfer.

This protocol represents an initial design point that is suitable for a single client/memory blade pair. We handle potentially out-of-order packets by attaching sequence numbers and transaction IDs to each packet, but assume a reliable network. While multiple clients could be supported with this protocol, some interesting design challenges begin to appear. One issue is that of fairness and congestion. A robust protocol would need some prioritization of clients (perhaps assigned from a global allocator) and a back pressure mechanism to ensure optimal behavior. It may be desirable, due to high access latencies, to include some amount of compute in memory, whether that be traditional atomic memory operations, or more general remote procedure calls. Another important component is that of confidentiality and authentication. While it would be straight-forward to encrypt payloads, this may interfere with atomics or other compute-in-memory features. Headers present additional challenges. Simple encryption of headers would not be sufficient because attackers could simply replay old requests to modify application state. Some form of nonce (a unique number added to

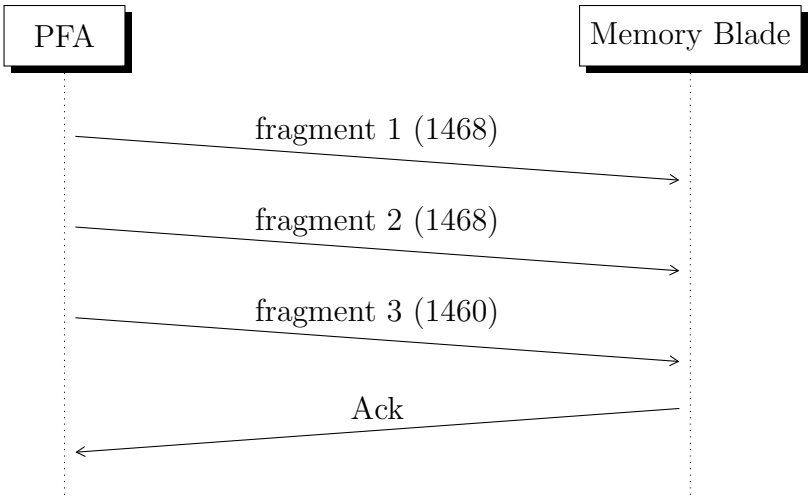


Figure 10: Page Write. Note that each fragment contains a header containing a "write" opcode and destination pageID, along with a unique transaction ID.

each header) may be required. While the design of a memory blade represents an interesting avenue of research, the design presented here is sufficient to evaluate the PFA and I will leave further discussion to future work.

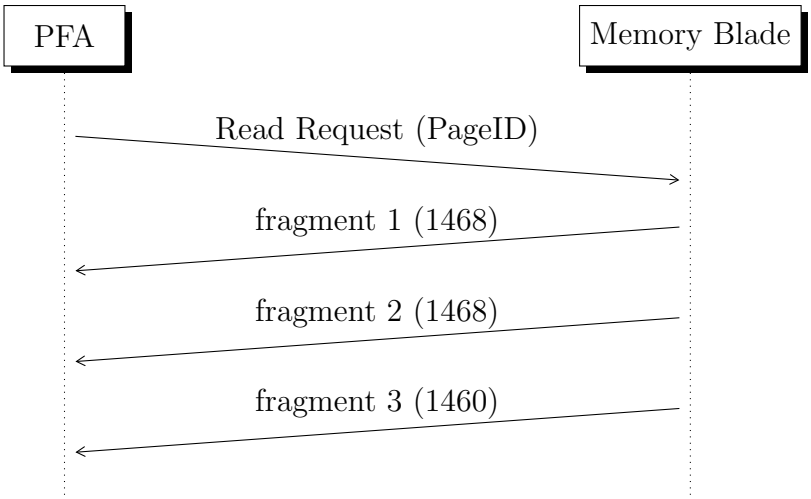


Figure 11: Page Read. Note that responses contain a header indicating the transaction ID of the corresponding read request.

4 Implementation

The PFA was implemented within the RISC-V ecosystem. RISC-V is an open-source instruction set with several open and closed-source implementations and ports for many common software components[35]. I used the RISC-V port of Linux 4.13 with a Buildroot generated user space.

4.1 PFA Reference Implementation

To accelerate software development, and to provide a golden-model of PFA behavior, I implemented the PFA first in a RISC-V ISA simulator called "Spike"[36]. Spike provides a functional simulation of a RISC-V core through a straightforward C++ interpreter, but does not provide any timing accuracy. Due to its simplicity, the PFA implementation required only a few weeks of implementation effort and less than 1000 LoC. With Spike, software development was able to proceed concurrently with the concrete hardware design. Furthermore, unit tests developed under Spike were used to validate the hardware implementation, reducing debugging effort. In all, the only software change that was needed to go from Spike to a concrete implementation was one extra TLB flush due to a difference in TLB design between Spike and the RISC-V implementation we used.

4.2 Hardware Implementation

The PFA prototype was implemented in the Chisel hardware construction language[37] and integrated with a simple in-order CPU called RocketCore[38]. The components were integrated using the RocketChip system-on-chip (SoC) generator[38]. I provide an overview of the relevant systems in the following sections. The current PFA prototype implements a subset of the specification described in Section 3.1. Specifically, it does not support multiple simultaneous evictions (the EvictQ has an effective depth of 1), however, it does allow for

asynchronous eviction. This prevents optimizations such as switching to other threads while many pages are simultaneously evicted (a single eviction does not take long enough to justify a context switch). The current prototype also does not support the full memory blade specification. In particular, it does not implement Ethernet headers which limits experiments to a single client/memory pair over a static network route. Instead, it sends a simplified form of packet header with only opcode and transaction ID. It also does not use a write acknowledgment, this means the OS must rate-limit writes to avoid overwhelming the memory blade. Without back-pressure, this rate-limiting must be pessimistic. However, the rate-limit only effects evictions which tend to occur slowly over time.

4.2.1 RocketCore and RocketChip

RocketChip[38] is a framework for generating SoCs. It includes on-chip interconnects, caches, and other utilities for chip construction. While the CPU is pluggable, we use only a single RocketCore in-order CPU for our experiments. Our implementation used dedicated 16KB instruction and data caches. While the node had access to several gigabytes of main memory, we artificially limited application memory using Linux cgroups (see Section 5 for details). A real WSC would likely include a mixture of simple cores, high single-thread performance cores (e.g., out of order), and accelerators. We hope to evaluate the Berkeley out-of-order core (BOOM[39]) and the Hwatcha vector accelerator[40] in the future when they become available in our simulation infrastructure.

4.2.2 Memory Blade

The memory blade that is used by the PFA was implemented as a bare-metal C program running on a RocketCore-based compute node. This memory blade is connected to the same network as our PFA-enabled compute node and traverses a simple switch in order to communicate. The current implementation uses a simple protocol in lieu of a full Ethernet stack. We are currently evaluating competing memory blade designs ranging from full hardware, to higher-level software implementations. However, this simple design is sufficient for evaluation of the core PFA functionality.

