We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs

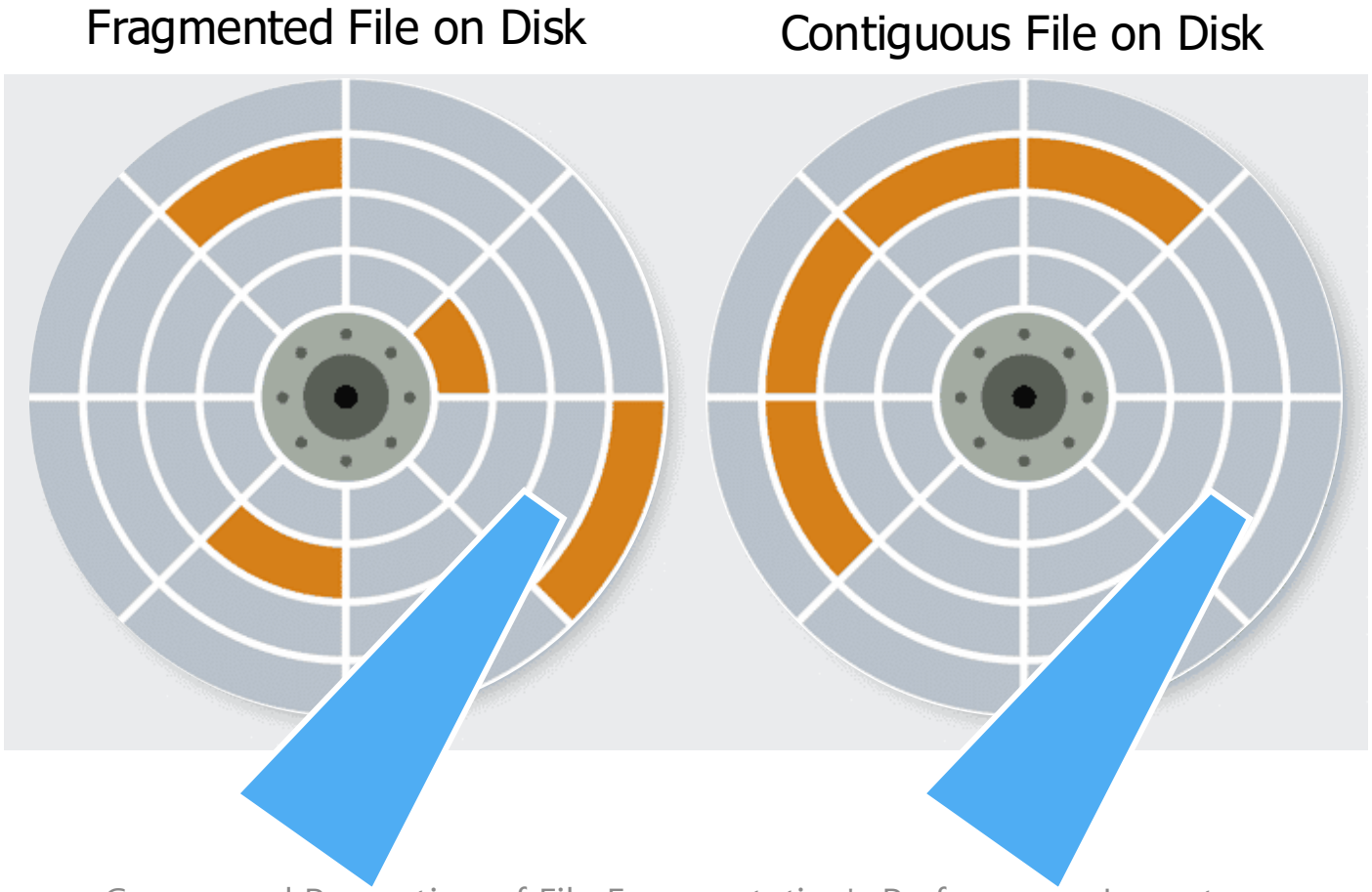
Yuhun Jun[1,2], Shinhyun Park[1], Jeong-Uk Kang[2], Sang-Hoon Kim[3] and Euiseong Seo[1]

[1]Sungkyunkwan University [2]Samsung Electronics [3]Ajou University

Unique Origin Unique Future

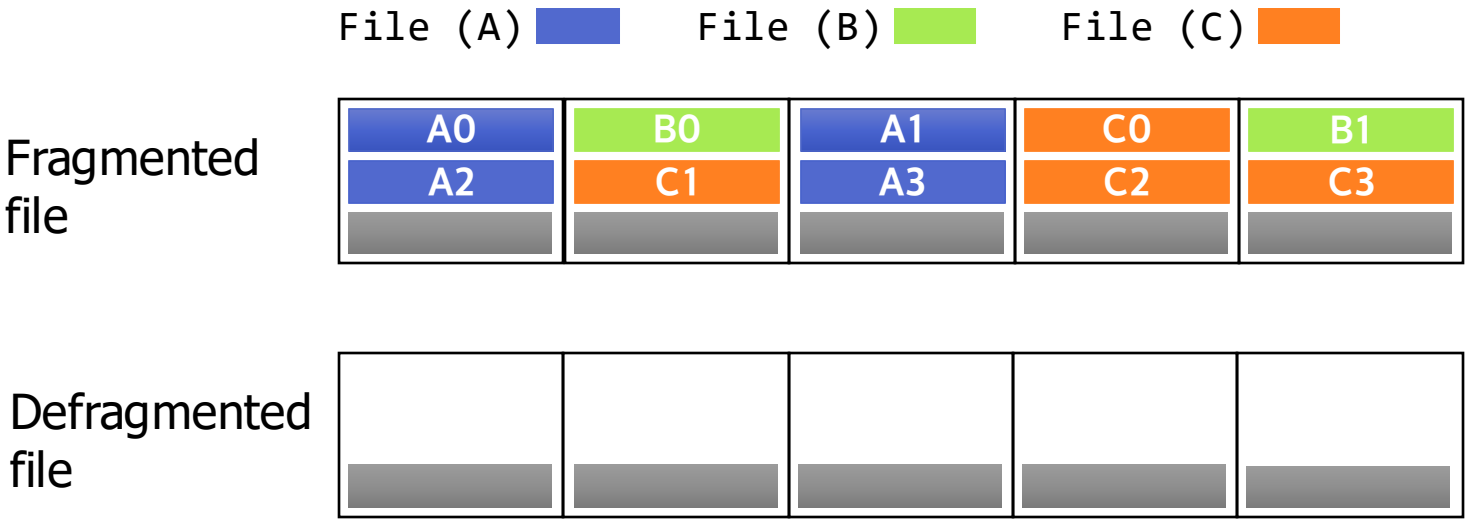
File Fragmentation

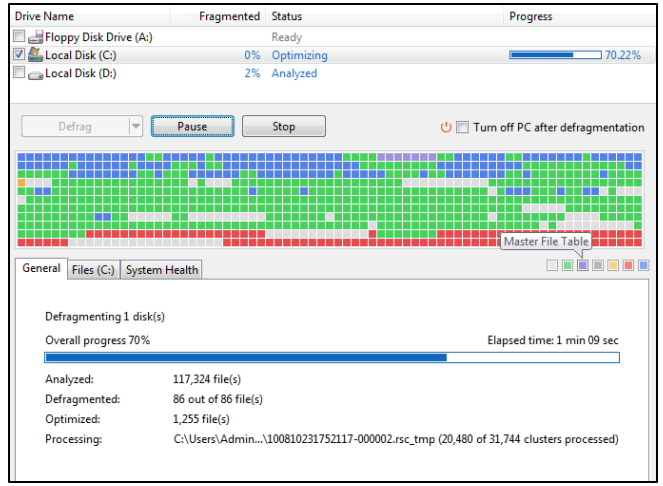
- Non-contiguously stored file ➔ Degraded read performance

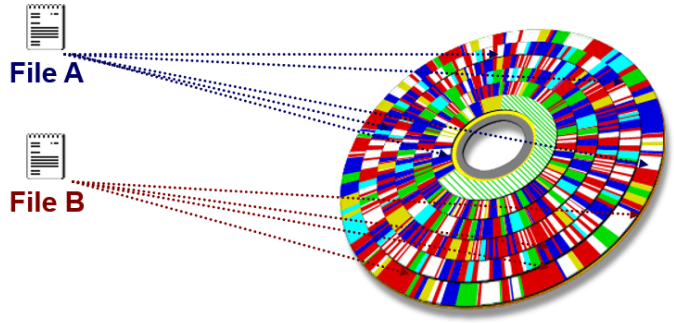


File Fragmentation

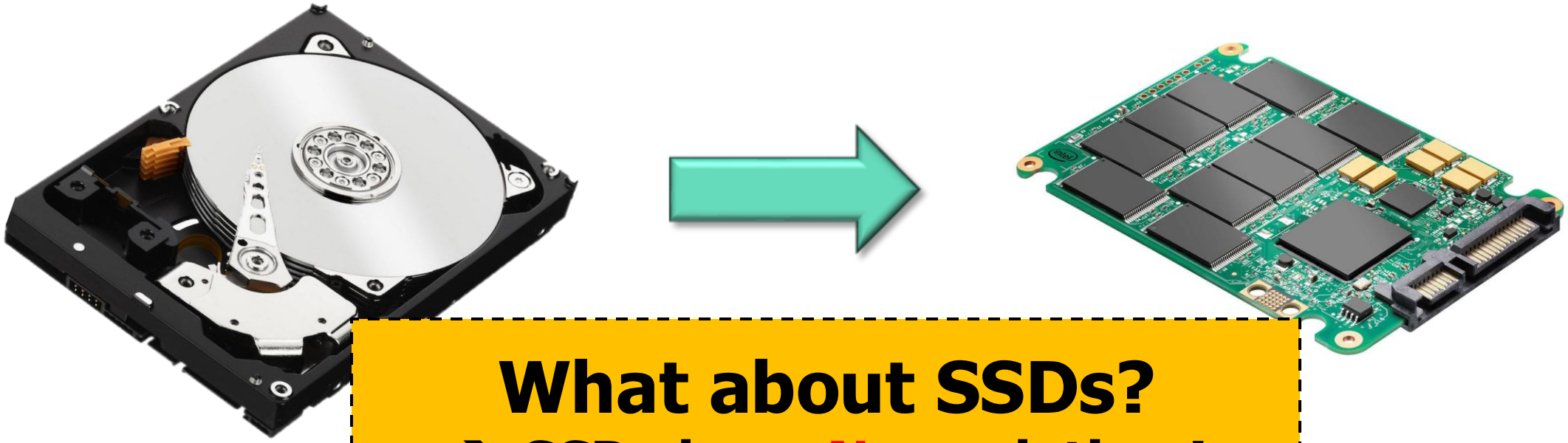
- To recover, costly defragmentation should be performed







File Fragmentation



What about SSDs?

➔ SSDs have No seek time!

➔ No performance drop?

File Fragmentation in SSD-Era

- Even in SSDs, still performance degradation occurs (observed reduction of 2x to 5x) *

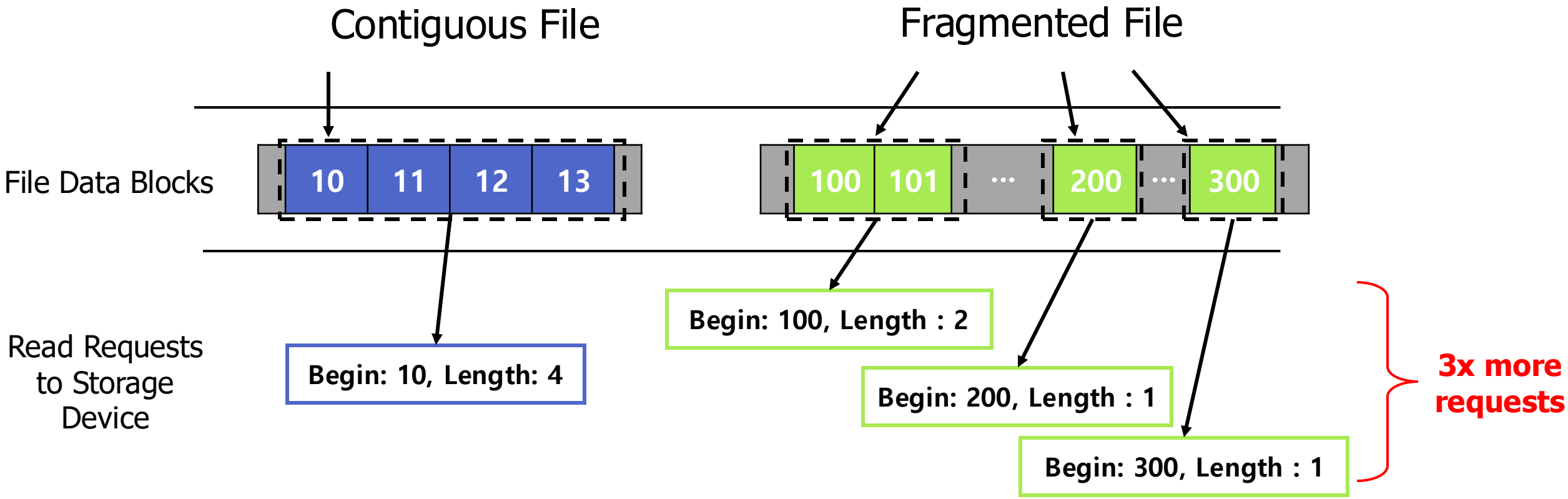
  * Conway *et al.* File systems fated for senescence? nonsense, says science! (FAST ' 17).
  * Kadekodi *et al.* Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems (ATC ' 18).
  * Conway *et al.* Filesystem Aging: It's more usage than fullness (Hotstorage ' 19).

