Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Project

Systems Group, Department of Computer Science, ETH Zurich

FPGA Acceleration for Storage Deduplication

by

Jiayong Li

Supervised by

Zhenhao He, Runbin Shi, Prof. Gustavo Alonso

April 2023 – July 2023

Informatik
Computer Science

Abstract

Data deduplication is crucial for optimizing storage resources and saving network bandwidth by identifying and eliminating duplicate data segments. In this project, we design and implement SSD deduplication logic on Field-Programmable Gate Arrays (FPGAs). FPGAs offer customizable hardware acceleration, making them a compelling platform for enhancing deduplication performance.

Our proposed FPGA-based deduplication algorithm employs SHA3-256 hashing as a unique page identifier and a Bloom-filter-accelerated hash table as an indexing mechanism to identify duplicate data chunks. The algorithm is implemented completely in hardware with high-degree parallelism by leveraging the FPGA's reconfigurability. As a result, we achieved a 10x improvement in latency compared to traditional software-based methods. And the throughput of the deduplication system can reach 12.7 GB/s (100 Gbps).

Acknowledgements

I would like to express my gratitude to Professor Gustavo Alonso and the Systems Group at ETH Zurich for providing me with the opportunity to undertake this Semester Project. This experience has been invaluable in expanding my understanding of the subject matter and fostering my academic growth.

I am deeply thankful to my dedicated and insightful supervisors, Zhenhao He and Runbin Shi, for their unwavering guidance, expert advice, and continuous support throughout this project. Their expertise and mentorship have been instrumental in shaping the direction and execution of my work.

I would also like to thank the Systems Group team for creating an intellectually stimulating and collaborative environment facilitating meaningful discussions and exchanging ideas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The exponential growth of data in various fields has underscored the need for efficient data storage and management techniques due to the relatively higher cost of Solid State Drives (SSDs) and their limited lifetime [1, 2]. As a result, data deduplication has emerged as a crucial strategy to mitigate these problems by identifying and eliminating duplicate data segments written to the flash, thereby optimizing storage utilization and extending the operational lifetime of flash memories.

Many software-based deduplication systems have been built and demonstrated significant space saving [3,4]. However, with the escalating demands for real-time data processing and enhanced system performance, there arises a need for innovative and accelerated deduplication methods [5]. This is where Field-Programmable Gate Arrays (FPGAs) come into play. FPGAs offer parallel processing and hardware customization capabilities, making them an attractive platform for accelerating computation-intensive tasks.

Nevertheless, existing FPGA-based approaches only use FPGAs to accelerate hash computations. Either assume a close integration of FPGAs within SSDs [6] or simply use FPGA as an offload engine which needs back-and-forth communications between host and FPGA [5]. Since most commercial SSDs do not have an FPGA integrated inside, those approaches either restrict deduplication enhancements to customized SSDs or prevent the exploitation of FPGAs' full potential.

Our work addresses these limitations by moving the deduplication logic to an independent abstraction layer between the host and SSD. This layer is dedicated to deduplication and handles everything related to it: fingerprint generation, fingerprint management, address translation, and garbage collection. With this abstraction layer, neither the host nor the SSD is aware of the existence of deduplication logic. Therefore, this is a drop-in solution for deduplication, as neither the host nor SSD needs to be modified.

Leveraging FPGA's reconfigurability, this layer is implemented completely in hardware. Our prototype reached 12.7 GB/s throughput for write requests and less than 30 μs latency for all write, read, and erase requests.

1.2 Our Contributions

1.2.1 Pure Hardware Implementation

A central contribution of our project is the realization of a purely hardware-based data deduplication algorithm. We have developed a dedicated hardware architecture for deduplication.

1.2.2 Independent Abstraction Layer

We introduce an independent abstraction layer encapsulating all necessary functions for seamless interaction with the normal host and SSD. This modular design enables easy integration into existing storage infrastructures.

1.2.3 Bloom-filter-enhanced Hash Table Design

Our project introduces a Bloom filter design that is able to handle deletion without space overhead to accelerate hash table lookup. We implement a per-bucket Bloom filter in the hash table. The Bloom filter will be reconstructed once the linked list lookup goes to the end. This is effectively a deletion mechanism without going to counting Bloom filter with memory space overhead.

1.2.4 Prototyping

Our prototype system is implemented on Xilinx Alveo u55c board and demonstrated remarkable performance metrics.

Chapter 2

Related Work

Data deduplication has garnered considerable attention, resulting in many software-based implementations and a growing interest in hardware acceleration. This section reviews relevant literature and highlights key aspects of existing approaches. We categorize the discussed papers into two groups: software-based implementations [7–15] and FPGA-accelerated methods [5, 6].

2.1 Software-based Implementations

Papers [7–15] present various software-based data deduplication techniques. These approaches primarily focus on optimizing duplicate data identification and storage efficiency within the Flash Translation Layer (FTL) of SSDs. While these works contribute valuable insights into the challenges and strategies for data deduplication, software-based systems do not scale with the increasing performance of SSD arrays [5].

2.2 FPGA-accelerated Methods

Papers [5,6] introduce FPGA to accelerate hash calculation, showcasing the potential of hardware platforms in enhancing specific deduplication components. [6] integrates hardware accelerators (FPGA for MD5) within the SSD and builds the rest of the deduplication system in FTL software. [5] offload hash calculation and page compression-decompression to FPGA. The rest of the deduplication system (fingerprint and address mapping management) remains in software implementation on the CPU side.

2.3 Garbage Collection Mechanism

Once a page is not referenced by any LBA, this page needs to be deleted from the flash. Only CAFTL [7] and DRACO [12] explicitly say how this garbage

collection is done in their systems. CAFTL uses two-level indirect mapping for the addresses and maintains each page's reference count in memory. Only the pages with no references can be recycled by the garbage collector. DRACO maintains a bit map that indicates if the page is referenced. They periodically check the address mapping and mark the invalid pages, and the in-place garbage collector will delete the unreferenced pages later.

2.4 Our Approach

In contrast to the aforementioned studies, our work represents a distinctive contribution in several key aspects. Firstly, our design pushes the hardware acceleration further by offering a complete hardware solution for data deduplication. The FPGA-based architecture ensures accelerated processing of deduplication tasks, resulting in enhanced efficiency and reduced processing times.

Secondly, our approach introduces a dedicated abstraction layer between the host and SSD. Our FPGA and driver provide a normal SSD interface for the host and do not need any modifications on the SSD side. This abstraction layer not only contains the necessary hash computation and fingerprint management path for the deduplication system but is also capable of handling instructions and mapping host LBA to the unique page LBA.

This abstraction layer also has its own garbage collection mechanism. Similar to CAFTL, our design employs a reference counter and in-line garbage collection. But we separate GC introduced by deduplication and original GC inside SSD. The reference counter is stored together with the in-memory address mapping table on FPGA and will update on each operation(write/erase). Once the reference counter of the corresponding page goes to zero, FPGA will issue an erase instruction to SSD. This process is managed purely by hardware, which minimizes the impact of deduplication on overall system performance and ensures consistently optimal resource utilization.