4.2.3 FireSim

We simulated the RTL using a cycle-accurate simulator called "FireSim"[41]. FireSim is an FPGA-accelerated simulator that runs on the Amazon cloud. It can simulate thousands of nodes with a cycle-accurate network and heterogeneous components. Many parameters of the simulation are tunable within FireSim. We decided on a 200 Gbit/s network with 2 µs link latency, leading to roughly 8us page access time to the memory blade. The network is in-order and reliable. A realistic Firebox system would include a much faster network with many more components interacting. We limit ourselves to this simple design in order to evaluate only fully synthesizable components (especially the limited throughput of our CPU platform). Future experiments will evaluate the effects of faster networks with improved compute and memory performance

CPU Type	Rocket (5-stage in order)
CPU Frequency	3.2 GHz
Caches	16 kB D$ and I$
NW Topology	Single Switch
NW Bandwidth	200 Gbit/s
NW Link Latency	2 µs

Figure 12: System parameters used for evaluation.

4.3 Linux Integration

We modified the Linux kernel (version 4.15[34]) to support the PFA. The majority of software development was done using the functional simulator. Linux is a mature open-source project with a long development history, resulting in many Linux-specific terms. I will define these terms throughout this discussion but you may refer to the glossary for a reference of terms.

4.3.1 Non-PFA Paging in Linux

I will now briefly describe how paging works in vanilla Linux. Note that the kernel internally uses the term swap in reference to all paging activity, I will use these terms interchangeably. For a more complete discussion of memory management in Linux, see [42]. Figures 13 and 14 map out the steps involved in evicting and fetching pages, respectively.

Page Reclaiming Linux manages memory limits on a per-task basis. In this case, a task refers to the kernel-specific abstraction of a process. Each task has its own resource limits which are exposed to system administrators through the control group (cgroup) interface. When a task approaches its assigned limit of a certain resource, it is throttled in a resource-specific manner. In the case of memory, the kernel attempts to free task-assigned memory. It will first attempt to shrink any file caches (especially clean disk blocks that can simply be deleted without requiring any disk activity). If shrinking caches is not enough, the kernel begins to page non-file backed pages (called "anonymous pages"). This is done using a pseudo-least recently used (LRU) eviction algorithm (Step 2 in Figure 13). Page reclaiming can be triggered in one of two ways (Step 1). If a hard memory limit is met, but more memory is needed to proceed, then page-reclaiming happens synchronously (called direct reclaim). However, Linux tries to avoid this scenario by running a background thread called kswapd that begins reclaiming pages when the application reaches a soft resource limit. Kswapd is usually idle, but can be woken up when the kernel detects a soft limit has been met. It also runs with a low priority to avoid wasting resources on speculative evictions (the application may never hit its hard limit).

Page Eviction Paging was originally intended to use hard disks as backing store, and this is reflected in the design of paging in Linux. To swap, one or more block devices must be formatted and mounted as swap devices. Linux then uses the block offset on this disk as a unique identifier for an evicted page (Step 3). In order to support more complex paging schemes (such as page compression, or heterogeneous memory), Linux introduced the

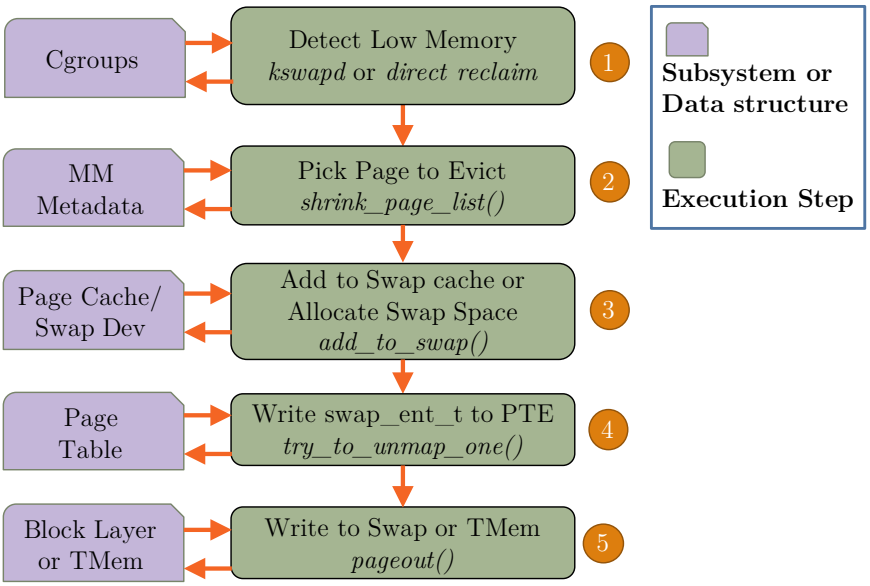


Figure 13: Baseline Linux page eviction (reclaim) code path.

transcendent memory (TMem) layer[43]. This scheme still uses disk offsets as identifiers, but completely bypasses the block layer. This is important because many optimizations in the block layer (e.g., write coalescing and block reordering) are not suitable for these alternative paging devices. Evictions do not immediately result in writes to TMem or a swap device. Instead, pages are stored in a data structure called the swap cache (Step 3). This swap cache helps reference count shared pages, and hedges against poor eviction choices. Once a page is no longer physically available, Linux replaces the corresponding PTE with a swap_entry_t which clears the valid bit, and uses the remaining bits to store the swap device ID (called "type" in the kernel) and block ID (called "offset") (Step 4). When changing a PTEs, most architectures require the OS to flush the TLB. This forces a page-table walk on the next access to this virtual address. Finally, the kernel begins a write to the swap device in the background (Step 5).

Page Fetch When a user program attempts to access a page that has been swapped out, the PTW notices the invalid PTE and issues a page fault trap to the OS (Step 1 in Figure 14). Note that the hardware does not examine the remaining bits (the swap_entry_t is purely a software construct). Upon receiving a page fault, Linux first determines if the requested virtual address has been assigned to this task. It does this by iterating through regions of virtual memory called virtual memory areas (VMAs) (Step 2). If a valid VMA is found, then the OS begins a page table walk to locate the corresponding PTE. There are several reasons that a page fault may occur, the OS must check the PTE to determine the cause (Step 3). Assuming the cause was an invalid PTE, the OS then searches the swap cache for this page (Step 4). This is in case some other process that shares it has already brought it in. If the page is not found, then a new frame is allocated and a transfer is initiated to read the page from the swap device (Step 5). If the page is found in TMem, then the transfer occurs synchronously, otherwise the process initiates the transfer and then yields to the scheduler,

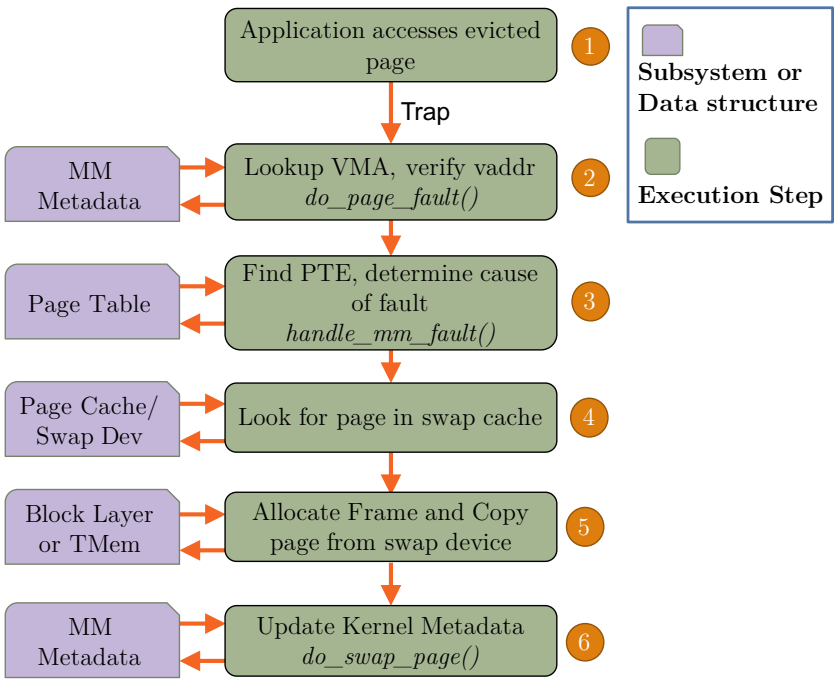


Figure 14: Baseline Linux page fetch code path.

