

© 2024 Neela Louise Kausik

TRUDI: TRUSTED USERSPACE DMA USING IPC

BY

NEELA LOUISE KAUSIK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the Grainger College of Engineering of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Advisor:

Professor Wen-mei Hwu

ABSTRACT

As the computer industry progresses in accelerating applications with accelerators like GPUs (Graphic Processing Units), the OS-kernel-level I/O software stack for storage data access and network communication has increasingly become the bottleneck of the end-to-end application execution. To alleviate this new bottleneck, high-performance systems are evolving towards user-space I/O services, where user-level libraries directly communicate with storage and I/O devices for data transfers, bypassing the operating system. Such systems pre-allocate and pin user memory, map for Direct Memory Access (DMA), and return the DMA physical address. The userspace application can use the returned DMA address to request the storage devices to directly read from and write to the pinned memory. However, there are currently no measures that prevent a malicious application from specifying a DMA address for memory belonging to another process. In this thesis, we propose TRUDI, a system design that enables a trusted user-level process to maintain DMA addresses and initiate I/O requests on behalf of an application process without exposing DMA addresses to the application process, thereby preventing unauthorized access to the physical memory of other processes. We provide the basic primitive of registering and sharing memory buffers between an untrusted application and a trusted process. These buffers can also be mapped for DMA with an I/O device by the trusted process, which maintains the needed DMA addresses in metadata. This allows the application to identify memory with a virtual address that the trusted entity can verify and translate. With this shared-memory primitive, we can build an isolated and high-throughput communication channel between the untrusted application and trusted process. We exemplify this by implementing a shared queue that allows CPU or GPU applications to communicate to the trusted process at high-throughput.

Subject Keywords: Direct Memory Access (DMA), security, GPUs

ACKNOWLEDGMENTS

I would like to thank Zaid Qureshi and Vikram Sharma Mailthody for their guidance, encouragement, and patience with me throughout the research and thesis process. I would also like to express my gratitude to Professor Wenmei Hwu for giving me the opportunity to do this research and for his help and advice. Finally, I would like to thank Brian Park and the rest of the IMPACT group for their support and inspiration.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	BACKGROUND	3
2.1	Paging	3
2.2	DMA and Pinned Memory	3
2.3	Inter-Process Communication	5
2.4	Relevance	5
CHAPTER 3	DESIGN AND IMPLEMENTATION	7
3.1	Unix Socket	7
3.2	Sharing Buffers	8
3.3	High-Throughput Queue	14
3.4	Putting it all Together	17
CHAPTER 4	ANALYSIS	19
4.1	Shared Buffer	19
4.2	Read/Write Request Queue	20
CHAPTER 5	CONCLUSION	22
REFERENCES	23

CHAPTER 1

INTRODUCTION

As the computing industry accelerates applications with accelerators like Graphic Processing Units (GPUs), the execution time due to the OS-kernel I/O software stacks has accounted for an increasing portion of the end-to-end application execution time. To avoid the costly kernel-level software stack overhead, such as user-space and kernel boundary crossings, multiple data copies, and software overheads in the OS kernel, modern systems are moving towards user-space I/O libraries and services [1, 2, 3]. This enables the application itself to communicate with I/O devices, for example to read data from storage, without the operating system. These systems pre-allocate memory and use APIs provided by the operating systems, e.g. `get_user_pages` [4], to pin user memory, Direct Memory Access (DMA) map it, and get the DMA address for the memory. The user-space application can then use the returned DMA address to program the device to read and write to the application’s pinned memory.

However, allowing an application to program other devices with DMA addresses can be very dangerous. When preparing a command for the device, a malicious user can specify whatever DMA address they want, as nothing is forcing them to use the specific DMA address for the memory they pinned. As an example, if the user is programming a storage controller to read data, if the user specifies an incorrect address, the storage controller can overwrite memory not belonging to that user, possibly even operating system memory or memory exposed by other devices in the system. Using write commands, they can use a storage controller to read data that does not belong to them, breaking the memory isolation provided by the operating system.

We propose TRUDI, a system design that allows a trusted user-space process to initiate I/O requests on behalf of a user-space application process, for its performance benefits, but that 1) does not expose DMA addresses to the user-space application, and 2) does not allow the application to communicate

devices using said DMA addresses. We need a mechanism to provide a virtual address for pinned memory to the application. Thus, our system enables the userspace application to register and share memory buffers with a trusted agent. The application only uses the virtual address to make user-initiated I/O requests through the trusted entity which translates the virtual address to the correct DMA address. If the address specified by the application is illegal, then the trusted entity can return an error as a response to the application's request. In our system, this trusted entity will be a trusted process running in the system. Furthermore, our system extends to accelerators like GPUs, enabling un-trusted GPU code to make requests to a trusted entity that also runs on the GPU, using the same buffer registration and identification scheme. This ability enables acceleration of services performed by the trusted process.

CHAPTER 2

BACKGROUND

This chapter describes the operating system and DMA features based on which the system described in this thesis is designed, and how they necessitate the new system design. We also discuss the relevant related work of this research.

