Project PBerry: FPGA Acceleration for Remote Memory

Irina Calciu
VMware Research

Ivan Puddu
ETH Zürich

Aasheesh Kolli
Penn State & VMware Research

Andreas Nowatzyk
Jayneel Gandhi
VMware Research

Onur Mutlu
ETH Zürich

Pratap Subrahmanyam
VMware

Abstract

Recent research efforts propose remote memory systems that pool memory from multiple hosts. These systems rely on the virtual memory subsystem to track application memory accesses and transparently offer remote memory to applications. We outline several limitations of this approach, such as page fault overheads and dirty data amplification. Instead, we argue for a fundamentally different approach: *leverage the local host's cache coherence traffic to track application memory accesses at cache line granularity.* Our approach uses emerging cache-coherent FPGAs to expose cache coherence events to the operating system. This approach not only accelerates remote memory systems by reducing dirty data amplification and by eliminating page faults, but also enables other use cases, such as live virtual machine migration, unified virtual memory, security and code analysis. All of these use cases open up many promising research directions.

CCS Concepts • **Hardware → Hardware accelerators**; • **Software and its engineering → Distributed memory**.

Keywords remote memory, cache coherence, FPGA

ACM Reference Format:
Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project PBerry: FPGA Acceleration for Remote Memory. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3317550.3321424

1 Introduction

Modern networks offer microsecond-scale latencies. These networks enable *remote memory* systems that transparently pool memory from multiple hosts [8, 69], decreasing per-server memory over-provisioning and improving overall memory utilization. These systems use the Virtual Memory Subsystem (VMS) in the operating system (OS) as the primary mechanism for two major tasks: (1) *demand paging* memory pages from remote hosts to cache them in local DRAM and (2) *eviction* of memory pages cached locally back to the remote host that owns the pages. Eviction uses the VMS for *dirty data tracking*, which improves network utilization by writing only the modified (*dirty*) pages back to the remote host.

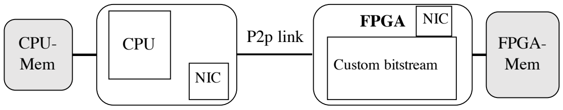

Figure 1. An FPGA connected to a CPU through a coherent point-to-point (p2p) interconnect. Both the CPU and the FPGA have DRAM attached, called CPU-Mem and FPGA-Mem, respectively. The FPGA has a NIC to connect to the network.

However, using the VMS to implement remote memory systems is antithetical to the VMS' primary purpose and results in high overheads (§2). We describe three major reasons for these overheads: (1) The key VMS limitation is that it uses 4KB or larger pages, while applications access and modify data at a finer granularity. This results in high amplification for dirty data tracking and high network utilization for both demand paging and eviction. Most of the data cached through demand paging remains unmodified compared to the original data in the remote host, yet a large fraction of it is marked dirty by page-granularity dirty data tracking and has to be copied over the network during eviction. For example, we show that between 76% and 96% of a 4KB page, on average across the modified pages, is incorrectly marked dirty due to page-granularity tracking. This amplification is even higher for larger pages (99% for 2MB pages). (2) Applications incur expensive page faults for demand paging, which are the main bottleneck for remote memory accesses [8], and additional write page faults, caused by write-protecting the pages for dirty data tracking. For Redis, a key-value store, write page faults decrease throughput by 13%, while increasing latency by 15%. (3) Applications incur additional delays indirectly caused by using the VMS, during the time when the OS is write-protecting pages, flushing TLBs, etc. The

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HotOS '19, May 13–15, 2019, Bertinoro, Italy

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6727-1/19/05.
https://doi.org/10.1145/3317550.3321424

time an application needs to be paused to write-protect its pages increases proportionally to the size of its memory.

To enable fast and efficient remote memory, the OS needs new low-overhead, fine-grained mechanisms to gain visibility into application memory read/write behavior. Our central insight is to use the hardware Cache Coherence (CC) protocol, which already tracks application reads and writes to memory at cache line granularity. Project PBerry *leverages information from hardware coherence mechanisms to augment the VMS for demand paging and dirty data tracking*, thereby reducing remote memory overheads. For example, we show how PBerry uses cache line miss requests to reduce the number of page faults incurred for demand paging remote memory (§5). Furthermore, tracking CC events enables monitoring application memory accesses at cache line granularity, allowing PBerry *to decouple memory access tracking from the VMS page size*, thereby eliminating the key limitation of the VMS. Tracking cache line write-backs allows PBerry to reduce dirty data amplification by 50-95% compared to 4KB VMS pages.