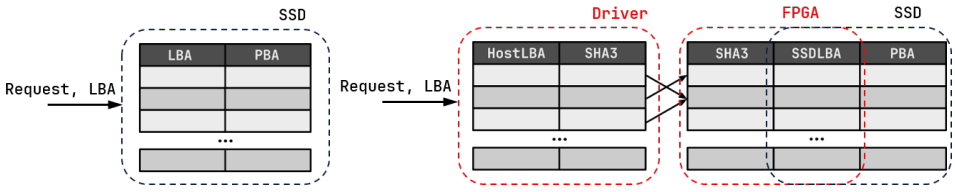
In summary, while prior works have explored both software-based and FPGA-accelerated deduplication techniques, our project distinguishes itself through a holistic hardware-based design encompassing the entire deduplication process. By providing an independent abstraction layer that contains everything needed for deduplication and is implemented in pure hardware, our approach offers a novel perspective on optimizing data deduplication for efficient storage management.

Chapter 3

Algorithm

3.1 Address Translation

In normal SSD, data is stored in fixed-sized pages and they are addressed by their Logical Block Address (LBA) from the host. Once a request comes, FTL will translate this address to Physical Block Address (PBA). PBA is the actual address of the page stored in flash memory.



(a) Normal SSD (b) Deduplication with SSD

Figure 3.1: Address Translation in Normal SSD and Deduplication SSD

To implement deduplication in SSD, we need to insert another layer that translates the page address to the unique page address. In our system, when a request comes, the driver first lookup which unique page the host LBA corresponds to in the SSD, and the FPGA then lookup which LBA that page is stored in the SSD. This address translation process is shown in Figure 3.1. The host and SSD are unaware of this layer between them and operate normally.

3.2 Computation Handling

3.2.1 Chunking

The first step of the algorithm is to chunk the input data stream into either fixed-sized or variable-sized chunks. In our system, we use 4kiB fixed-sized

chunks.

3.2.2 Fingerprinting

Traditionally, data reduction is done by compression algorithms such as LZ77/LZ88 [16, 17]. These algorithms identify redundancy for short strings by first computing a weak hash and then comparing hash-matched strings byte-to-byte. Due to their space and time complexity, they are not widely used in large-scale storage deduplication [1].

In chunk-level deduplication systems, unique pages are characterized by their cryptographically secure hash signature (in our case, SHA3-256). Since the hash collision probability of SHA-256 is much smaller than the hard disk drive error in ZB and YB scale [1], we can safely regard this hash as a collision-free function and use it as a unique fingerprint.

3.2.3 Indexing

On the FPGA, we are given SHA3-256 of the unique pages, and we need to maintain a translation between this hash value to the SSD LBA and the reference counter. As shown in Figure 3.2, this is done by a hash table.

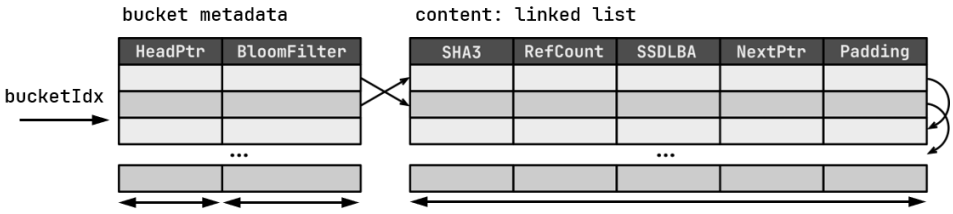


Figure 3.2: Hash Table

We use the last 25 bits of SHA3-256 value as the bucket index, and each bucket contains a linked list. We always keep the average length of the linked list = 8. Different storage size corresponds to different bucket count. Each bucket has a 32-bit Bloom filter to accelerate this linked list lookup. The Bloom filter uses the first 15 bits of SHA3-256 as three 5-bit hash functions(k=3).

3.2.4 Metadata Management

As shown in Figure 3.1, we need to manage two metadata tables. The driver maintains the first table. It contains the mapping from the host LBA to unique pages. The second table is maintained on FPGA as an in-memory table. It contains information on unique pages: reference counter and where it is stored in SSD (SSD LBA). On each update, the corresponding part in the second table will also be sent to SSD as part of the page header.

3.3 Instruction Handling

This section introduces how write, erase, and read instructions are treated in the driver and FPGA. Figure 3.3 gives an overall view of instruction handling.

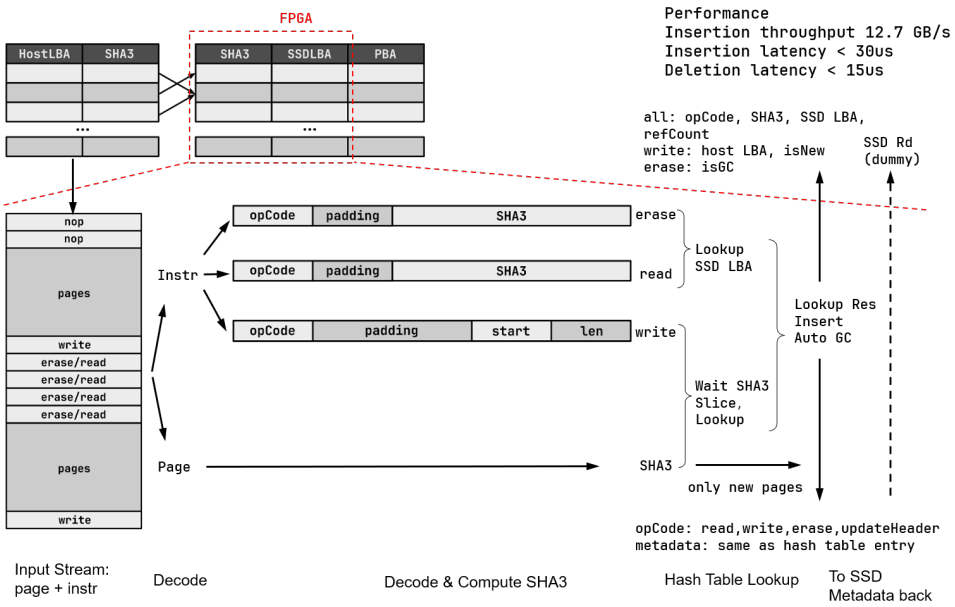


Figure 3.3: Instruction Handling

3.3.1 Read Instruction

For read instruction, the host gives a start LBA and length. Then, the driver will do a lookup for the SHA3 values. Those SHA3 values are sent to FPGA to look up their LBA seen by SSD. After that, SSD will transfer back to the corresponding pages.

3.3.2 Erase Instruction

For erase instruction, the host gives a start LBA and length. Then, the driver will do a lookup for the SHA3 values. Those SHA3 values are sent to FPGA to look up their LBA seen by SSD and decrease the reference counter by one. SSD will receive a new reference count and update header instruction if the reference count is not zero. If the reference count is zero after deletion, SSD will receive an erase instruction on the corresponding page.

