Cache Coherency in ARMv8-A for Cross-Architectural DSM Systems

Zhengyi Chen

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

[TODO] . . .

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Zhengyi Chen*)

Acknowledgements

[TODO]:

For unbounded peace and happiness among all peoples of the world.

May we, one day, be able to see each other as equals.

Contents

Chapter 1

Introduction

...

This thesis paper builds upon an ongoing research effort in implementing a tightly coupled cluster where HMM abstractions allow for transparent RDMA access from accelerator nodes to local data and migration of data near computation, leveraging different consistency model and coherency protocols to amortize the communication cost for shared data. More specifically, this thesis explores the following:

- The effect of cache coherency maintenance, specifically OS-initiated, on RDMA programs.
- Discussion of memory models and coherence protocol designs for a single-writer, multi-reader RDMA-based DSM system.

Chapter 2

Background

Though large-scale cluster systems remain the dominant solution for request and data-level parallelism [27], there have been a resurgence towards applying HPC techniques (e.g., DSM) for more efficient heterogeneous computation with tighter-coupled heterogeneous nodes providing (hardware) acceleration for one another [10, 48, 36]. Orthogonally, within the scope of one motherboard, *heterogeneous memory management (HMM)* enables the use of OS-controlled, unified memory view across both main memory and device memory [26], all while using the same libc function calls as one would with SMP programming, the underlying complexities of memory ownership and data placement automatically managed by the OS kernel. However, while HMM promises a distributed shared memory approach towards exposing CPU and peripheral memory, applications (drivers and front-ends) that exploit HMM to provide ergonomic programming models remain fragmented and narrowly-focused. Existing efforts in exploiting HMM in Linux predominantly focus on exposing global address space abstraction to GPU memory – a largely non-coordinated effort surrounding both *in-tree* and proprietary code [16, 2]. Limited effort have been done on incorporating HMM into other variants of accelerators in various system topologies.

Orthogonally, allocation of hardware accelerator resources in a cluster computing environment becomes difficult when the required hardware accelerator resources of one workload cannot be easily determined and/or isolated as a "stage" of computation. Within a cluster system there may exist a large amount of general-purpose worker nodes and limited amount of hardware-accelerated nodes. Further, it is possible that every workload performed on this cluster asks for hardware acceleration from time to time, but never for a relatively long time. Many job scheduling mechanisms within a cluster *move data near computation* by migrating the entire job/container between general-purpose and accelerator nodes [60, 54]. This way of migration naturally incurs large overhead – accelerator nodes which strictly perform computation on data in memory without ever needing to touch the container's filesystem should not have to install the entire filesystem locally, for starters. Moreover, must *all* computations be performed near data? *Adrias*[50], for example, shows that RDMA over fast network interfaces (25 Gbps $\times$ 8), when compared to node-local setups, result in negligible impact on tail latencies but high impact on throughput when bandwidth is maximized.

The rest of the chapter is structured as follows:

- We identify and discuss notable developments in software-implemented DSM systems, and thus identify key features of contemporary advancements in DSM techniques that differentiate them from their predecessors.
- We identify alternative (shared memory) programming paradigms and compare them with DSM, which sought to provide transparent shared address space among participating nodes.
- We give an overview of coherency protocol and consistency models for multi-sharer DSM systems.
- We provide a primer to cache coherency in ARM64 systems, which *do not* guarantee cache-coherent DMA, as opposed to x86 systems [67].

2.1 Experiences from Software DSM

A majority of contributions to software DSM systems come from the 1990s [6, 12, 33, 30]. These developments follow from the success of the Stanford DASH project in the late 1980s – a hardware distributed shared memory (specifically NUMA) implementation of a multiprocessor that first proposed the *directory-based protocol* for cache coherence, which stores the ownership information of cache lines to reduce unnecessary communication that prevented previous multiprocessors from scaling out [40].

While developments in hardware DSM materialized into a universal approach to cache-coherence in contemporary many-core processors (e.g., *Ampere Altra*[3]), software DSMs in clustered computing languished in favor of loosely-coupled nodes performing data-parallel computation, communicating via message-passing. Bandwidth limitations with the network interfaces of the late 1990s was insufficient to support the high traffic incurred by DSM and its programming model [69, 46].

New developments in network interfaces provides much improved bandwidth and latency compared to ethernet in the 1990s. RDMA-capable NICs have been shown to improve the training efficiency sixfold compared to distributed *TensorFlow* via RPC, scaling positively over non-distributed training [34]. Similar results have been observed for *APACHE Spark* [47] and *SMBDirect* [41]. Consequently, there have been a resurgence of interest in software DSM systems and programming models [53, 11].

2.1.1 Munin: Multi-Consistency Protocol

Munin[12] is one of the older developments in software DSM systems. The authors of Munin identify that *false-sharing*, occurring due to multiple processors writing to different offsets of the same page triggering invalidations, is strongly detrimental to the performance of shared-memory systems. To combat this, Munin exposes annotations as part of its programming model to facilitate multiple consistency protocols on top of release consistency. An immutable shared memory object across readers, for example, can be safely copied without concern for coherence between processors. On the other hand, the *write-shared* annotation explicates that a memory object is written by multiple

processors without synchronization – i.e., the programmer guarantees that only false-sharing occurs within this granularity. Annotations such as these explicitly disables subsets of consistency procedures to reduce communication in the network fabric, thereby improving the performance of the DSM system.

Perhaps most importantly, experiences from Munin show that *restricting the flexibility of programming model can lead to more performant coherence models*, as exhibited by the now-foundational *Resilient Distributed Database* paper [72] which powered many now-popular scalable data processing frameworks such as *Hadoop MapReduce* [4] and *APACHE Spark* [5]. "To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory [based on]... transformations rather than... updates to shared state" [72]. This allows for the use of transformation logs to cheaply synchronize states between unshared address spaces – a much desired property for highly scalable, loosely-coupled clustered systems.

2.1.2 Treadmarks: Multi-Writer Protocol

Treadmarks[6] is a software DSM system developed in 1996, which featured an intricate *interval*-based multi-writer protocol that allows multiple nodes to write to the same page without false-sharing. The system follows a release-consistent memory model, which requires the use of either locks (via `acquire`, `release`) or barriers (via `barrier`) to synchronize. Each *interval* represents a time period in-between page creation, `release` to another processor, or a `barrier`; they also each correspond to a *write notice*, which are used for page invalidation. Each `acquire` message is sent to the statically-assigned lock-manager node, which forwards the message to the last releaser. The last releaser computes the outstanding write notices and piggy-backs them back for the acquirer to invalidate its own cached page entry, thus signifying entry into the critical section. Consistency information, including write notices, intervals, and page diffs, are routinely garbage-collected which forces cached pages in each node to become validated.

Compared to *Treadmarks*, the system described in this paper uses a single-writer protocol, thus eliminating the concept of "intervals" – with regards to synchronization, each page can be either in-sync (in which case they can be safely shared) or out-of-sync (in which case they must be invalidated/updated). This comes with the following advantage:

- Less metadata for consistency-keeping.
- More adherent to the CPU-accelerator dichotomy model.
- Much simpler coherence protocol, which reduces communication cost.

In view of the (still) disparate throughput and latency differences between local and remote memory access [11], the simpler coherence protocol of single-writer protocol should provide better performance on the critical paths of remote memory access.

2.1.3 Hotpot: Single-Writer & Data Replication

Newer works such as *Hotpot*[63] apply distributed shared memory techniques on persistent memory to provide "transparent memory accesses, data persistence, data reliability, and high availability". Leveraging on persistent memory devices allow DSM applications to bypass checkpoints to block device storage [63], ensuring both distributed cache coherence and data reliability at the same time [63].

We specifically discuss the single-writer portion of its coherence protocol. The data reliability guarantees proposed by the *Hotpot* system requires each shared page to be replicated to some *degree of replication*. Nodes who always store latest replication of shared pages are referred to as "owner nodes", which arbitrate other nodes to store more replications in order to reach the degree of replication quota. At acquisition time, the acquiring node asks the access-management node for single-writer access to shared page, who grants it if no other critical section exists, alongside list of current owner nodes. At release time, the releaser first commits its changes to all owner nodes which, in turn, commits its received changes across lesser sharers to achieve the required degree of replication. These two operations are all acknowledged back in reverse order. Once all acknowledgements are received from owner nodes by commit node, the releaser tells them to delete their commit logs and, finally, tells the manager node to exit critical section.

The required degree of replication and logged commit transaction until explicit deletion facilitate crash recovery at the expense of worse performance over release-time I/O. While the study of crash recovery with respect to shared memory systems is out of the scope of this thesis, this paper provides a good framework for a **correct** coherence protocol for a single-writer, multiple-reader shared memory system, particularly when the protocol needs to cater for a great variety of nodes each with their own memory preferences (e.g., write-update vs. write-invalidate, prefetching, etc.).

2.1.4 MENPS: A Return to DSM

MENPS[22] leverages new RDMA-capable interconnects as a proof-of-concept that DSM systems and programming models can be as efficient as *partitioned global address space* (PGAS) using today's network interfaces. It builds upon *TreadMark*'s [6] coherence protocol and crucially alters it to a *floating home-based* protocol, based on the insight that diff-transfers across the network is comparatively costly compared to RDMA intrinsics – which implies preference towards local diff-merging. The home node then acts as the data supplier for every shared page within the system.

Compared to PGAS frameworks (e.g., MPI), experimentation over a subset of *NAS Parallel Benchmarks* shows that MENPS can obtain comparable speedup in some of the computation tasks, while achieving much better productivity due to DSM's support for transparent caching, etc. [22]. These results back up their claim that DSM systems are at least as viable as traditional PGAS/message-passing frameworks for scientific computing, also corroborated by the resurgence of DSM studies later on[50].

2.2 Alternatives to DSM

While the feasibility of transparent DSM systems over multiple machines on the network has been made apparent since the 1980s, predominant approaches to "scaling-out" programs over the network relies on the message-passing approach [66]. The reasons are twofold:

1. Programmers would rather resort to more intricate, more predictable approaches to scaling-out programs over the network [66]. This implies manual/controlled data sharding over nodes, separation of compute and communication "stages" of computation, etc., which benefit performance analysis and engineering.
2. Enterprise applications value throughput and uptime of relatively computationally inexpensive tasks/resources [27], which requires easy scalability of tried-and-true, latency-inexpensive applications. Studies in transparent DSM systems mostly require exotic, specifically-written programs to exploit global address space, which is fundamentally at odds in terms of reusability and flexibility required.

2.2.1 PGAS

Partitioned Global Address Space (PGAS) is a parallel programming model that (1) exposes a global address space to all machines within a network and (2) explicates distinction between local and remote memory [19]. Oftentimes, message-passing frameworks, for example *OpenMPI*, *OpenFabrics*, and *UCX*, are used as backends to provide the PGAS model over various network interfaces/platforms (e.g., Ethernet and Infiniband)[65, 58].