resulting in a context switch. When the transfer is complete, the kernel changes the PTE from a swap_entry_t to a valid PTE with permissions defined by the VMA. Finally, the kernel updates page-tracking meta-data (Step 6). This includes the LRU lists maintained by the eviction algorithm, VMA membership, and a number of other kernel subsystems. Note that several of these updates require synchronization with other kernel threads. Once all bookkeeping is complete, and the PTE is updated, the kernel flushes the TLB, and restarts the application.

4.3.2 PFA Modifications

The introduction of a PFA changes a number of the assumptions underlying baseline paging behavior. Figure 15 summarizes these changes.

Frame Allocation and Permissions Linux uses the faulting virtual address to make a number of decisions during the page fetch process. For instance, the permission bits are taken from the VMA. VMA information is also used to decide which physical frame to use (this is particularly important in NUMA systems). With the PFA, however, the OS must decide on this information at *eviction* time. Pre-allocating physical frames is not an issue in our system because there is only one core, and frame selection does not depend on the VMA. Permission bit selection is more problematic. Our current approach is to assign permission bits to a remote page based on the VMA permissions at the time of eviction, we then update those permissions while performing bookkeeping. In practice, this is unlikely to cause problems as permissions rarely change. Furthermore, Linux is able to correct inappropriately restrictive permissions during page-faults. However, there may be security concerns if permissions are

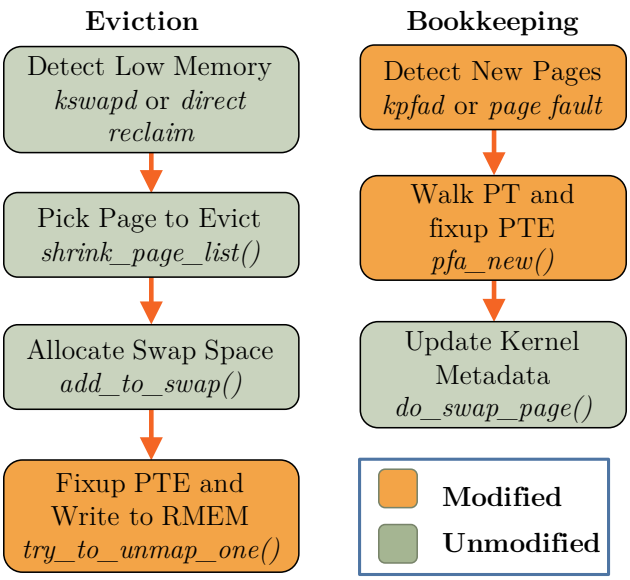


Figure 15: Major changes to Linux paging to accommodate the PFA. Most subsystems could be re-used without change. On the eviction path, all that was changed was the PTE update (to write a remote PTE instead of a swap_entry_t), and the write to disk (to write to remote memory instead). The bookkeeping path is now triggered either from the page-fault handler (due to a PFA service request) or from kpfad. The core bookkeeping function remains unmodified.

made more restrictive while a page is remote. This vulnerability exists in the window between page fetch and bookkeeping. To mitigate this concern, the OS would need to be modified to update remote PTEs when changing VMA permissions.

Under these simplifying assumptions, we are able to allocate frames proactively. The current implementation always refills the FreeQ during bookkeeping. To simplify the bookkeeping procedure, we track each allocated frame in a FIFO. This allows the bookkeeping code to simply pop this FIFO to find which frame was used for each new page (the PFA always drains the FreeQ in FIFO order). Furthermore, this FIFO allows for aggressive sanity checks that aided greatly in system debugging.

Asynchronous Bookkeeping In normal paging, the *do_swap_page()* function is able to update meta-data as soon as a page is fetched. With the PFA, we delay this bookkeeping for a bounded but potentially non-trivial period of time. Many of these bookkeeping tasks are in support of heuristics or resource accounting (e.g., LRU lists for eviction, or memory utilization metrics). Delaying these tasks reduces the accuracy of various algorithms, but does not result in incorrect behavior. Others are needed for correct execution (e.g., VMA membership or shared page tracking for copy-on-write). We address these correctness issues by performing bookkeeping preemptively before accessing any of the related algorithms. These tasks may be fairly common, but they are unlikely to actually involve a recently fetched page. To avoid preemptively performing bookkeeping, we use one of the reserved bits in the PTE protection field to indicate a page that has been recently fetched but not yet processed. This bit gets

set at eviction time, but is cleared during bookkeeping.

Swap Device and Block ID Allocation Linux assumes that all swap activity is backed by a block device and it uses the physical address on this device to identify all evicted pages. This block ID is needed during the bookkeeping process to identify the page. To address this problem we make a number of simplifying assumptions.

1. **A real swap device is available** (even though it is not used). We use a ram-based file system (ramfs) to trick the kernel into thinking it has a large disk attached. This uses no actual physical memory.

2. **There is only one swap device.** This allows us to not track device ID. This is achieved by making the ramfs sufficiently large to address all swap activity.

3. **Block IDs are contiguous on the integers $(\mathbf{0}, \mathbf{2^{28}}]$.** This allows us to pack the block ID into the remote PTE format (see Section 3.1.1). We achieve this by ensuring that the ramfs is the same size as our memory blade (and less than 2^{28} pages). Since block IDs correspond to physical offsets on the swap device, we are guaranteed to never see an invalid block ID.

While these assumptions hold, we are able to compress the swap_entry_t into a 28 bit PageID by eliding the type, and using the offset directly. Finally, we avoid overheads in the block layer by implementing the PFA as a TMem device. Since bookkeeping is asynchronous, and eviction occurs earlier in the process (due to ordering constraints with PTE modifications), this TMem plugin simply returns immediately. The current implementation evicts synchronously. This is because the expected write time is much smaller than a scheduling quantum and asynchronous eviction would result in wasteful context switches. Future implementations may attempt to overlap eviction with low-latency tasks such as bookkeeping.