- ***request*** splitting caused by fragmentation increases kernel I/O stack overhead **

  ** Park and Eom. Fragpicker: A new defragmentation tool for modern storage devices (SOSP ' 21)

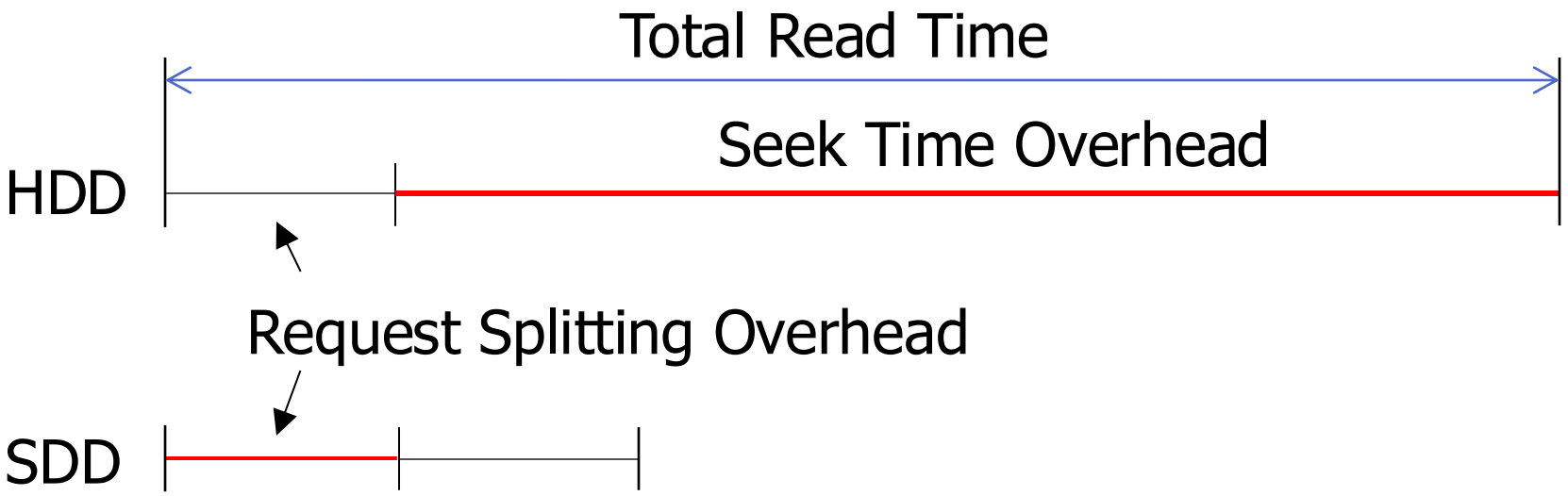
File Fragmentation in SSD-Era

- ***Request splitting***



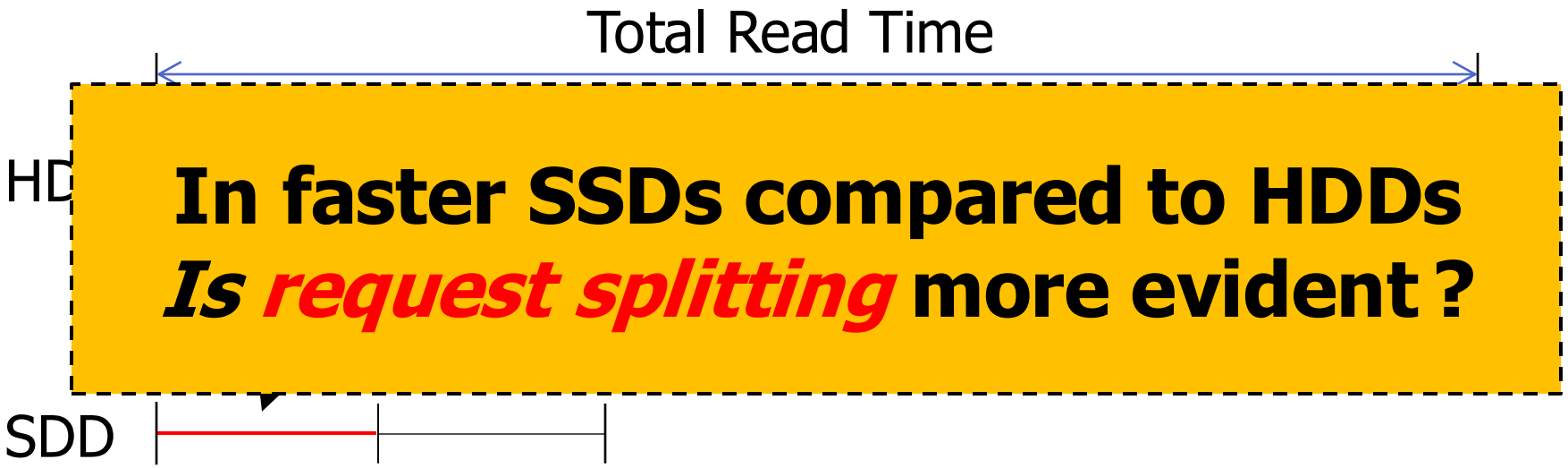
File Fragmentation in SSD-Era

- ***Request splitting***



File Fragmentation in SSD-Era

- ***Request splitting***

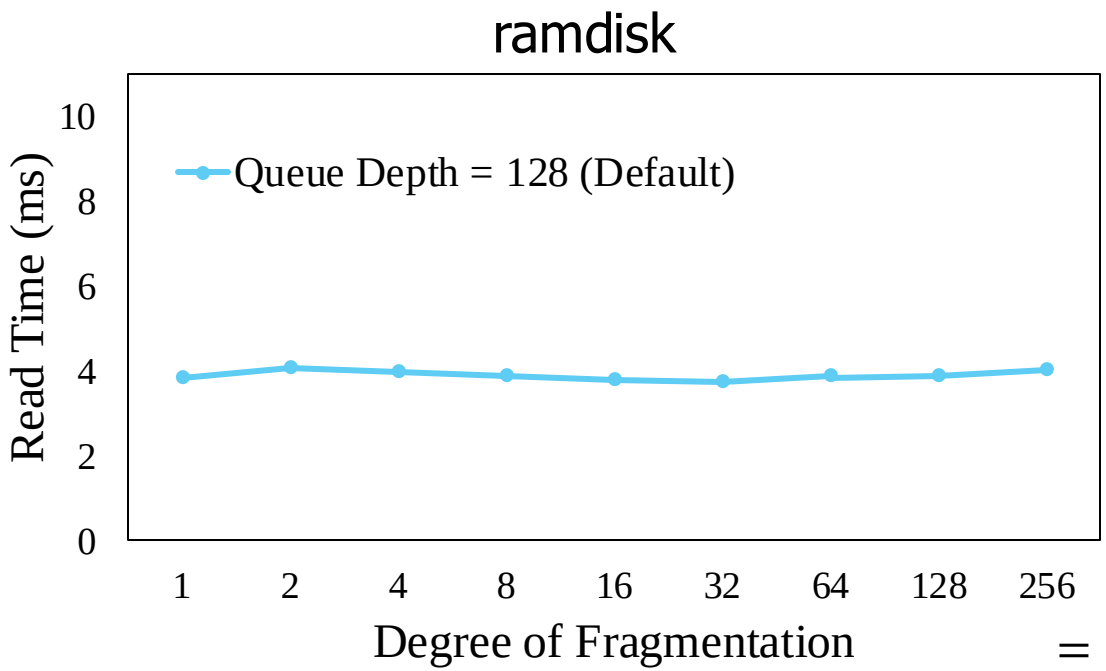


Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

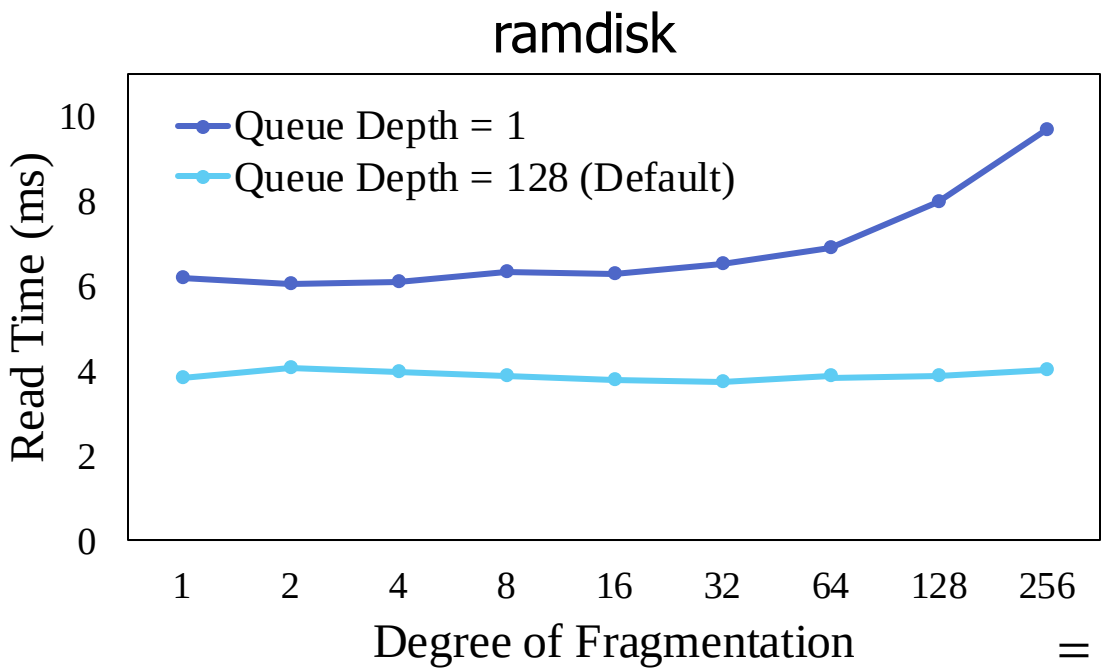


$$= \frac{\text{Actual Number of Extents of File}}{\text{Ideal Number of Extents of File (contiguous case)}}$$

Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

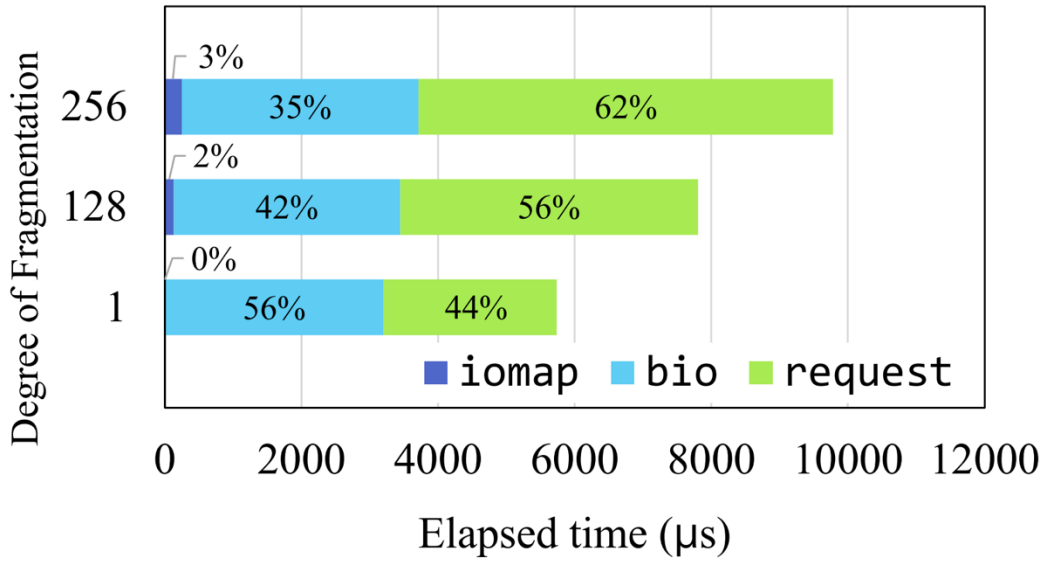
→ Impact seen with forced queue depth of 1

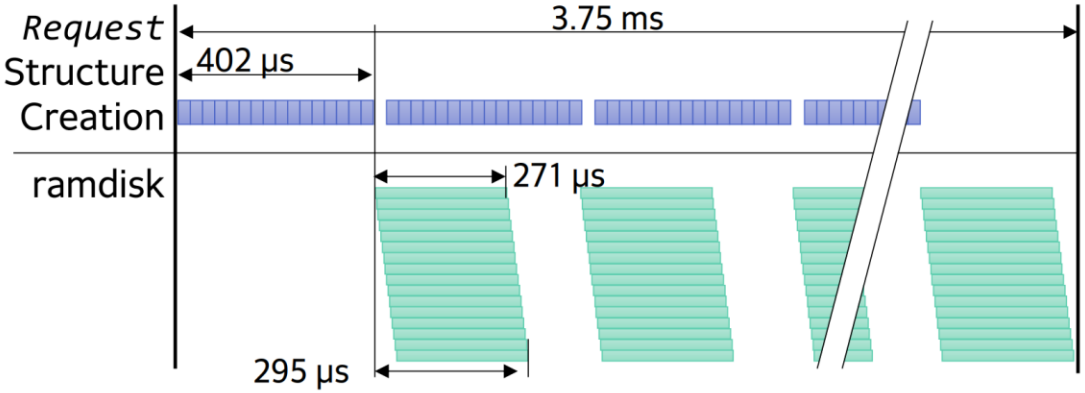


$$= \frac{\text{Actual Number of Extents of File}}{\text{Ideal Number of Extents of File (contiguous case)}}$$