Notably, implementation of a *global* address space across machines on top of machines already equipped with their own *local* address space (e.g., cluster nodes running commercial Linux) necessitates a global addressing mechanism for shared/shared data objects. DART[74], for example, utilizes a 128-bit "global pointer" to encode global memory object/segment ID and access flags in the upper 64 bits and virtual addresses in the lower 64 bits for each (slice of) memory object allocated within the PGAS model. A *non-collective* PGAS object is allocated entirely local to the allocating node's memory, but registered globally. Consequently, a single global pointer is recorded in the runtime with corresponding permission flags for the context of some user-defined group of associated nodes. Comparatively, a *collective* PGAS object is allocated such that a partition of the object (i.e., a sub-array of the repr) is stored in each of the associated node – for a k-partitioned object, k global pointers are recorded in the runtime each pointing to the same object, with different offsets and (intuitively) independently-chosen virtual addresses. Note that this design naturally requires virtual addresses within each node to be *pinned* – the allocated object cannot be re-addressed to a different virtual address, thus preventing the global pointer that records the local virtual address from becoming spontaneously invalidated.

Similar schemes can be observed in other PGAS backends/runtimes, albeit they may opt to use a map-like data structure for addressing instead. In general, despite both PGAS and DSM systems provide memory management over remote nodes, PGAS frameworks

provide no transparent caching and transfer of remote memory objects accessed by local nodes. The programmer is still expected to handle data/thread movement manually when working with shared memory over network to maximize their performance metrics of interest.

2.2.2 Message Passing

Message Passing remains the predominant programming model for parallelism between loosely-coupled nodes within a computer system, much as it is ubiquitous in supporting all levels of abstraction within any concurrent components of a computer system. Specific to cluster computing systems is the message-passing programming model, where parallel programs (or instances of the same parallel program) on different nodes within the system communicate via exchanging messages over network between these nodes. Such models exchange programming model productivity for more fine-grained control over the messages passed, as well as more explicit separation between communication and computation stages within a programming subproblem.

Commonly, message-passing backends function as *middlewares* – communication runtimes – to aid distributed software development [66]. Such a message-passing backend expose facilities for inter-application communication to frontend developers while transparently providing security, accounting, and fault-tolerance, much like how an operating system may provide resource management, scheduling, and security to traditional applications [66]. This is the case for implementing the PGAS programming model, which mostly rely on common message-passing backends to facilitate orchestrated data manipulation across distributed nodes. Likewise, message-passing backends, including RDMA API, form the backbone of many research-oriented DSM systems [22, 29, 11, 35].

Message-passing between network-connected nodes may be *two-sided* or *one-sided*. The former models an intuitive workflow to sending and receiving datagrams over the network – the sender initiates a transfer; the receiver copies a received packet from the network card into a kernel buffer; the receiver's kernel filters the packet and (optionally) [59] copies the internal message into the message-passing runtime/middleware's address space; the receiver's middleware inspects the copied message and performs some procedures accordingly, likely also involving copying slices of message data to some registered distributed shared memory buffer for the distributed application to access. Despite it being a highly intuitive model of data manipulation over the network, this poses a fundamental performance issue: because the process requires the receiver's kernel AND userspace to exert CPU-time, upon reception of each message, the receiver node needs to proactively exert CPU-time to move the received data from bytes read from NIC devices to userspace. Because this happens concurrently with other kernel and userspace routines in a concurrent system, a preemptable kernel may incur significant latency if the kernel routine for packet filtering is pre-empted by another kernel routine, userspace, or IRQs.

Comparatively, a "one-sided" message-passing scheme, for example RDMA, allows the network interface card to bypass in-kernel packet filters and perform DMA on registered memory regions. The NIC can hence notify the CPU via interrupts, thus allowing the

kernel and the userspace programs to perform callbacks at reception time with reduced latency. Because of this advantage, many recent studies attempt to leverage RDMA APIs for improved distributed data workloads and creating DSM middlewares [47, 34, 22, 29, 11, 35].

2.3 Consistency Model and Cache Coherence

Consistency model specifies a contract on allowed behaviors of multi-processing programs with regards to a shared memory [52]. One obvious conflict, which consistency models aim to resolve, lies within the interaction between processor-native programs and multi-processors, all of whom needs to operate on a shared memory with heterogeneous cache topologies. Here, a well-defined consistency model aims to resolve the conflict on an architectural scope. Beyond consistency models for bare-metal systems, programming languages [9, 8, 49, 55] and paradigms [6, 29, 11] define consistency models for parallel access to shared memory on top of program order guarantees to explicate program behavior under shared memory parallel programming across underlying implementations.

Related to the definition of a consistency model is the coherence problem, which arises whenever multiple actors have access to multiple copies of some datum, which needs to be synchronized across multiple actors with regards to write-accesses [52]. While less relevant to programming language design, coherence must be maintained via a coherence protocol [52] in systems of both microarchitectural and network scales. For DSM systems, the design of a correct and performant coherence protocol is of especially high priority and is a major part of many studies in DSM systems throughout history [12, 6, 57, 22, 18].

2.3.1 Consistency Model in DSM

Distributed shared memory systems with node-local caching naturally implies the existence of the consistency problem with regards to contending read/write accesses. Indeed, a significant subset of DSM studies explicitly characterize themselves as adhering to one of the well-known consistency models to better understand system behavior and to provide optimizations in coherence protocols [6, 30, 12, 22, 68, 11, 37], each adhering to a different consistency model to balance between communication costs and ease of programming.

In particular, we note that DSM studies tend to conform to either release consistency [6, 22, 12] or weaker [30], or sequential consistency [13, 68, 37, 20], with few works [11] pertaining to moderately constrained consistency models in-between. While older works, as well as works which center performance of their proposed DSM systems over existing approaches [22, 11], favor release consistency due to its performance benefits (e.g., in terms of coherence costs [22]), newer works tend to adopt stricter consistency models, sometimes due to improved productivity offered to programmers [37].

We especially note the role of balancing productivity and performance in terms of selecting the ideal consistency model for a system. It is common knowledge that weaker

	Sequential	TSO	PSO	Release	Acquire	Scope
Home; Invalidate	[37, 20, 73]			[63, 22]	[28]	[30]
Home; Update						
Float; Invalidate				[22]		
Float; Update						
Directory; Inval.	[68]					
Directory; Update						
Dist. Dir.; Inval.	[13]		[11]	[12]	[12, 6]	
Dist. Dir.; Update				[12]		

Table 2.1: Coherence Protocol vs. Consistency Model in Selected Disaggregated Memory Studies. “Float” short for “floating home”. Studies selected for clearly described consistency model and coherence protocol.

consistency models are harder to program with, at the benefit of less (implied) coherence communications resulting in better throughput overall – provided that the programmer could guarantee correctness, a weaker consistency model allows for less invalidation of node-local cache entries, thereby allowing multiple nodes to compute in parallel on (likely) outdated local copy of data such that the result of the computation remains semantically correct with regards to the program. This point was made explicit in *Munin* [12], where (to reiterate) it introduces the concept of consistency “protocol parameters” to annotate shared memory access pattern, in order to reduce the amount of coherence communications necessary between nodes computing in distributed shared memory. For example, a DSM object (memory object accounted for by the DSM system) can be annotated with “delayed operations” to delay coherence operations beyond any write-access, or shared without “write” annotation to disable write-access over shared nodes, thereby disabling all coherence operations with regards to this DSM object. Via programmer annotation of DSM objects, the Munin DSM system explicates the effect of weaker consistency in relation to the amount of synchronization overhead necessary among shared memory nodes. To our knowledge, no other more recent DSM works have explored this interaction between consistency and coherence costs on DSM objects, though relatedly *Resilient Distributed Dataset (RDD)* [72] also highlights its performance and flexibility benefits in opting for an immutable data representation over disaggregated memory over network when compared to contemporary DSM approaches.

2.3.2 Coherence Protocol

Coherence protocols hence becomes the means over which DSM systems implement their consistency model guarantees. As table 2.1 shows, DSM studies tends to implement write-invalidated coherence under a *home-based* or *directory-based* protocol framework, while a subset of DSM studies sought to reduce communication overheads and/or improve data persistence by offering write-update protocol extensions [12, 63].

2.3.2.1 Home-Based Protocols

Home-based protocols define each shared memory object with a corresponding "home" node, under the assumption that a many-node network would distribute home-node ownership of shared memory objects across all hosts [30]. On top of home-node ownership, each mutable shared memory object may be additionally cached by other nodes within the network, creating the coherence problem. To our knowledge, in addition to table 2.1, this protocol and its derivatives had been adopted by [23, 42, 30, 53, 63, 22].

We identify that home-based protocols are conceptually straightforward compared to directory-based protocols, centering communications over storage of global metadata (in this case ownership of each shared memory object). This leads to greater flexibility in implementing coherence protocols. A shared memory object at its creation may be made known globally via broadcast, or made known to only a subset of nodes (0 or more) via multicast. Likewise, metadata storage could be cached locally to each node and invalidated alongside object invalidation or fetched from a fixed node with respect to one object. This implementation flexibility is further taken advantage of in *Hotpot*[63], which refines the "home node" concept into *owner node* to provide replication and persistence, in addition to adopting a dynamic home protocol similar to that of [22].

2.3.2.2 Directory-Based Protocols

Directory-based protocols instead take a shared database approach by denoting each shared memory object with a globally shared entry describing ownership and sharing status. In its non-distributed form (e.g., [68]), a global, central directory is maintained for all nodes in network for ownership information: the directory hence becomes a bottleneck for imposing latency and bandwidth constraints on parallel processing systems. Comparatively, a distributed directory scheme may delegate responsibilities across all nodes in network mostly in accordance to sharded address space [29, 11]. Though theoretically sound, this scheme performs no dynamic load-balancing for commonly shared memory objects, which in the worst case would function exactly like a non-distributed directory coherence scheme. To our knowledge, in addition to table 2.1, this protocol and its derivatives had been adopted by [12, 6, 62, 21, 29].

2.3.3 DMA and Cache Coherence

The advent of high-speed RDMA-capable network interfaces introduce introduce opportunities for designing more performant DSM systems over RDMA (as established in 2.2.2). Orthogonally, RDMA-capable NICs on a fundamental level perform direct memory access over the main memory to achieve one-sided RDMA operations to reduce the effect of OS jittering on RDMA latencies. For modern computer systems with cached multiprocessors, this poses a potential cache coherence problem on a local level – because RDMA operations happen concurrently with regards to memory accesses by CPUs, which stores copies of memory data in cache lines which may [39, 67] or may not [25, 15] be fully coherent by the DMA mechanism, any DMA operations

performed by the RDMA NIC may be incoherent with the cached copy of the same data inside the CPU caches (as is the case for accelerators, etc.). This issue is of particular concern to the kernel development community, who needs to ensure that the behaviors of DMA operations remain identical across architectures regardless of support of cache-coherent DMA [15]. Likewise existing RDMA implementations which make heavy use of architecture-specific DMA memory allocation implementations, implementing RDMA-based DSM systems in kernel also requires careful use of kernel API functions that ensure cache coherency as necessary.