To gain access to the CC protocol, PBerry relies on emerging cache-coherent FPGAs [58] (Figure 1). These FPGAs share coherent memory with a CPU through a point-to-point interconnect, such as Intel UPI [54], CXL [72] or CCIX [1] (§3). The FPGA not only gets to participate in the CC protocol, but it is also able to *observe* and *influence* its behavior with small changes to the coherence blocks on the FPGA. Using these capabilities, we build PBerry (§4) and describe the implementation of an efficient remote memory system (§5). While our focus is on remote memory, we believe that PBerry can benefit a wide variety of use cases, such as those in unified virtual memory, security and live virtual machine migration (§6), which open up a myriad of promising research directions. Finally, we present preliminary emulation results, which show that PBerry can reduce the dirty data amplification by 50-95% and eliminate write page faults (§7).

2 Current systems limitations

Recent remote memory systems [7, 8, 28, 70] rely on virtual memory for two tasks: (1) demand paging from a remote host, which caches remote pages in local memory (*remote memory caching*) and (2) eviction from the local memory back to the remote host (*local memory reclamation*). Virtual memory is additionally used for tracking application modifications to pages cached in local memory (*dirty data tracking*) to reduce the amount of data written back to the remote host, as only *dirty* pages need to be written back.

As remote memory operates at microsecond latencies [7] in contrast to traditional millisecond-latency devices [11], the overhead of page faults in caching remote pages is prohibitive [25, 46]: e.g., for each GB of memory accessed from a remote host, an application can experience a 44% degradation in performance due to page faults [8]. In addition, we identify three major problems in using virtual memory for dirty data tracking:

(1) Dirty data amplification. Memory management on current operating systems assumes a fixed page granularity (e.g., 4 KB for regular pages, 2 MB or 1 GB for large pages). Using virtual memory for tracking dirty data results in *dirty data amplification*, because an entire page is marked dirty, even if only a small part of it was actually modified. This impacts systems that copy the dirty data over the network (e.g., remote memory, live migration), as the network bandwidth and additional power are wasted by copying unnecessary data. Dirty data amplification also impacts persistent memory applications, which can flush modified data at cache line granularity [36], but are restricted to tracking dirty data at page granularity through virtual memory. To overcome this challenge, the Linux community added a *go-faster* flag to mmap. This allows applications to manage their own dirty cache lines, resulting in a 10X performance improvement compared to a traditional *msync* that flushes pages [24], despite heavy criticism that *go-faster* breaks POSIX semantics.

We use Pin [4] to quantify the dirty data amplification for 1) Redis [5], a data structure server, running a random and a sequential workload, and 2) Metis [50], an in-memory MapReduce framework running linear regression and histogram. We consider three granularities for dirty data tracking (4KB page, 2MB page and 64B cache line); for each one we report the amplification as the percentage of the block (i.e., page or cache line) that is clean, but incorrectly reported as dirty (due to actually considering the *entire* block as dirty). Table 1 shows that even regular 4KB pages cause large amplification (82-96%). Large pages increase the amplification to over 99%. In practice, systems that use virtual memory for dirty data tracking choose to break large pages into 4 KB pages during tracking to reduce the dirty data amplification, despite overheads due to TLB flushes, TLB misses and additional translations (e.g., live migration [74]). Instead, tracking at cache line granularity (64B) results in much smaller amplification (17-60%) and less dirty data.

Application	Mem (GB)	Amplif. % (4 KB)	Amplif. % (2 MB)	Amplif. % (64 B)
Redis-random	3.93	96.82	99.997	31.70
Redis-seq	0.13	82.91	99.83	17.18
Linear regression	40	91.73	99.98	18.81
Histogram	40	88.02	99.95	60.20

Table 1. Dirty data amplification at different granularities.