Analysis of Request Splitting Overhead

- Request splitting overhead

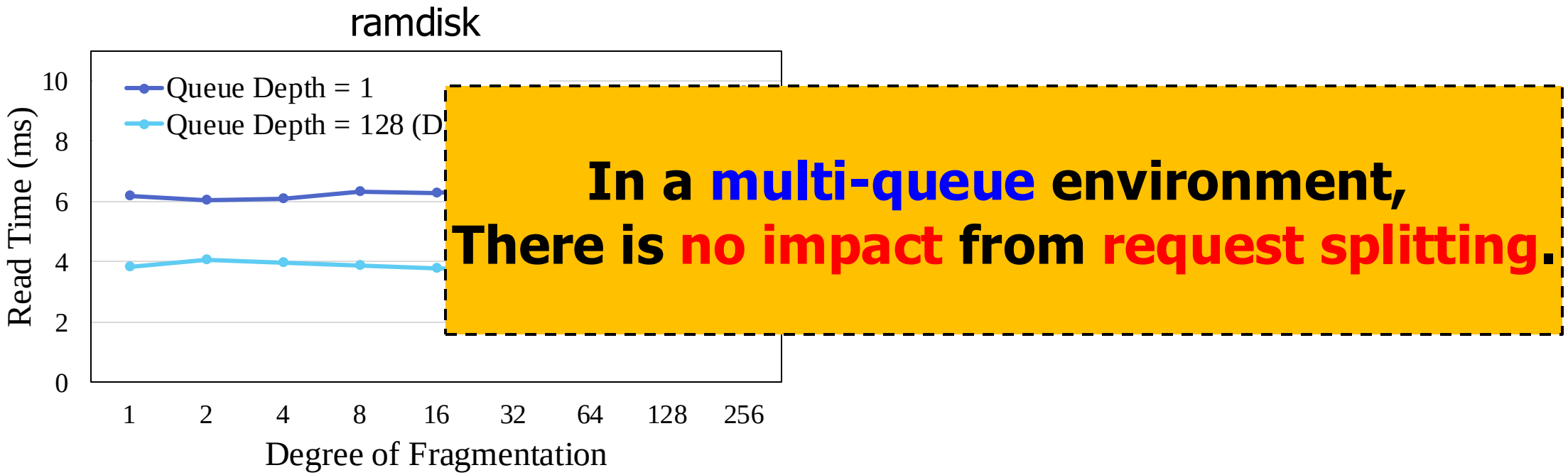




Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

→ Impact seen with forced queue depth of 1

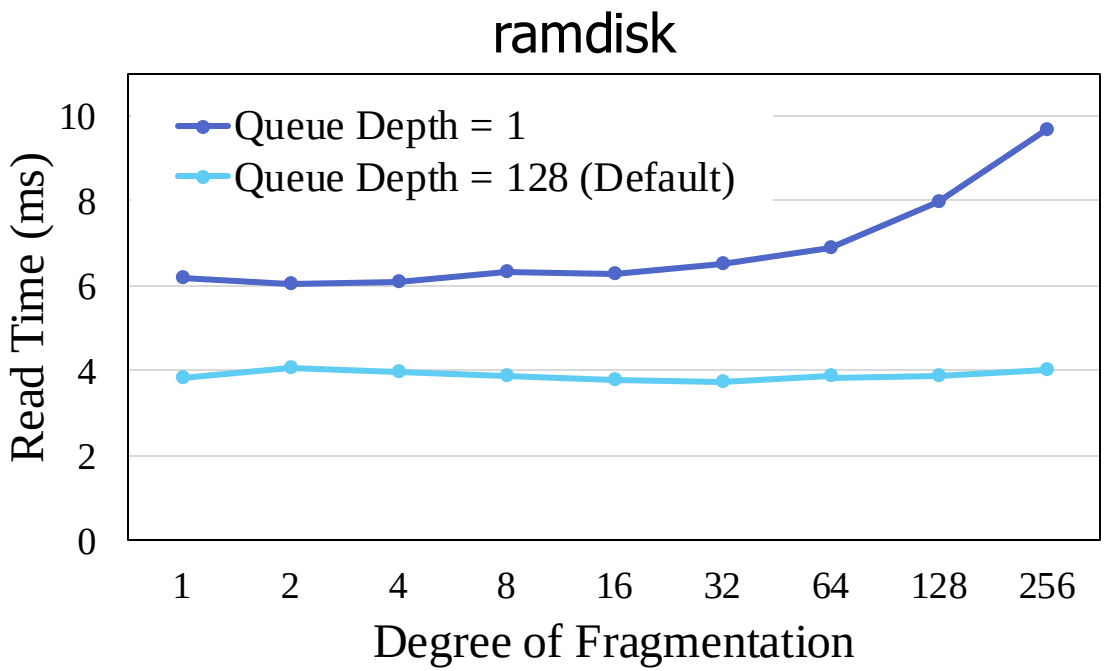


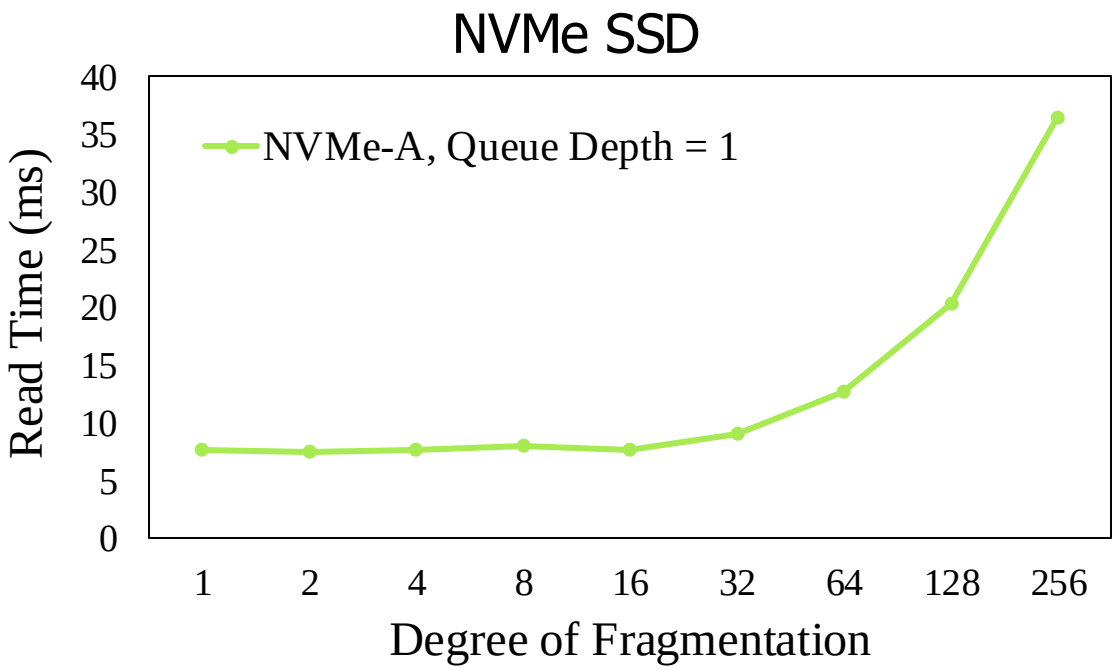
**In a multi-queue environment,
There is no impact from request splitting.**

Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

→ Impact seen with forced queue depth of 1; No request splitting impact in multi-queue (default)



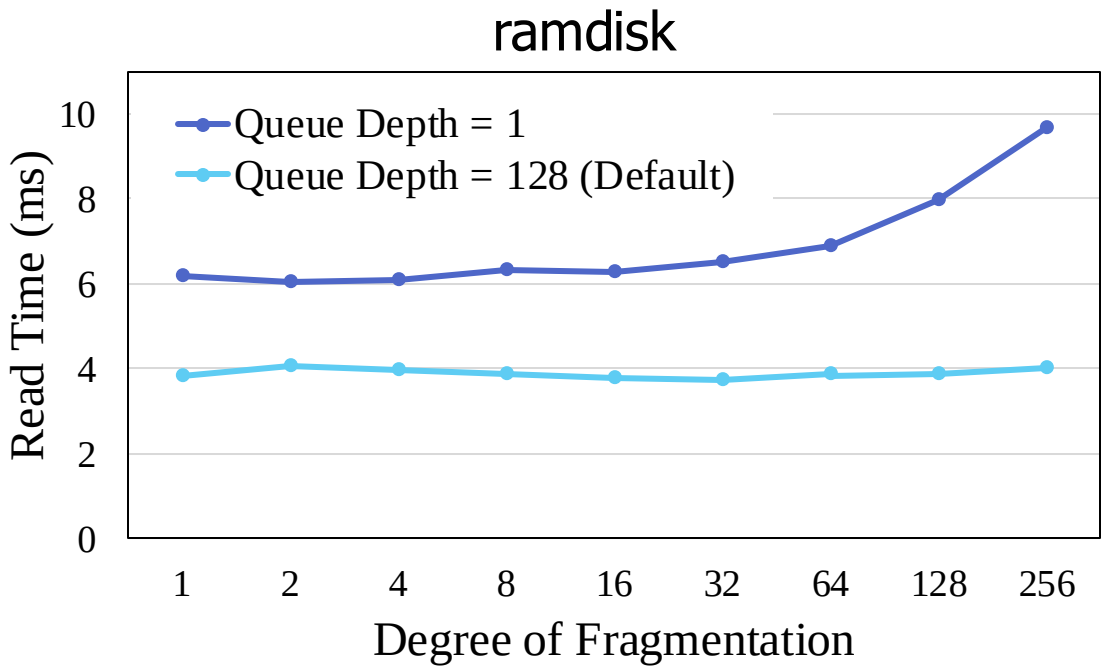


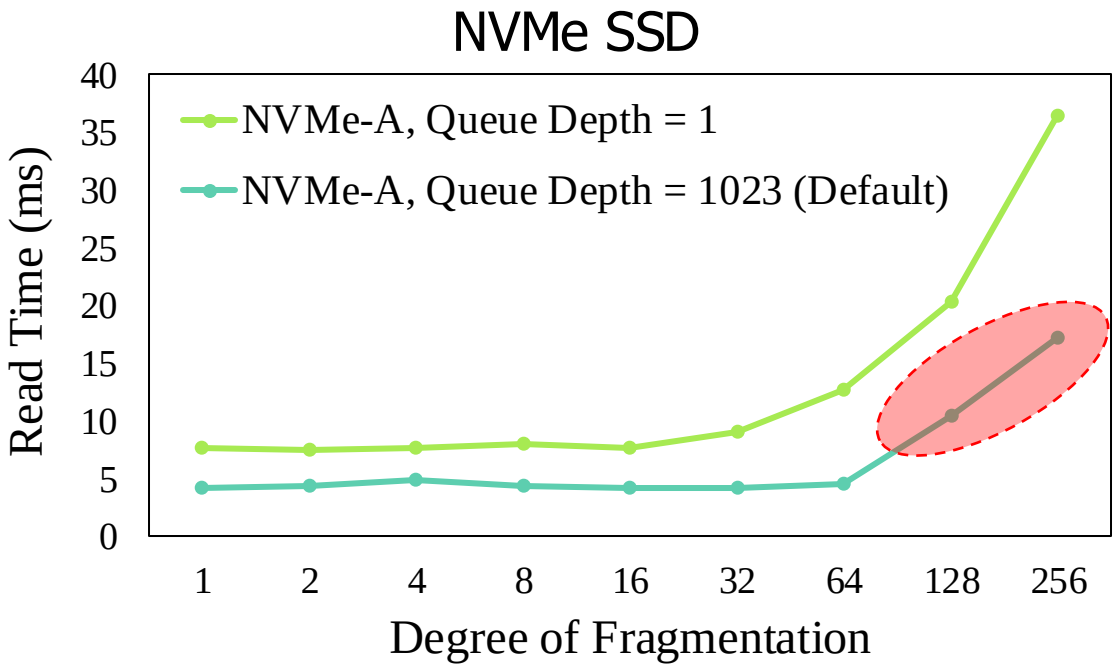
Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

→ Impact seen with forced queue depth of 1; No request splitting impact in multi-queue (default)

Commercial SSDs showed performance drop in fragmentation



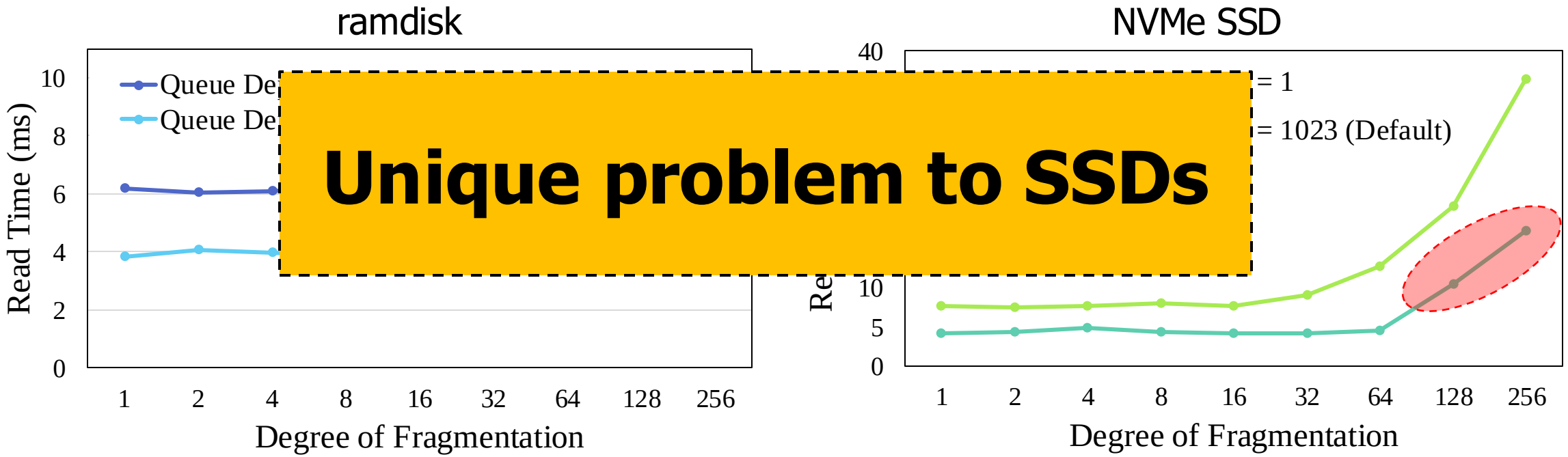


Analysis of Request Splitting Overhead

- Does *request splitting* impact ramdisks more than SSDs?

→ Impact seen with forced queue depth of 1; No request splitting impact in multi-queue (default)

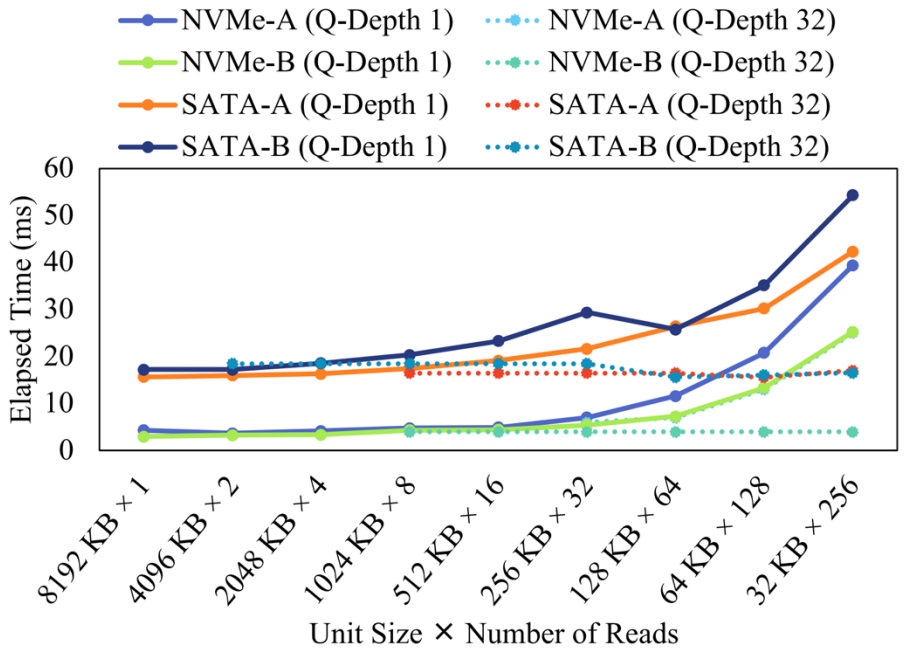
Commercial SSDs showed performance drop in fragmentation



Analysis of Interface Overhead

- Does *PCIe Interface* extend the read latency?

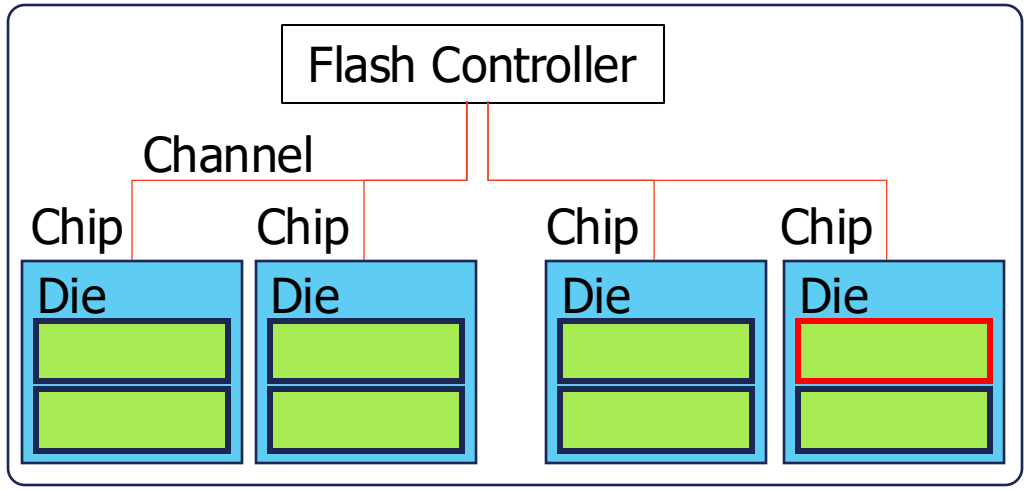
→ When we read the same number of sectors with different unit request sizes, the performance barely varied