2.3.4 Cache Coherence in ARMv8-A

We specifically focus on the implementation of cache coherence in ARMv8-A. Unlike x86 which guarantees cache-coherent DMA [67, 15], the ARMv8-A architecture (and many other popular ISAs, for example *RISC-V*) *does not* guarantee cache-coherency of DMA operations across vendor implementations. ARMv8 defines a hierarchical model for coherency organization to support *heterogeneous* and *asymmetric* multi-processing systems [7].

Definition 1 (cluster)**.** A *cluster* defines a minimal cache-coherent region for Cortex-A53 and Cortex-A57 processors. Each cluster usually comprises of 1 or more core as well as a shared last-level cache.

Definition 2 (sharable domain)**.** A *sharable domain* defines a vendor-defined cache-coherent region. Sharable domains can be *inner* or *outer*, which limits the scope of broadcast coherence messages to *point-of-unification* and *point-of-coherence*, respectively.

Usually, the *inner* sharable domain defines the domain of all (closely-coupled) processors inside a heterogeneous multiprocessing system (see 5); while the *outer* sharable domain defines the largest memory-sharing domain for the system (e.g. inclusive of DMA bus).

Definition 3 (Point-of-Unification)**.** The *point-of-unification* (*PoU*) under ARMv8 defines a level of coherency such that all sharers inside the **inner** sharable domain see the same copy of data.

Consequently, *PoU* defines a point at which every core of a ARMv8-A processor sees the same (i.e., a *unified*) copy of a memory location regardless of accessing via instruction caches, data caches, or TLB.

Definition 4 (Point-of-Coherence)**.** The *point-of-coherence* (*PoC*) under ARMv8 defines a level of coherency such that all sharers inside the **outer** sharable domain see the same copy of data.

Consequently, *PoC* defines a point at which all *observers* (e.g., cores, DSPs, DMA engines) to memory will observe the same copy of a memory location.

2.3.4.1 Addendum: *Heterogeneous* & *Asymmetric* Multiprocessing

Using these definitions, a vendor could build *heterogeneous* and *asymmetric* multiprocessing systems as follows:

Definition 5 (Heterogeneous Multiprocessing)**.** A *heterogeneous multiprocessing* system incorporates ARMv8 processors of diverse microarchitectures that are fully coherent with one another, running the same system image.

Definition 6 (Asymmetric Multiprocessing)**.** A *asymmetric multiprocessing* system needs not contain fully coherent processors. For example, a system-on-a-chip may contain a non-coherent co-processor for secure computing purposes [7].

2.3.5 ARMv8-A Software Cache Coherence in Linux Kernel

Because of the lack of hardware guarantee on hardware DMA coherency (though such support exists [56]), programmers need to invoke architecture-specific cache-coherency instructions when porting DMA hardware support over a diverse range of ARMv8 microarchitectures, often encapsulated in problem-specific subroutines.

Notably, kernel (driver) programming warrants programmer attention to software-maintained coherency when userspace programmers downstream expect data-flow, interspersed between CPU and DMA operations, to follow program ordering and (driver vendor) specifications. One such example arises in the Linux kernel implementation of DMA memory management API [51][1]:

Definition 7 (DMA Mappings)**.** The Linux kernel DMA memory allocation API, imported via

```
1 #include <linux/dma-mapping.h>
```

defines two variants of DMA mappings:

- *Consistent* DMA mappings:

  They are guaranteed to be coherent in-between concurrent CPU/DMA accesses without explicit software flushing. [2]

- *Streaming* DMA mappings:

  They provide no guarantee to coherency in-between concurrent CPU/DMA accesses. Programmers need to manually apply coherency maintenance subroutines for synchronization.

Consistent DMA mappings could be trivially created via allocating non-cacheable memory, which guarantees *PoC* for all memory observers (though system-specific fastpaths exist).

[1] Based on Linux kernel v6.7.0.

[2] However, it does not preclude CPU store reordering, so memory barriers remain necessary in a multiprocessing context.

On the other hand, streaming DMA mappings require manual synchronization upon programmed CPU/DMA access. Take single-buffer synchronization on CPU after DMA access for example:

```
/* In kernel/dma/mapping.c */
void dma_sync_single_for_cpu(
    struct device *dev,          // kernel repr for DMA device
    dma_addr_t addr,             // DMA address
    size_t size,                 // Synchronization buffer size
    enum dma_data_direction dir, // Data-flow direction
) {
    /* Translate DMA address to physical address */
    phys_addr_t paddr = dma_to_phys(dev, addr);

    if (!dev_is_dma_coherent(dev)) {
        arch_sync_dma_for_cpu(paddr, size, dir);
        arch_sync_dma_for_cpu_all(); // MIPS quirks, nop for ARM64
    }

    /* Miscellaneous cases...*/
}
```

```
/* In arch/arm64/mm/dma-mapping.c */
void arch_sync_dma_for_cpu(
    phys_addr_t paddr,
    size_t size,
    enum dma_data_direction dir,
) {
    /* Translate physical address to (kernel) virtual address */
    unsigned long start = (unsigned long)phys_to_virt(paddr);

    /* Early exit for DMA read: no action needed for CPU */
    if (dir == DMA_TO_DEVICE)
        return;

    /* ARM64-specific: invalidate CPU cache to PoC */
    dcache_inval_poc(start, start + size);
}
```

This call-chain, as well as its mirror case which maintains cache coherency for the DMA device after CPU access:

```
dma_sync_single_for_device(struct device *, dma_addr_t, size_t,
↪  enum dma_data_direction)
```

, call into the following procedures, respectively:

```c
/* Exported @ arch/arm64/include/asm/cacheflush.h */
/* Defined @ arch/arm64/mm/cache.S */
/* All functions accept virtual start, end addresses. */

/* Invalidate data cache region [start, end) to PoC.
 *
 * Invalidate CPU cache entries that intersect with [start, end),
 * such that data from external writers becomes visible to CPU.
 */
extern void dcache_inval_poc(
    unsigned long start, unsigned long end
);

/* Clean data cache region [start, end) to PoC.
 *
 * Write-back CPU cache entries that intersect with [start, end),
 * such that data from CPU becomes visible to external writers.
 */
extern void dcache_clean_poc(
    unsigned long start, unsigned long end
);
```

2.3.5.1 Addendum: `enum dma_data_direction`

The Linux kernel defines 4 direction `enum` values for fine-tuning synchronization behaviors:

```c
/* In include/linux/dma-direction.h */
enum dma_data_direction {
    DMA_BIDIRECTION = 0, // data transfer direction uncertain.
    DMA_TO_DEVICE = 1,   // data from main memory to device.
    DMA_FROM_DEVICE = 2, // data from device to main memory.
    DMA_NONE = 3,        // invalid repr for runtime errors.
};
```

These values allow for certain fast-paths to be taken at runtime. For example, `DMA_TO_DEVICE` implies that the device reads data from memory without modification, and hence precludes software coherence instructions from being run when synchronizing for CPU after DMA operation.

2.3.5.2 Use-case: Kernel-space *SMBDirect* Driver

An example of cache-coherent in-kernel RDMA networking module over heterogeneous ISAs could be found in the Linux implementation of *SMBDirect*. *SMBDirect* is an

extension of the *SMB* (*Server Message Block*) protocol for opportunistically establishing the communication protocol over RDMA-capable network interfaces [70].

We focus on two procedures inside the in-kernel SMBDirect implementation:

Before send: `smbd_post_send` `smbd_post_send` is a function downstream of the call-chain of `smbd_send`, which sends SMBDirect payload for transport over network. Payloads are constructed and batched for maximized bandwidth, then `smbd_post_send` is called to signal the RDMA NIC for transport.

The function body is roughly as follows:

```
1  /* In fs/smb/client/smbdirect.c */
2  static int smbd_post_send(
3      struct smbd_connection *info, // SMBDirect transport context
4      struct smbd_request *request, // SMBDirect request context
5  ) {
6      struct ib_send_wr send_wr; // Ib "Write Request" for payload
7      int rc, i;
8
9      /* For each message in batched payload */
10     for (i = 0; i < request->num_sge; i++) {
11         /* Log to kmesg ring buffer... */
12
13         /* RDMA wrapper over DMA API1 */
14         ib_dma_sync_single_for_device(
15             info->id->device,       // struct ib_device *
16             request->sge[i].addr,   // u64 (as dma_addr_t)
17             request->sge[i].length, // size_t
18             DMA_TO_DEVICE,          // enum dma_data_direction
19         );
20     }
21
22     /* Populate `request`, `send_wr`... */
23
24     rc = ib_post_send(
25         info->id->qp, // struct ib_qp * ("Queue Pair")
26         &send_wr,     // const struct ib_recv_wr *
27         NULL,         // const struct ib_recv_wr ** (err handling)
28     );
29
30     /* Error handling... */
31
32     return rc;
33 }
```

Line 13 writes back CPU cache lines to be visible for RDMA NIC in preparation for

DMA operations when the posted *send request* is worked upon.

Upon reception: `recv_done` `recv_done` is called when the RDMA subsystem works on the received payload over RDMA.

Mirroring the case for `smbd_post_send`, it invalidates CPU cache lines for DMA-ed data to be visible at CPU cores prior to any operations on received data:

```
/* In fs/smb/client/smbdirect.c */
static void recv_done(
    struct ib_cq *cq, // "Completion Queue"
    struct ib_wc *wc, // "Work Completion"
) {
    struct smbd_data_transfer *data_transfer;
    struct smbd_response *response = container_of(
        wc->wr_cqe,           // ptr: pointer to member
        struct smbd_response, // type: type of container struct
        cqe,                  // name: name of member in struct
    ); // Cast member of struct into containing struct (C magic)
    struct smbd_connection *info = response->info;
    int data_length = 0;

    /* Logging, error handling... */

    /* Likewise, RDMA wrapper over DMA API1 */
    ib_dma_sync_single_for_cpu(
        wc->qp->device,
        response->sge.addr,
        response->sge.length,
        DMA_FROM_DEVICE,
    );

    /* ... */
}
```

Chapter 3

Software Coherency Latency

Coherency must be maintained at software level when hardware cache coherency cannot be guaranteed for some specific ISA (as established in subsection 2.3.5). There is, therefore, interest in knowing the latency of coherence-maintenance operations for performance engineering purposes, for example OS jitter analysis for scientific computing in heterogeneous clusters and, more pertinently, comparative analysis between software and hardware-backed DSM systems (e.g. [50, 68]). Such an analysis is crucial to being well-informed when designing a cross-architectural DSM system over RDMA.