4.3.3 kpfad

The most basic implementation of PFA support in Linux simply performs bookkeeping tasks whenever the internal queues of the PFA fill up. This effectively batches page bookkeeping, but it does not allow the kernel to choose when the bookkeeping occurs. To leverage idle periods in program execution, or unused hardware threads, we implement a background bookkeeping daemon called kpfad. Kpfad is triggered by an adaptive timer that attempts to discover the average time between full queues. It does this by increasing the wait time by a small amount every time it runs, and decreasing the time whenever the application is interrupted due to full queues. While kpfad gives increased flexibility and efficiency on a lightly loaded system, it causes strictly more overhead than interrupt-driven bookkeeping when the application has enough work to keep all hardware threads busy (since the adaptive timer is not perfect). To avoid this, kpfad is run with very low priority (similar to the page-out daemon kswapd). Unlike kswapd, however, kpfad does not get triggered by a soft limit. We expect the adaptive timer scheme, coupled with low priority, to be sufficient to avoid significant overhead.

4.3.4 Baseline Swapping

We modified Linux to use the remote memory blade while paging. This was done by implementing a software interface to the remote memory blade as a TMem device. The swapping mechanism uses a custom NIC driver that provides zero-copy semantics and bypasses the normal Linux networking stack.

5 Evaluation

5.1 Experimental Design

Our evaluation is based on two benchmarks with significantly different access patterns. The first is quicksort (Qsort). This benchmark first allocates a large array of random numbers, and then sorts it using the well-known quicksort algorithm. Quicksort is a divide and conquer algorithm that automatically partitions the input array into small local blocks before performing a final sort. This leads to excellent cache behavior and predictable access patterns. Furthermore, by allocating the input array dynamically, quicksort performs no file I/O, so it is never blocked on I/O or other OS interactions.

The other benchmark is a de-novo genome assembly benchmark (Gen). Gen begins by loading a large text file that represents raw genome data. Raw genome data consists of short, overlapping, sequences of base-pairs called "contigs", the goal is to align these overlapping contigs into a single contiguous sequence representing a genome. This is done by loading contigs into a large hash table and probing into it repeatedly to find matching sequences. This leads to very little locality and unpredictable access patterns. Furthermore, Gen performs file I/O on the input, which allows for more complex OS interactions.

5.2 End-to-End Performance

The benchmarks were both run under a cgroup in Linux in order to reduce the available memory and emulate a system where applications would need to share limited local memory. This is the same mechanism that system administrators use today to control application memory consumption (e.g., in containers). In this experiment, we disable kpfad in order to isolate the batching of new-page management from the scheduling flexibility offered by kswapd's asynchrony. The PFA was configured to allow up to 64 outstanding page faults before bookkeeping was performed.

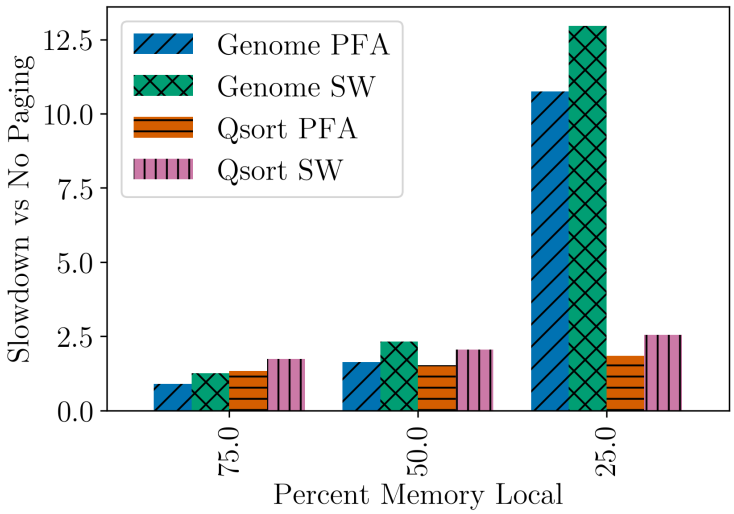


Figure 16: PFA vs Baseline without kswapd. Applications run approximately 20-40% faster when the PFA is enabled.

Both applications use 64MB of memory at their peak. We then varied the cgroup memory limit from 100% (64MB) down to 25% (16MB), triggering increasing levels of paging. For both benchmarks, the PFA reduces end to end run time by up to 1.4x.

5.2.1 Analysis

We now analyze the sources of this performance improvement.

Fetch Times We begin our analysis by looking at the key metric of average fetch time. This is the time between when an application attempts to access a remote page, and when it is able to continue processing. In this experiment, we use a simplified memory blade and network implementation with a constant 4 µs access latency in order to better understand local overheads. Figure 18 plots the time for accessing a single remote page on an unloaded system. We classify time into four categories:

- **Trap:** The time for the hardware to detect an invalid access and context switch to the OS.
- **Proc:** The time spent processing the page locally (overhead).
- **NIC:** The time spent interacting with the network interface.
- **MemBlade:** The time spent on the network and in the memory blade.

Recall from Section 3 (and Figure 6 in particular) that the PFA moves some of this processing (especially **Proc**) to an independent kernel thread; we account for this in a later section.

Note that the trap overhead is a very small fraction of total time (just 113 ns). This is a result of using a simple in-order RISC core like Rocket. It is likely that this overhead may

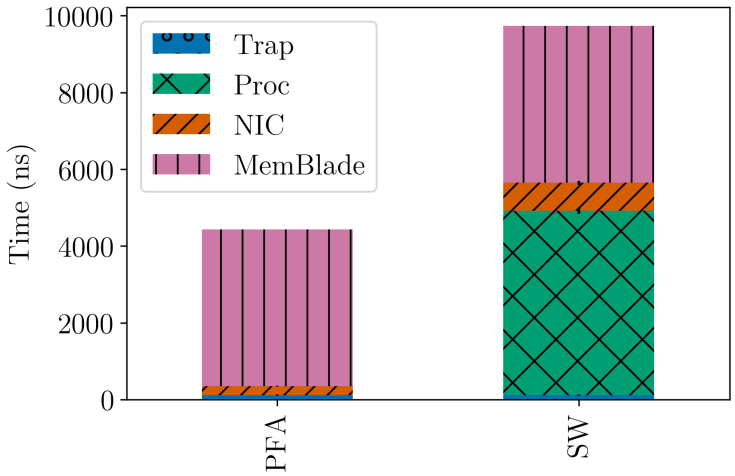


Figure 17: Breakdown of time in fetching a single remote page. All data are the average of 10 runs. Error bars represent standard deviation (but are almost too small to be seen). Note that local processing time (including the trap and NIC interaction) only accounts for 8% of time with the PFA, but accounts for over 50% of time for the baseline.

be more significant on a more complex architecture like server-grade x86 cores. Next, note that the time spent on the network and in the memory blade accounts for less than half the time in the baseline implementation, but completely dominates the PFA fetch time. This effect will be even more pronounced as network and memory blade performance improves. For example, if **MemBlade** time were reduced to 1129 ns to simulate a 1 Tbit/s link with 1 µs round-trip latency (as predicted in Section 2.1 for photonic networks), then client-side processing would account for 83% of time in SW but only 23% of time with the PFA. Finally, note that the **NIC** time in software is larger than the total time in the PFA. We believe this is due to a more efficient hardware to hardware interface between the PFA and the NIC. While not visible in the figure, the actual PFA-specific processing takes only 1 cycle in hardware, the remaining time is split between detecting and delivering the remote PTE to the PFA (**Trap**), and interacting with the NIC (**NIC**). The total time to fetch a page with the PFA is 2.2 times faster than in SW, but this does not tell the whole story; The PFA does not eliminate the work that is done during SW **Proc**, it simply moves it to another thread. Likewise, the 113 ns trap overhead may seem small, but this does not account for the effect that cache pollution from the handler has on the application when it restarts.