Time for reading 8 MB of data through raw device I/O operations

SSD Performance Background

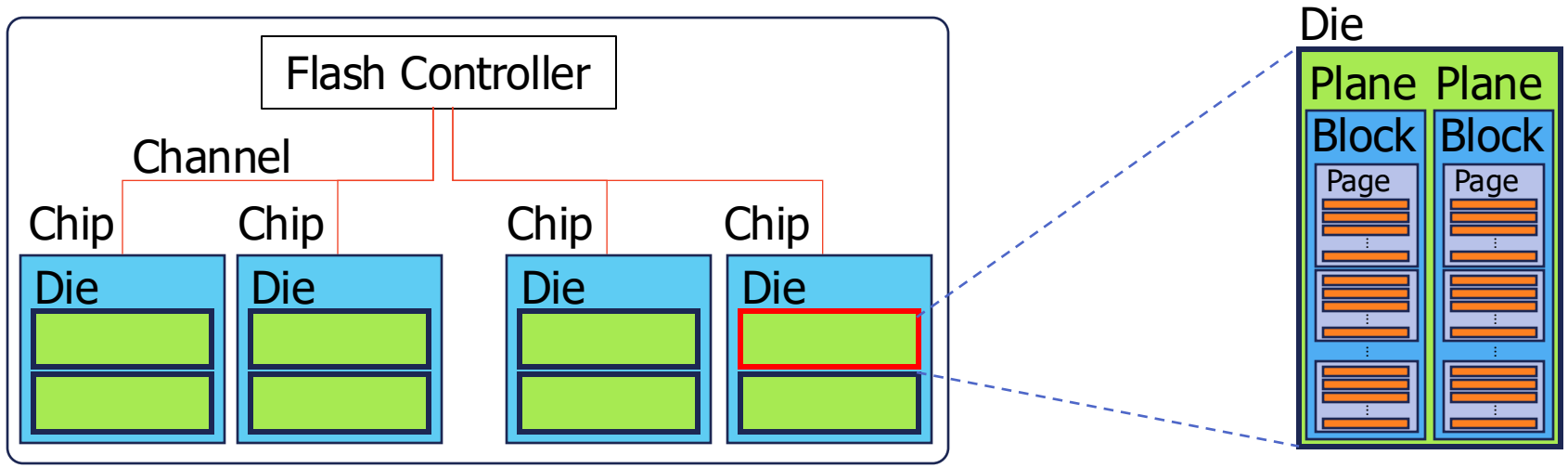
- High performance from operating multiple NAND Flashes simultaneously



Structure of NAND Flash inside SSD
(2-Channel 2-chip 2-Die SSD Total 8-Dies in an SSD)

SSD Performance Background

- High performance from operating multiple NAND Flashes simultaneously

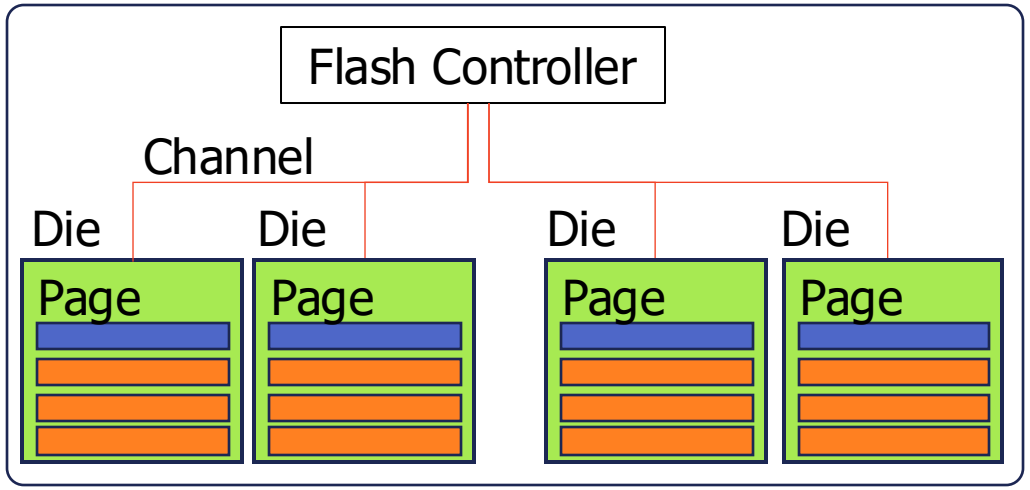


Structure of NAND Flash inside SSD
(2-Channel 2-chip 2-Die SSD Total 8-Dies in an SSD)

Page writing/reading suspends other operations issued to the same die

SSD Performance Background

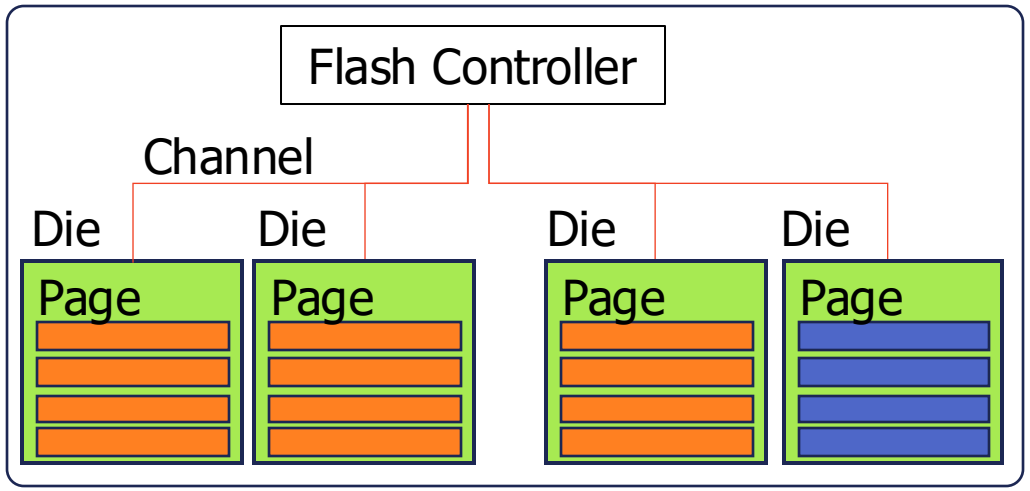
- High performance from operating multiple dies simultaneously
- For write and read operations, as many dies as possible should be utilized



Structure of NAND **dies** inside SSD
(2-Channel 2-Die SSD, Total 4-Dies in an SSD)

SSD Performance Background

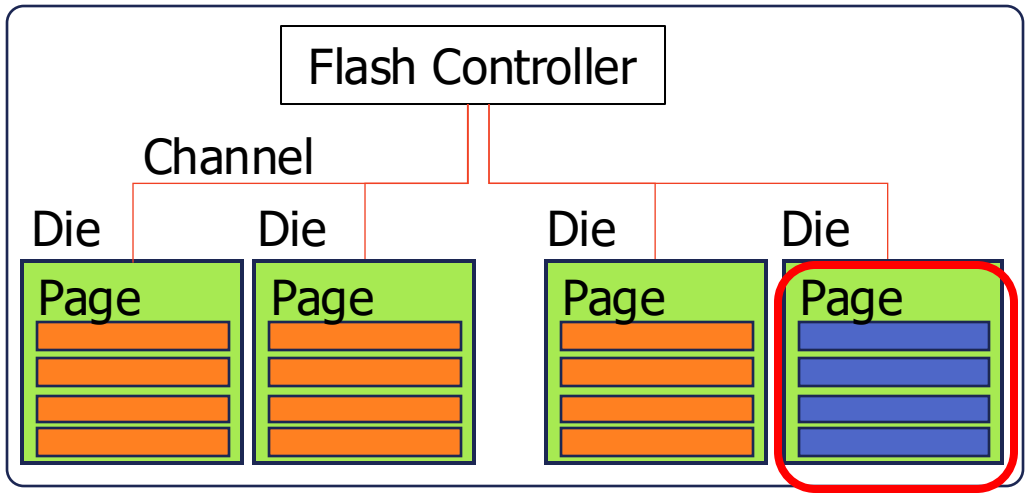
- High performance from operating multiple dies simultaneously.
- For write and read operations, as many dies as possible should be activated
- Focusing on a single die during reading ➔ Reduced parallelism



Structure of NAND **dies** inside SSD
(2-Channel 2-Die SSD, Total 4-Dies in an SSD)

SSD Performance Background

- High performance from operating multiple dies simultaneously.
- For write and read operations, as many dies as possible should be activated
- Focusing on a single die during reading ➔ Reduced parallelism

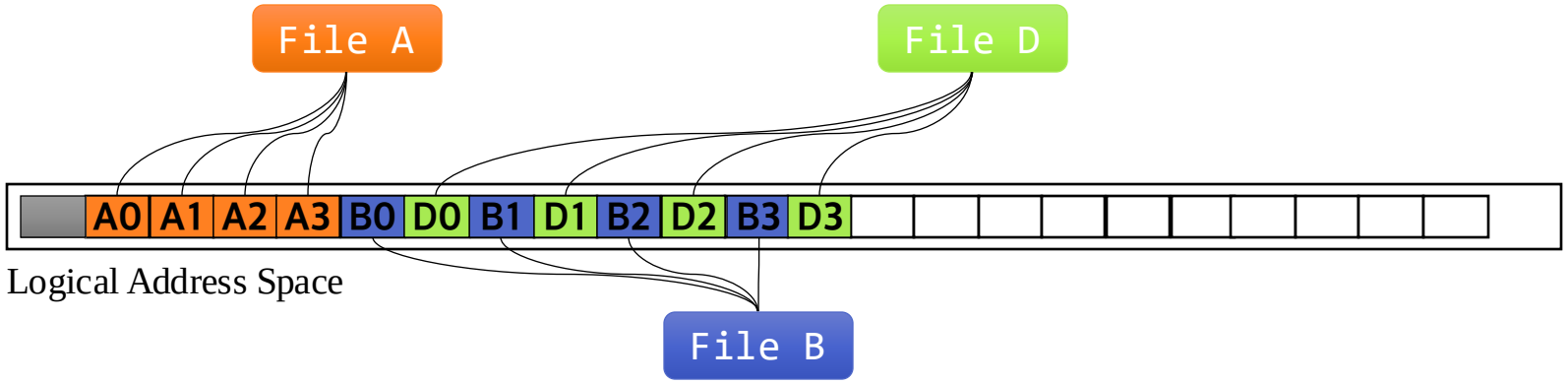


Read Collisions

Structure of NAND **dies** inside SSD
(2-Channel 2-Die SSD, Total 4-Dies in an SSD)

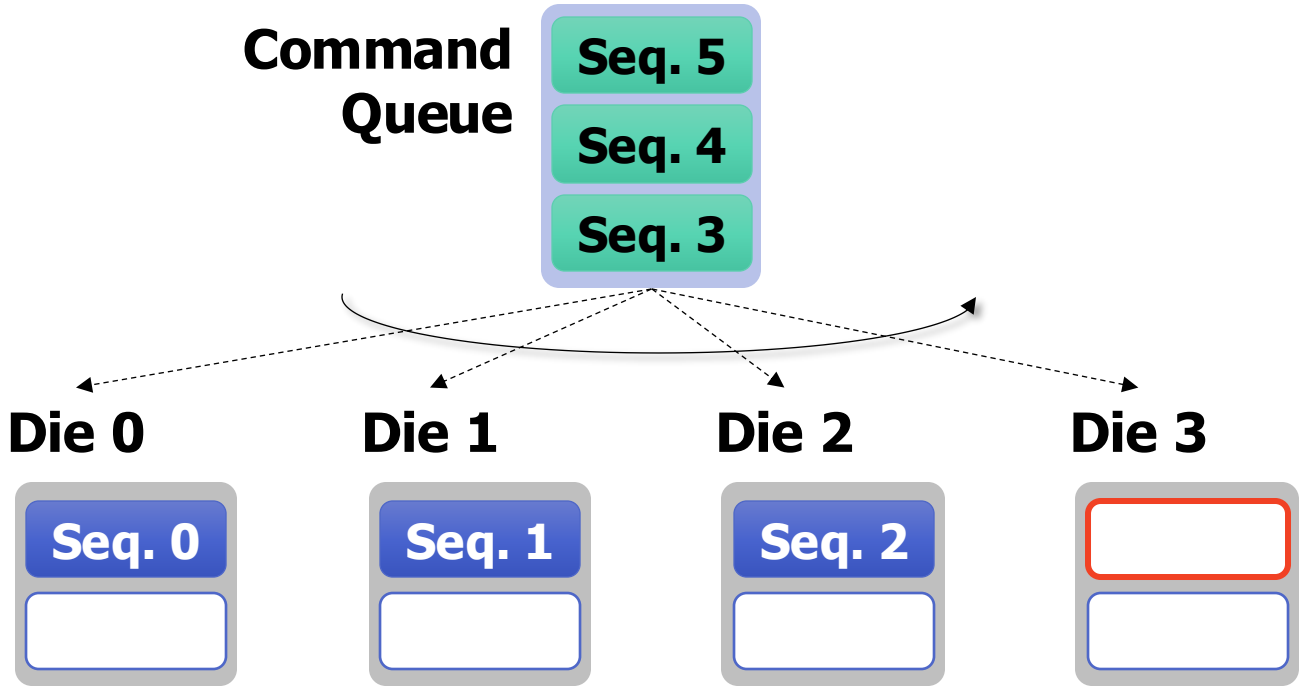
File Fragmentation Scenarios

- File fragmentation occurs when multiple files are appended in an alternating manner



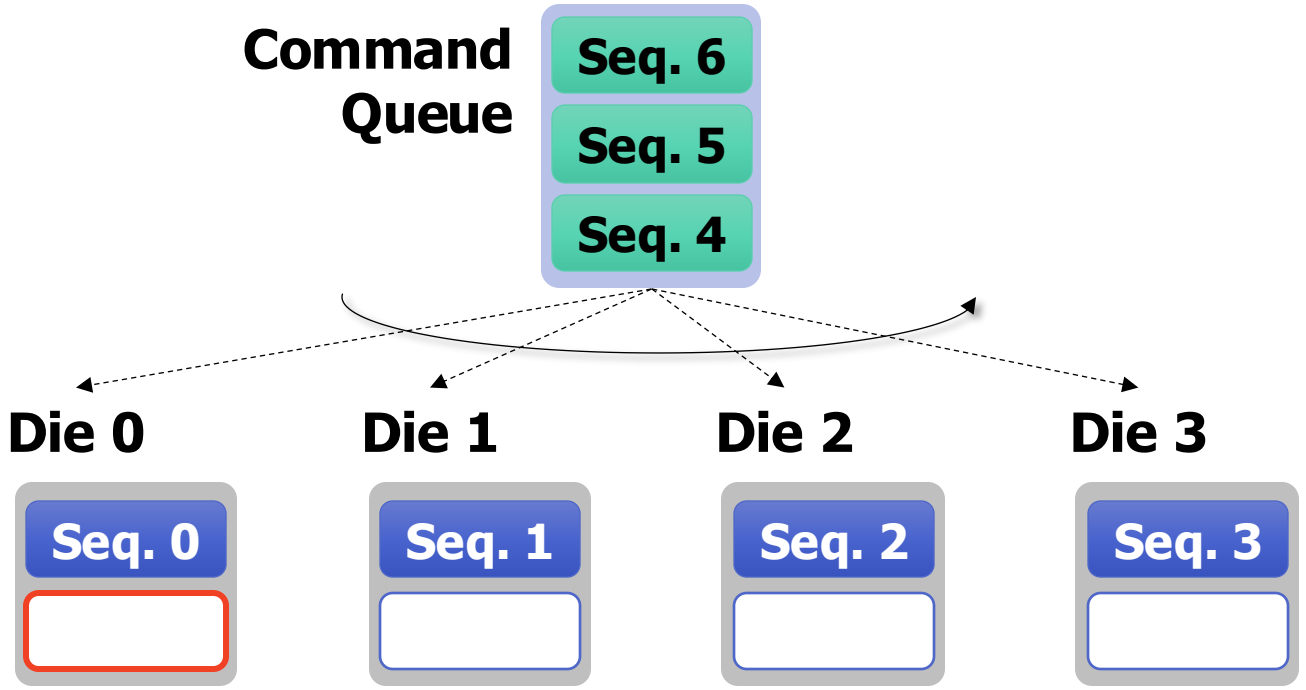
Misaligned Die Allocation from Fragmentation