The purpose of this chapter is hence to provide a statistical analysis over software coherency latency in ARM64 systems by instrumenting hypothetical scenarios of software-initiated coherence maintenance in ARM64 test-benches.

The rest of the chapter is structured as follows:

- **Experiment Setup** covers the test-benches used for instrumentation, including the kernel version, distribution, and the specifications of the instrumented (bare-metal/virtual) machine.
- **Methodology** covers the kernel module and workload used for instrumentation and experimentation, including changes made to the kernel, the kernel module, and userspace programs used for experimentation.
- **Results** covers the results gathered during instrumentation from various test-benches, segmented by experiment.
- **Discussion** identifies key insights from experimental results, as well as deficiencies in research method and possible directions of future works.

3.1 Experiment Setup

3.1.1 QEMU-over-x86: `star`

The primary source of experimental data come from a virtualized machine: a virtualized guest running a lightly-modified Linux v6.7.0 preemptive kernel with standard non-graphical Debian 12 distribution installed to provide userspace support. Table 3.1

describes the specifics of the QEMU-emulated ARM64 test-bench, while table 3.2 describes the specifics of its host.

Processors	QEMU virt-8.2 ($3 \times$ 2-way SMT; emulates Cortex-A76)
Frequency	2.0 GHz (*sic.*[3])
CPU Flags	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm lrcpc dcpop asimddp
NUMA Topology	1: $\{P_0, \ldots, P_5\}$
Memory	1: 4GiB
Kernel	Linux 6.7.0 (modified) SMP Preemptive
Distribution	Debian 12 (bookworm)

Table 3.1: Specification of `star`

Processors	AMD Ryzen 7 4800HS ($8 \times$ 2-way SMT)	
Freuqnecy	2.9 GHz (4.2 GHz Turbo)	
NUMA Topology	1: $\{P_0, \ldots, P_{15}\}$	
Cache Structure	L3	$P_0 \ldots P_7$: 4MiB, $P_8 \ldots P_{15}$: 4MiB
	L2	Per core[4]: 512KiB
	L1	Per core: d-cache 32KiB, i-cache 32KiB
Memory	1: 40 GiB DDR4-3200 SO-DIMM	
Filesystem	ext4 on Samsung SSD 970 EVO Plus	
Kernel	Linux 6.7.9 (arch1-1) SMP Preemptive	
Distribution	Arch Linux	

Table 3.2: Specification of Host

3.1.2 *Ampere Altra*: `rose`

Processors	Ampere Altra (32 core; Neoverse N1 microarch.)	
Frequency	1.7 GHz (3.0 GHz max)	
NUMA Topology	1: $\{P_0, \ldots, P_{31}\}$	
Cache Structure	L2	Per core: 1MiB
	L1	Per core: d-cache 64KiB, i-cache 64KiB
Memory	1: 256 GiB DDR4-3200 DIMM ECC	
Kernel	Linux 6.7.0 (modified) SMP Preemptive	
Distribution	Ubuntu 22.04 LTS (Jammy Jellyfish)	

Table 3.3: Specification of `rose`

Additional to virtualized testbench, I have had the honor to access `rose`, a ARMv8 server rack system hosted by the [TODO] PLACEHOLDER at the *Informatics Forum*, through

[3] As reported from `lscpu`. Likely not reflective of actual emulation performance.

[4] i.e., per 2 threads. For example: P_0, P_1 comprises one core.

the invaluable assistance of my primary advisor, *Amir Noohi*, for instrumentation of similar experimental setups on server-grade bare-metal systems.

The specifications of `rose` is listed in table 3.3.

3.2 Methodology

3.2.1 Exporting `dcache_clean_poc`

As established in subsection 2.3.5, software cache-coherence maintenance operations (e.g., `dcache_[clean|inval]_poc`) are wrapped behind DMA API function calls and are hence unavailable for direct use in drivers. Moreover, instrumentation of assembly code becomes non-trivial when compared to instrumenting C function symbols, likely due to automatically stripped assembly symbols in C object files. Consequently, it becomes impossible to utilize the existing instrumentation tools available in the Linux kernel (e.g., `ftrace`) to trace assembly routines.

In order to convert `dcache_clean_poc` to a traceable equivalent, a wrapper function `__dcache_clean_poc` is created as follows:

```
/* In arch/arm64/mm/flush.c */
#include <asm/cacheflush_extra.h>

/* ... */

void __dcache_clean_poc(ulong start, ulong end)
{
    dcache_clean_poc(start, end); // see arch/arm64/mm/cache.S
}
EXPORT_SYMBOL(__dcache_clean_poc);
```

Correspondingly, the header `arch/arm64/include/asm/cacheflush_extra.h` is created to export the symbol `__dcache_clean_poc` into kernel module namespace. This has the additional benefit of creating a corresponding `ftrace` target, allowing the symbol to be instrumented using existing Linux instrumentation mechanisms. The entirety of modifications done to the in-tree v6.7.0 kernel culminates to a 44-line patch file (inclusive of metadata, context, etc.). It is expected that the introduction of additional symbols would increment the function latency by (at least) the amount of time necessary to fetch the instruction, but such latency is expected to be miniscule when compared to cache coherency operations.

3.2.2 Kernel Module: `my_shmem`

To simulate module-initiated cache coherence behavior over allocated kernel buffers, a kernel module, `my_shmem`, is written such that specially-written userspace programs could cause the kernel to invoke `__dcache_clean_poc` at will.

3.2.2.1 `my_shmem`: Design

The `my_shmem` module is a utility for (lazily) allocating one or more kernel-space pages, re-mapping them into the userspace for reading/writing operations, and invoking cache-coherency operations *as if* accessed via DMA on unmap.

To emulate streaming DMA mapping allocation, the module is designed to allocate memory directly from the *page allocator*, as required by the kernel documentation's guideline, *What Memory is DMA'able?*[51]:

> If you acquired your memory via the page allocator (i.e. `__get_free_page*()`) or the generic memory allocators (i.e. `kmalloc()` or `kmem_cache_alloc()`) then you may DMA to/from that memory using the addresses returned from those routines.

To enable page sharing between user-space processes, the module implements a allocation accounting mechanism for re-mapping existing allocations to multiple user-space address spaces on-demand. Specifically, it involves:

- Allocation of contiguous pages to some user-specified order (i.e., 2^{order} pages).
- Correct re-mapping behavior of existing allocations, for example computing the correct offset when re-mapping a multi-page allocation during any given page-fault, which may not be aligned with the first page in the allocation.
- Software cache coherency maintenance on removal of mapping from any user-space program. This is intended to simulate the behavior of DMA API in a system without any specific DMA hardware.

The module should hence support userspace programs to be able to perform as follows:

1. Open the "device" file as exposed by the kernel module.
2. `mmap` on the opened file descriptor, as per POSIX syscall API.
3. Allocate memory due to load/store actions within the `mmap`-ed memory mapping.
4. Close the memory mapping, which initiates a simulated software cache coherency maintenance operation.

3.2.2.2 `my_shmem`: Implementation

To implement the features as specified, `my_shmem` exposes itself as a character device file `/dev/my_shmem`; implements *file operations* `open`, `mmap`, and `release`; and implements *vm operations* `close` and `fault`.

Additionally, the parameter `max_contiguous_alloc_order` is exposed as a writable parameter file inside *sysfs* to manually control the number of contiguous pages allocated per module allocation.

The entire kernel module used for experiment amount to around 400 lines of kernel-space code.

Data Structures The primary functions of `my_shmem` is to provide correct accounting of current allocations via the kernel module in addition to allocating on-demand. Hence, to represent a in-kernel allocation of multi-page contiguous buffer, define `struct my_shmem_alloc` as follows:

```
struct my_shmem_alloc {
    struct page *page; // GFP alloc repr, points to HEAD page
    ulong alloc_order; // alloc buffer length: 2^alloc_order
    struct list_head list; // kernel repr of doubly linked list
};
```

`.list` defines the Linux kernel implementation of a element of a generically-typed doubly linked list, such that multiple allocations could be kept during the lifetime of the module. The corresponding linked list is defined as follows:

```
static LIST_HEAD(my_shmem_allocs);
```

To book-keep the real amount of pages allocated during the module's lifetime, define:

```
static size_t my_shmem_page_count;
```

Finally, to ensure mutual exlusion of the module's critical sections while running inside a *SMP* (*Symmetric Multi-Processing*) kernel, define mutex:

```
static DEFINE_MUTEX(my_shmem_allocs_mtx);
```

This protects all read/write operations to `my_shmem_allocs` and `my_shmem_page_count` against concurrent module function calls.

File Operations The Linux kernel defines *file operations* as a series of module-specific callbacks whenever the userspace invokes a corresponding syscall on the (character) device file. These callbacks may be declared inside a `file_operations` struct[17], which provides an interface for modules on file-related syscalls:

```
/* In include/linux/fs.h */
struct file_operations {
    struct module *owner;
    /* ... */
    int (*mmap) (
        struct file *,          // opened (device) file
        struct vm_area_struct * // kernel repr of mapping
    ); // Downstream of syscall: mmap
    /* ... */
    int (*open) (
        struct inode *, // inode of file to be opened
```

```
        struct file *   // opened (generic) file
    ); // Downstream of libc: open
    /* ... */
    int (*release) (
        struct inode *, // inode of file to be closed
        struct file *   // to be closed
    ); // Downstream of libc: close
    /* ... */
} __randomize_layout;
```

The corresponding structure for the particular module is hence defined as follows:

```
/* In my_shmem.c */
static const struct file_operations my_shmem_fops = {
    .owner = THIS_MODULE,
    .open = my_shmem_fops_open,
    .mmap = my_shmem_fops_mmap,
    .release = my_shmem_fops_release,
};
```

Implementation of `.open` is simple. It suffices to install the module-specific `struct file_operations` (i.e., `my_shmem_fops`) into the `struct file` passed in argument, which is constructed downstream via kernel's generic file opening mechanisms.

Likewise for `.release`, which does nothing except to print a debug message into the kernel ring buffer.