3.3.3 Write Instruction

For write instruction, the host will give a start LBA, length, and the data to write. First of all, the driver will check if all the places are free. If not, the driver will issue an erase instruction first. From the FPGA side, all write instructions are written to free blocks, and only newly inserted pages' reference counters need to be updated. Similar to erase, FPGA will tell SSD to insert a page if it is new and only tell SSD to update the header if the page already exists.

3.3.4 Dependencies Between Instructions

If there is no GC mechanism in page deletion, there are no dependencies between write, erase, and read instructions since they are only plus one or minus one in the reference counters. As a result, dependencies only exist between read and erase when the erase instruction triggers GC(erase a page that is only referenced by one place). This can be easily avoided in the driver software, so our hardware implementation has no dependency checks.

Chapter 4

Hardware Architecture

In the previous chapter, we described our basic algorithm for deduplication and how it works with different instructions. This chapter will show how each part of the algorithm is implemented in hardware.

The key component of our system is the DedupCore. Its overall hardware architecture is shown in Figure 4.2. DedupCore communicates with the host via Coyote [18] shell. Coyote handles basic communications with the host, and requests come to DedupCore through DMA Engine. DedupCore will send responses (metadata and read pages) back to Coyote in the end.

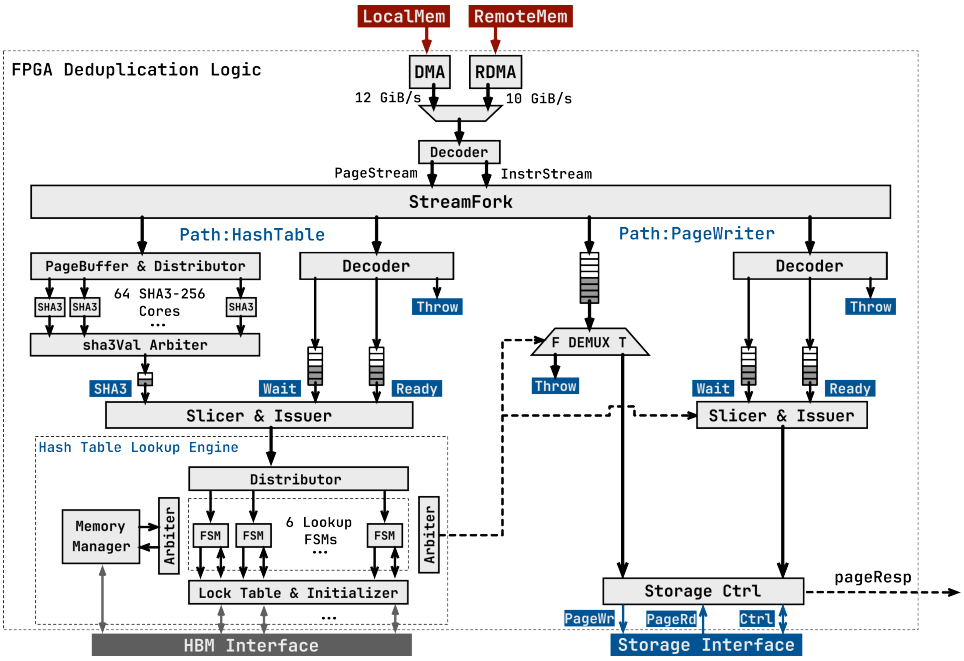


Figure 4.1: FPGA Deduplication Logic

When a request stream comes, the first step is to separate it into an

instruction stream and a data stream. After that, each stream is forked into two identical streams: one goes to the hash table path, and the other one goes to the page writer path. The hash table path is responsible for hash computation and metadata management. Page writer first buffers input pages and instructions to wait for results from the hash table and then decide what to do to the pages and instructions.

4.1 Hash Table Path

The hash table path consists of the SHA3 core group, instruction decoding logic, instruction slicing and issuing logic, and hash table lookup engine.

4.1.1 SHA3 Core Group

The whole DedupCore is written in Scala with SpinalHDL [19], and the SHA3 cores are used from SpinalCrypto [20]. In the RTL simulation, one SHA3 core needs around 3360 clock cycles to calculate the SHA3-256 value for a 4 kiB page. The input data width is 512b, corresponding to 64 cycles per page. To match the SHA3 core's throughput to the input throughput, we use 64 SHA3 cores to compute the hash in parallel.

4.1.2 Instruction Decoder

The decoder will decode the type of instruction (write/erase/read/nop) and decide which queue the instruction should go to. In the hash table, the write instruction needs to wait for the SHA3 value calculation. The erase and read instructions are ready to be issued to the hash table since they already have SHA3 values. Nop will be thrown away.

4.1.3 Instruction Slicer and Issuer

The issuer's job is to decide which instruction to issue to the lookup engine. For write instruction, it needs to be sliced and wait for SHA3 value before being sent to hash table lookup. In the current implementation, there are no complicated strategies inside the issuer. The instruction order seen by the hash table is the same as the input.

4.1.4 Lookup Engine

Lookup Engine receives an opcode and a SHA3. It will look up this unique page in the hash table and update the metadata depending on the instruction. If a page's reference count goes to zero, this page will be deleted from the hash table.