- Pages to be written are allocated from dies in a round-robin manner



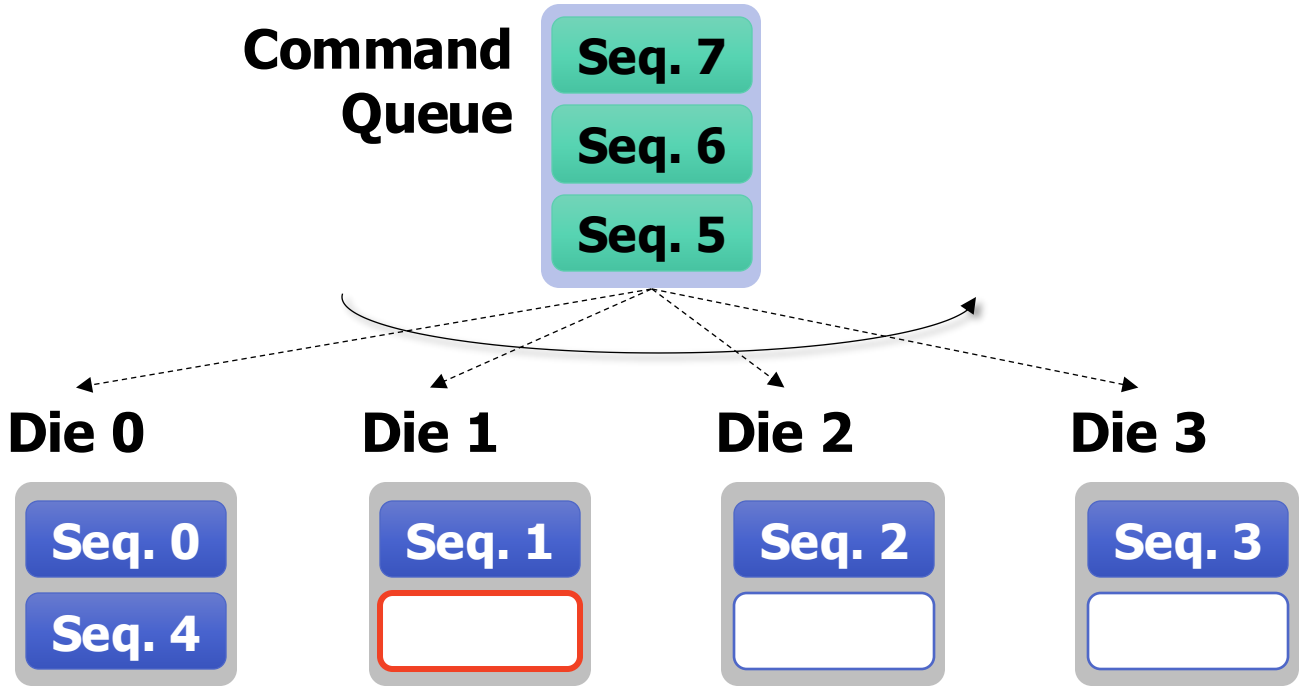
Misaligned Die Allocation from Fragmentation

- Pages to be written are allocated from dies in a round-robin manner



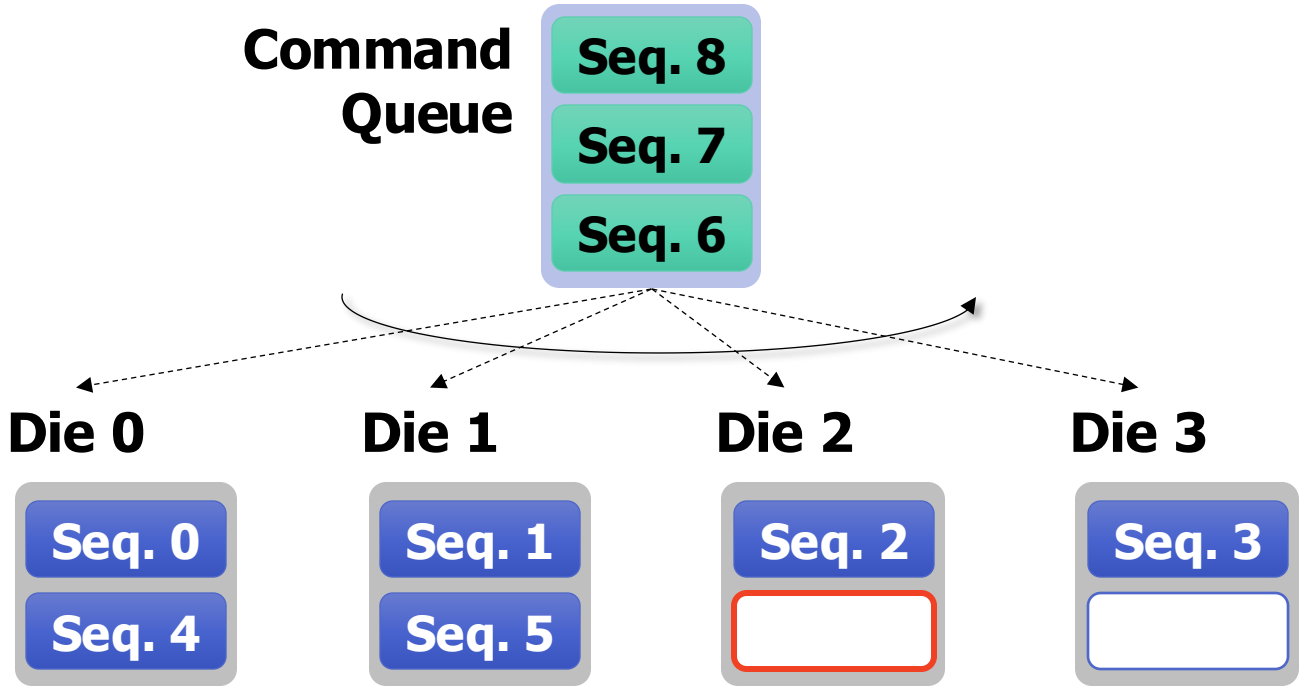
Misaligned Die Allocation from Fragmentation

- Pages to be written are allocated from dies in a round-robin manner



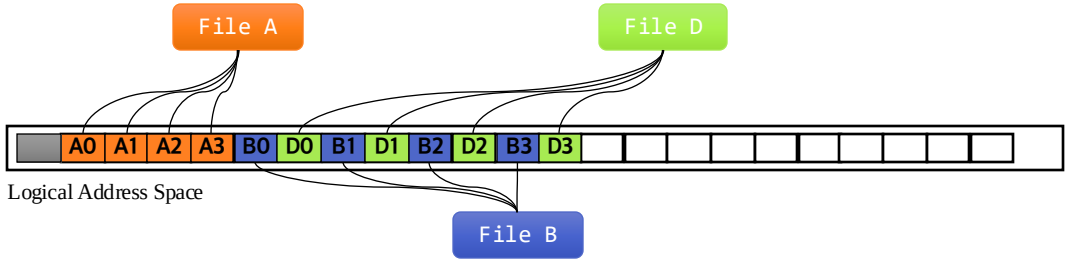
Misaligned Die Allocation from Fragmentation

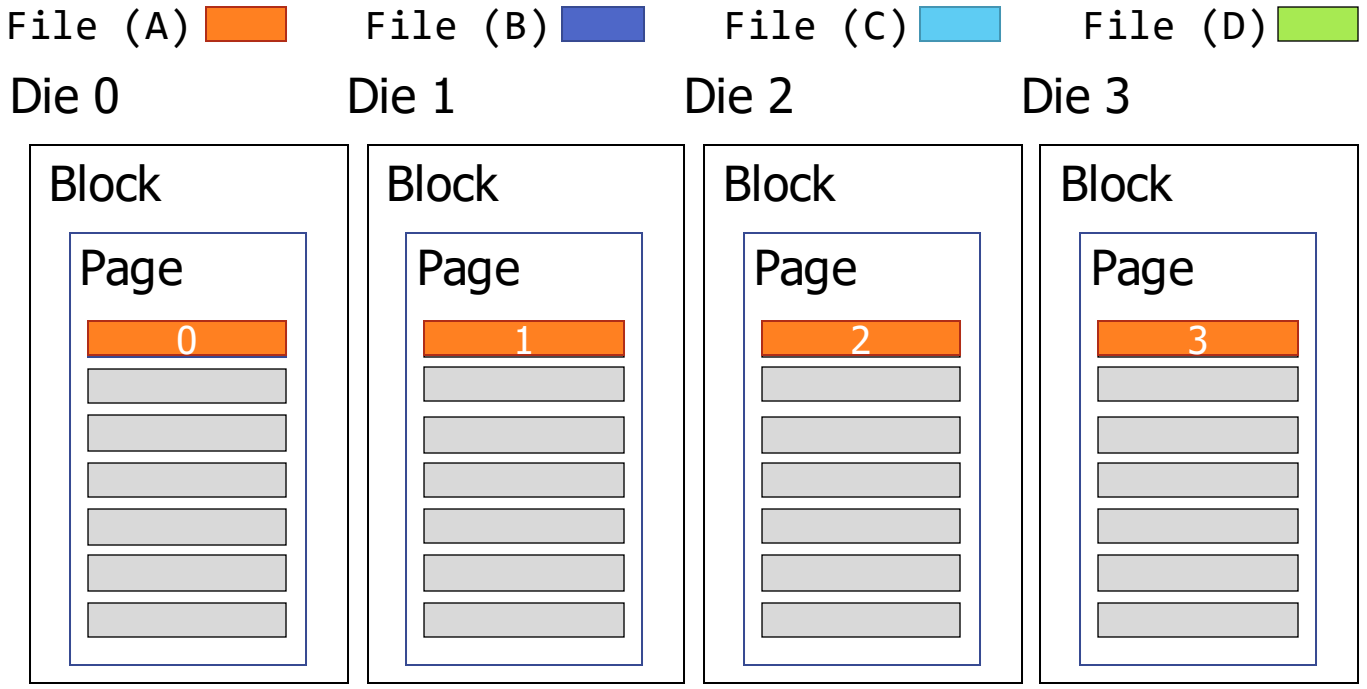
- Pages to be written are allocated from dies in a round-robin manner



Misaligned Die Allocation from Fragmentation

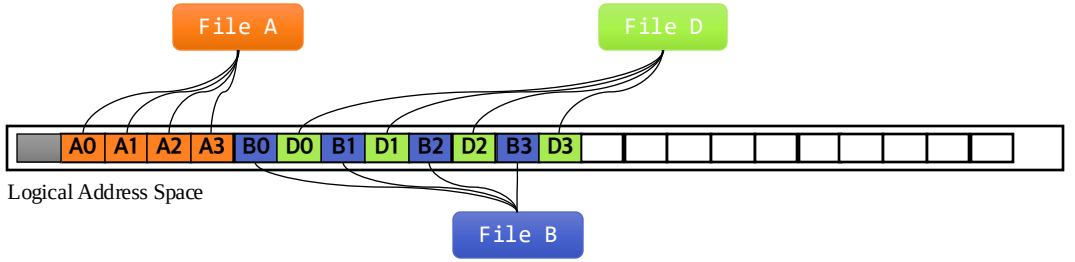
- Alternating appends causes misaligned die allocation

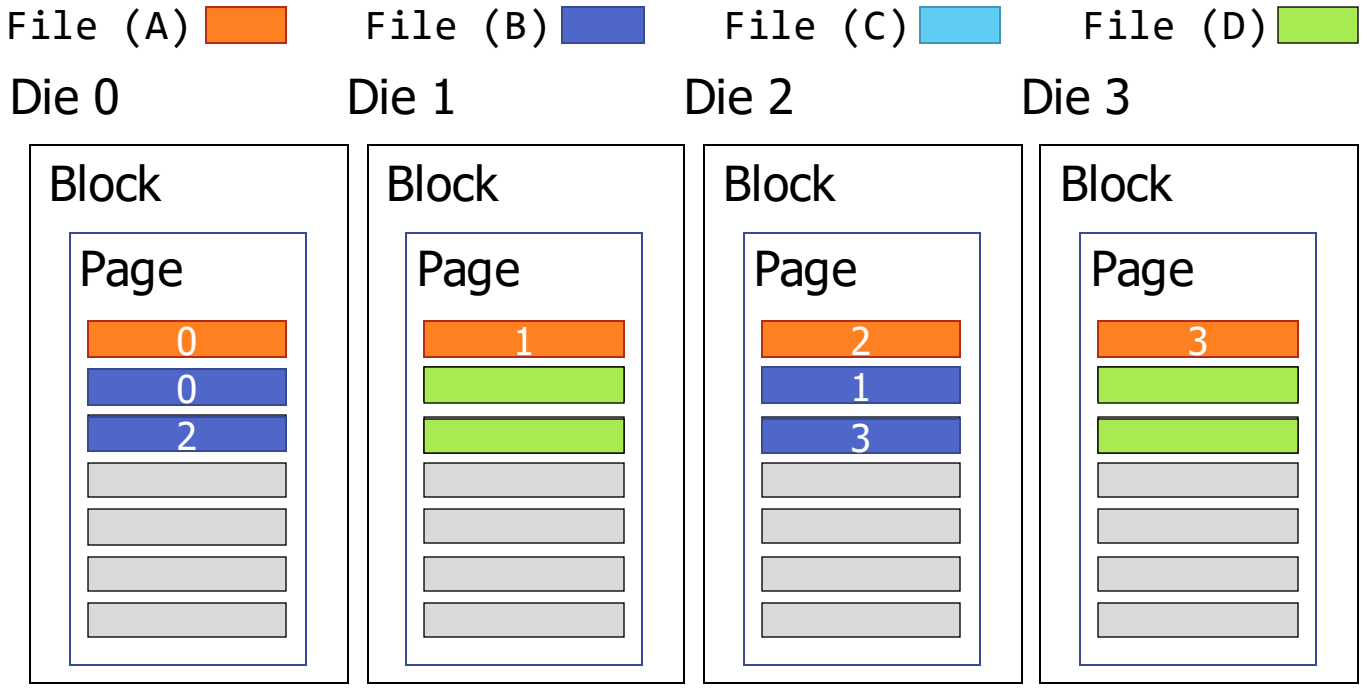




Misaligned Die Allocation from Fragmentation

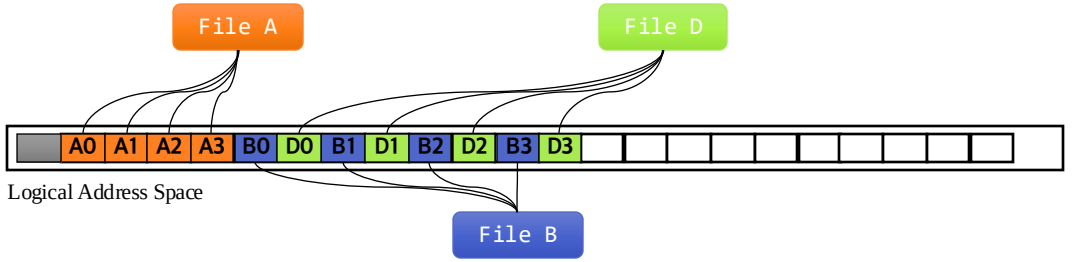
- Alternating appends causes misaligned die allocation