Total Page Faults One key function of the PFA is to reduce the number of page faults due to paging. Recall from Section 2.3 that there are many causes for faults (e.g., to perform copy-on-write), in Figure 19 I plot the number of paging-related faults each benchmark experiences as a fraction of total faults. The first thing to note is that the number paging-related faults decreases by approximately 64 when the PFA is used. This is because the PFA interrupts the OS to perform bookkeeping only when its queues are full (every 64 fetches in this experiment). However, these only account for 45% of faults, even in the worst-case (our simplest benchmark, Qsort) with 25% local memory. The more complex Gen benchmark has even fewer paging-related faults (as a fraction of total). While there are certainly some

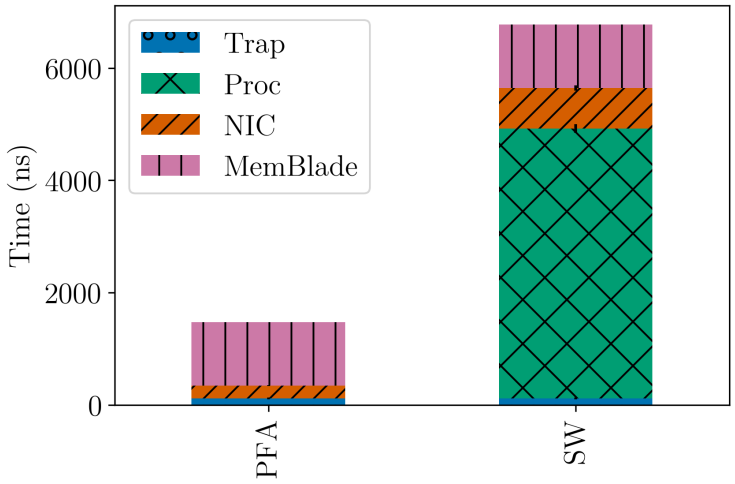


Figure 18: Breakdown of time in fetching a single remote page from a hypothetical fast memory blade with 1 µs page read latency. As network and memory technology improves, the relative benefit of the PFA increases (from 2.2x faster with the baseline memory blade to 4.6x faster with the optimistic memory blade).

savings due to fewer kernel crossings, they are not frequent nor long enough to explain all the performance benefits we see end-to-end.

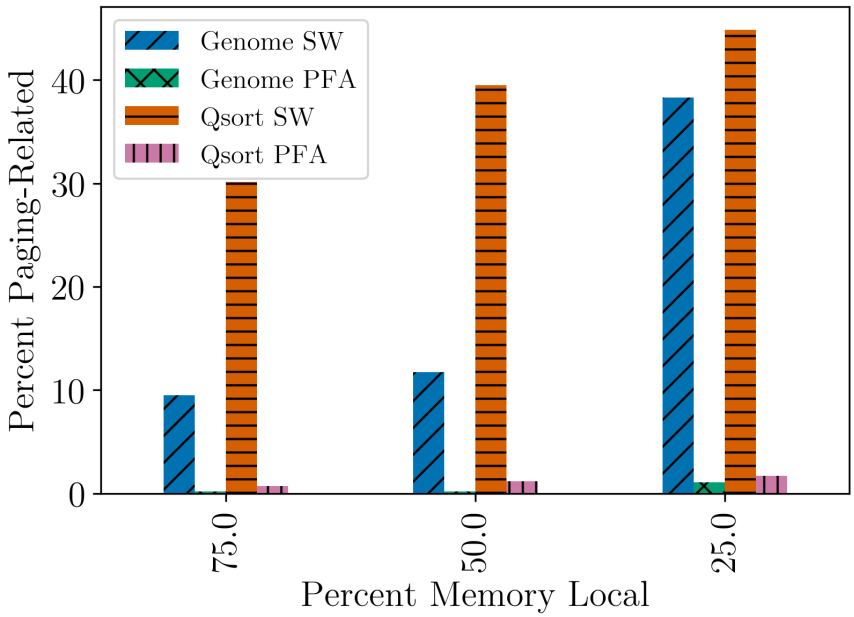


Figure 19: Number of paging-related faults as a fraction of total faults experienced.

Bookkeeping Time While we do reduce the number of paging-related faults, the kernel still needs to perform bookkeeping on the same number of pages. This batching means that more work is performed per page fault with the PFA. Figure 20 shows total time spent bookkeeping, regardless of the number of page faults. What we see is that while the number of evicted pages is the same in both configurations, using the PFA leads to a 2.5x reduction in bookkeeping time on average. The same code path is executed for each new page, but the PFA batches these events, leading to improved cache locality for the OS, and fewer cache-polluting page-faults for the application. The result is that, even in the worst case, the PFA spends less than half its time handling paging-related faults, while the baseline spends about 80%.

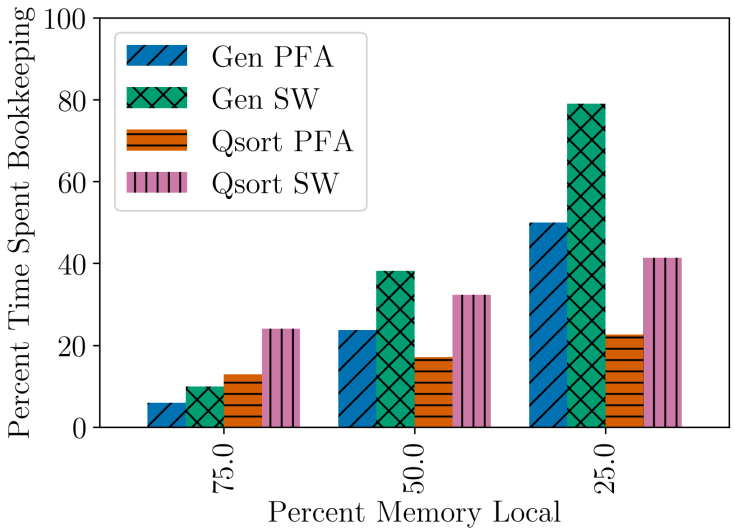


Figure 20: Proportion of Time Spent Bookkeeping

Scaling Figure 21 shows the improvement in end-to-end runtime due to the PFA on our applications. While the improvement is significant (up to 40%), the savings are constant (there is no asymptotic improvement). This is because the PFA does not change any of the caching algorithms, and therefor experiences the same number of faults. This means that applications (such as Gen) that are not particularly cache-friendly can see significant slowdowns in a disaggregated environment, even with the PFA. Ultimately, the PFA pushes the boundaries of what is possible with cache-like interfaces, but it cannot change their fundamental limitations. Applications like Gen will need deeper changes to be viable on a disaggregated system.