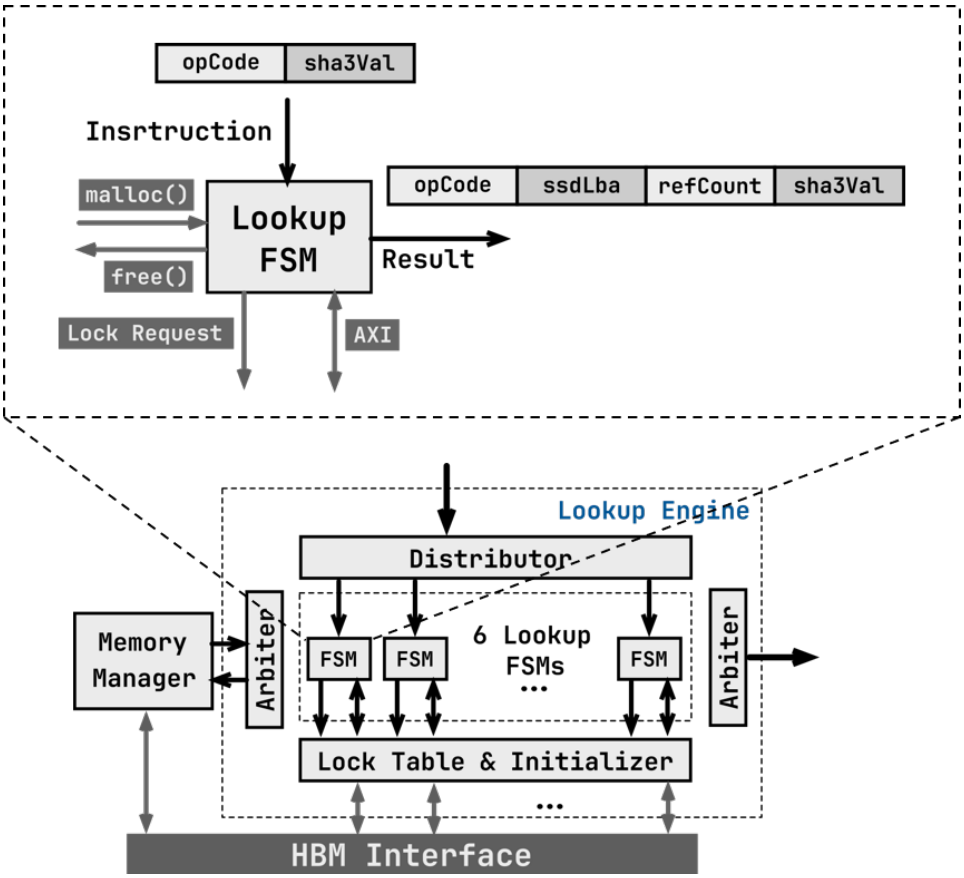


Figure 4.2: Lookup Engine and Lookup FSM

Lookup Engine will look up the hash table and tell the page writer where the page was/will be stored, the current reference counts, and the original instruction. Memory Manager will dynamically manage memory and SSD space and provide the Lookup Engine with necessary functionalities (malloc and free).

Lookup FSM

When an instruction comes, it will be dispatched to 1 of 6 FSMs. The last 25 bits of SHA3 value will be used as bucket index in the hash table, and the first 15 bits will be used as three 5-bit hash values for each bucket's Bloom filter. This is shown in Figure 4.3.

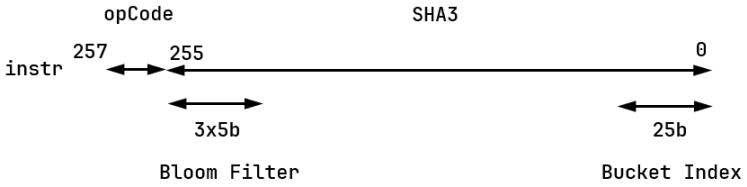


Figure 4.3: Bucket Index and Bloom Filter Hash

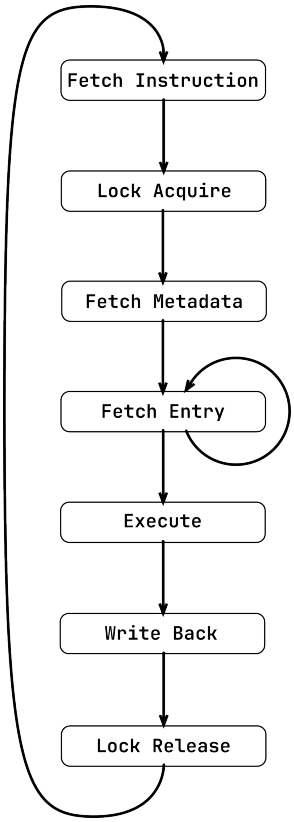


Figure 4.4: State Transfer Diagram for Lookup FSM

Figure 4.4 shows the state transfer diagram for each state machine. After fetching the next instruction from the distributor, state machines will communicate with the Lock Manager to acquire a lock on the corresponding

bucket. Once the lock is acquired, state machines will go to the memory to fetch bucket metadata and each element in the linked list to see if there is a match.

State machines will go to the Execute state if there is a match or they reach the end of the linked list. Depending on the lookup results and instructions, the state machine will execute different operations on the current entry, previous entry, and bucket metadata. Those operations are summarized in Table 4.1. In the end, the state machine will release the lock in the Lock Manager and output all metadata related to the current instruction.

Table 4.1: State Machine Operations for Each Instruction at Execute and Write Back State

	found	not found
write	increase reference count	insert new entry with reference 1 at head
	update current entry	write back new entry update metadata
erase	decrease reference count if reference = 0: GC (delete entry), modify previous entry/metadata.	illegal
	if not GC: update current entry. if GC: update previous entry/metadata	
read	read metadata	illegal
	no write back	

Bloom Filter

We must go to the end of a linked list before we can claim that the current page is new. To accelerate new page lookup, we employ a Bloom filter in each bucket.

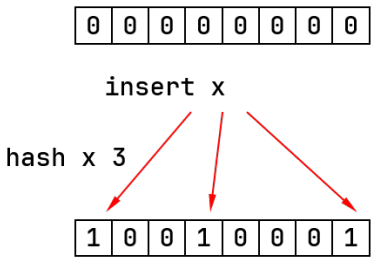


Figure 4.5: Principle of Bloom Filter

Bloom filter is a data structure that can tell the element definitely does not exist or probably exits in a set. Figure 4.5 shows the basic principle of the Bloom filter. When a data item is inserted into the filter, its corresponding hash values are employed to set the corresponding bits in the array. In our design, each bucket employs a 32-bit Bloom filter with 3 5-bit hash functions(first 15 bits in SHA3-256 value). During lookup, Bloom filter will return true (exist in the set) if all bits are already 1. There are no false negatives, but false positives (FP) can occur due to hash collisions.

Deletion is not possible in the Bloom filter. To implement deletion instruction in the Bloom filter, a general approach is to use the counting Bloom filter. Instead of using 1 bit per entry, Counting Bloom filters implement a counter in each entry. The counter will increase when insertion and decrease in deletion.

The drawbacks of counting Bloom filters are also obvious: 1. Space overhead: counting Bloom filters uses significantly larger memory spaces. For example, a 16-bit counting Bloom filter uses 16x space in memory compared to a normal Bloom filter. 2. Control overhead: We must ensure the element we want to delete exists in the set. When a counter overflows, we cannot modify this entry anymore.

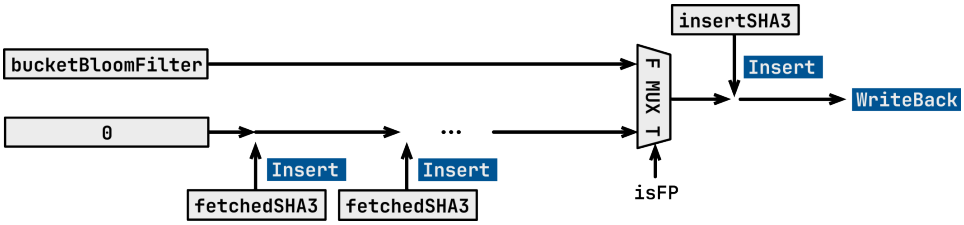


Figure 4.6: Principle of Bloom Filter Reconstruction

Our system employs a per-bucket normal Bloom filter design with false positive detection and reconstruction logic. This design harnesses the inherent properties of Bloom filters, allowing us to detect new elements quickly and bypass the linked list lookup. Although deletion is not possible in a normal Bloom filter, this design allows us to reconstruct the whole Bloom filter every time FP happens.