To implement `.mmap`, the kernel module attempts to *re-map as much allocations into the given `struct vm_area_struct` as possible without making any allocation*. This centralizes allocation logic into the page fault handler, which is described later in 3.2.2.2.

```
static int my_shmem_fops_mmap(
    struct file *filp,
    struct vm_area_struct *vma
) {
    int ret = 0;
    const ulong vma_pg_count =
        (vma->vm_end - vma->vm_start) » PAGE_SHIFT;
    struct page *pg;
    ulong tgt_addr = vma->vm_start; // Current remap target addr
    ulong src_head_pfn; // Current remap source: head PFN
    ulong src_pg_nr;    // Current remap source: length
    ulong vma_remainder_count = vma_pg_count; // vma: remain pgs

    /* Lock mutex... */
```

```
    /* Iterate over allocations, remap as much as possible */
    struct my_shmem_alloc *curr;
    list_for_each_entry(curr, &my_shmem_allocs, list) {
        /* exit if all of vma is mapped */
        if (tgt_addr >= vma->vm_end)
            break;

        /* decrement page offset until alloc intersects */
        if (vma_pgoff > ORDER_TO_PAGE_NR(curr->alloc_order)) {
            vma_pgoff -= ORDER_TO_PAGE_NR(curr->alloc_order);
            continue;
        }

        /* intersects, hence compute PFN to remap */
        pg = curr->page;
        get_page(pg); // increment alloc. refcount
        src_head_pfn = page_to_pfn(pg) + vma_pgoff;
        src_pg_nr = min(
            vma_remainder_count,
            ORDER_TO_PAGE_NR(curr->alloc_order) - vma_pgoff
        );
        ret = remap_pfn_range(
            vma,                    // remap target VM area
            tgt_addr,               // page-aligned tgt addr
            src_head_pfn,           // kernel PFN as source
            src_pg_nr * PAGE_SIZE, // size of remap region
            vma->vm_page_prot,      // page protection flags
        );
        /* if (ret): goto error handling... */
        /* Prepare for next iteration */
        tgt_addr += src_pg_nr * PAGE_SIZE;
        vma_remainder_count -= src_pg_nr;
    }

    /* return or error handling... */
}
```

VM Operations On `mmap`, the Linux kernel installs a new *VMA* (*Virtual Memory Area*) as the internal representation for the corresponding mapping in process address space[17]. Likewise file operations, kernel modules may implement callbacks in `vm_operations_struct` to define module-specific operations per VMA access at userspace:

```c
/* In include/linux/mm.h */
struct vm_operations_struct {
    /* ... */
    void (*close)(struct vm_area_struct * area);
    /* ... */
    vm_fault_t (*fault)(
        struct vm_fault *vmf // Page fault descriptor
    ); // Page fault handler
    /* ... */
};
```

The corresponding structure for the particular module is hence defined as follows:

```c
/* In my_shmem.c */
static const struct vm_operations_struct my_shmem_vmops = {
    .close = my_shmem_vmops_close,
    .fault = my_shmem_vmops_fault,
};
```

Function `.fault` is implemented such that allocations are performed lazily until the number of pages allocated inside the module superseeds the faulting page offset wrt. its mapping. A simple implementation of this is to, given the number of pages allocated is insufficient to service this page fault, continuously allocate unitl this condition becomes valid:

```c
static vm_fault_t my_shmem_vmops_fault(struct vm_fault *vmf)
{
    vm_fault_t ret = VM_FAULT_NOPAGE; // See 1
    ulong tgt_offset = vmf->vma->vm_pgoff + vmf->pgoff;

    /* Lock mutex... */
    for (;;) {
        /* When we already allocated enough, remap */
        if (tgt_offset < my_shmem_page_count)
            return __my_shmem_fault_remap(vmf); // See 2

        /* Otherwise, allocate 2^order pages and retry */
        struct my_shmem_alloc *new_alloc_handle = kzalloc(
            sizeof(struct my_shmem_alloc),
            GFP_KERNEL, // kernel-only allocation rule flag
        );
        /* if (!new_alloc_handle) goto error handling... */

        struct page *new_alloc_pg = alloc_pages(
            GFP_USER, // user-remapped kernel alloc rule flag
```

```
21              max_contiguous_alloc_order,
22          ); // Alloc 2^order pages
23          /* if (!new_alloc_pg) goto error handling... */
24
25          /* Fill in handle data */
26          new_alloc_handle->page = new_alloc_pg;
27          new_alloc_handle->alloc_order = max_contiguous_alloc_order;
28          /* Add `new_alloc_handle` to `my_shmem_allocs`... */
29
30          /* Prepare for next iteration */
31          my_shmem_page_count +=
32              ORDER_TO_PAGE_NR(new_alloc_handle->alloc_order);
33      }
34
35      /* Error handling... */
36  }
```

Several implementation quirks that warrant attention are as follows:

1. `my_shmem_vmops_fault` returns `VM_FAULT_NOPAGE` on success. This is due to the need to support multi-page contiguous allocation inside the kernel module for performance analysis purposes.

   Usually, the `vm_operations_struct` API expects its `.fault` implementations to assign `struct page *` to `vmf->page` on return. Here, `vmf->page` represents the page-aligned allocation that is to be installed into the faulting process's page table, thereby resolving the page fault.

   However, this expectation causes a conflict between the module's ability to allocate multi-page contiguous allocations and its ability to perform page-granularity mapping of underlying allocations (no matter the size of the allocation). Because *GFP*-family of page allocators use `struct page` as the representation of the *entire* allocation (no matter the number of pages actually allocated), it is incorrect to install the `struct page` representation of a multi-page contiguous allocation to any given page fault in case that the page fault offset is misaligned with the alignment of the allocation (an example of such case arising could be found at 3.1).

   Consequently, `VM_FAULT_NOPAGE` is raised to indicate that *`vmf->page` would not be assigned with a reasonable value, and the callee guarantees that corresponding page table entries would be installed when control returns to caller*. The latter guarantee is respected with the use of `remap_pfn_range`, which eventually calls into `remap_pte_range`, thereby modifying the page table.

2. `__my_shmem_fault_remap` serves as inner logic for when outer page fault handling (allocation) logic deems that a sufficient number of pages exist for handling the current page fault. As its name suggests, it finds and remaps the correct

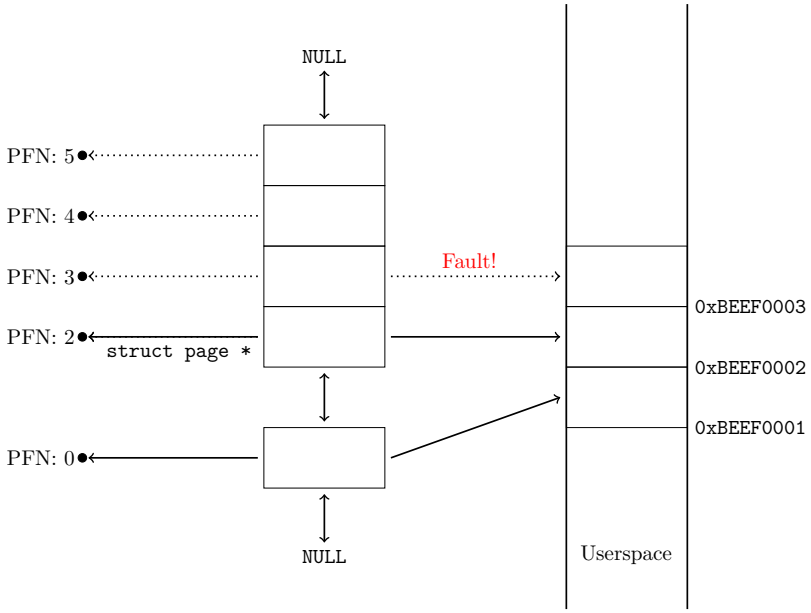

Figure 3.1: Misaligned Kernel Page Remap. Left column represents physical memory (addressed by PFN); center column represents in-module accounting of allocations; right column represents process address space.

allocation into the page fault's parent VMA (assuming that such allocation, of course, exists).

The logic of this function is similar to `my_shmem_fops_mmap`. For a code excerpt listing, refer to [TODO] Appendix ???.

Function `.close` emulates synchronization behavior whenever a VMA is removed from a process's address space (e.g., due to `munmap`). Given a removed VMA as argument, it computes the intersecting allocations and invokes `dcache_clean_poc` on each such allocations. While this results in conservative approximation of cleaned cache entries, it is nevertheless good for instrumentation purposes, as the amount of pages cleaned per invocation becomes invariable with respect to how the VMA was remapped – a misaligned VMA will not result in less pages being flushed in a given allocation.

```
1  static void my_shmem_vmops_close(struct vm_area_struct *vma)
2  {
3      size_t vma_pg_count =
4          (vma->vm_end - vma->vm_start) » PAGE_SHIFT;
5      size_t vma_pg_off = vma->vm_pgoff;
6
7      /* Lock mutex... */
8      struct my_shmem_alloc *entry;
9      list_for_each_entry(entry, &my_shmem_allocs, list) {
10         const ulong entry_pg_count =
11             ORDER_TO_PAGE_NR(entry->alloc_order);
12
13         /* Loop till entry intersects with start of VMA */
```

```
14          if (vma_pg_off > entry_pg_count) {
15              vma_pg_off -= entry_pg_count;
16              continue;
17          }
18
19          /* All of VMA cleaned: exit */
20          if (!vma_pg_count)
21              break;
22
23          /* entry intersects with VMA - emulate clean */
24          struct page *pg = entry->page;
25          ulong kvaddr_bgn = (ulong) page_address(pg);
26          ulong kvaddr_end =
27              kvaddr_bgn + entry_pg_count * PAGE_SIZE;
28          __dcache_clean_poc(kvaddr_bgn, kvaddr_end); // See 14
29          put_page(pg); // decrement refcount
30
31          /* Prepare for next iteration */
32          vma_pg_count -= min(
33              entry_pg_count - vma_pg_off,
34              vma_pg_count
35          );
36          if (vma_pg_off != 0) // ~ first intersection
37              vma_pg_off = 0;
38      }
39
40      /* cleanup... */
41  }
```

***sysfs* Parameter** Finally, `my_shmem` exposes a tunable *sysfs* parameter for adjusting the number of pages allocated per allocation in `my_shmem_vmops_fault`. The parameter, `max_contiguous_alloc_order`, defines the order o for allocation from page allocator such that, for each allocation, 2^o contiguous pages are allocated at once.

To adjust the parameter (for example, set $o \leftarrow 2$), one may run as follows in a sh-compatible terminal:

```
$ echo 2 > \
    /sys/module/my_shmem/parameters/max_contiguous_alloc_order
```

Consequently, all allocations occuring after this change will be allocated with a 4-page contiguous granularity. Upon further testing, the maximum value allowed here is 10 (i.e., $2^{10} = 1024$ 4K pages).