(2) Application slowdown due to write page faults. When dirty data tracking is enabled, writable pages need to be write-protected, which leads to additional page faults, called *write page faults*, which are necessary to remove the write-protection. Therefore, applications experience lower throughput and higher latency, e.g., for Redis we see a 13.3% decrease in throughput and a 15.24% increase in latency due to write page faults (§7).

(3) Application pause time while write-protecting pages. Write-protecting pages requires finding and modifying page table entries and flushing the TLBs. Thus, the OS needs to pause the application during this time. Write-protecting pages can be done in batches, to minimize the impact on the application. However, the total time spent write-protecting pages increases proportionally to the size of the application's in-memory data and can be prohibitive for latency-critical applications. For example, Metis running linear regression on 40 GB data is paused for 7 ms, while the OS is write-protecting all pages (§7).

3 Cache-coherent FPGAs (ccFPGAs)

Field Programmable Gate Arrays (FPGAs) are integrated circuits that consist of programmable logic blocks. Unlike Application-Specific Integrated Circuits (ASICs) that are designed once and manufactured for a specific purpose, FPGAs can be re-configured after manufacturing, and their usage can evolve over time between different applications or different versions of the same application. In this paper, we focus our attention on cache-coherent FPGAs (ccFPGAs): those that share memory with a CPU and access to such memory employs a cache-coherence protocol to keep the memory consistent across the CPU and the FPGA. For such architectures, the FPGA is connected to the CPU using a point-to-point interconnect implementing a cache coherence protocol (Fig. 1), such as MESI [61] or one of its variants. Various such platforms have been proposed, such as IBM's CAPI [59] for the POWER8/POWER9, the Convey Computer [21] and Enzian, a research computer [2]. Other cache-coherent interconnects between a CPU and an FPGA are expected to become available commercially in the near future: Intel UPI [54] connecting an Intel Xeon to a Stratix X FPGA [37, 47], CCIX [1] connecting an AMD/ARM CPU to a Xilinx FPGA, and the new interconnect based on the CXL [72] specification. The FPGA might have its own memory attached (*FPGA-Mem*), but this is not required (§4.1). We call the main system memory *CPU-Mem*. We assume the FPGA package contains network interface card (NIC) logic and an open implementation of the coherence protocol.

4 Project PBerry

PBerry's key goal is to enable fast and transparent application memory access tracking at cache line granularity. PBerry also accelerates local and remote data copy. We present the PBerry design: an FPGA module (§4.1), a software component (§4.2) and a software emulation component (§4.3). We discuss a few alternative choices and outline their tradeoffs. We describe how to use PBerry for remote memory in §5.

4.1 PBerry FPGA Module (PBF)

The PBerry FPGA Module (PBF) runs on a cache-coherent FPGA, which already implements a memory agent that manages memory attached to the FPGA (FPGA-Mem) or a cache agent that manages coherent CPU memory shared with the FPGA. We discuss two alternative solutions, one based on the memory agent and one based on the cache agent.

Memory agent solution. Consider an application running on the CPU and accessing data from FPGA-Mem. Last Level Cache (LLC) misses and write-backs from the CPU are transmitted over the coherent link to the memory agent responsible for FPGA-Mem. The memory agent tracks the coherence traffic in order to respond to requests from CPU caches. PBF uses the memory agent to discover information about memory accesses and then stores and exposes these events to the OS (§4.1.2). For example, tracking cache line write-backs allows PBF to identify dirty data at cache line granularity. However, a cache line write-back is asynchronous, as it occurs when the CPU cache evicts the cache line. Therefore, PBF might have stale information about a cache line that is still cached in the CPU caches. To determine the ground truth during a page eviction, PBF snoops the cache line, invalidating it in the CPU cache and forcing the write-back to happen. The downside of the memory agent approach is that PBF's visibility is limited to data in FPGA-Mem, which is limited in size and incurs higher CPU access latency.