Figure 4.6 shows the hardware architecture of Bloom filter reconstruction. Besides the Bloom filter read from metadata, our state machine will recalculate a new Bloom filter during the hash table lookup. FP means when a new element comes, the Bloom filter says it exists in the linked list instead of saying it is new. This means the FSM will go to the end of the linked list, and we will write back the newly constructed Bloom filter to DRAM. This reconstruction mechanism is effectively a deletion operation without introducing space and control overhead.

Lock Manager

To guarantee the correctness of Lookup FSMs' parallel execution, we employ a Lock Manager in our system. Figure 4.7 shows its hardware architecture.

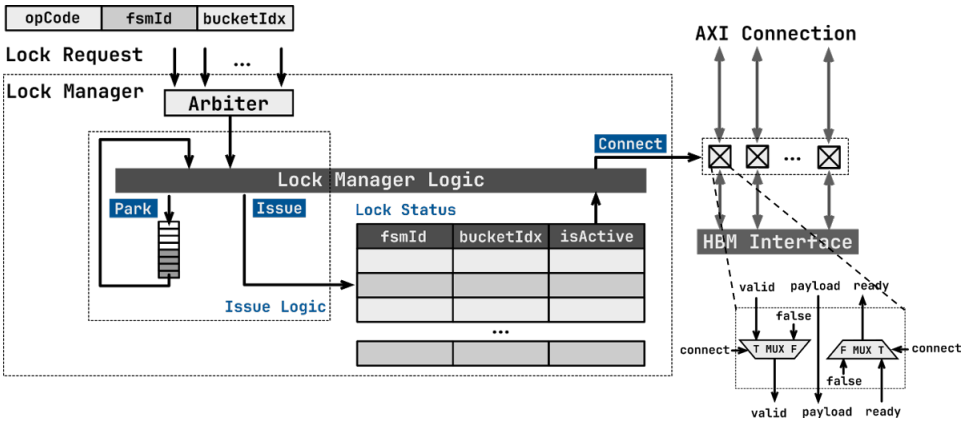


Figure 4.7: Lock Manager

We use (strict) two-phase locking (2PL) in our system. Since each FSM will lookup the linked list in each bucket and write back bucket metadata and some entries, this locking mechanism is equivalent to granting the FSM an exclusive lock on the whole bucket. This is why the state machine enters only once lock acquire/release state every lookup. On the lock manager side, it only accepts state machine ID and bucket index as lock requests.

Another design principle is to minimize the communication overhead between the Lock Manager and Lookup FSMs. Instead of sending a grant or a wait response back to the state machines, Lock Manager will directly control the AXI connection between FSM and DRAM. Only FSMs with active locks are allowed to connect to DRAM.

When a new lock request comes, the Lock Manager will check if other FSMs lock the bucket. If not, the new request will be registered as active in the lock status table. Otherwise, this request will go to the Parking Queue. The request at the head of the Parking Queue will be checked every cycle to see if it is possible to grant a lock to it, and it always has priority compared to new requests.

Memory Manager

Memory Manager is a unit that manages free space in both SSD and DRAM. There is a one-to-one correspondence between hash table entries in DRAM and unique page blocks in SSD. Therefore, we can manage both of them by one unit if we force them to have the same layout.

Our Memory Manager is a huge FIFO that contains all free block addresses. We need one 32-bit block address for one 512-bit hash table entry. This means the space used by this free index list is $32/512 = 6\%$ of the DRAM. This is not a big overhead to DRAM space, but we cannot keep all free indexes on-chip. As shown in Figure 4.8, conceptually, our Memory Manager is a big FIFO go through DRAM.

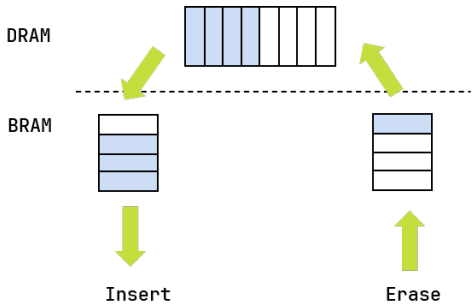


Figure 4.8: Free Index List

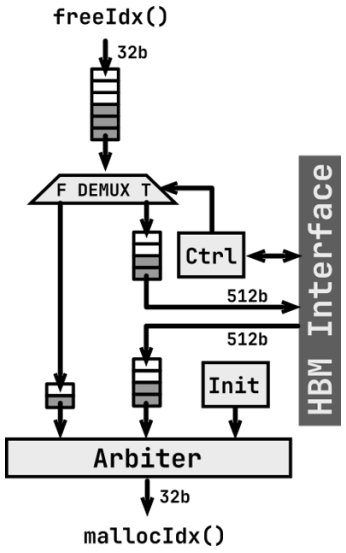


Figure 4.9: Memory Manager

The architecture of this unit is shown in Figure 4.9. After reset, new indexes are generated by the Init Counter. When an FSM frees a memory position, Memory Manager will first try to buffer the index in an on-chip buffer. If too many indexes are freed up, those indexes will be sent to DRAM. We use double buffering and prefetching when communicating with DRAM to ensure we are always ready to accept the freed index and malloc new index.

4.2 Page Writer Path

The first part of the Page Writer is some buffers for input pages and instructions. Pages are buffered and waiting for results from the hash table. Only new pages will be sent to the storage controller. Instructions are classified and buffered. The hash table will send the metadata to the page writer. The storage controller's job is to react differently to the instructions depending on the metadata. For write instructions, the page will be written to storage if the reference count is 1. For erase instructions, the page in SSD will be deleted if the reference count is 0. Read instructions will always be sent to SSD. Otherwise, SSD will only receive the update header instruction.

For the page header, we keep it the same as the hash table entry. The new page header is always sent to SSD.

Chapter 5

Results

This chapter will show the workload generation and the performance results from experiments.

5.1 Workload

Figure 5.1 shows the hash table in our experiments. We use 32768 buckets (reminder: average length of linked list = 8) corresponding to 1 GiB SSD space in our prototype.

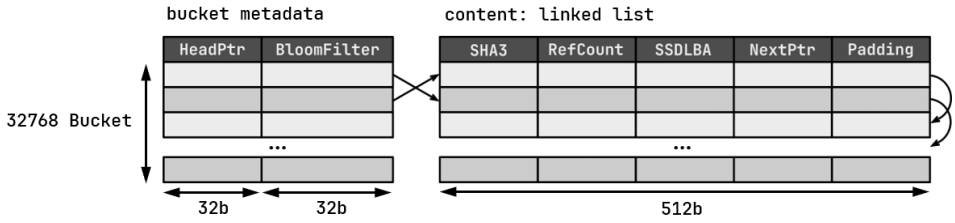


Figure 5.1: Hash Table Bucket Count in Experiments

While the SHA3 core group's throughput is constant, the Lookup Engine's throughput highly depends on the workload. The following two factors may affect the throughput:

1. Hash table fullness. More items inserted in the hash table means a longer linked list we need to lookup. We would like to know how our performance is affected by this.