2.1 Paging

Paging is an efficient memory management technique in which the operating system (OS) divides memory into pages of constant size that can be retrieved from secondary storage when needed. This creates the illusion of infinite memory capacity, or virtual memory. The paging process utilizes page tables in order to translate virtual addresses to physical addresses.

As the user processes work on different parts of the virtual memory data, the paging mechanism needs to bring in the new parts that are actively in use. Since the physical memory is of limited size, some of the previously used virtual memory data needs to be purged from the physical memory to make room. In a modern computing system, there are usually many user processes running. The paging activities for some of the processes are often conducted concurrently with the application compute and I/O activities of other processes. It is important that the paging activities do not accidentally corrupt the result of the I/O activities that are running concurrently.

2.2 DMA and Pinned Memory

Direct Memory Access (DMA) is a mechanism provided by devices such as storage controllers and Graphic Processing Units (GPUs) to move data in and out of the physical memory in the system. This mechanism frees

the operating system running on the CPU from the task of moving data in the system. As a result, the data movement activities can be overlapped with other computation or system management tasks running on the CPU, increasing the productivity of the system.

However, when another device moves data to or from a page in the physical memory, the operating system may accidentally reallocate the physical memory page to another virtual page and corrupt the data being transferred. Therefore, the pages being accessed by DMA devices must be pinned in physical memory during the access. In a virtual memory system, once a virtual memory page is pinned in the physical memory, it will remain in its currently mapped physical memory page indefinitely until it is unpinned.

The OS provides user application processes with the ability to pin pages in memory and map them for DMA for a device. The DMA mapping of a virtual memory page returns a DMA (physical) address that can be used by DMA devices to access the pinned pages in memory. Historically, memory pinning, DMA mapping, and programming the DMA between the device and memory are all handled by the OS on behalf of applications.

For example, when reading data from storage, the user-space application will make a system call with one of the arguments being the virtual address of a user-level buffer. The OS will identify a pre-allocated and DMA-mapped pinned memory region for the DMA. It will use the DMA address for that region to program the storage controller when asking for requested data. The storage controller will write the requested data in the pinned buffer and the OS will copy the data from the pinned buffer to the buffer provided by the user application. Finally, the user application can access the data in its buffer using virtual addresses.

With the emergence of accelerators and new storage devices, the overhead incurred by the OS when accessing storage data has become a major bottleneck for many accelerated applications. Due to the difficulties in accelerating and debugging OS code, computing systems are increasingly moving the storage access services from the OS level to the user level [2, 5]. For example, in the recent BaM system, the library that programs the storage devices for DMA is provided to the user-level application as a header-only library.

However, allowing untrusted user-level applications to manage DMA addresses, which are physical addresses, leads to the security vulnerability described in Chapter 1. Since a user application can manipulate physical ad-

dresses and present them as DMA addresses, it can potentially read and write to any physical address in the system, including those that are allocated to other user processes, through DMA activities. To eliminate such a vulnerability, one needs to limit the visibility and handling of DMA addresses to only trusted user service processes that perform the storage accesses on behalf of untrusted user application processes.

2.3 Inter-Process Communication

TRUDI utilizes two different forms of inter-process communication (IPC): Unix domain socket and shared memory. First, for Unix domain socket, communication occurs exclusively in the OS kernel. Two processes can connect via Unix socket by opening the socket's file descriptor on both ends. Though Unix sockets can be beneficial for their simplicity and security (one can change the permissions on a socket file descriptor), they involve many system calls. This increases overhead [6] compared to shared memory communication. For shared memory IPC, two processes need to open the same region of memory. Each process can read and write to that memory in order to communicate.

2.4 Relevance

Userspace-initiated I/O is becoming commonplace for performance [1, 2, 3]. Thus, efficiently providing memory isolation is key to preventing malicious actors trying to read and corrupt memory of other processes. Furthermore, modern accelerators like GPUs and FPGAs also enable userspace-initiated I/O [7, 8]. This increases the already large attack surface exposed by DMA vulnerabilities. Previous exploits in this area have been demonstrated [9, 10].

Currently, there is no performant mechanism for accelerators like GPUs to make requests to the trusted OS. The only current methods of doing so are Unified Virtual Memory (UVM) and Heterogeneous Memory Management. UVM has performance limitations, as its page fault handling incurs substantial software overhead, and GPU threads can only communicate to a CPU OS [2, 11].

The work in this thesis provides a performant framework where trusted user-level processes map and maintain DMA addresses on behalf of an untrusted user application processes. The trusted user process returns a handle for the maintained DMA access to the user application process. When the user application needs to request storage data access, it presents the handle to the trusted user process, which in turn programs the storage devices with the maintained DMA address. This approach eliminates the need to expose the DMA address to the user application process. Just as importantly, it avoids having to trust a user application process to present a valid, legitimate DMA address for storage data.

CHAPTER 3

DESIGN AND IMPLEMENTATION

This chapter describes the components of the system design in detail. TRUDI consists of a client, or user, process requesting operations from a server process. Section 3.1 describes how communication is established via Unix socket. Next, Section 3.2 details how the client can share buffers with the server, which registers them for use in future requests. Finally, Section 3.3 shows how inter-process communication can be maintained via a high-throughput queue, a use case of the buffer sharing mechanism. Figure 3.1 shows a high-level overview of the system design. Later, Chapter 4 will analyze how this system design addresses the fundamental security issues we introduced previously in Chapter 1.