Cache agent solution. An alternative PBF implementation is to extend the cache agent on the FPGA instead of the memory agent. In this scenario, PBF uses the cache agent to mimic having a large cache that can fit the tracked memory regions, by modifying the FPGA cache structure to house only the cache line tags, without storing the cache line data. PBF issues a read snoop request for each cache line that needs to be tracked and effectively "caches" it (in reality only its tag is stored). A subsequent CPU access to a cache line will result in a coherence transaction sent to PBF, which marks the cache line as accessed or dirty, depending on the type of transaction received (read or write, respectively), and invalidates the cache line. PBF needs to re-snoop that cache line later to enable tracking again. This approach allows PBF to track memory from both CPU-Mem and FPGA-Mem, but it requires changing the cache agent and the cache structure to allow storing only tags, without data. Therefore, in the rest of this paper we focus on the *memory agent* solution.

4.1.1 PBF primitives

PBF can track three different types of pages: application data pages, application code pages, and kernel pages (such as page table pages). We propose an interface that exposes the following PBF interactions with the cache coherence protocol to the OS: (1) **Read tracking** - PBF keeps track of cache lines requested by applications in shared mode and discovers the application's access pattern. (2) **Write tracking** - PBF keeps track of cache lines that have been evicted from the CPU caches with modified data (dirty data tracking). (3) **Local copy** - PBF can copy cache lines in the background to a new location in local memory. This feature can enable an efficient cache line granularity copy-on-write mechanism. (4) **Remote copy** - PBF can also copy cache lines or pages to a remote host. This is useful for enabling use cases such as remote memory or live migration. (5) **Inplace transform** - PBF can transform cache lines in local memory. PBF avoids races with a CPU requesting the cache line being transformed by delaying the return of the cache line to the CPU or by aborting the transformation. This is useful for modifying the layout of the data in memory, for compression [63, 78], deduplication [80], etc.

4.1.2 Data structures used for tracking application reads and writes

PBF uses two Bloom filters [14] to collect the addresses of the tracked cache lines in a space-efficient manner: a regular Bloom filter for cache line addresses and a counting Bloom filter for page addresses. In addition, PBF uses a buffer shared with a software daemon (§4.2) to communicate the addresses to the OS. The buffer contains the physical address and the size of the data (cache line or page). This buffer could grow arbitrarily large, so we limit its size by changing the granularity at which PBF reports dirty data as follows. If the cache line address has not been reported before (it is not in the cache line Bloom filter), but the corresponding page has been reported a threshold N times (in the page counting Bloom filter), then PBF writes the page address in the buffer and gives up on the cache line granularity tracking for that page. Otherwise, PBF adds the cache line address to the buffer. This works well because applications generally access only a few cache lines out of a page. For example, in the Redis-random workload, 78% of pages have fewer than 5 dirty cache lines. For this workload, PBF could select the threshold to be 5. Subsequently, for all pages with more than 5 cache lines reported, PBF will choose page granularity. Our approach admits false positives, due to the use of Bloom filters. A false positive results in reporting at page granularity instead of cache line granularity, which can impact performance, but not correctness.

4.2 PBerry kernel daemon (PBK)

PBerry supports a wide variety of use cases (§6), which are all enabled by the PBF primitives (§4.1.1). PBerry's software component, the PBK software daemon that runs in kernel mode, manages PBF and uses the primitives to implement the various use cases. We describe how PBK implements remote memory in §5. Across use cases, PBK fulfills the following responsibilities: (1) PBK loads the PBF bitstream on the cache-coherent FPGA. (2) It configures the communication channels with PBF. These channels consists of shared buffers that PBF uses to communicate accessed and dirty data addresses (§4.1.2). A separate buffer is also used for reporting errors encountered by PBF. (3) PBK configures any PBF parameters and policies necessary for a particular use case. (4) It also sends control commands, such as requesting which memory range to track or to copy. PBK's role is to manage only the control path. The data path is low-latency because PBF handles it in hardware.