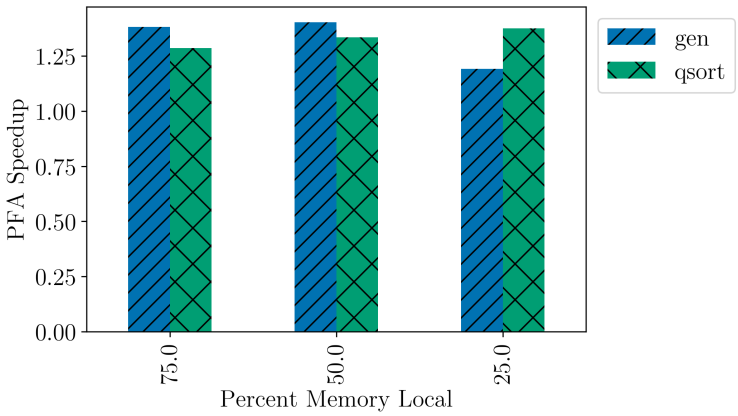


Figure 21: Total runtime improvement due to PFA

6 Future Work

So far, we have only experimented with a single client and memory blade. However, this does not capture all of the effects that contribute to performance. In particular, it would be valuable to understand congestion at both the memory blade, and in the network. Another effect that may surface as we experiment with more distributed applications is tail latency. The introduction of a remote page fault could increase tail latency significantly and would require mitigation.

Another topic not addressed by the current PFA and memory blade designs is that of management and security. How should memory blade capacity be allocated to applications? How can such allocations be authenticated? Simple payload encryption may not be sufficient to protect applications from attackers with network-level access.

Finally, while caching can be very effective for some workloads, it is not appropriate everywhere. Even within one workload, caching may be appropriate for some data structures, but not for others. To allow users maximum flexibility to choose between performance and convenience, we plan to implement a hybrid cache/scratchpad interface to remote memory (similar to [44]). In this system, application memory would default to demand paging, but certain portions could be pinned in specially allocated regions of local memory. The application would then be responsible for directly writing to and from remote memory.

7 Conclusion

Disaggregated memory systems promise to simplify deployment, allocation, and scheduling on next-generation WSCs, but bring significant performance and complexity challenges. This complexity cannot be mitigated with shallow application changes because assumptions about system performance run deep. While caching attempts to avoid these challenges, we find that virtual memory paging is no different; the OS mechanisms implementing paging were designed in a world with millisecond-level access latencies and are not suitable for the microsecond access times offered by remote memory. The PFA allows us to make the deep changes needed to accommodate this new environment. By improving end-to-end application performance by up to 40%, the PFA enables a greater range of applications to run on limited local memory. However, caching is a general purpose approach, and applications with large working sets and poor locality will always suffer from increased main memory access times. To take full advantage of disaggregated memory, we will need a mix of interfaces, both implicit, and explicit.

Acknowledgements

I would like to first thank Emmanuel Amaro who was integral to the design, motivation, and hardware implementation of the PFA. Howard Mao contributed significantly to general infrastructure and implemented the memory blade. The FireSim simulator was the work of many in the Berkeley architecture research group (BAR), especially Sagar Karandikar. I would also like to thank those in the BAR group, and the RISC-V Foundation for supplying the underlying infrastructure for this research. Finally, I acknowledge my advisers Prof. Randy Katz and Prof. John Kubiatowicz, as well as Prof. Krste Asanovic, for their guidance throughout this project.

Appendices

A Reproducibility

All code for the experiments in this thesis is available on Github (some repositories may be private or move to other URLs, contact the author for access):

- Software Environment
  - Repository: https://github.com/firesim/firesim-software
  - Branch: nathan_pfa
  - Commit: 442324046a08cd51e0b8b9198effac7b649196a5
- Simulation Platform
  - Repository: https://github.com/firesim/firesim
  - Branch: pfa
  - Commit: 2c7e113b4f317ad6ddece12705667c79ea30fb1f

B PFA Specification

B.1 Overview

The page-fault accelerator (PFA) is a device for RISC-V based cpus to store virtual memory pages to remote memory and fetch them automatically on a page-fault. With a PFA, the latency-sensitive critical path of swapping-in a page from remote memory is handled in hardware, while more complex behaviors like page-replacement policies are handled asynchronously by the OS.

B.1.1 Basic Usage

1. Eviction:
   (a) The OS identifies pages that should be stored remotely.
   (b) It evicts them explicitly by writing to the evict queue.
   (c) The OS stores a page identifier in the PTE and marks it as remote.
2. Provide Free Frames:
   (a) The OS identifies one or more physical frames that can be used to house newly fetched pages.
   (b) It gives them to the PFA through the free queue.
3. Page Fault:

(a) Application code issues a load/store for the (now remote) page.

(b) The PFA automatically and synchronously brings the page from remote memory and stores it in a free frame.

(c) The PFA clears the remote bit in the pte.

(d) The PFA pushes the virtual address of the fetched page to the new page queue.

(e) The application is restarted.

4. New Page Management

   (a) The OS periodically queries the new page queue and performs any necessary bookkeeping.

B.1.2 Free Frame and New Page Queue Management

The OS should ensure that there are sufficient free frames in the free queue to ensure smooth operation. If a remote page is requested and there are no free frames, the PFA will trap to the OS with a conventional page-fault. The OS must enqueue one or more free-frames before returning from the interrupt. This may involve evicting pages synchronously in the page-fault handler.

Similarly, the OS needs to drain the new page queue periodically to ensure it does not overflow. This will also trap to the OS with a conventional page fault.

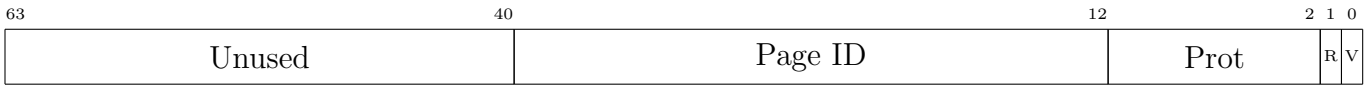
The OS can differentiate a regular page-fault interrupt from a "PFA out of free-frames" interrupt by checking if the requested page has the remote bit set in its PTE. To tell the difference between a full free-frames queue and full new-pages queue, the OS can query the FREE_STAT and NEW_STAT ports.

B.1.3 Limitations

- The current PFA design does not support multiple cores.
  - **Example**: Queues cannot be atomically queried for free space and pushed to.
  - **Example**: PFA directly modifies page table with no regard for locking or other MMUs.
- The PFA does not handle shared pages.

B.2 RISCV Standards

User Spec: 2.1 (RV64 only)
Priv Spec: 1.10 (Sv39 or Sv48 only)



B.3 PTE

Remote pages use a unique PTE format:

Fields:

- **Valid (V):** Valid Flag
  - **1** Indicates that this page is valid and shall be interpreted as a normal Sv48 PTE.
  - **0** indicates that this PTE is invalid (the remote bit will be checked).
- **Remote (R):** Remote memory flag.
  - **1** indicates the page is in remote memory.
  - **0** indicates that this page is simply invalid (access triggers a page fault).
  - *Note*: This is an incompatible change from the RISC-V privileged spec 1.10 which specifies that bits 1-63 are don't cares if the valid bit is 0. This is compatible in-practice with the current RISC-V Linux implementation.
- **Protection:** Protection bits to use after a page is fetched. These match the first 10 bits of a standard PTE.
  - *Note:* This includes a valid bit which may differ from the Remote PTE valid bit. If this is 'invalid', the PFA will fetch the remote page, but then trigger a page-fault anyway.
- **Page ID:** A unique identifier for this page.
  - Must match a pageID that was evicted and not-yet-fetched.