Figure 3.1: System Design Overview

3.1 Unix Socket

A Unix socket is used to establish client and server connection. It transports the buffer registration commands (Section 3.2.3) and shared memory file descriptor (fd), so that shared memory buffers can be created on both processes.

The server process is the first to be initiated, with the `socket()` system call, followed by `bind()` to assign it to an address [12, 13]. Then, `listen()` marks the socket as passive, so that when a client’s connection request arrives, `accept()` completes the connection [14, 15].

The client process connects to an existing server process through the `socket()` and `connect()` system calls [16], giving the client a file descriptor, `socketfd`, to use for communicating with the server. When a client connects, the server creates a client handler process by calling the `fork()` system call [17]. After `fork()`, every client's server process has its own copy of the server state, including the mapping for buffers. This allows for isolation between each client handler.

After connecting, the client can use `write()` on the file descriptor for the socket to send data to the server and use `read()` on the same socket to get responses from the server. The server can similarly communicate with the client via `write()` and `read()` on the file descriptor it gets when a client connects. Below is an example of client code sending a file descriptor via Unix socket to a server (Line 3) and waiting for its response (Line 4):

```
1 char buffer[256];
2 sprintf(&buffer[0], "%d", fd); //put data into buffer
3 write(socketfd, buffer, strlen(buffer)); //write to socket fd
4 read(socketfd, buffer, 255); //read response
```

Listing 3.1: Unix socket example

This scheme can be used by the client to send predefined commands, e.g. `register_buffer`, to the server, who will respond appropriately.

3.2 Sharing Buffers

Shared memory between client and server is allocated for each buffer. The method for doing so differs for CPU and GPU, but the process is always initiated on the client side.

3.2.1 CPU Shared Memory

The Linux `dmabuf` subsystem allows for a buffer that corresponds to some memory to be shared across the system for DMA [18]. The buffer can be mapped to user space, kernel space, or external devices. Linux currently includes the driver `/dev/udmabuf`, obviating the need to create a new kernel driver to utilize `dmabuf` functionality.

For CPU applications, the process of creating and memory-mapping a shared buffer is shown in Listing 3.2. First, the client opens `/dev/udmabuf` (Line 16). Then, `memfd_create()` creates a file for the memory region (Line 17), which is sealed with `fcntl()` (Line 18) [19, 20, 21]. Finally, an `ioctl()` call (Line 28) on the `udmabuf` device creates and opens the `dma_buf` file for the memory region created with `memfd_create()` [22]. The return value is a file descriptor for the `dma_buf` which will then be used in `mmap()` to map the memory region backed by the `dma_buf` into the process's address space [23]. Note that the `MAP_SHARED` flag must be included. For TRUDI, we adapted from the usage of the original developers of the `udmabuf` device [24, 25, 26, 27].

```

1 struct udmabuf_create {
2     uint32_t memfd;
3     uint32_t flags;
4     uint64_t offset;
5     uint64_t size;
6 };
7
8 //Create ioctl identifier for udmabuf_create
9 #define UDMABUF_CREATE _IOW('u', 0x42, struct udmabuf_create)
10
11 struct udmabuf_create create;
12 int devfd, memfd, fd;
13 off_t size;
14
15 //Open udmabuf driver and create shared buffer
16 devfd = open("/dev/udmabuf", O_RDWR);
17 memfd = memfd_create("udmabuf-test", MFD_ALLOW_SEALING);
18 fcntl(memfd, F_ADD_SEALS, F_SEAL_SHRINK);
19 size = getpagesize() * NUM_PAGES;
20 ftruncate(memfd, size);
21 memset(&create, 0, sizeof(create));
22
23 //Open and memory-map dma_buf file
24 create.memfd = memfd;
25 create.offset = 0;
26 create.size = size;
27 create.flags = 0;
28 fd = ioctl(devfd, UDMABUF_CREATE, &create);
29 void * mmap_addr = (mmap(NULL, 4096, PROT_READ|PROT_WRITE,

```

30

```
MAP_SHARED, fd, 0));
```

Listing 3.2: CPU udmabuf

Then, the `dma_buf` `fd` is sent to the server through the Unix socket. A file descriptor (`fd`) is a process-specific pointer to kernel data [28]. Therefore, before memory-mapping, the server must translate the received `fd` to its process-specific `fd`. As Listing 3.3 shows, the server translates the client’s `fd` using `SYS_pidfd_getfd` (Line 10 in Listing 3.3) [29]. In order to determine the client’s `pid`, which is needed to translate the `fd`, the server must check the socket options using `getsockopt()` (Line 6) [30]. This prevents a malicious application from specifying another process’s `pid` to register a buffer it did not create.

```
1 socklen_t len;
2 struct ucred creds;
3 len = sizeof(struct ucred);
4
5 //Get client credentials
6 getsockopt(socketfd, SOL_SOCKET, SO_PEERCRED, &creds, &len);
7 int pid = creds.pid;
8
9 //Use client's pid to translate fd
10 int newfd = syscall(SYS_pidfd_getfd, syscall(SYS_pidfd_open,
11     pid, 0), fd, 0);
```