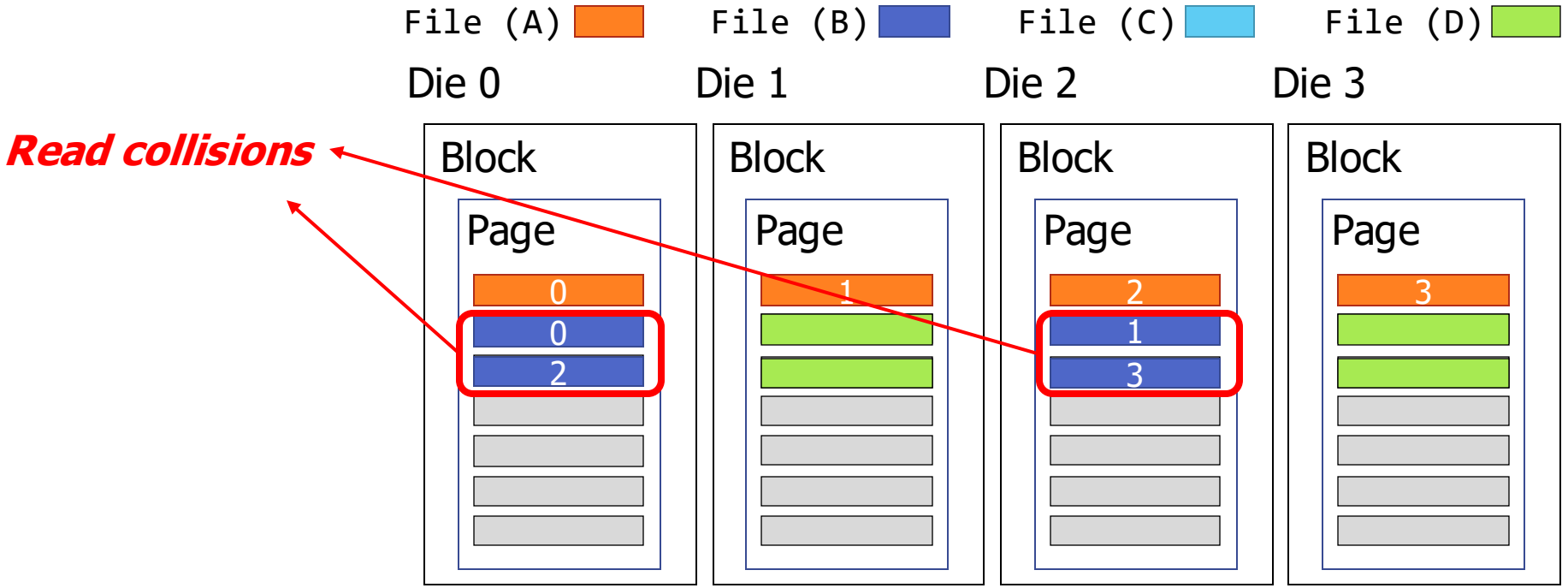




Misaligned Die Allocation from Fragmentation

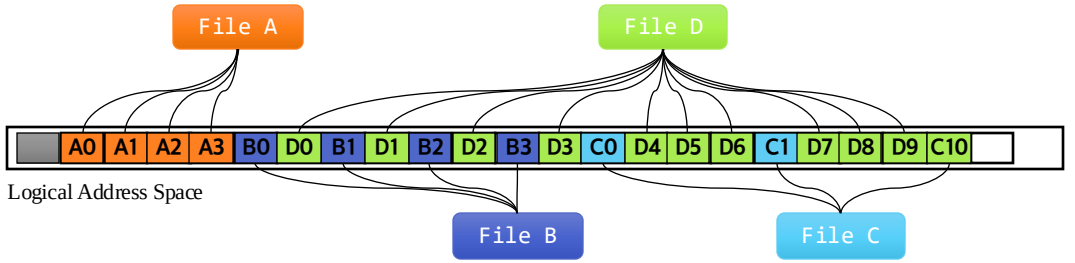
- Alternating appends causes misaligned die allocation

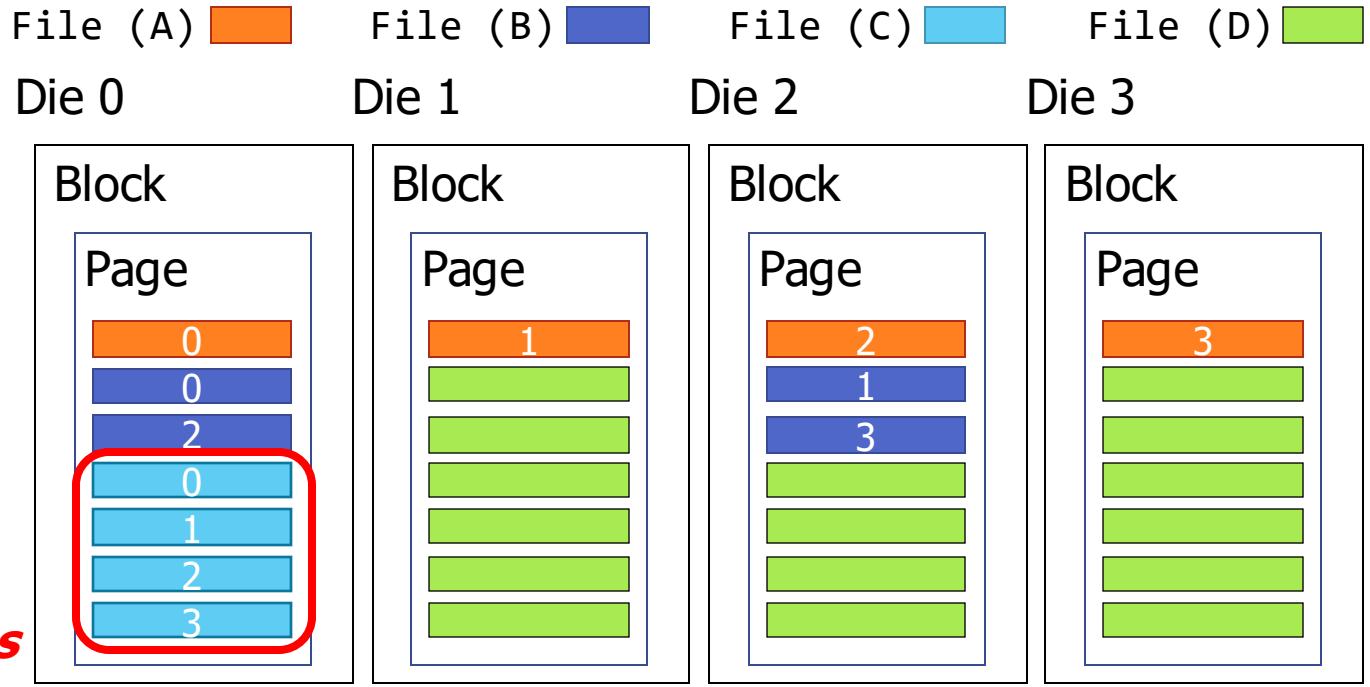




Misaligned Die Allocation from Fragmentation

- Alternating appends causes misaligned die allocation

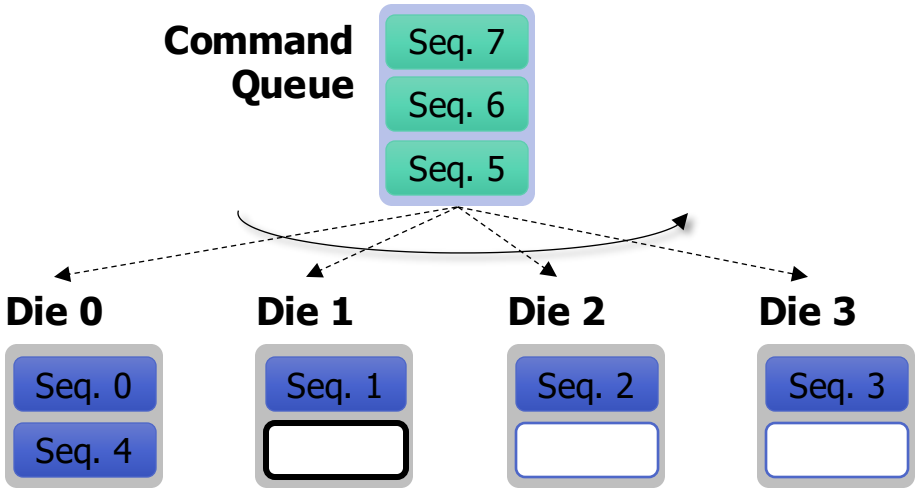




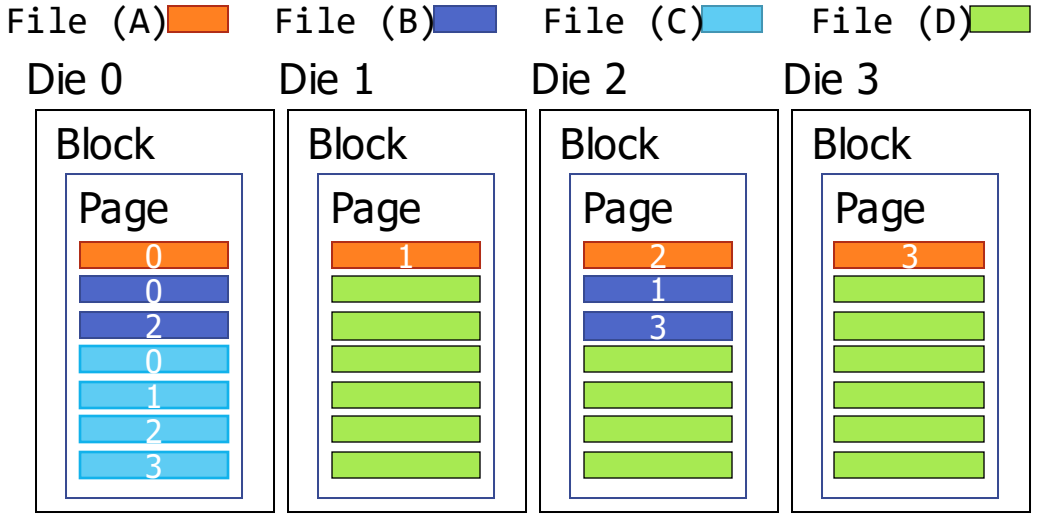
Read collisions

Misaligned Die Allocation from Fragmentation

- Alternating appends causes misaligned die allocation
  ➔ ***Read collisions***



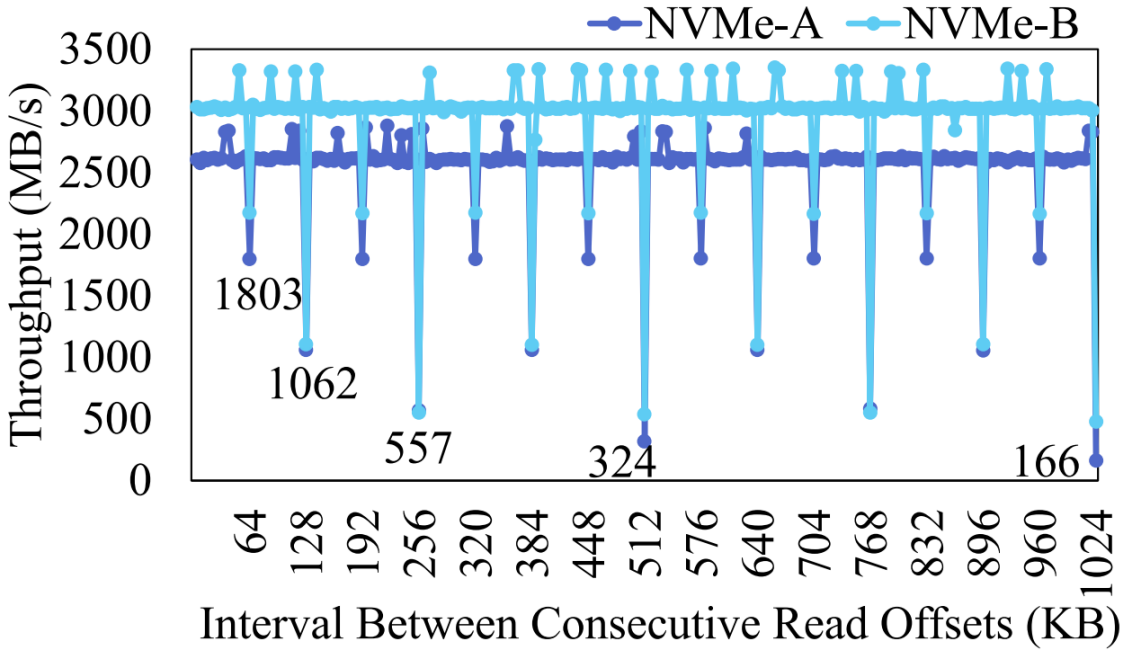
Die allocation in a round-robin manner



Concurrent writes to multiple files cause misaligned die allocation

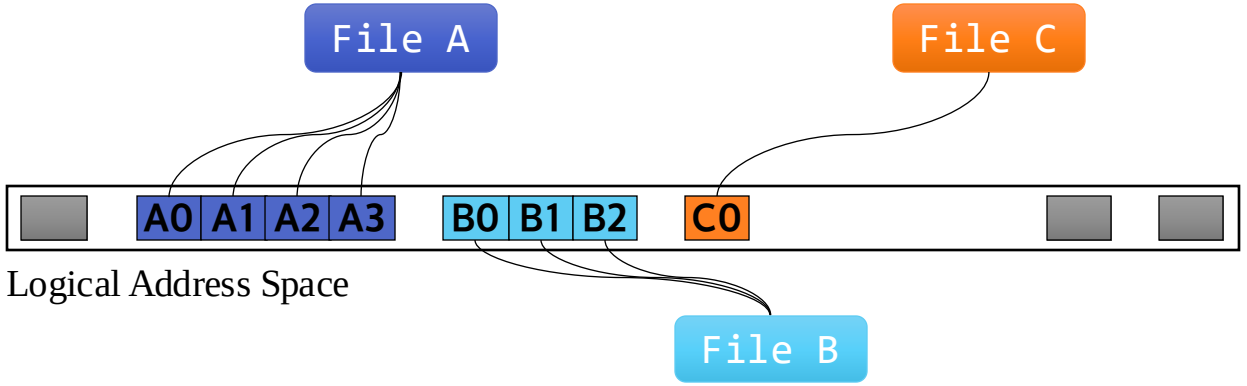
Misaligned Die Allocation from Fragmentation

- Experiments with commercial SSDs clearly verified our conjecture

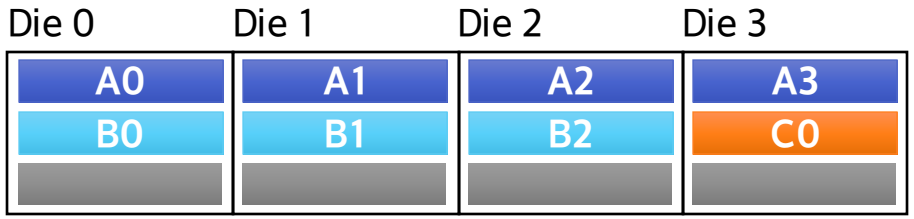


Throughput while varying the interval between starting points of consecutive read operations