3.2.3 Instrumentation: `ftrace` and `bcc-tools`

We use two instrumentation frameworks to evaluate the latency of software-initiated coherency operations. `ftrace` is the primary kernel tracing mechanism across multiple (supporting) architectures, which supports both *static* tracing of tracepoints and *dynamic* tracing of function symbols:

- **Static** tracepoints describe tracepoints compiled into the Linux kernel. They are defined by kernel programmers and is otherwise known as *event tracing*.
- **Dynamic** `ftrace` support is enabled by self-modifying the kernel code to replace injected placeholder nop-routines with `ftrace` infrastructure calls. This allows for function tracing of all function symbols present in C object files created for linkage. [61]

Because we do not inline `__dcache_clean_poc`, we are able to include its symbol inside compiled C object files and hence expose its internals for dynamic tracing.

`bcc-tools`, on the other hand, provide an array of handy instrumentation tools that is compiled just-in-time into *BPF* programs and ran inside a in-kernel virtual machine. Description of how BPF programs are parsed and run inside the Linux kernel is documented in the kernel documentations [43]. The ability of `bcc`/`libbpf` programs to interface with both userspace and kernelspace function tracing mechanisms make `bcc-tools` ideal as a easy tracing interface for both userspace and kernelspace tracing.

3.2.4 Userspace Programs

Finally, two simple userspace programs are written to invoke the corresponding kernelspace callback operations – namely, allocation and cleaning of kernel buffers for simulating DMA behaviors. To achieve this, it simply `mmap`s the amount of pages passed in as argument and either reads or writes the entirety of the buffer (which differentiates the two programs). A listing of their logic is at [TODO] Appendix ???.

3.3 Results

3.3.1 Controlled Allocation Size; Variable Allocation Count

Experiments are first conducted over software coherency operation latencies over variable `mmap`-ed memory area sizes while keeping the underlying allocation sizes to 4KiB (i.e., single-page allocation). All experiments are conducted on `star` on `mmap` memory areas ranged from 16KiB till 1GiB, in which we control the number of sampled coherency operations to 1000. Data gathering is performed using the `trace-cmd` front-end for `ftrace`. The results of the experiments conducted is listed in figure 3.2.

Additionally, we also obtain the latencies of TLB flushes due to userspace programs, as listed in figure 3.3.

Notes on Long-Tailed Distribution

We identify that a long-tailed distribution of latencies exist for both figures (3.2, 3.4). For software coherency operations, we identify this to be partially due to *softirq* preemption (notably, RCU maintenance), which take higher precendence compared to "regular" kernel routines. A brief description of *processor contexts* defined in the Linux kernel is listed in Appendix ???.

For TLB operations, we identify the cluster of long-runtime TLB flush operations (e.g., around 10^4 μs) to be interference from `mm` cleanup on process exit.

Moreover, latencies to software coherency operations are highly system-specific. On `rose`, data gathered from similar experimentations have shown to be $1/10$-th of the latencies gathered from `star`, which (coincidentially) reduces the likelihood of long-tailed distributions forming due to RCU *softirq* preemption.

3.3.2 Controlled Allocation Count; Variable Allocation Size

We also conduct experiments over software coherency operations latencies over fixed `mmap`-ed memory area sizes while varying the underlying allocation sizes. This is achieved by varying the allocation order – while 0-order allocation allocates $2^0 = 1$ page per allocation, a 8-order allocation allocates $2^8 = 256$ contiguous pages per allocation. All experiments are conducted on `star`. The results for all experiments are gathered using `bcc-tools`, which provide utilities for injecting *BPF*-based tracing routines. The results of these experimentations are visualized in figure 3.4, with $N \geq 64$ per experiment.

Order	25p	50p (Median)	75p	99p
0	5.968	9.808	15.808	58.464
2	8.960	13.152	17.776	39.184
4	19.216	21.120	23.648	123.984
6	67.376	70.352	74.304	103.120
8	278.784	303.136	324.048	1783.008
10	1050.752	1141.312	1912.576	2325.104

Table 3.4: Coherency op latency of Variable-order Contiguous Allocation. Time listed in μs. $N = 100$ across allocation orders.

3.4 Discussion

Figures 3.2, 3.4 exhibits that, in general, coherency maintenance operation is **unrelated with the size of the mapped memory area** and **correlated with how large a single contiguous allocation is made**. We especially note that the runtime of each software-initiated coherency maintenance operation **does not grow linearly with allocation size**. Given that both axis of figure 3.4 is on a log-scale, with the "order" axis interpretable as a $\log_2$ scale of number of contiguous 4K pages, a perfect linear correlation between allocation size and latency would see a roughly linear interpolation between the data

points. This is obviously not the case for figure 3.4, which sees software coherency operation latency increasing drastically once order ≥ 6 (i.e., 64 contiguous pages), but remain roughly comparable for smaller orders.

On the other hand, linearly increasing coherency operation latencies exhibited for higher-order allocations have their runtimes amortized by two factors:

1. Exponentially-decreasing number of buffers (allocations) made in the underlying kernel module, which corresponds to less memory allocation calls made during runtime.
2. Latency of contiguous allocation operations (i.e., `alloc_pages`) **does not** grow significantly in relation to the size of the allocation.

Due to both factors, it remains economic to allocate larger contiguous allocations for DMA pages that are subject to frequent cache coherency maintenance operations than applying a "scatter-gather" paradigm to the underlying allocations.

3.4.1 *Hugepages* and RDMA-based DSM

Hugepage is an architectural feature that allows an aligned, larger-than-page-size contiguous memory region to be represented using a single TLB entry. x86-64, for example, supports (huge)pages to the size of 4KiB, 2MiB, or 1GiB [31]. ARM64 supports a more involved implementation of TLB entries, allowing it to represent more variable pages sizes in one TLB entry (up to 16GiB) [32]. Hypothetically, using hugepages as backing store for very large RDMA buffers reduces address translation overhead, either by relieving TLB pressure or through reduced page table indirections [71].

Specifically, the kernel developers identify the following factors that allow *hugepages* to create faster large-working-set programs [64]:

1. TLB misses run faster.
2. A single TLB entry corresponds to a much larger section of virtual memory, thereby reducing miss rate.

In general, performance critical computing applications dealing with large memory working sets will be running on top of *hugetlbfs* – hugepage mechanism exposed by the Linux kernel to userspace [64]. Alternatively, the use of hugepages could be dynamically and transparently enabled and disabled in userspace using *transparent hugepages* supported by contemporary Linux kernels [64]. This enhances programmer productivity in userspace programs when relying on a hypothetical *transparent hugepage*-enabled in-kernel DSM system for heterogenenous data processing tasks on variable-sized buffers, though few in-kernel mechanisms actually incorporate *transparent hugepages* support – at the time of writing, only anonymous *vma*s (e.g., stack, heap, etc.) and *tmpfs/shmem* incorporates *transparent hugepage* [64].

We identify *transparent hugepage* support as one possible direction to improving in-kernel DSM system performance. Traditionally, userspace programs who really wishes to allocate hugepages rely on *libhugetlbfs* as interface to the Linux kernel's *hugetlbfs* mechanism. These techniques remain heavily reliant on programmer discretion which

is fundamentally at odds with what the parent project of this paper envisions: a remote compute node is exposed as a DMA-capable accelerator to another, whereby two compute nodes could transparently perform computation on each other's memory via heterogeneous memory management mechanism. Because this process is transparent to the userspace programmer (who only have access to e.g., `/dev/my_shmem`), ideally the underlying kernel handler to `/dev/my_shmem` should abstract away the need for hugepages for very large allocations (since this is not handled by *libhugetlbfs*). Furthermore, transparent hugepage support would also hypothetically allow for shared pages to be promoted and demoted on ownership transfer time, thereby allowing for dynamically-grained memory sharing while maximizing address translation performance.

Furthermore, further studies remains necessary to check whether the use of (transparent) hugepages significantly benefit a real implementation of an in-kernel DSM system. Current implementation for `alloc_pages` does not allow for allocation of hugepages even when the allocation order is sufficiently large. Consequently, future studies need to examine alternative implementations that incorporate transparent hugepages into the DSM system. One candidate that could allow for hugepage allocation, for example, is to directly use `alloc_pages_mpol` instead of `alloc_pages`, as is the case for the current implementation of *shmem* in kernel.

3.4.2 Access Latency Post-*PoC*

This chapter solely explores latencies due to software cache coherency operations. In practice, it may be equally important to explore the latency incurred due to read/write accesses after *PoC* is reached, which is almost always the case for any inter-operation between CPU and DMA engines.

Recall from section 2.3.5 that ARMv8-A defines *Point* of Coherency/Unification within its coherency domains. In practice, it often implies an actual, physical *point* to which cached data is evicted to:

- Consider a ARMv8-A system design with a shared L2/lowest-level cache that is also snooped by the DMA engine. Here, the *Point-of-Coherency* could be defined as the shared L2 cache to which higher-level cache entries are cleaned or invalidated.
- Alternatively, a DMA engine may be capable of snooping all processor caches. The *Point-of-Coherency* could then be defined merely as the L1 cache, with some overhead depending on how the DMA engine accesses these caches.

Further studies are necessary to examine the latency after coherency maintenance operations on ARMv8 architectures on various systems, including access from DMA engine vs. access from CPU, etc.

3.4.3 Reflection

We identify the following weaknesses within our experiment setup that undermines the generalizability of our work.

What About `dcache_inval_poc`? Due to time constraints, we were unable to explore the latencies posed by `dcache_inval_poc`, which will be called whenever the DMA driver attempts to prepare the CPU to access data modified by DMA engine. Further studies that expose `dcache_inval_poc` for similar instrumentation should be trivial, as the steps necessary should mirror the case for `dcache_clean_poc` listed above.

Do Instrumented Statistics Reflect Real Latency? It remains debateable whether the method portrayed in section 3.2, specifically via exporting `dcache_clean_poc` to driver namespace as a traceable target, is a good candidate for instrumenting the "actual" latencies incurred by software coherency operations.

For one, we specifically opt not to disable IRQ when running `__dcahce_clean_poc`. This mirrors the implementation of `arch_sync_dma_for_cpu`, which:

1. is (at least) called under process context.
2. does not disable IRQ downstream.

Similar context is also observed for upstream function calls, for example `dma_sync_single_for_device`. As a consequence, kernel routines running inside IRQ/*softirq* contexts are capable of preempting the cache coherency operations, hence preventing early returns. The effect of this on tail latencies have been discussed in section 3.3.1.

On the other hand, it may be argued that analyzing software coherency operation latency on a hardware level better reveals the "real" latency incurred by coherency maintenance operations during runtime. Indeed, latencies of `clflush`-family of instructions performed on x86 chipsets measured in units of clock cycles [38, 24] amount to around 250 cycles – significantly less than microsecond-grade function call latencies for any GHz-capable CPUs. We argue that because an in-kernel implementation of a DSM system would more likely call into the exposed driver API function calls as opposed to individual instructions – i.e., not writing inline assemblies that "reinvent the wheel" – instrumentation of relatively low-level and synchronous procedure calls is more crucial than instrumenting individual instructions.