Listing 3.3: Server fd translation

3.2.2 GPU Shared Memory

CUDA now allows buffer sharing with `dma_buf` [31]. In order to take advantage of this functionality, one must check if it is supported as shown in Listing 3.4.

```
1 cuDeviceGetAttribute(&attr,
2     CU_DEVICE_ATTRIBUTE_DMA_BUF_SUPPORTED, cudev);
```

Listing 3.4: Checking for CUDA `dma_buf` support

If supported, a user can obtain a handle to memory in the form of a `dma_buf` as shown in Listing 3.5.

```
1 cuMemGetHandleForAddressRange((void *)&handle, ptr, sz,
```



```
2     CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD , OULL)
```

Listing 3.5: Obtaining dma_buf fd

For GPU applications, a shared buffer is created with the CUDA Driver API [32], rather than the Linux API described above. The shared memory is allocated and memory mapped by the client as shown in Listing 3.6. First, `cuMemGetAllocationGranularity` is used to compute the total size of the memory region to be allocated (Line 11). Then, `cuMemAddressReserve` reserves an address range (Line 16) before the memory can be created and memory mapped with `cuMemCreate` and `cuMemMap`, respectively (Lines 17-18). Memory access flags must be set appropriately with `cuMemSetAccess` (Line 2). Finally, the client obtains the buffer handle by calling `cuMemGetHandleForAddressRange` with the flag `CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD` (Line 27). This handle is essentially a file descriptor and is still sent over the Unix socket (Listing 3.1).

```
1 size_t sz;
2 CUdeviceptr ptr;
3 CUmemAccessDesc accessDesc;
4 CUmemGenericAllocationHandle hdl;
5
6 //Get allocation granularity
7 CUmemAllocationProp prop = {};
8 prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
9 prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
10 prop.location.id = dev;
11 CUresult ret = cuMemGetAllocationGranularity(&aligned_sz ,
12                                             &prop, CU_MEM_ALLOC_GRANULARITY_MINIMUM);
13 sz = ((size + aligned_sz - 1) / aligned_sz) * aligned_sz;
14
15 //Allocate and MemMap
16 cuMemAddressReserve(&ptr, sz, OULL, OULL, OULL);
17 cuMemCreate(&hdl, sz, &prop, 0);
18 cuMemMap(ptr, sz, OULL, hdl, OULL);
19
20 //Set Access
21 accessDesc.location.id = device;
22 accessDesc.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
23 accessDesc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
24 cuMemSetAccess(ptr, sz, &accessDesc, 1ULL);
25
```

```

26 //Get handle
27 cuMemGetHandleForAddressRange((void *)&handle, ptr, sz,
28     CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD, 0ULL);

```

Listing 3.6: GPU shared buffer code

In addition to the `handle`, the size is also sent to the server via the socket. The server must also get its own version of the handle using `SYS_pidfd_getfd` (Line 10 of Listing 3.7), which is then used to import the allocation with `cuMemImportFromShareableHandle()` (Line 12). The server completes `cuMemAddressReserve()` with its own `ptr` (Line 15). At that point, `cuMemMap()` and `cuMemSetAccess()` are invoked (Lines 16 and 22). Note that an additional call to `cuMemGetHandleForAddressRange()` would be needed to obtain the `dma_buf` file descriptor, which the server can use for DMA mapping.

```

1 //Received from socket
2 int handle;
3 size_t sz;
4
5 CUdeviceptr ptr;
6 CUmemAccessDesc accessDesc;
7 CUmemGenericAllocationHandle hdl;
8
9 //Translate handle and import memory
10 int newhandle = syscall(SYS_pidfd_getfd, syscall(
11     SYS_pidfd_open, pid, 0), handle, 0);
12 cuMemImportFromShareableHandle(&hdl, (void *)(uintptr_t)
13     newhandle, CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR);
14
15 //Memory map
16 cuMemAddressReserve(&ptr, sz, 0ULL, 0ULL, 0ULL);
17 cuMemMap(ptr, sz, 0ULL, hdl, 0ULL);
18
19 //Set Access
20 accessDesc.location.id = device;
21 accessDesc.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
22 accessDesc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
23 cuMemSetAccess(ptr, sz, &accessDesc, 1ULL);

```

Listing 3.7: Server opens shared buffer

3.2.3 Buffer Registration

Upon creation of the shared buffer, the server “registers” the buffer in the `buffer_map`. This maps each buffer to a handle, which is sent back to the client. The client must use a handle to identify a registered buffer when communicating requests to the server. The client can create several shared buffers with the server, each of which will have a corresponding handle. To do so, the client would specify the commands “`register_buffer`” or “`unregister_buffer`”. When a buffer is registered, it is added to `buffer_map`, and the corresponding handle is sent back to the client. If the client requests to unregister a buffer, the server removes the entry for the requested handle, responding with an error if no such entry exists. Listing 3.8 shows the fields of buffer registration requests.

```
1 struct register_req{
2     char op;           // "R" or "U" (register or unregister)
3     uint32_t fd;       // Buffer fd if "R", else ignored
4     size_t size;      // buffer size if "R", else ignored
5     uint32_t handle;   // Handle if "U", else ignored
6 };
```