4.3 Emulating PBF (PBSim)

To understand PBF's benefits and overheads, we designed and implemented PBSim, an emulator for PBF's write tracking primitive (§4.1.1). PBSim uses the *ptrace* Linux functionality [39] to identify an application's dirty data at cache line granularity. To track an application, PBSim becomes the application's tracer and pauses it to copy its entire memory. Subsequently, PBSim resumes the application, then continuously "diffs" its own copy with the application's memory, to determine dirty cache lines. This approach identifies the *virtual addresses* of the dirty cache lines, unlike PBF, which identifies *physical addresses.* In order to accurately mimic PBF's behavior, PBSim uses the application's pagemap file (via procfs) to translate the virtual addresses to physical addresses. In addition, PBF tracks data located in FPGA-Mem, which incurs higher latency compared to CPU-Mem. To capture this effect, PBSim runs on one NUMA node, while executing the application on a separate NUMA node. PBSim reports the emulated time of the application, which is the time the application spends *actually executing*, discarding the time elapsed while the application was paused by PBSim.

5 Using PBerry for remote memory

Virtual memory performs two functions for remote memory: (1) dirty data tracking for memory reclamation and (2) demand paging. PBerry's software component, PBK (§4.2), implements both functions and uses PBF primitives to accelerate them. Similar to prior systems [7, 28, 70], PBK caches remote memory in the local host's memory. Unlike prior systems, PBK primarily uses FPGA-Mem as the software-managed cache. This enables PBF read and write tracking at cache line granularity (§4.1.1), reducing the amplification and eliminating page faults. However, accessing pages in FPGA-Mem incurs higher latencies. To reduce latency overheads, pages that are not tracked, such as read-only pages or pages with frequent updates to many cache lines, are cached in CPU-Mem.

Memory reclamation and dirty data tracking. PBK handles memory reclamation and reclaims enough local memory so that demand paging from remote hosts can always allocate memory. PBK also sets up PBF for write tracking memory regions in FPGA-Mem. PBF collects the addresses of the dirty cache lines from write-backs to memory, using the memory agent solution (§4.1). PBK also uses the PBF remote copy primitive (§4.1.1) to reclaim memory pages, which works as follows. PBK specifies a memory range in FPGA-Mem, and PBF aggregates the dirty cache lines in the specified range and writes them to the remote host using its remote copy primitive.

Demand paging. PBK manages demand paging allocations proactively, marking remote pages as present in FPGA-Mem, but does not fetch the remote data. Doing so avoids a later page fault, when the application accesses that page. Instead, the application access generates a cache miss for FPGA-Mem, which is routed to PBF. PBF detects the cache miss and fulfills the cache line request from the remote host. This approach is analogous to the *Critical Word First* technique [32], used in hardware cache hierarchies. Critical Word First refers to fetching the word on which the miss occurred first and loading it directly into a CPU register, while fetching the rest of the cache line in the background. Similarly, PBF fetches the *Critical Cache Line First*, pre-fetching the rest of the page asynchronously.

Implementing the Critical Cache Line First approach is challenging because (1) PBF needs to respond to the cache coherence request within a limited amount of time, depending on the cache coherence protocol and (2) PBF needs to be able to effectively handle failures of the network or of the remote host. Not responding to the cache coherence protocol in time can trigger a machine check exception, which is notoriously hard to handle due to potential deadlocks and system instability [42]. To address these challenges, we propose a solution that falls back to software to handle errors. PBK moves page table pages to FPGA-Mem and PBF tracks coherence events for those pages. Before an application accesses a data page, it needs to obtain the virtual to physical address translation information, thus a read access will be generated to the page tables, which is intercepted by PBF. Then, PBF can delay responding to the cache miss event and pre-fetch the necessary pages (based on the requested page table entries [13]) from the remote host. If the prefetch fails for any reason, including timeout, PBF modifies the page table entries to mark the missing pages as not present before allowing the response to return to the CPU. This will force the data access to cause a page fault and PBK will handle the failure scenario. When a remote memory page is reclaimed from the local cache, a TLB shootdown ensures that there are no remaining cached translations of this page in any TLB, and that a future reference will have to access memory through PBF.[1]

6 Other use cases

We list multiple use cases for PBerry (Table 2), grouped by the PBF primitives used and types of pages tracked (§4.1.1). Each use case opens new research directions.

Pre-copy live virtual machine (or process) migration [3, 20] employs dirty data tracking in a similar fashion to remote memory. Post-copy live virtual machine (or process) migration [34] uses demand paging from a remote host. Both use cases can benefit from PBerry accelerating dirty data tracking and demand paging.