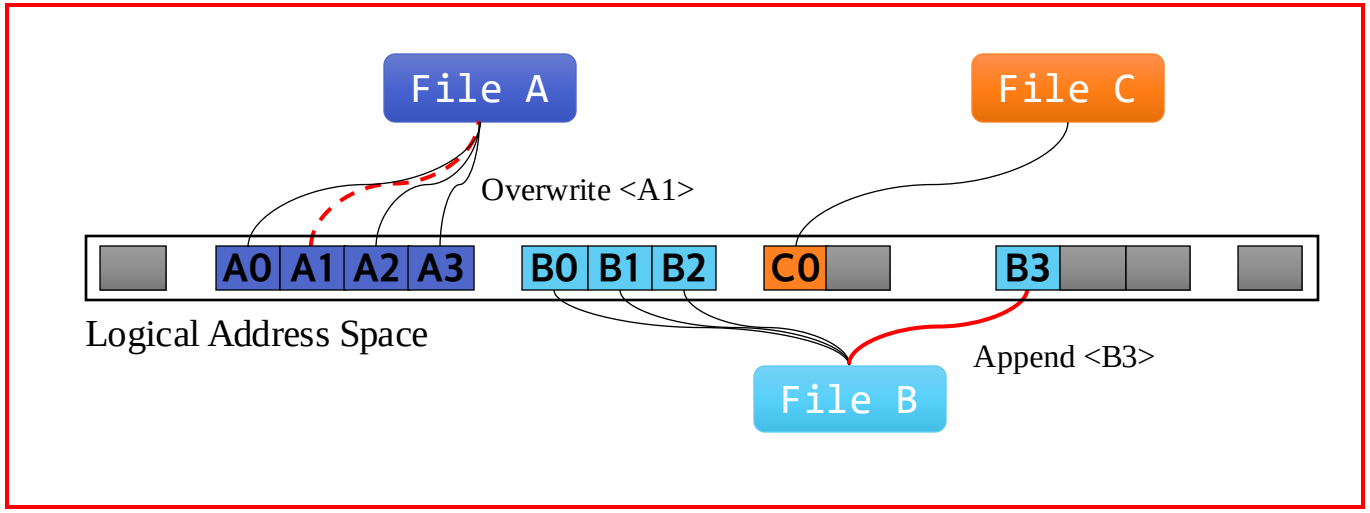
Misaligned Die Allocation from Overwrites



- File A Append <A0, A1, A2, A3>
- File B Append <B0, B1, B2>
- File C Append <C0>

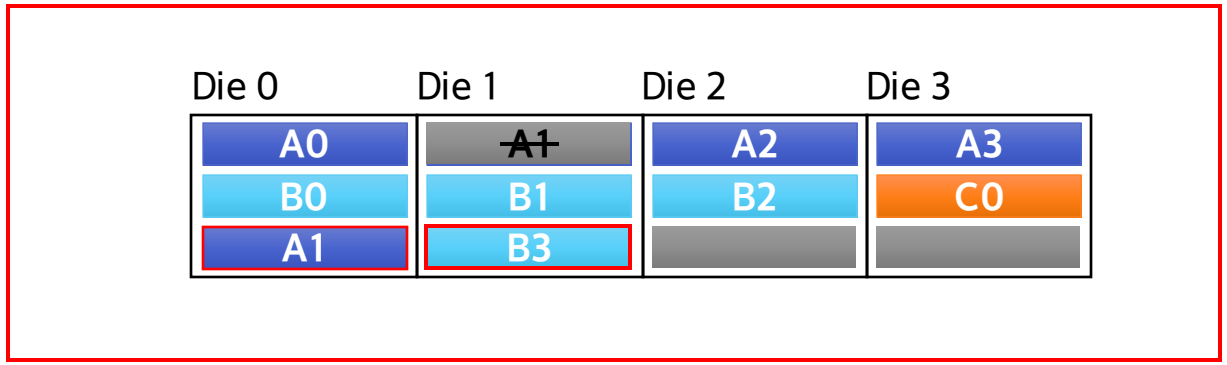


Misaligned Die Allocation from Overwrites



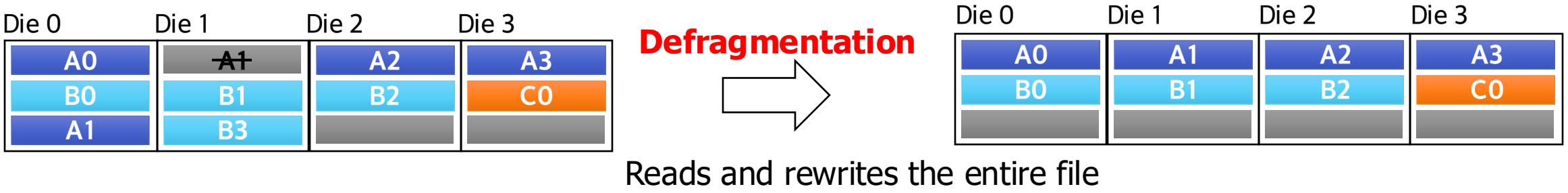
- File A Append <A0, A1, A2, A3>
- File B Append <B0, B1, B2>
- File C Append <C0>

- File A Overwrite <A1>
- File B Append <B3>



Conventional Approach

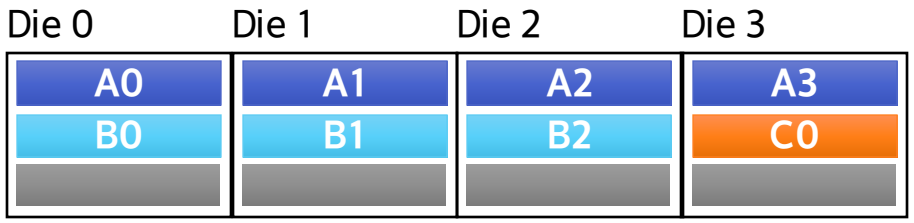
- Appends and overwrites lead to misaligned die allocation
- While defragmentation addresses this issue, it incurs significant costs



Reads and rewrites the entire file

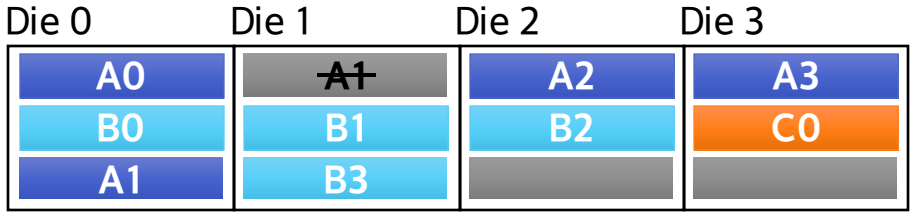
Our Approach

- Prevents misaligned allocation on the fly



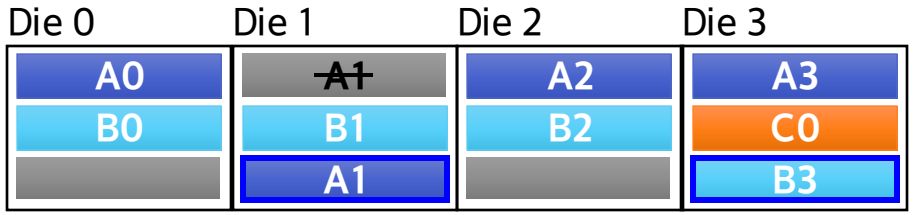
- File A Append <A0, A1, A2, A3>
- File B Append <B0, B1, B2>
- File C Append <C0>

AS-IS



TO-BE

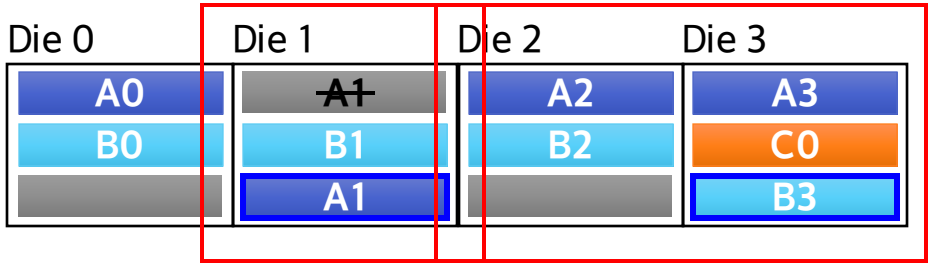
- File A Overwrite <A1>
- File B Append <B3>



Our Approach

- Prevents misaligned allocation on the fly

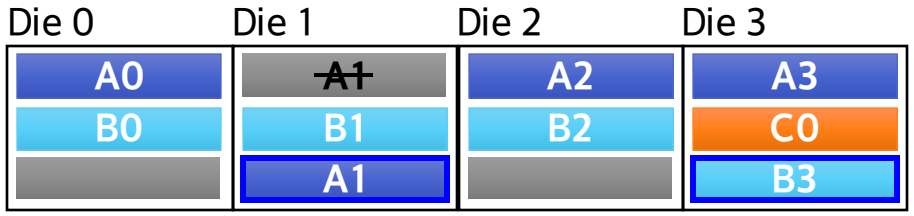
```
- File A Overwrite <A1>
- File B Append <B3>
```

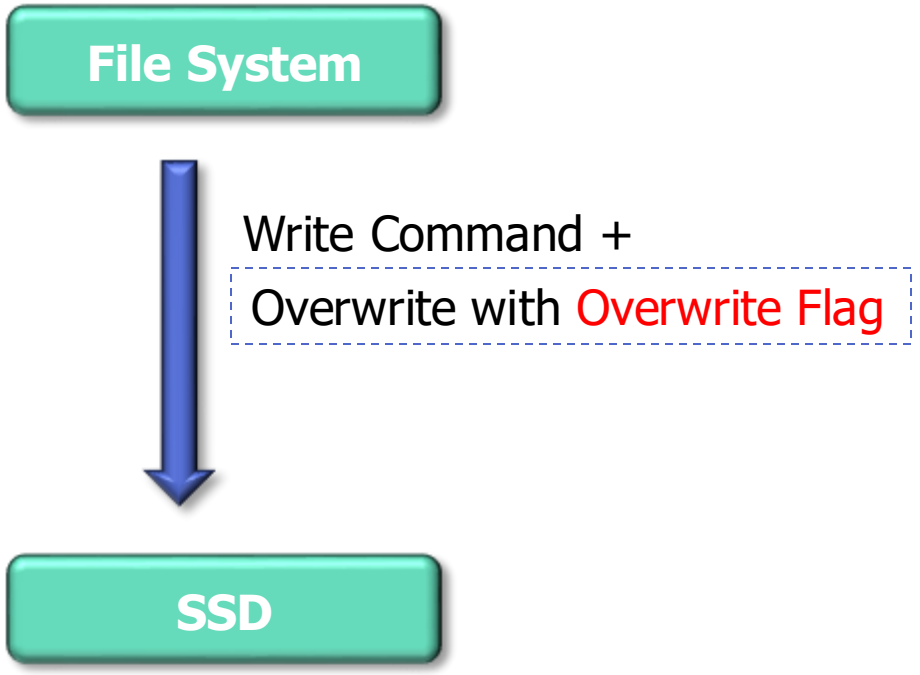


Our Approach

- File system provides file information to the SSD
- Overwrite to same die

- File A Overwrite <A1> (with OW flag)

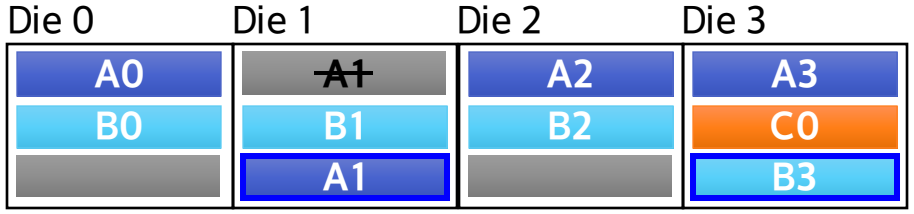


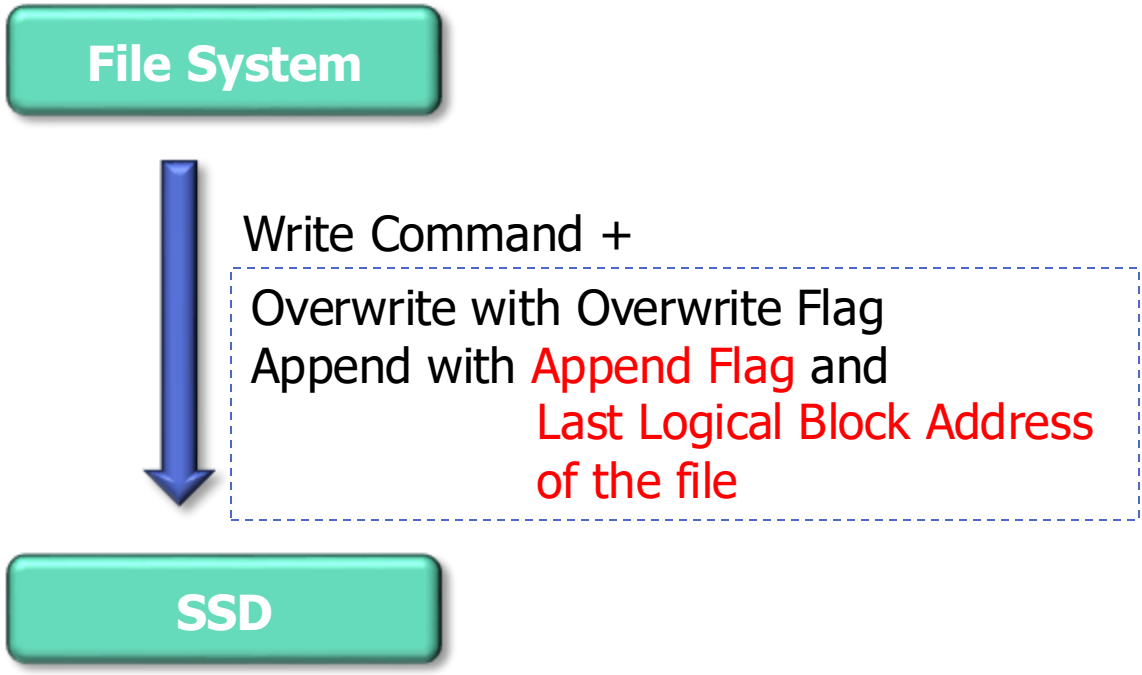


Our Approach

- File system provides file information to the SSD
- Overwrite to same die
- Append to the die next to last written one

```
- File A Overwrite <A1> (with OW flag)
- File B Append <B3> (with AP flag, B2)
```





Our Approach

- The hints are sent through an unused field of the NVMe write command

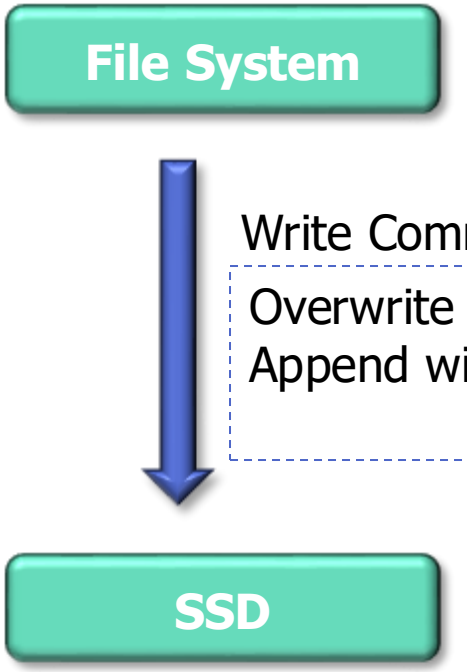


Figure 406: Write – Command Dword 12

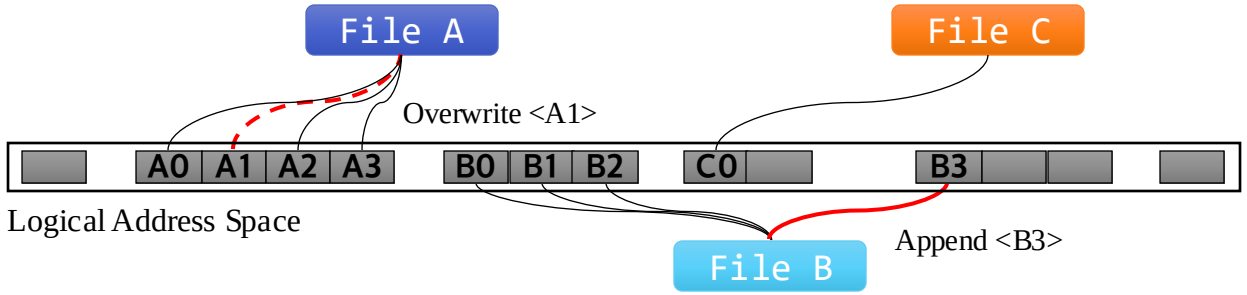
Bits	Description
31	**Limited Retry (LR):** If set to '1', the controller should apply limited retry efforts. If cleared to '0', the controller should apply all available error recovery means to write the data to the NVM.
30	**Force Unit Access (FUA):** If set to '1', then for data and metadata, if any, associated with logical blocks specified by the Write command, the controller shall write that data and metadata, if any, to non-volatile media before indicating command completion. There is no implied ordering with other commands. If cleared to '0', then this bit has no effect.
25:24	Reserved
19:16	Reserved
15:00	**Number of Logical Blocks (NLB):** This field indicates the number of logical blocks to be written. This is a 0's based value.