B.4 MMIO

Name	Value
BASE	0x10017000
FREE	BASE
FREE_STAT	BASE + 8
EVICT	BASE + 16
EVICT_STAT	BASE + 24
NEW_PGID	BASE + 32
NEW_VADDR	BASE + 40
NEW_STAT	BASE + 48
INIT_MEM	BASE + 56

Basic PFA MMIO behavior is described below. Operations marked "Illegal" will result in a load/store access fault.

B.4.1 FREE

Provide free frames to the PFA for use when fetching remote pages.

Load Illegal

Store Expected Value: physical address (paddr) of frame to publish

Write the physical address of an unused memory frame to FREE to publish it for use in subsequent remote fetches.

The FREE queue is bounded. Users may query FREE_STAT before pushing to ensure there is sufficient space. Storing to FREE while the queue is full is illegal.

B.4.2 FREE_STAT

Query status of free queue.

Load Returned Value: Number of unused slots in the free queue. Returning 0 means it is illegal to store to FREE.

Store Illegal

B.4.3 EVICT

Evict pages to remote memory and check if the eviction is complete.

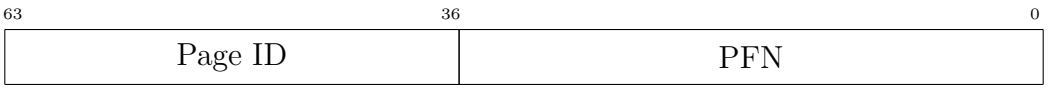
Load Illegal

Store Expected Value: Packed eviction uint64 containing pfn and pgid

Eviction requires two values (packed into a single 8-byte value, see below):

- pfn: Page frame number. This is the physical address of the page shifted 12 bit to the right (since the first 12 bit are always 0 in page-aligned addresses).
- Page ID: A unique 28 bit value to be associated with the page. Must be unique among all currently evicted pages (pgids may be reused after being seen in the newq)

The two values must be packed into a single 8-byte value as follows:



Eviction is asynchronous. Multiple pages may be enqueued for eviction simultaneously. Users may check EVICT_STAT before storing to ensure there is sufficient room in the queue. Storing to EVICT while full is illegal.

Note: In principle, it may be possible for remote memory to be full (or exceed allocations). It is expected that the OS will track this out of band. A store to EVICT when remote memory is full is illegal.

B.4.4 EVICT_STAT

Query status of evict queue.

Load Returned Value: Number of unused slots in the evict queue. Returning 0 means it is illegal to store to EVICT. Returning EVICT_MAX (size of evict Q) means all pages have been successfully evicted.

Store Illegal

B.4.5 NEW_PGID

Check which pages have been fetched automatically by the PFA (useful for OS bookkeeping)

Load Returned Value: Page ID of oldest fetched page that has not been reported (FIFO order).
Note: It is illegal to load from an empty new queue. You must check NEW_STAT before loading from NEW.
Note: Unlike EVICT, NEW always reports every fetched page. Since it may be bounded, it is important for SW to drain this queue periodically. A full new queue will result in a page-fault being delivered to the OS.

Store Illegal

B.4.6 NEW_VADDR

Same as NEW_PGID but returns the vaddr of fetched pages.

Load Returned Value: virtual address of oldest fetched page that has not been reported (FIFO order).
Note: It is up to the user to keep these queues in sync. Ideally they would both be loaded from at the same time.

Store Illegal

B.4.7 NEW_STAT

Query status of new page queue.

Load Returned Value: Number of new pages in the queue.
Note: It is undefined which size (NEW_VADDR or NEW_PGID) is being reported. It is required to pop both queues together.

Store Illegal

B.4.8 INIT_MEM

Provide 1 page of scratch memory to the PFA. Software must not write to this page after registering it with the PFA. There is no way to revoke or de-initialize the PFA.

Store Expected Value: Physical address of 1 page of memory to be used by the PFA

Load Illegal

C Memory Blade Protocol

C.1 Ethernet header

Field (Size in Bytes)	Bits	Values	Description
src_mac (6)			
dst_mac (6)			
ethertype (2)			
vlan_id (2)		Optional - set if previous ethertype is 0x8100	
ethertype (2)		Optional - carries inner ethertype if vlan tag present	

C.2 Request Packet

Field (Size in Bytes)	Bits	Values	Description
version (1)		0x1 - draft	Protocol version
op_code (1)		0x0 - page read 0x1 - page write 0x2 - word read 0x3 - word write 0x4 - atomic add 0x5 - compare and swap	Operation identifier
part_id (1)			Message part number
reserved (1)			
transact_id (4)			Transaction ID
page_num (8)			
payload Page Read (0) Page Write (1368/1368/1360) Word Read (8) Word Write (16) Atomic Add (16) CAS (24)			

C.3 Atomic/Small Payload

Field (Size in Bytes)	Bits	Values	Description
offset + disable + size (2)	15-4 offset 3-2 reserved 1-0 size	For size: 0 - 8 bits 1 - 16 bits 2 - 32 bits 3 - 64 bits	Offset within page + Log2 of size in bytes
Reserved (6)			
arg0 (8)			The data to swap/add
arg1 (8)			The data to compare

C.4 Response Packet

Field (Size in Bytes)	Bits	Values	Description
version (1)		0x1 - draft	Protocol version
resp_code (1)		0x80 - success w/ page 0x81 - success w/o data 0x82 - success w/ word ¿= 0x83 - error	Response code
part_id (1)			Message part number
reserved (1)			
transact_id (4)			Transaction ID
payload w/ page (1368/1368/1360) w/ word (8)			

Glossary

anonymous page A page that does not contain disk-backed information. This is primarily "heap" memory (e.g. memory allocated through malloc()) 20

bookkeeping The internal OS-specific tasks related to bringing in a new page. This typically includes updating meta-data and other page-tracking activities. 11, 12

cgroup The per-task (or group of tasks) resource management system in the Linux kernel. 20

dissaggregation The WSC design that moves compute resources (such as memory, disk, and CPUs) into dedicated resource-blades that are connected through a high-performance network. 5

EvictQ Queue of pages to be evicted by the PFA. Populated by the OS when it needs to free local physical memory. 13

frame Synonym for page frame *Glossary:* page frame

FreeQ Queue of free frames to be used by the PFA to service page-faults. 13, 15, 23

kpfad A background daemon that opportunistically performs bookkeeping and maintenance for the PFA. 23, 24

kswapd A background kernel thread that opportunistically performs bookkeeping. 20, 24

least recently used An algorithm that attempt to pick pages that have not been used recently. 20, 22, 41

maximum transfer unit The largest contiguous packet that a network is capable of transmitting. 16, 41

memory blade A dedicated memory node in a disaggregated system. The memory blade exists solely to server memory requests. A memory blade may be custom-designed for this purpose, or may simply expose an RDMA interface. 16