2. Ratio of new pages. While we assume old pages randomly occur in the linked list, we always need to go to the end of a linked list when we lookup for the new pages. We want to know how our Bloom filter can accelerate the new page lookup.

5.2 Throughput vs. #FSM

Figure 5.2 shows the system's insertion and deletion throughput when the hash table is full. The throughput is obtained by randomly inserting/deleting 16384 old pages in one request batch.

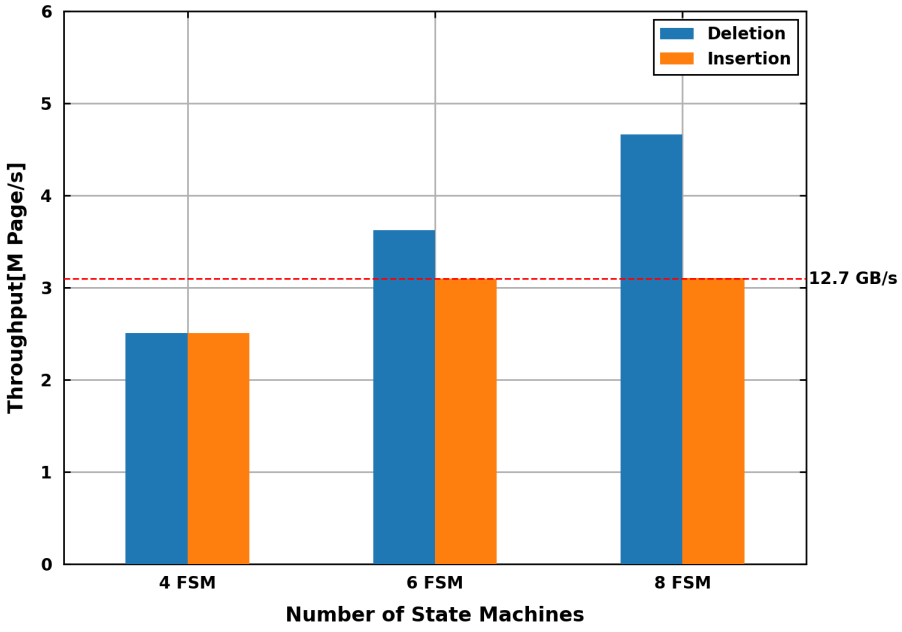
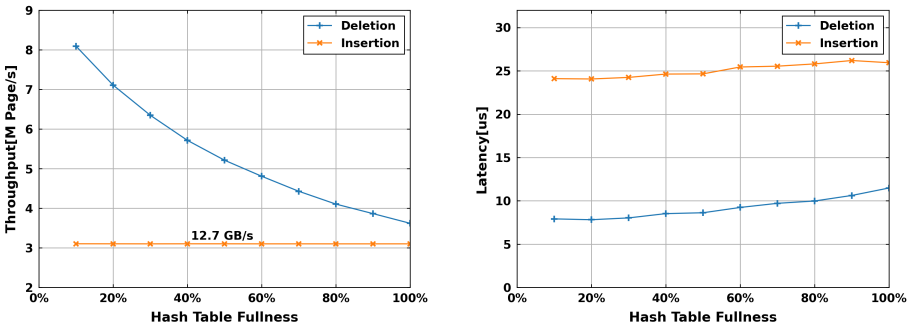


Figure 5.2: Throughput vs. #FSM when Hash Table is 100% Full

When using 4FSM, our system's throughput is limited by the hash table lookup. For 6 and 8 FSMs, While deletion throughput goes higher, insertion throughput remains constant as it is limited by the DMA throughput. We need at least 6 FSMs for hash table lookup to saturate DMA throughput.

5.3 Throughput and Latency vs. Hash Table Fullness when Using 6 FSMs

Figure 5.3 shows how throughput and latency vary with hash table fullness when we use 6 FSM and requests only contain old pages. Throughput is obtained by inserting/deleting 16384 random old pages in one request batch. Latency is obtained by inserting/deleting 16 random old pages in one request batch(FPGA will respond to the host in batches of 16 requests in our system).



(a) Throughput (b) Latency

Figure 5.3: Performance vs. Hash Table Fullness for 6 FSM

The higher the fullness, the longer the linked list we need to lookup. That is why the deletion throughput decreases with the increase of the hash table fullness. For the insertion case, we are always limited by the DMA throughput. In the latency case, with a longer linked list lookup, we have an increase in latency. The difference between insertion and deletion latency is $\sim 15\mu$s. This is introduced by SHA3-256 hash value calculation (3360 cycles).

5.4 Worst Case Throughput: Effect of Bloom Filter

To show the effect of the Bloom filter, Figure 5.4 gives the worst-case insertion throughput. Worst-case means all the lookups must go to the linked list's end. Here, we first insert random pages into the system until the hash table only has 16384 entries available. Then, 16384 new pages are inserted into the system.

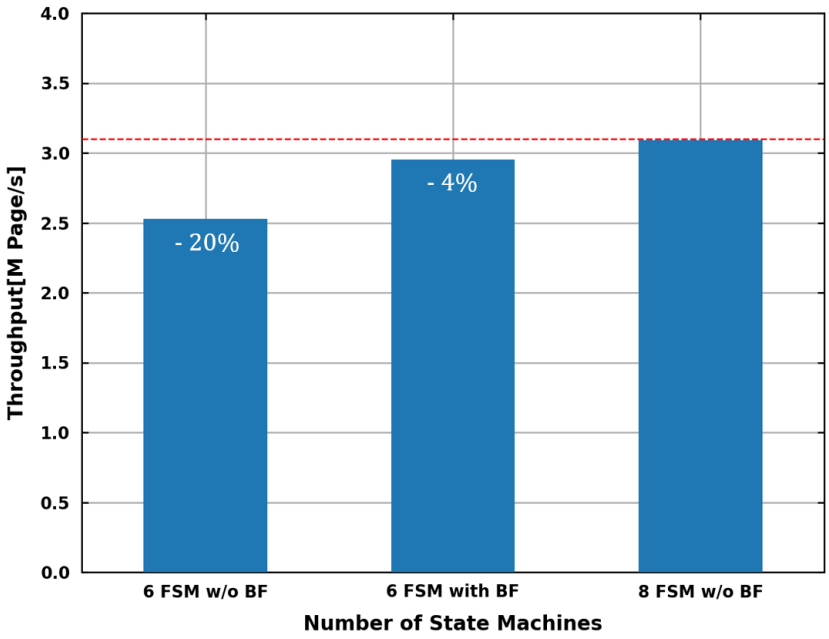


Figure 5.4: Effect of the Bloom Filter

There is no throughput drop for the 8 FSM case since we have enough compute resources. For normal 6 FSM (no Bloom filter) cases, there is a 20% throughput drop. Introducing the Bloom filter alleviates this to only a 4% throughput drop.

5.5 Throughput vs. Deduplication Percentage

Figure 5.5 shows our system's throughput in different deduplication percentages. The deduplication percentage means the ratio of old pages in the coming write requests. 0% means all pages are new and 100% means all pages are old page. The hash table is inserted only to have 16384 entries

available to show the results better. In this case, 0% deduplication percentage corresponds to the worst case scenario from the previous section.