Listing 3.8: Buffer registration command

Every client handler in the server keeps a map of registered buffers, and for each buffer it maintains metadata in the `buffer_map_entry` shown in Listing 3.9.

```
1 struct buffer_map_entry{
2     uint32_t fd;       // DMA buffer fd
3     uint64_t dma_addr; // DMA address if applicable
4     size_t size;      // buffer size
5 };
```

Listing 3.9: `buffer_map` entry struct

The `buffer_map` enables the server to check which buffers a client has created and therefore has access to. In the future, if the client requests to read or write with a handle that does not exist in the map, the server will respond with an error. Additionally, the server will check the `size` element to determine if a request has been made with either a size or offset value that is out of range, responding with an error as well.

3.3 High-Throughput Queue

While read and write requests can be communicated through the existing Unix socket, this communication has a couple of shortcomings. First, it limits the usability to CPU application threads, meaning other accelerators such as GPUs cannot use it. Second, communication through the socket requires system calls which have high overhead [6]. In this section, we will show how we can use the buffer sharing mechanism to create an isolated communication channel. Once a connection is established between the two processes, the overhead that sockets incur from context switching is no longer necessary, as the two processes can communicate through shared memory. Figure 3.2 shows an overview of the primary system design with an additional queue.

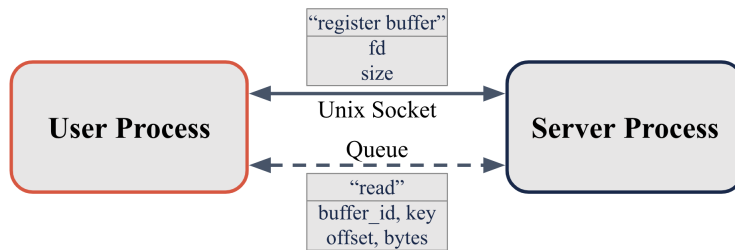


Figure 3.2: System Design Overview with Queue

3.3.1 Instantiation

A queue is set up using the same methods shown in Section 3.2, except both the client and server must cast the memory map result to type `Queue`, shown in Listing 3.10. Queues are differentiated with separate registration commands “`register_queue`” and “`unregister_queue`”, and are registered in a `queue_map` similar to the `buffer_map`.

3.3.2 Sample Implementation

The sample queue implementation, class `Queue`, is shown in Listing 3.10.

```
1 class Queue {
2     //Physical array of data elements
3     data_op data[QUEUE_SIZE];
```

```

4     //Atomically updated tickets for enqueueing and dequeuing
5     atomic<uint32_t> head_ticket, tail_ticket;
6     //Atomically updated physical queue holding ticket turns
7     atomic<uint32_t> pos_tickets[QUEUE_SIZE];
8     public:
9         void enqueue(data_op buf_info, response data...);
10        void dequeue(data_op *buf_info);
11 };

```

Listing 3.10: Queue struct

The queue holds elements of type `data_op`, exhibited in Listing 3.11.

```

1 struct data_op{
2     char op;                //read or write
3     unsigned buffer_id;    //id for registered buffer
4     uint64_t offset;       //offset into buffer
5     uint64_t bytes;        //number of bytes for read or write
6     uint64_t key;         //location of read or write data
7 };

```

Listing 3.11: `data_op` struct

The location of read and write data is an implementation choice for the server. It could come from a storage device, over the network, a filesystem, etc. The `buffer_id` should be a handle for a previously registered buffer, which the server will translate to perform the request, responding with an error message if needed.

In this sample implementation, on `enqueue()` or `dequeue()`, each thread will receive a “virtual ticket”, as shown Listing 3.12. For `enqueue`, the ticket is calculated by incrementing the tail (Line 2), and for `dequeue`, the ticket is calculated by incrementing the head (Line 18). The virtual ticket is then used to determine its corresponding turn and position in the physical tickets array. The turn is used to coordinate between the client and server actions on the data array. It can be one of the following: `enqueue`, `dequeue`, or `receive` response.

```

1 void Queue::enqueue(data_op buf_info, response data...) {
2     auto my_virtual_ticket = tail_ticket.fetch_add(1);
3     //set turn to enqueue state
4     auto turn = (my_virtual_ticket / QUEUE_SIZE) * 3;
5     auto pos = my_virtual_ticket % QUEUE_SIZE;
6     while (turn != pos_tickets[pos].load()); //wait for turn
7     data[pos] = buf_info;