NewQ Queue of new-page descriptors populated by the PFA on every page fault and drained by the OS for bookkeeping. 13

non-uniform memory access A system where memory is cache-coherently available to multiple CPUS, but with varying access latencies and bandwidths (a type of multi-socket machine). 7–9, 41

non-volatile memory Storage devices with near-DRAM performance, and byte-addressablity, that do not lose their data when powered off. 5, 41

page A fixed-size logical group of data (typically 4 KiB). Sometimes called "virtual page". 9

page fault accelerator The proposed hardware-accelerator that handles page-faults for remote pages automatically. 5, 6, 41

page table A hardware-visible tree in main memory that contains translations from virtual to physical addresses. 9, 21

page table entry A single entry of the page-table. Each PTE refers to a single virtual page. 9, 14, 21, 23, 41

page table walker A hardware device to automatically walk the page-table and locate PTEs for a particular virtual address. 10, 21, 41

pageID A unique identifier for a page in remote memory. Acts as a remote-memory address. 15, 17

paging The process of storing logical pages on an external storage device in order to free physical memory. Also called "swapping". 5

remote direct memory access A system where memory is directly addressable between multiple nodes through a network interface. RDMA systems are not typically cache-coherent. 5, 8, 9, 41

swap Historically used to refer to the process of moving an entire process's memory image to disk, Linux uses "swapping" to refer to all paging. see 20

swap_entry_t A Linux-specific value stored in evicted PTEs that contains information on where to locate an evicted page. 21–24

system in package A system where all the necessary components (NIC, cpu, memory) are grouped into the same physical package (but not necessarily on the same chip) 5, 6, 41

task Linux kernel internal abstraction of a process. 12, 20

total cost of ownership A metric that includes not only up-front costs of a system, but the total cost to own and operating that system for its effective lifespan. 7, 41

transcendent memory A layer in the Linux paging subsystem that stores pages in specialized memory that may not be disk-backed. 21, 24, 25, 41

translation look-aside buffer A cache of virtual to physical address translations. 9, 21, 22, 41

virtual memory area Contiguous region of virtual memory used by Linux to simplify memory management. 21–23, 41

warehouse-scale computer Generic term referring to tightly-integrated clusters of machines deployed in the data center. 5–7, 19, 31, 41

Acronyms

LRU least recently used 20, 22

MTU maximum transfer unit 16

NUMA non-uniform memory access 7–9

NVM non-volatile memory 5

PFA page fault accelerator 5, 6

PTE page table entry 9, 14, 21, 23

PTW page table walker 10, 21

RDMA remote direct memory access 5, 8, 9

SiP system in package 5, 6

TCO total cost of ownership 7

TLB translation look-aside buffer 9, 21, 22

TMem transcendent memory 21, 24, 25

VMA virtual memory area 21–23

WSC warehouse-scale computer 5–7, 19, 31

References

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson, "The berkeley networks of workstations (now) project," in *Proceedings of the 40th IEEE Computer Society International Conference*, COMPCON '95, (Washington, DC, USA), pp. 322–, IEEE Computer Society, 1995.

[2] K. Asanović, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers," FAST 2014.

[3] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dReDBox project vision," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 690–695, March 2016.

[4] HP Labs, "The machine." https://www.labs.hpe.com/the-machine, 2017.

[5] Huawei, "High throughput computing data center architecture - thinking of data center 3.0." www.huawei.com/ilink/en/download/HW_349607, 2014.

[6] Intel, "Intel rack scale design." https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2017.

[7] Facebook, "Disaggregated rack." http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf, 2013.

[8] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 649–667, USENIX Association, 2017.

[9] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 249–264, USENIX Association, 2016.

[10] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat caches for scale-out servers," *IEEE Micro*, vol. 37, pp. 90–103, Mar. 2017.

[11] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A Fully Associative, Tagless DRAM Cache," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 211–222, ACM, 2015.

[12] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avižienis, S. Lin, B. R. Moss, R. Kumar, F. Pavanello, A. H. Atabaki, H. M. Cook, A. J. Ou, J. C. Leu, Y.-H. Chen, K. Asanović, R. J. Ram, M. A. Popović, and V. M. Stojanović, "Single-chip microprocessor that communicates directly using light," *Nature*, vol. 528, pp. 534–538, dec 2015.

[13] C. Batten, A. Joshi, J. Orcutt, C. Holzwarth, M. Popovic, J. Hoyt, F. Kartner, R. Ram, V. Stojanovic, and K. Asanovic, "Building manycore processor-to-dram networks with monolithic cmos silicon photonics," *IEEE Micro*, vol. PP, no. 99, p. 11, 2016.

[14] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: Its time for a redesign," *Proc. VLDB Endow.*, vol. 9, p. 528539, Mar 2016.

[15] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, p. 241251. ISCA 97, ACM, 1997.

[16] "Sgi uv 3000, uv 30: Big brains for no-limit computing," tech. rep., Silicon Graphics International Corp., 2016.

[17] "RoCE in the Data Center," tech. rep., Mellanox, October 2014.

[18] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," RFC 5040, RFC Editor, October 2007.

[19] "Deploying hpc cluster with mellanox infiniband interconnect solutions," tech. rep., Mellanox Technologies, 2017.

[20] "Edr infiniband." https://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Friday/friday_01.pdf, 2015. 11th Annual OpenFabrics Alliance Workshop.

[21] "Ccix consortium." https://www.ccixconsortium.com, 2017.

[22] B. Wile, "Coherent accelerator processor interface (capi) for power8 systems," tech. rep., IBM Systems and Technology Group, September 2014.

[23] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 3–18, ACM, 2014.

[24] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, *System-level implications of disaggregated memory*, p. 112. IEEE, 2012.

[25] Gen-Z Consortium, "Gen-z overview," tech. rep., 2016.

[26] "What is Linux Memory Policy?." https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt, 2017.

[27] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*. 1999.

[28] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, *Latency-tolerant software distributed shared memory*, p. 291305. 2015.

[29] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, (Santa Clara, CA), pp. 1–16, USENIX, 2014.

[30] A. Dragojevi, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, *No Compromises: Distributed Transactions with Consistency, Availability, and Performance*, p. 5470. ACM, 2015.

[31] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, *HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter*, p. 521534. ACM, 2017.

[32] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, *Mojim: A Reliable and Highly-Available Non-Volatile Memory System*, p. 318. ACM, 2015.

[33] K. A. . Andrew Waterman, ed., *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. RISC-V Foundation, May 2017.

[34] "Linux kernel." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/, 2017.

[35] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," Tech. Rep. UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[36] Y. L. Andrew Waterman, "Risc-v isa simulator." https://github.com/riscv/riscv-isa-sim, 2017.

[37] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi, *Chisel: constructing hardware in a scala embedded language*, p. 12161225. ACM, 2012.

[38] K. Asanović, R. Avižienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[39] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Tech. Rep. UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[40] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovic, "The hwacha vector-fetch architecture manual, version 3.8.," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262*, 2015.

[41] S. Karandikar and M. Champion, "Bringing datacenter-scale hardware-software co-design to the cloud with firesim and amazon ec2 f1 instances." https://aws.amazon.com/blogs/compute/bringing-datacenter-scale-hardware-software-co-design-to-the-cloud-with-firesim-and-amazon-ec2-f1-instances/, 2017. AWS Compute Blog.

[42] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Second Edition*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 3 ed., 2005.

[43] D. Magenheimer, "Transcendent memory in a nutshell." https://lwn.net/Articles/454795/, 2011.

[44] H. Cook, K. Asanovi, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," Tech. Rep. UCB/EECS-2009-131, EECS Department, University of California, Berkeley, Sep 2009.