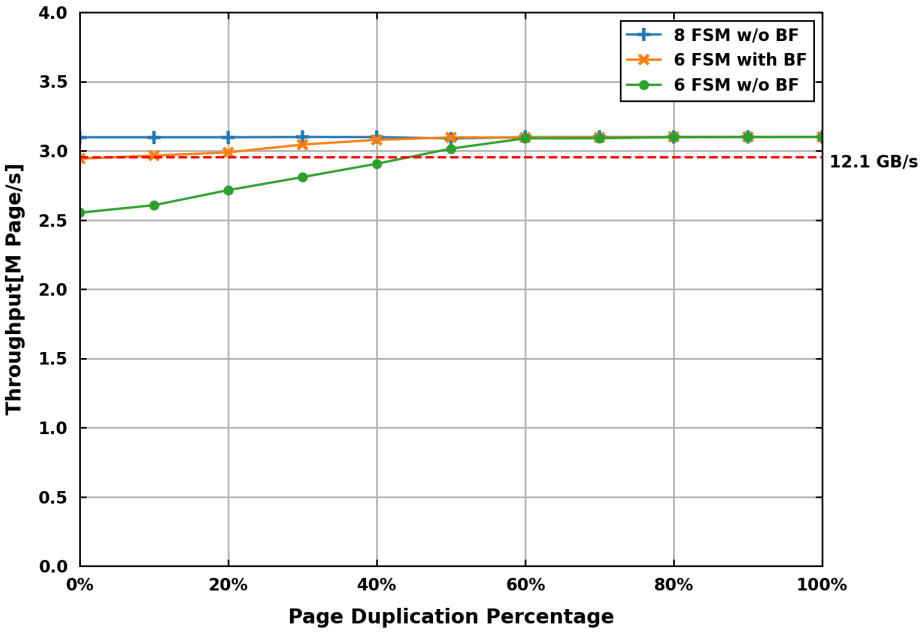


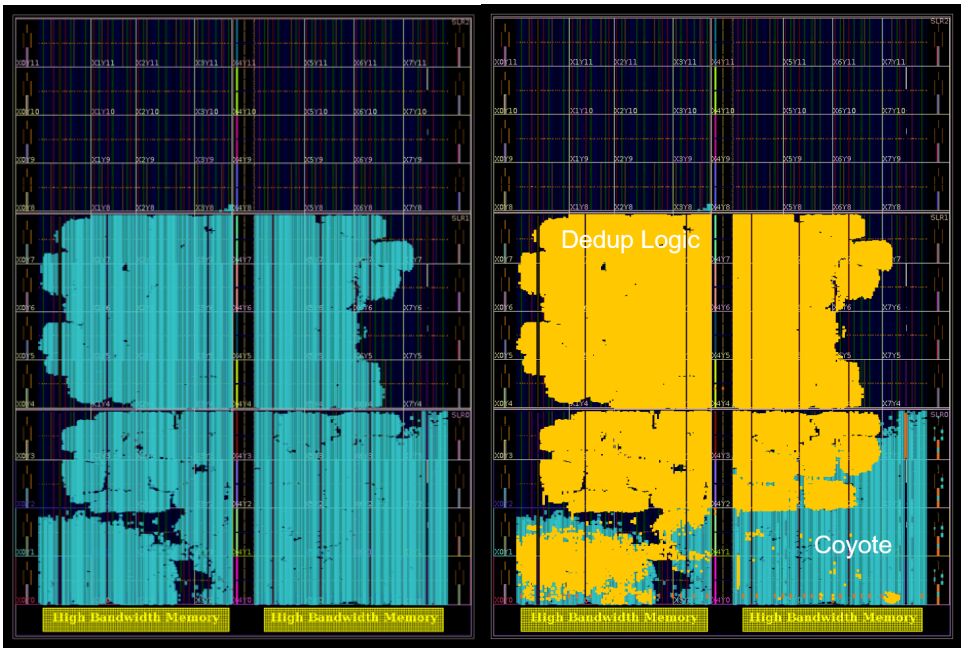
Figure 5.5: Throughput vs. Deduplication Percentage

While throughput is a constant in the 8 FSM case, there is a drop when less than 60% of the coming pages are old in the 6 FSM case without Bloom filter. With Bloom filter, the drop only visible when deduplication percentage is less than 40% and still can reach 12.1 GB/s in the worst case.

Chapter 6

FPGA Floorplan

6.1 Floorplan and Resource Utilization of the Whole System



(a) (b)

Figure 6.1: FPGA Floorplan of the Whole System

Figure 6.1 shows the FPGA floorplan, and resource utilization is shown in Figure 6.2. The whole system uses 42.3% CLB and 16.3% BRAM resources on the FPGA.

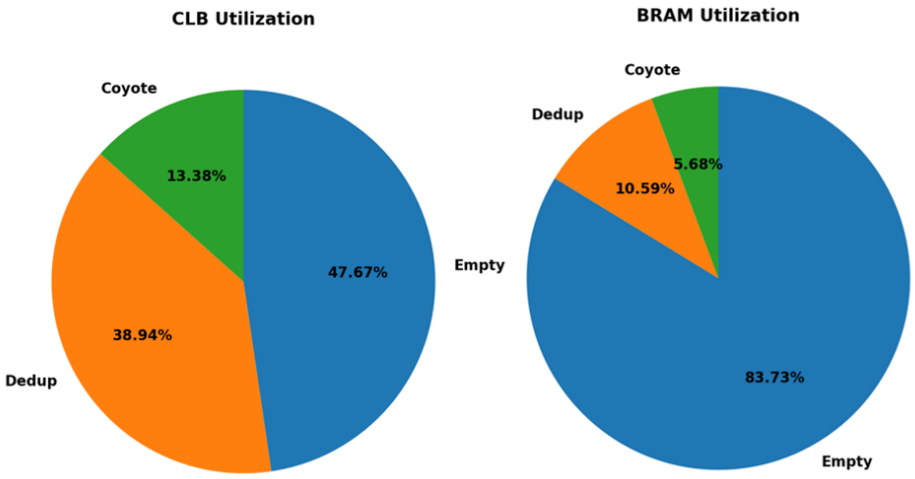
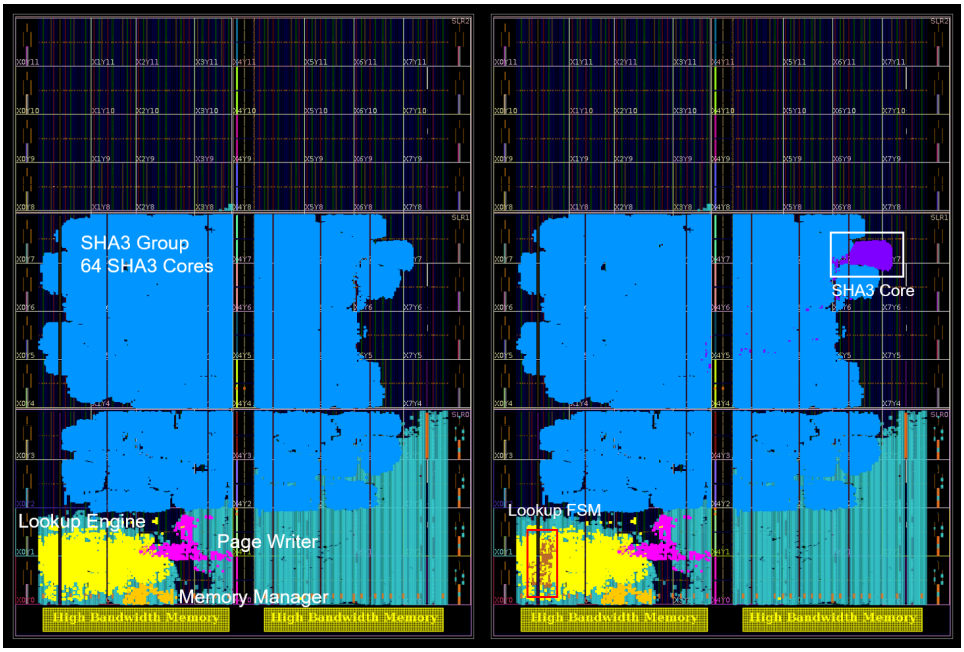