```

```

8   pos_tickets[pos].fetch_add(1); //add 1 for dequeue state
9   //wait for receive response turn value
10  while ((turn + 2) != pos_tickets[pos].load());
11  /*get response data from data[pos]
12     ...
13  */
14  pos_tickets[pos].fetch_add(1);
15 }
16
17 void Queue::dequeue(data_op * buf_info) {
18     auto my_virtual_ticket = head_ticket.fetch_add(1);
19     //set turn to dequeue state
20     auto turn = ((my_virtual_ticket / QUEUE_SIZE) * 3) + 1;
21     auto pos = my_virtual_ticket % QUEUE_SIZE;
22     while (turn != pos_tickets[pos].load()); //wait for turn
23     *buf_info = data[pos];
24     /*set fields in data[pos] to indicate response
25        ...
26     */
27     pos_tickets[pos].fetch_add(1); //add 1 for receive
    response state
28 }

```

Listing 3.12: Code for enqueue and dequeue

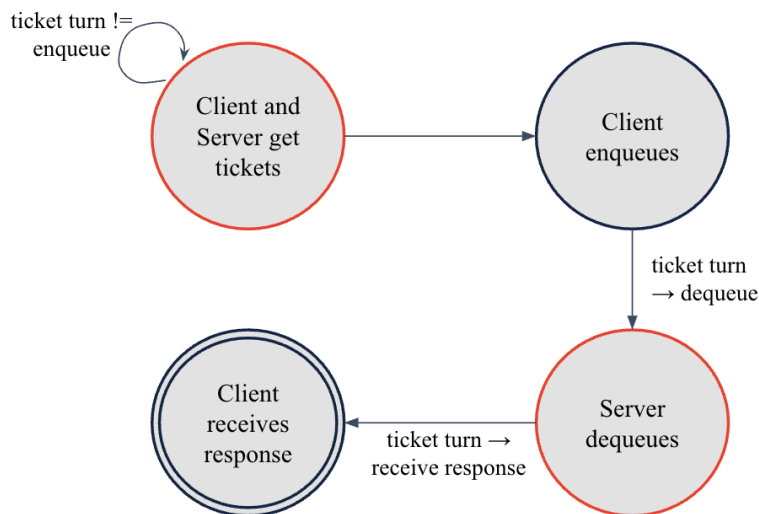


Figure 3.3: Queue State Machine

Figure 3.3 outlines the flow of the queue. When the client wants to enqueue a request, it must wait for the turn value in its entry in the physical tickets array

(`pos_tickets[pos]`) to be equal to its turn value (Line 6). It then increments `pos_tickets[pos]` to indicate that it is the turn of the server thread with the same virtual ticket to dequeue (Line 8), and waits for a response turn value in the same physical ticket array entry (Line 10). When the server is done dequeuing and handling the data operation, it increments `pos_tickets[pos]` (Line 27) to indicate that it is the turn of the corresponding client thread to receive the response (Line 11). This implementation ensures that every thread's request is handled in order and efficiently.

3.4 Putting it all Together

Listing 3.13 shows a simplified example of a client process utilizing the system design. The client creates a 4KB buffer to be shared with the trusted server to be mapped for DMA. It prepares (Lines 11-14) and sends (Line 17) a buffer registration request, and receives the buffer handle (Line 18-20). On buffer registration, the server can map the buffer for DMA with an I/O device as needed. Then, the client creates (Line 26) and registers (Line 29-35) a queue. Finally, it can use the newly registered queue to send a write request to write back data in the previously registered buffer (Line 49).

```
1 size_t buff_size = 4096;
2 size_t queue_size = 8192;
3 int resp;
4 int buff_fd, queue_fd;
5 int buff_handle;
6
7 // Create a buffer
8 /* Allocate shared buffer (Listing 3.2/3.6) to get handle */
9
10 //Fill in buffer registration request fields
11 struct register_req;
12 register_req.op = "R"; //Register buffer
13 register_req.fd = buff_fd;
14 register_req.size = buff_size;
15
16 //Send request and receive handle for buffer
17 write(socketfd, register_req, sizeof(register_req));
18 read(socketfd, resp, sizeof(resp));
19 if(resp < 0) printf("Buffer registration error");
```

```

20 else buff_handle = resp;
21
22 //Create a Queue
23 /* Allocate another shared buffer for queue */
24
25 //Cast buffer to Queue type
26 Queue *req_queue = new ((void *) ptr) Queue();
27
28 //Fill in queue registration request fields
29 struct register_req;
30 register_req.op = "QR"; //Register queue
31 register_req.fd = queue_fd;
32 register_req.size = queue_size;
33
34 //Send request and receive response
35 write(socketfd, register_req, sizeof(register_req));
36 read(socketfd, resp, sizeof(resp));
37 if(resp < 0) printf("Queue registration error");
38
39 //Fill in data operation request fields
40 struct data_op request;
41 request.op = "W";
42 request.buffer_id = buff_handle;
43 request.offset = 0;
44 request.bytes = sizeof(data_op);
45 request.key = request_key; //Source of write data
46
47 /* In client kernel */
48 //Enqueue write request
49 req_queue->enqueue(request);

```

Listing 3.13: User process queue example

CHAPTER 4

ANALYSIS

4.1 Shared Buffer

The simple use case of the shared memory that we have enabled is a buffer that is shared between processes. More specifically, this buffer can be a DMA buffer, meaning that higher performance memory can be shared between GPU processes and even be used for data transfers with I/O devices.