PBF write tracking of **application data pages** enables efficient dirty data tracking at a cache line granularity without employing any page faults. This can be used to realize an efficient copy-on-write. Combined with PBF's ability to copy data, it can achieve more efficient unified virtual memory (UVM).[2] Read tracking enables PBF to determine the application's access pattern, which is useful in implementing smart prefetchers for remote memory and in detecting security attacks, such as RowHammer [41] or cache-based side-channel attacks [19, 30].

PBF can also track read accesses to **application code pages**, which enables tracing and code coverage analysis [77]. Allowing PBF to modify these code pages enables dynamic code rewriting [62]. PBF can also be used for replicating data between hosts [82] or within a host [18], checkpointing [16], persistent memory logging [79], memory compression [63] and encryption [33], near-memory processing [55], offloading memory management tasks, etc.

Use case	Page type(s)	PBF primitives
Rem. mem. swapping/ memory reclamation (§5)	data, page tables	WT,RC
Rem. mem. caching (§5)	data, page tables	RT, RC, IPT
Live migration	data, page tables	RT, WT, RC, IPT
Copy-on-write	data	WT, LC
Unified virtual memory	data	WT, LC, RC
Security attack detection	data	RT, WT
Tracing code analysis	code	RT
Binary translation	code	IPT
Dynamic code rewriting	code	IPT
Replication	data	WT, LC, RC
Checkpointing	data	WT, LC, RC
Persistent memory	data	WT, LC, RC, IPT
Compression	data	RT, WT, IPT
Encryption	data	RT, WT, IPT
Near-memory compute	data	RT, WT, IPT

WT = Write Tracking, RT = Read Tracking, RC = Remote Copy, LC = Local Copy, IPT = In-Place-Transform

Table 2. Example PBerry use cases.

7 Preliminary results

We evaluate PBerry's potential to accelerate dirty data tracking for remote memory reclamation. We use PBSim (§4.3) to emulate PBF write tracking (§4.1.1). We use page write-protection as a baseline. We want to understand (1) what is the *dirty data amplification reduction* from tracking data at cache line granularity instead of 4KB page granularity? (2) what is the application execution time with and without write page faults? and (3) what is the time spent by the baseline to write-protect pages?

We measure Redis with a random and a sequential workload, and Metis running linear regression and histogram. We

[1] A TLB shootdown is necessary when reclaiming a page in a virtual memory based remote memory system too, so it is not an additional overhead introduced by PBerry.

[2] CUDA provides UVM for transparent data movement between CPU and GPU [29]

use the Memtier client with 1:1 *get* and *set* to drive Redis. We change the distribution of keys, and keep all other arguments unchanged between the different experiments. We use a host with 2 Intel Skylake processors running at 2.2GHz, with 56 cores (incl. hyperthreading), 187GB DRAM, running Linux kernel 4.4. We report the results in Table 3. We show the memory used by each application, and the reduction in the dirty data amplification with PBerry compared to baseline 4KB page granularity. Our results show that PBerry significantly reduces the dirty data amplification (by 50-95% for the applications we study). Next, we report the application execution time as indicated by PBSim. This is the *emulated time*, obtained by removing the time the application is paused for PBSim to copy the application data (§4.3). We compare PBSim with a version that write protects pages (PBSim+WP). We observe an increase in the execution time of the application when using write page faults to detect dirty data compared to our approach that does not rely on page faults: e.g., the execution time for Redis-random increases from 168s to 231s. Finally, we report the time spent write protecting all pages (WP) in the PBSim+WP experiment. WP time quickly increases with the size of the memory that needs to be write protected,[3] up to 7.3 ms for 40GB.

Application	Mem (GB)	% Amplif. reduction	PBSim (s)	PBSim +WP(s)	WP (ms)
Redis-random	3.93	89	168	231	0.2
Redis-seq	0.13	50	69	72	0.2
Lin. regr.	40	95	38	39	7.3
Histogram	40	50	38	39	7.2

Table 3. Dirty data tracking results using PBSim.