Figure 6.2: CLB and BRAM Resource Utilization of the Whole System

6.2 Floorplan and Resource Utilization inside DedupCore

Figure 6.3 shows the floorplan for components inside DedupCore. Resource usage inside DedupCore is shown in Figure 6.4. In the DedupCore, resource is mainly (∼90%) used by the 64x SHA3 cores.



(a) (b)

Figure 6.3: FPGA Floorplan of DedupCore Components

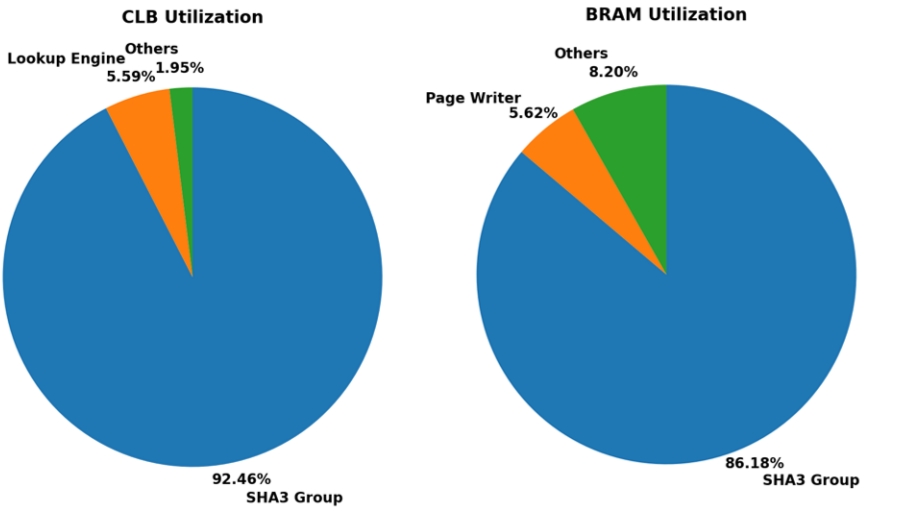


Figure 6.4: CLB and BRAM Resource Usage inside DedupCore

Chapter 7

Conclusion

In this project, we design and implement a data deduplication system on FPGA. This system works as an independent layer between the host and SSD, which is different from all the existing implementations.

We integrated SHA3 core groups, instruction handling logic, and hash table lookup logic on FPGA. We use reference counters and an in-line garbage collection mechanism to remove pages with no references. In the hash table, the linked list lookup is accelerated by our per-bucket Bloom filter. We detect false positives and reconstruct the Bloom filter as a deletion mechanism.

With 6 FSM and Bloom filter, we can saturate DMA throughput in normal cases. And reach 12.1 GB/s in the worst cases. For 8 FSM, we can reach 12.7 GB/s in any corner case. The insertion latency of the FPGA logic is smaller than 30 μs for writing and smaller than 15 μs for erasure and reading.

Bibliography

[1] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[2] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in ssds: Model and quantitative analysis. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2012.

[3] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data Deduplication—Large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, Boston, MA, June 2012. USENIX Association.

[4] Cornel Constantinescu, Joseph Glider, and David Chambliss. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference*, pages 393–402, 2011.

[5] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. Cidr: A cost-effective in-line data reduction system for terabit-per-second scale ssd arrays. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 28–41, 2019.

[6] Zheng Guo Chen, Nong Xiao, Fang Liu, Yu Xuan Xing, and Zhen Sun. Using fpga to accelerate deduplication on high-performance ssd. In *Materials Science and Intelligent Technologies Applications*, volume 1042 of *Advanced Materials Research*, pages 212–217. Trans Tech Publications Ltd, 11 2014.

[7] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A Content-Aware flash translation layer enhancing the lifespan of flash memory

based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.

[8] Ramin Gholami Taghizadeh, Reza Gholami Taghizadeh, Fahimeh Khakpash, Mohammadreza Binesh Marvasti, and Seyyed Amir Asghari. Ca-dedupe: content-aware deduplication in ssds. *The Journal of Supercomputing*, 76(11):8901–8921, Nov 2020.

[9] Jin-Yong Ha, Young-Sik Lee, and Jin-Soo Kim. Deduplication with block-level content-aware chunking for solid state drives (ssds). In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1982–1989, 2013.

[10] Zhengguo Chen, Zhiguang Chen, Nong Xiao, and Fang Liu. Nf-dedupe: A novel no-fingerprint deduplication scheme for flash-based ssds. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 588–594, 2015.

[11] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 187–202. USENIX Association, February 2021.

[12] Bon-Keun Seo, Seungryoul Maeng, Joonwon Lee, and Euiseong Seo. Draco: A deduplicating ftl for tangible extra capacity. *IEEE Computer Architecture Letters*, 14(2):123–126, 2015.

[13] Yoshihiro Tsuchiya and Takashi Watanabe. Dblk: Deduplication for primary block storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2011.

[14] Zhichao Yan, Hong Jiang, Song Jiang, Yujuan Tan, and Hao Luo. Sesdedup: a case for low-cost ecc-based ssd deduplication. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 292–298, 2019.

[15] Binqi Zhang, Chen Wang, Bing Bing Zhou, and Albert Y. Zomaya. Inline data deduplication for ssd-based distributed storage. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 593–600, 2015.

[16] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[17] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[18] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.

[19] SpinalHDL. https://github.com/SpinalHDL/SpinalHDL. Accessed: 2023-08-24.

[20] SpinalCrypto. https://github.com/SpinalHDL/SpinalCrypto. Accessed: 2023-08-24.