Lack of Hardware Diversity The majority of data gathered throughout the experiments come from a single, virtualized setup which may not be reflective of real latencies incurred by software coherency maintenance operations. While similar experiments have been conducted in bare-metal systems such as `rose`, we note that `rose`'s *Ampere Altra* is certified *SystemReady SR* by ARM [45] and hence supports hardware-coherent DMA access (by virture of *ARM Server Base System Architecture* which stipulates hardware-coherent memory access as implemented via MMU) [44], and hence may not be reflective of any real latencies incurred via coherency maintenance.

On the other hand, we note that a growing amount of non-hardware-coherent ARM systems with DMA-capable interface (e.g., PCIe) are quickly becoming mainstream. Newer generation of embedded SoCs are starting to feature PCIe interface as part of their I/O provisions, for example *Rockchip*'s *RK3588* [14] and *Broadcom*'s *BCM2712*

[1], both of which were selected for use in embedded and single-board systems, though (at the time of writing) with incomplete kernel support. Moreover, desktop-grade ARM CPUs and SoCs are also becoming increasingly common, spearheaded by *Apple*'s *M*-series processors as well as *Qualcomm*'s equivalent products, all of which, to the author's knowledge, **do not** implement hardware coherence with their PCIe peripherals. Consequently, it is of interest to evaluate the performance of software-initiated cache coherency operations commonly applied in CPU-DMA interoperations on such non-*SystemReady SR* systems.

Orthogonally, even though the *virt* emulated platform does not explicitly support hardware-based cache coherency operations, the underlying implementation of its emulation on x86 hosts is not explored in this study. Because (as established) the x86 ISA implements hardware-level guarantee of DMA cache coherence, if no other constraints exist, it may be possible for a "loose" emulation of the ARMv8-A ISA to define *PoC* and *PoU* operations as no-ops instead, though this theory cannot be ascertained without any cross-correlation with *virt*'s source code. Figure 3.4 also disputes this theory, as a mapping from ARMv8-A *PoC* instructions to x86 no-op instructions would likely not cause differing latency magnitude over variable-sized contiguous allocations.

Inconsistent Latency Magnitudes Across Experiments We recognize .... We deduce this is due to one important variable across all experiments that we failed to control – power supply to host machine.

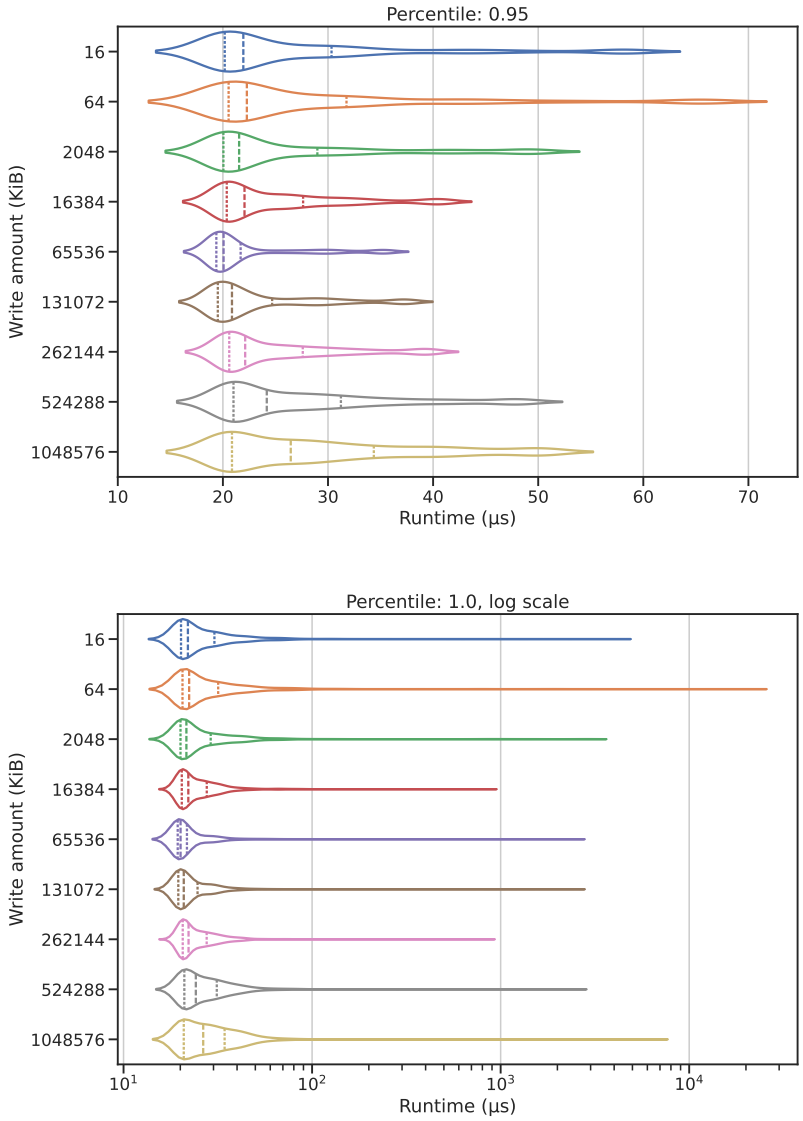

Figure 3.2: Coherency operation latency. Allocation on per-page basis. Vertical lines represent 25th, 50th, and 75th percentiles respectively.

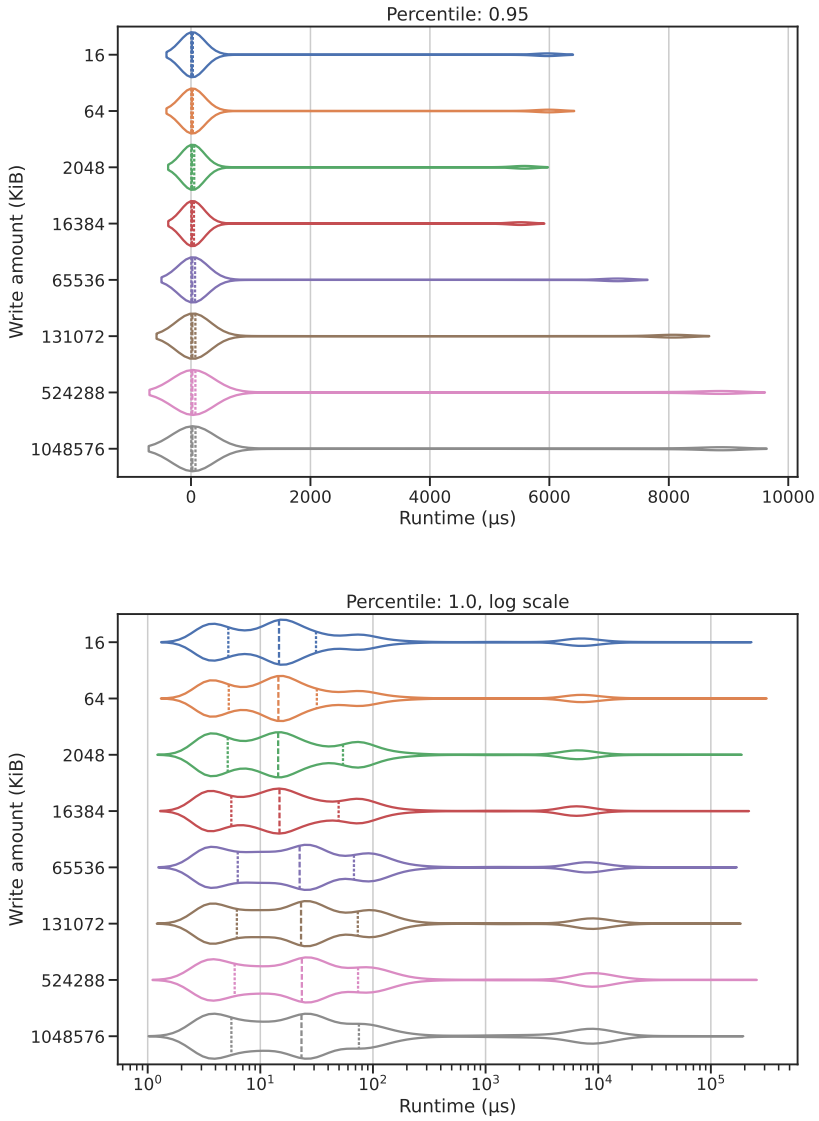

Figure 3.3: TLB operation latency. Allocation on per-page basis. Vertical lines represent 25th, 50th, and 75th percentiles respectively.

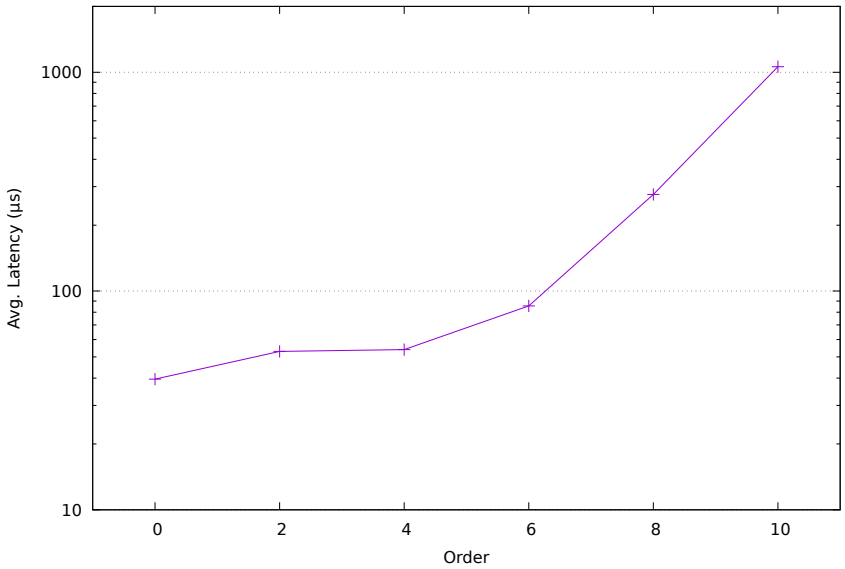

Figure 3.4: Average coherency op latency of variable-order contiguous allocation.

Chapter 4

Conclusion

4.1 Summary

4.2 Future Work

Bibliography

[1] 2023. URL: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf.

[2] URL: https://www.phoronix.com/search/Heterogeneous%20Memory%20Management.

[3] URL: https://uawartifacts.blob.core.windows.net/upload-files/Altra_Max_Rev_A1_DS_v1_15_20230809_b7cdce449e_424d129849.pdf.

[4] URL: https://hadoop.apache.org/.

[5] URL: https://spark.apache.org/.

[6] Cristiana Amza et al. "Treadmarks: Shared memory computing on networks of workstations". In: *Computer* 29.2 (1996), pp. 18–28.