Because each buffer that is shared is registered in the `buffer_map`, they can also be used in read/write requests from the client. It is important to note that each forked server opens its corresponding client's shared memory and has its own copy of the `buffer_map`. Therefore, this design provides a security guarantee by means of isolation between processes. Each process can only make I/O requests through the trusted server, which determines if the requested handle is present in its isolated `buffer_map`. This effectively blocks external processes from performing DMA to or from locations in system memory that do not belong to them. Additionally, traditional virtual memory already prevents a client process from accessing another process's buffer, as it would not be in the client's address space. The trusted server also keeps all DMA addresses stored in the isolated `buffer_map`. As a result, it can securely map buffers for DMA, whilst not exposing the DMA address or the ability to program DMAs to any client process.

Further, the server validates all client requests. It responds with an error if a client specifies an unregistered buffer handle, as well as a size or offset that is out of range. This prevents access to not only a buffer's backing memory, but also to arbitrary system memory by means of an overflow attack. Consequently, the trusted server can validate requests, perform appropriate address translations, and securely program an I/O device as requested by the client.

As discussed in Section 2.4, GPU threads were previously only able to communicate with the CPU OS as a trusted entity. Now that we have enabled buffer registration, as well as the queue registration which will be analyzed in more detail in Section 4.2, we have created a new mechanism. TRUDI facilitates communication between GPU threads and a trusted GPU process, with all of the security benefits identified above.

4.2 Read/Write Request Queue

A use case of this shared memory method is the secure queue mechanism shown in Listing 3.12. Because queues are isolated between processes, no outside process can inject read or write requests to any registered buffer or queue that does not belong to them.

Figure 4.1 shows the performance of the queue implementation we presented in the last chapter. The experiment is set up with a queue of depth 512, each client thread submitting one queue request, and 512 server processing GPU threads running on an NVIDIA A100 GPU. As we increase the number of client threads, the throughput can reach a peak of 15 million requests per second. The significant drop in throughput after 2^{14} threads is likely attributed to improper inter and intra-warp contention management in our sample queue implementation. Since the queue serves as an example for the use of the shared buffer mechanism, we leave the performance tuning of the queue implementation for future work.

The queue is an example of how the central mechanism we have created can be used to enable an isolated, performant communication channel between user processes and the trusted server process for both CPU and GPU.

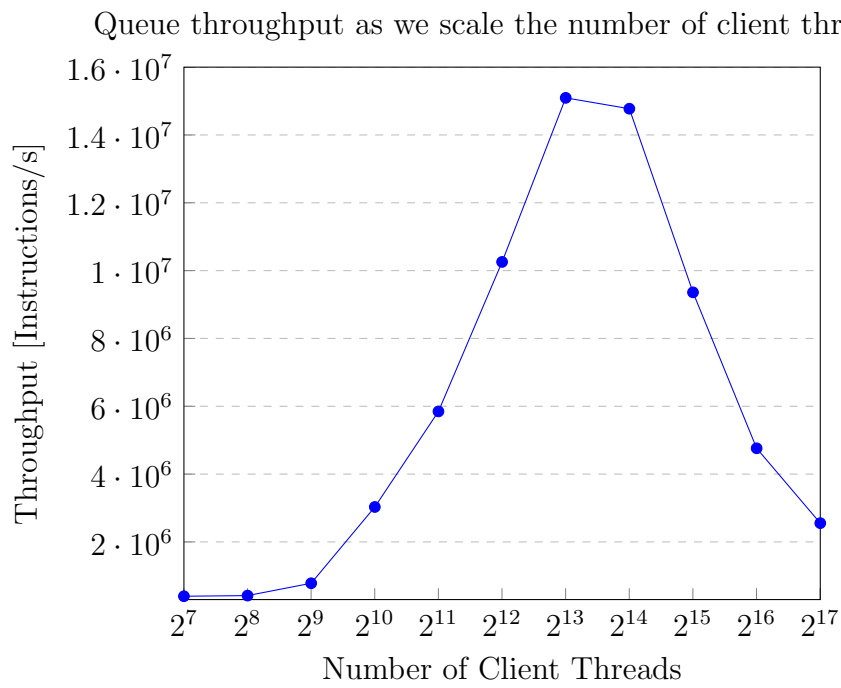


Figure 4.1: Scaling client threads with 512 server threads and 512 as queue size

CHAPTER 5

CONCLUSION

The system design that we have created achieves shared memory between a user application and a trusted server, such that a shared memory buffer can be registered by the server for future secure memory access. With the additional ability to create a queue from a shared buffer, one can build high-performance trusted services, even for accelerators like GPUs. This memory sharing method is novel in that two GPU-accelerated processes can communicate, in contrast to other methods, such as UVM, in which GPU threads can only communicate with a CPU OS. We showed how the buffer sharing mechanism guarantees security through process isolation, server validation of client requests, and concealment of DMA addresses. Case study analysis of the high-throughput queue shows how TRUDI enables the creation of a high-throughput, secure trusted process that can program I/O devices on behalf of a user application. Thus, userspace-initiated I/O can continue to be used for its performance benefits, without the risk of the fundamental DMA address vulnerability.