8 Related work

FPGAs are increasingly being used in the datacenter to accelerate applications like databases [31, 60, 73], key-value stores [15, 44], machine learning [49, 71], and web search [64]. FPGAs are also being used to efficiently manage infrastructure, to enable bump-in-the-wire architectures and software defined networks, e.g., Mellanox's hybrid NIC [51], NetFPGA [26] and Microsoft's Project Catapult [6, 52, 53]. FPGA sharing [40] and virtualization will drive down costs by improving utilization. New programming abstractions are improving adoption by simplifying the FPGA development process [10, 43]. Cache-coherent FPGAs have been previously used to accelerate Fast Fourier Transform [27] and to provide a prototyping platform for innovation in the memory subsystem of POWER servers [76].

Seminal distributed shared memory work [9, 12, 45, 65, 66] uses software techniques based on virtual memory or hardware techniques based on cache coherence. Unlike this work, we do not provide cache coherence across hosts. Instead, we use a cache-coherent FPGA *attached to the local host* to track applications' memory accesses.

Identifying sub-page granularity memory accesses can be achieved using a variety of approaches. First, the application can report its memory accesses by using a specific API [22, 23, 57], or it can rely on source code annotations or compiler support. These methods require changing the application or at least recompiling the source code, which is not always feasible. Second, run-time techniques, such as dynamic binary instrumentation [4, 17], can be used to instrument memory reads and writes transparently at run-time, without modifying the application or re-compiling the source code, but incur prohibitive overheads [48]. Lastly, hardware support can lower the overhead but still achieve transparency. Prior hardware extensions [56, 67, 68, 75, 81, 83] enable fine-grain dirty data tracking. Intel recently added Page Modification Logging (PML) [38], to log page updates in hardware and make them available to a hypervisor in batches of up to 512 entries. Per-page dirty bits and PML still incur dirty data amplification as they track data at page granularity. Intel also announced sub-page protection [35] that enables write-protecting 128-byte blocks within a page. This technique makes dirty tracking worse, since it can cause as many as 32 page faults for a 4KB page instead of a single page-wide fault, while PBerry eliminates the need for page faults by using the cache coherence protocol.

9 Conclusion

We introduced Project PBerry, a software-hardware co-designed system based on exposing cache coherence events from the local host to the operating system. PBerry relies on cache-coherent FPGAs to track cache misses and cache line write-backs in order to accelerate remote memory. We also discussed other use cases that PBerry enables beyond remote memory, such as live migration, unified virtual memory, security and code analysis. These use cases open up a myriad of new research directions.

Acknowledgements

We are grateful to the anonymous HotOS reviewers, as well as Nadav Amit, Richard Brunner, Jon Howell, Curt Kolovson, Chris Rossbach and Lalith Suresh for their thoughtful feedback that substantially improved the paper.

References

[1] CCIX. https://www.ccixconsortium.com.
[2] Enzian, a research computer built by the Systems Group at ETH Zürich. http://www.enzian.systems/index.html.
[3] P.Haul. https://criu.org/P.Haul.
[4] Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[5] Redis: open-source, in-memory data structure store. https://redis.io.
[6] Serving DNNs in real time at datacenter scale with Project Brainwave. https://www.microsoft.com/en-us/research/uploads/prod/2018/03/mi0218_Chung-2018Mar25.pdf.

[3] We under-approximate WP, as we only measure the first time each page is write-protected.

[7] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference (ATC)*, Boston, MA, 2018.

[8] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[9] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. 2010.

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, March 2017.

[12] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 1990.

[13] Abhishek Bhattacharjee. Translation-triggered prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, July 1970.

[15] M. Blott and K. Vissers. Dataflow architectures for 10 Gbps line-rate key-value-stores. In *IEEE Hot Chips 25 Symposium (HCS)*, 2013.

[16] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[17] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *International Conference on Virtual Execution Environments (VEE)*, 2012.

[18] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[19] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49(C), December 2016.

[20] Christopher Clark, Keir Fraser, Steven H, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[21] Convey Computer. The Convey HC-2 Computer. Architectural Overview. https://www.micron.com/~/media/documents/products/white-paper/wp_convey_hc2_architectual_overview.pdf, 2012.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation (NSDI)*, April 2014.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[24] Jake Edge. DAX, mmap(), and a "go faster" flag. https://lwn.net/Articles/684828/.

[25] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.