[7] ARM. *ARMő Cortexő-A Series Programmer's Guide for ARMv8-A*. 2015. URL: https://developer.arm.com/documentation/den0024/a.

[8] Hans J Boehm and Lawrence Crowl. *C++ Atomic Types and Operations*. 2007. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html.

[9] *BS ISO/IEC 9899:2011: Information technology. Programming languages. C*. eng. 2013.

[10] Javier Cabezas et al. "GPU-SM: shared memory multi-GPU programming". In: *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. 2015, pp. 13–24.

[11] Qingchao Cai et al. "Efficient distributed memory management with RDMA and caching". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1604–1617.

[12] John B Carter, John K Bennett, and Willy Zwaenepoel. "Implementation and performance of Munin". In: *ACM SIGOPS Operating Systems Review* 25.5 (1991), pp. 152–164.

[13] David Chaiken, John Kubiatowicz, and Anant Agarwal. "LimitLESS directories: A scalable cache coherence scheme". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 224–234. ISBN: 0897913809. DOI: 10.1145/106972.106995. URL: https://doi.org/10.1145/106972.106995.

[14] Rockchip Electronics Co. Ltd. *RK3588*. 2022. URL: https://www.rock-chips.com/a/en/products/RK35_Series/2022/0926/1660.html.

[15] Jonathan Corbet. 2021. URL: https://lwn.net/Articles/855328/.

[16] Jonathan Corbet. *Heterogeneous memory management meets EXPORT_SYMBOL_GPL()*. 2018. URL: https://lwn.net/Articles/757124/.

[17] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[18] Maria Couceiro et al. "D2STM: Dependable distributed software transactional memory". In: *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2009, pp. 307–313.

[19] Mattias De Wael et al. "Partitioned global address space languages". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–27.

[20] Zhuocheng Ding. "vDSM: Distributed Shared Memory in Virtualized Environments". In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*. 2018, pp. 1112–1115. DOI: 10.1109/ICSESS.2018.8663720.

[21] Noel Eisley, Li-Shiuan Peh, and Li Shang. "In-network cache coherence". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, pp. 321–332.

[22] Wataru Endo, Shigeyuki Sato, and Kenjiro Taura. "MENPS: a decentralized distributed shared memory exploiting RDMA". In: *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE. 2020, pp. 9–16.

[23] Brett Fleisch and Gerald Popek. "Mirage: A coherent distributed shared memory design". In: *ACM SIGOPS Operating Systems Review* 23.5 (1989), pp. 211–223.

[24] Agner Fog. "Instruction tables". In: *Technical University of Denmark* (2018).

[25] Davide Giri, Paolo Mantovani, and Luca P Carloni. "NoC-based support of heterogeneous cache-coherence models for accelerators". In: *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE. 2018, pp. 1–8.

[26] Mark Harris. *Unified memory for cuda beginners*. 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[27] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[28] Stephen Alan Holsapple. *DSM64: A Distributed Shared Memory System in User-Space*. California Polytechnic State University, 2012.

[29] Yang Hong et al. "Scaling out NUMA-aware applications with RDMA-based distributed shared memory". In: *Journal of Computer Science and Technology* 34 (2019), pp. 94–112.

[30] Weiwu Hu, Weisong Shi, and Zhimin Tang. "JIAJIA: A software DSM system based on a new cache coherence protocol". In: *High-Performance Computing and Networking: 7th International Conference, HPCN Europe 1999 Amsterdam, The Netherlands, April 12–14, 1999 Proceedings 7*. Springer. 1999, pp. 461–472.

[31] *HugeTLB Pages*. 2023. URL: https://www.kernel.org/doc/html/v6.7/admin-guide/mm/hugetlbpage.html.

[32] *HugeTLBpage on ARM64*. 2023. URL: https://www.kernel.org/doc/html/v6.7/arch/arm64/hugetlbpage.html.

[33] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. "Thread migration and its applications in distributed shared memory systems". In: *Journal of Systems and Software* 42.1 (1998), pp. 71–87.

[34] Chengfan Jia et al. “Improving the performance of distributed tensorflow with RDMA”. In: *International Journal of Parallel Programming* 46 (2018), pp. 674–685.

[35] Stefanos Kaxiras et al. “Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 3–14. ISBN: 9781450335508. DOI: 10.1145/2749246.2749250. URL: https://doi.org/10.1145/2749246.2749250.

[36] Ahmed Khawaja et al. “Sharing, Protection, and Compatibility for Reconfigurable Fabric with {AmorphOS}”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 107–127.

[37] Sang-Hoon Kim et al. “DeX: Scaling Applications Beyond Machine Boundaries”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 864–876. DOI: 10.1109/ICDCS47774.2020.00021.

[38] Sowoong Kim, Myeonggyun Han, and Woongki Baek. “MARF: A Memory-Aware CLFLUSH-Based Intra-and Inter-CPU Side-Channel Attack”. In: *European Symposium on Research in Computer Security*. Springer. 2023, pp. 120–140.

[39] Toddj Kjos et al. *Hardware cache coherent input/output*. eng. PALO ALTO, 1996.

[40] Daniel Lenoski et al. “The stanford dash multiprocessor”. In: *Computer* 25.3 (1992), pp. 63–79.

[41] Feng Li et al. “Accelerating relational databases by leveraging remote memory and RDMA”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 355–370.

[42] Kai Li and Richard Schaefer. “Shiva: An operating system transforming a hypercube into a shared-memory machine”. In: (1989).

[43] *libbpf Overview*. 2023. URL: https://www.kernel.org/doc/html/v6.7/bpf/libbpf/libbpf_overview.html.

[44] Arm Ltd. *Arm Server Base System Architecture 7.1*. 2022. URL: https://developer.arm.com/documentation/den0029/h.

[45] Arm Ltd. *SystemReady SR*. 2024. URL: https://www.arm.com/architecture/system-architectures/systemready-certification-program/sr.

[46] Honghui Lu et al. “Message passing versus distributed shared memory on networks of workstations”. In: *Supercomputing’95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. IEEE. 1995, pp. 37–37.

[47] Xiaoyi Lu et al. “Accelerating spark with RDMA for big data processing: Early experiences”. In: *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*. IEEE. 2014, pp. 9–16.

[48] Jiacheng Ma et al. “A hypervisor for shared-memory FPGA platforms”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 827–844.

[49] Jeremy Manson and Brian Goetz. 2004. URL: https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

[50] Dimosthenis Masouros et al. “Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 855–869.

[51] David S Miller, Richard Henderson, and Jakub Jelinek. *Dynamic DMA mapping Guide*. 2024. URL: https://www.kernel.org/doc/html/v6.7/core-api/dma-api-howto.html.

[52] Vijay Nagarajan et al. *A primer on memory consistency and cache coherence*. Springer Nature, 2020.

[53] Jacob Nelson et al. “{Latency-Tolerant} software distributed shared memory”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 291–305.

[54] SeungYong Oh and JongWon Kim. “Stateful Container Migration employing Checkpoint-based Restoration for Orchestrated Container Clusters”. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. 2018, pp. 25–30. DOI: 10.1109/ICTC.2018.8539562.

[55] *Ordering in core::sync::atomic - Rust*. 2024. URL: https://doc.rust-lang.org/core/sync/atomic/enum.Ordering.html.

[56] Neil Parris. *Extended system coherency: Cache Coherency Fundamentals*. 2013. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/extended-system-coherency--part-1--cache-coherency-fundamentals.

[57] Christian Pinto et al. “Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 868–880.

[58] *Platform-Specifc Notes*. 2023. URL: https://chapel-lang.org/docs/platforms/index.html#.

[59] The FreeBSD Project. *FreeBSD manual pages*. 2021. URL: https://man.freebsd.org/cgi/man.cgi?query=bpf&manpath=FreeBSD+14.0-RELEASE+and+Ports.

[60] Manuel Rodrguez-Pascual et al. “Job migration in hpc clusters by means of checkpoint/restart”. In: *The Journal of Supercomputing* 75 (2019), pp. 6517–6541.

[61] Steven Rostedt. *ftrace - Function Tracer*. Ed. by Du Changbin. 2023. URL: https://www.kernel.org/doc/html/v6.7/trace/ftrace.html#dynamic-ftrace.

[62] Ioannis Schoinas et al. “Sirocco: Cost-effective fine-grain distributed shared memory”. In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE. 1998, pp. 40–49.

[63] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. “Distributed Shared Persistent Memory”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 323–337. ISBN: 9781450350280. DOI: 10.1145/3127479.3128610. URL: https://doi.org/10.1145/3127479.3128610.

[64] *Transparent Hugepage Support*. 2023. URL: https://www.kernel.org/doc/html/v6.7/admin-guide/mm/transhuge.html.

[65] *upcc.1*. 2022. URL: https://upc.lbl.gov/docs/user/upcc.html.

[66] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[67] Arjan van de Ven. *Background on ioremap, cacheing, cache coherency on x86*. 2008. URL: https://lkml.org/lkml/2008/4/29/480.

[68] Qing Wang et al. "Concordia: Distributed Shared Memory with In-Network Cache Coherence". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 277–292. ISBN: 978-1-939133-20-5. URL: https://www.usenix.org/conference/fast21/presentation/wang.

[69] Paul Werstein, Mark Pethick, and Zhiyi Huang. "A performance comparison of dsm, pvm, and mpi". In: *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE. 2003, pp. 476–482.

[70] Xelu86 et al. *SMB Direct*. 2024. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-direct.

[71] Jian Yang, Joseph Izraelevitz, and Steven Swanson. "{FileMR}: Rethinking {RDMA} Networking for Scalable Persistent Memory". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 111–125.

[72] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[73] Jin Zhang et al. "Giantvm: A type-ii hypervisor implementing many-to-one virtualization". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 30–44.

[74] Huan Zhou et al. "DART-MPI: An MPI-based implementation of a PGAS runtime system". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 2014, pp. 1–11.

Appendix A

Terminologies

This chapter provides a listing of all terminologies used in this thesis that may be of interest or warrant a quick-reference entry during reading.

Appendix B

More on The Linux Kernel

This chapter provides some extra background information on the Linux kernel that may have been mentioned or implied but bears insufficient significance to be explained in the Background chapter of this thesis.

B.1 Processor Context

B.2 `enum dma_data_direction`

B.3 Use case for `dcache_clean_poc`: *smbdirect*

Appendix C

Cut & Extra Work

This chapter provides a brief summary of some work that was done during the writing of the thesis, but the author decided against inclusion of into the submitted work. It also explains some assumptions made with regards to the title of this thesis that the author find to have weakness on second thought.

C.1 Replacement Policy

C.2 Coherency Protocol

C.3 Listing: Userspace

C.4 *Why did you do ⋆?*