REFERENCES

- [1] J. Edge, “DMA and `get_user_pages()`,” 2018. [Online]. Available: <https://lwn.net/Articles/774411/>
- [2] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W. mei Hwu, “GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture,” in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, New York, NY, USA, Mar. 2023, pp. 325–339.
- [3] SPDK, “Direct Memory Access (DMA) from Userspace.” [Online]. Available: <https://spdk.io/doc/memory.html>
- [4] H. J. Koch, “Userspace I/O drivers in a realtime context,” unpublished.
- [5] Z. Liang, J. Lombardi, M. Chaarawi, and M. Hennecke, “DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory,” in Supercomputing Frontiers, vol. 12082, June 2020.
- [6] G. Sauthoff, “On the costs of syscalls,” 2021. [Online]. Available: <https://gms.tf/on-the-costs-of-syscalls.html>
- [7] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, “Designing a true direct-access file system with DevFS,” in Proceedings of the 16th USENIX Conference on File and Storage Technologies, Oakland, CA, USA, Feb. 2018, p. 241–255.
- [8] F. Daoud, A. Watad, and M. Silberstein, “GPUrdma: GPU-side library for high performance networking from GPU kernels,” Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, no. 6, p. 1–8, 2016.
- [9] CodeAurora, “Privilege Escalation Vulnerability in Graphics Driver (CVE 2016-2067),” 2016. [Online]. Available: <https://www.codeaurora.org/security-advisory/privilege-escalation-vulnerability-in-graphics-driver-cve-2016-2067>

- [10] J. Taft, “GPU Security Exposed,” Presentation, Black Hat Europe 2016, London, UK, Nov. 2016.
- [11] T. Allen and R. Ge, “In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, Nov. 2021.
- [12] Linux manual page, “socket(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/socket.2.html>
- [13] Linux manual page, “bind(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/bind.2.html>
- [14] Linux manual page, “listen(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/listen.2.html>
- [15] Linux manual page, “accept(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/accept.2.html>
- [16] Linux manual page, “connect(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/connect.2.html>
- [17] Linux manual page, “fork(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/fork.2.html>
- [18] The Linux Kernel Archives, “Buffer Sharing and Synchronization (dma-buf).” [Online]. Available: <https://docs.kernel.org/driver-api/dma-buf.html>
- [19] Linux manual page, “memfd(2),” 2019. [Online]. Available: https://man7.org/linux/man-pages/man2/memfd_create.2.html
- [20] J. Corbet, “Sealed files,” 2014. [Online]. Available: <https://lwn.net/Articles/593918/>
- [21] Linux manual page, “fcntl(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/fcntl.2.html>
- [22] Linux manual page, “ioctl(2),” 1979. [Online]. Available: <https://man7.org/linux/man-pages/man2/ioctl.2.html>
- [23] Linux manual page, “mmap(2),” 2001. [Online]. Available: <https://man7.org/linux/man-pages/man2/mmap.2.html>
- [24] G. Hoffmann, “Add udmabuf misc device,” 2018. [Online]. Available: <https://lwn.net/Articles/758903/>

- [25] QEMU, “virtio-gpu-udmabuf.c,” 2022. [Online]. Available: <https://github.com/qemu/qemu/blob/a95260486aa7e78d7c7194eba65cf03311ad94ad/hw/display/virtio-gpu-udmabuf.c>
- [26] QEMU, “udmabuf.c,” 2021. [Online]. Available: <https://github.com/qemu/qemu/blob/a95260486aa7e78d7c7194eba65cf03311ad94ad/ui/udmabuf.c>
- [27] QEMU, “udmabuf.h,” 2021. [Online]. Available: <https://github.com/qemu/qemu/blob/d451e32ce8e1eef1b7d05d9f532113e9618f1fc1/include/standard-headers/linux/udmabuf.h>
- [28] C. Sridharan, “Seamless file descriptor transfer between processes with pidfd and pidfd_getfd,” 2021. [Online]. Available: <https://copyconstruct.medium.com/seamless-file-descriptor-transfer-between-processes-with-pidfd-and-pidfd-getfd-816afcd19ed4>
- [29] Linux manual page, “pidfd_getfd(2),” 2020. [Online]. Available: https://man7.org/linux/man-pages/man2/pidfd_getfd.2.html
- [30] Linux manual page, “getsockopt(2),” 2008. [Online]. Available: <https://man7.org/linux/man-pages/man2/setsockopt.2.html>
- [31] C. D. API, “cuMemGetHandleForAddressRange,” 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MEM.html#group__CUDA__MEM_1g51e719462c04ee90a6b0f8b2a75fe031
- [32] C. D. API, “Virtual memory management,” 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html
- [33] Blue Frost Security, “CVE-2023-2008 - Analyzing and exploiting a bug in the udmabuf driver,” 2023. [Online]. Available: <https://labs.bluefrostsecurity.de/blog/cve-2023-2008.html>
- [34] M. Almasri, A. Masood, E. Richter, and K. Wu, “High-Throughput Multi-Producer Multi-Consumer Queues on GPUs,” unpublished.
- [35] J. Corbet, “Security requirements for new kernel features,” 2022. [Online]. Available: <https://lwn.net/Articles/902466/>