[26] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA: An open platform for teaching how to build Gigabit-rate network switches and routers. *IEEE Transactions on Education*, 2008.

[27] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. Accelerating arithmetic kernels with coherent attached fpga coprocessors. In *Design, Automation & Test in Europe (DATE)*, 2015.

[28] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with Infiniswap. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[29] Mark Harris. Unified Memory in CUDA 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[30] Zecheng He and Ruby B. Lee. How secure is your cache against side-channel attacks? In *International Symposium on Microarchitecture (MICRO)*, 2017.

[31] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. A flexible k-means operator for hybrid databases. In *"International Conference on Field Programmable Logic and Applications (FPL)"*, 2018.

[32] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.

[33] Michael Henson and Stephen Taylor. Memory encryption: A survey of existing techniques. *ACM Computing Surveys*, March 2014.

[34] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *Operating Systems Review*, July 2009.

[35] Intel. EPT-based Sub-Page Permissions. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[36] Intel. Intel® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[37] Intel. Intel® Xeon®+FPGA Platform for the Data Center. http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf.

[38] Intel. Page Modification Logging for Virtual Machine Monitor White Paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf.

[39] Daniel Jacobowitz. ptrace() event tracing. https://lwn.net/Articles/10369/.

[40] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, 2018.

[41] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[42] Andi Kleen. Machine check handling on Linux. https://www.halobates.de/mce.pdf.

[43] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[44] Maysam Lavasani, Hari Angepat, and Derek Chiou. An FPGA-based in-line accelerator for Memcached. *IEEE Computer Architecture Letters*, 2014.

[45] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, November

1989.

[46] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, February 2012.

[47] Liu Ling, Neal Oliver, Chitlur Bhushan, Wang Qigang, Alvin Chen, Shen Wenbo, Yu Zhihong, Arthur Sheiman, Ian McCallum, Joseph Grecco, Henry Mitchel, Liu Dong, and Prabhat Gupta. High-performance, energy-efficient platforms using in-socket fpga accelerators. In *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2009.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[49] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[50] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, May 2010.

[51] Mellanox. Mellanox Innova™ IPsec 4 Lx Ethernet Adapter Card User Manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_Innova_IPsec_4_Lx_Ethernet_Adapter_Card_User_Manual_rev_1_3.pdf.

[52] Microsoft. Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult.

[53] Microsoft. SDN for the Cloud. https://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf.

[54] David Mulnix. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[55] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 2019.

[56] Vijay Nagarajan and Rajiv Gupta. Architectural support for shadow memory in multiprocessors. In *International Conference on Virtual Execution Environments (VEE)*, 2009.

[57] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference (ATC)*, July 2015.

[58] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011.

[59] OpenCAPI consortium. http://opencapi.org.

[60] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[61] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *International Symposium on Computer Architecture (ISCA)*, 1984.

[62] Mathias Payer, Boris Bluntschli, and Thomas R. Gross. Dynsec: On-the-fly code rewriting and repair. In *Hot Topics in Software Upgrades*, 2013.

[63] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[64] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[65] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[66] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1994.

[67] Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. The dirty-block index. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[68] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[69] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, 2018.

[70] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[71] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[72] Navin Shenoy. A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data.

[73] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppioDB: A hardware accelerated database. In *International Conference on Management of Data (SIGMOD)*, 2017.

[74] Mario Smarduch. Enhanced Live Migration For Intensive Memory Loads. https://events.static.linuxfound.org/sites/events/files/slides/CloudOpen-Japan-2015.pdf.

[75] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[76] Bharat Sukhwani, Thomas Roewer, Charles L. Haymes, Kyu-Hyoun Kim, Adam J. McPadden, Daniel M. Dreps, Dean Sanner, Jan Van Lunteren, and Sameh Asaad. Contutto: A novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor. In *International Symposium on Microarchitecture (MICRO)*, 2017.

[77] A. Tran, M. Smith, and J. Miller. A hardware-assisted tool for fast, full code coverage analysis. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[78] Irina Chihaia Tuduce and Thomas R. Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, April 2005.

[79] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[80] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[81] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[82] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[83] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *International Symposium on Memory Management (ISMM)*, 2010.