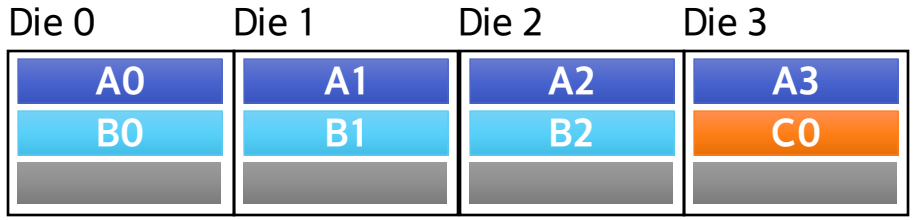
Figure 106: Command Format – Admin and NVM Command Set

Bytes	Description
07:04	**Namespace Identifier (NSID):** This field specifies the namespace that this command applies to. If the namespace identifier is not used for the command, then this field shall be cleared to 0h. The value FFFFFFFFh in this field is a broadcast value (refer to section 6.1), where the scope (e.g., the NVM subsystem, all attached namespaces, or all namespaces in the NVM subsystem) is dependent on the command. Refer to Figure 141, Figure 142, and Figure 350 for commands that support the use of the value FFFFFFFFh in this field. Specifying an inactive namespace identifier (refer to section 6.1.4) in a command that uses the namespace identifier shall cause the controller to abort the command with status Invalid Field in Command, unless otherwise specified. Specifying an invalid namespace identifier (refer to section 6.1.2) in a command that uses the namespace identifier shall cause the controller to abort the command with status Invalid Namespace or Format, unless otherwise specified. If the namespace identifier is used for the command (refer to Figure 141 and Figure 142), the value FFFFFFFFh is not supported for that command, and the host specifies a value of FFFFFFFFh, then the
15:08	Reserved
	implementations.

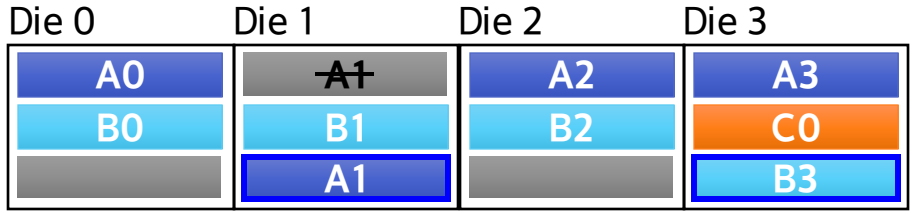
Our Approach

- Prevents misaligned allocation on the fly





- File A Append <A0, A1, A2, A3>
- File B Append <B0, B1, B2>
- File C Append <C0>



- File A Overwrite <A1> (with OW flag)
- File B Append <B3> (with AP flag, B2)

Evaluation

- Environment

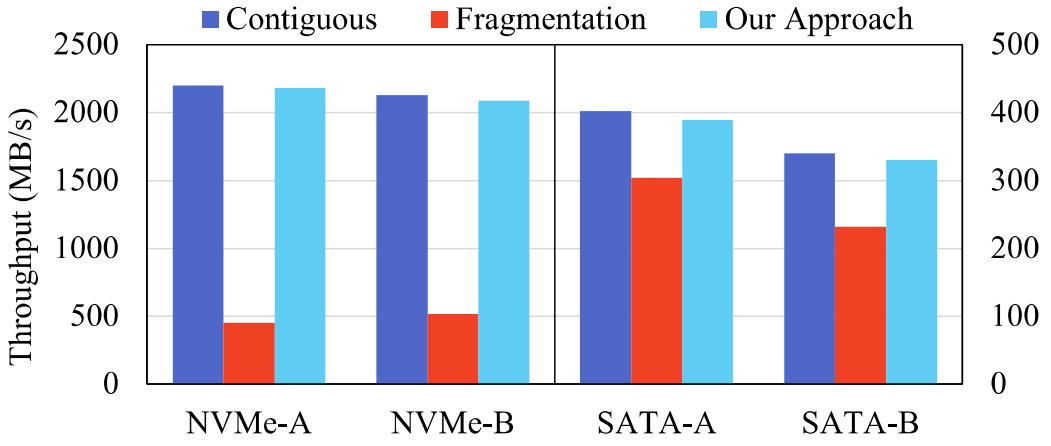
System Configuration	
Processor	Intel Xeon Gold 6138 2.0 GHz, 160-Core
Chipset	Intel C621
Memory	DDR4 2666 MHz, 32 GB x16
OS	Ubuntu 20.04 Server (kernel v5.15.0)
File system	Ext4

NVMeVirt Emulator [22]	
Interface	PCIe Gen 3 x4
Capacity	60 GB
Channel Count	4
Dies per Channel	2
Read/Write Unit Size	32 KB
Read Time	36 μs
Write Time	185 μs

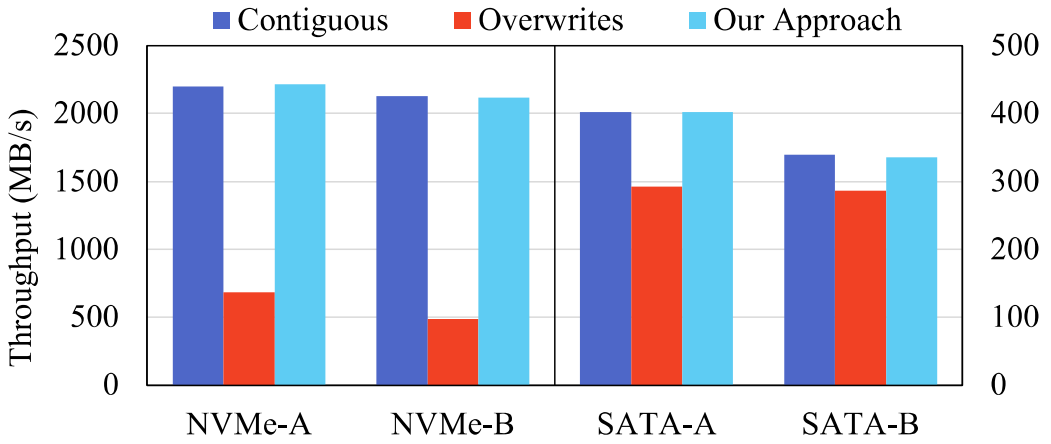
- Our approach was validated using commodity SSDs
- Evaluation our approach with SSD emulator
  - Modified Ext4 and NVMe driver to transmit info through NVMe Write Command
  - NVMeVirt adjusts die allocation policy using this information

Evaluation

- Reading 8MB file
  - Written by 32KB x 256 times
- Commodity SSDs
  - Samsung 980 Pro 1TB – NVMe-A
  - WD Black SN850 1TB – NVMe-B
  - Samsung 870 Evo 500GB – SATA-A
  - WD Blue SA510 500GB– SATA-B



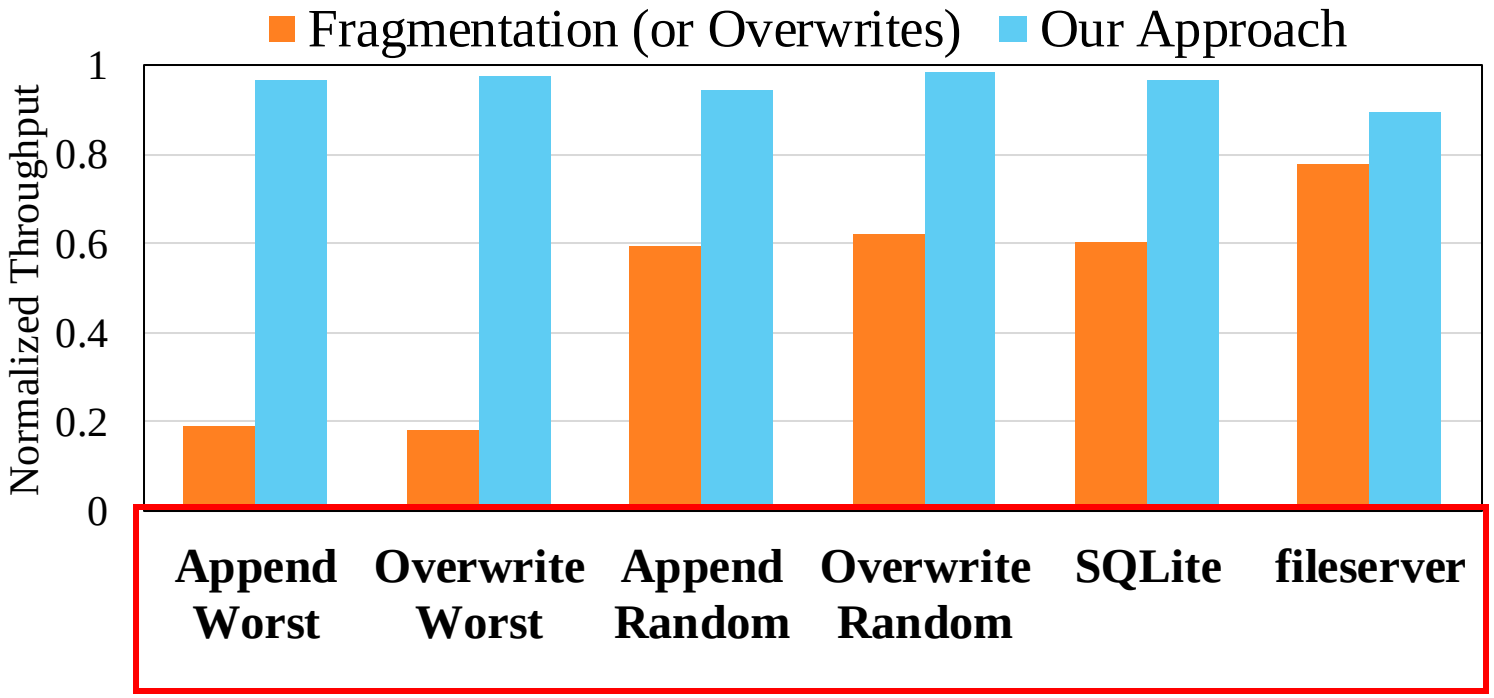
(a) Append Write



(b) Overwrite

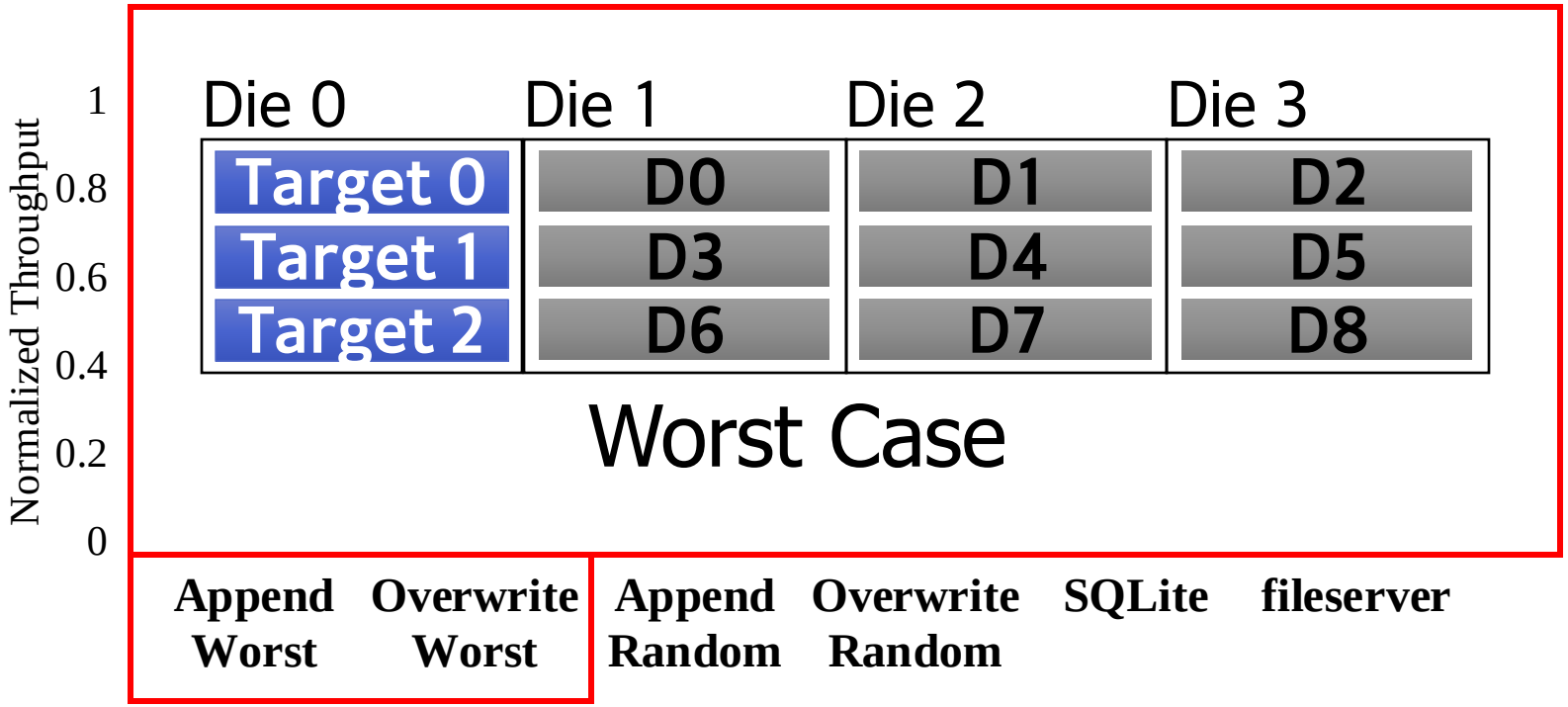
Evaluation

- Hypothetical workloads, SQLite and Filebench
  - SQLite: 16KB records x 10,000 while writing 100KB chunks to dummy files
    5,005 DoF
  - Fileserver: 10 threads perform 32 KB rand. appends for 1 Min, and
    seq. reads over 128KB x 10,000 files



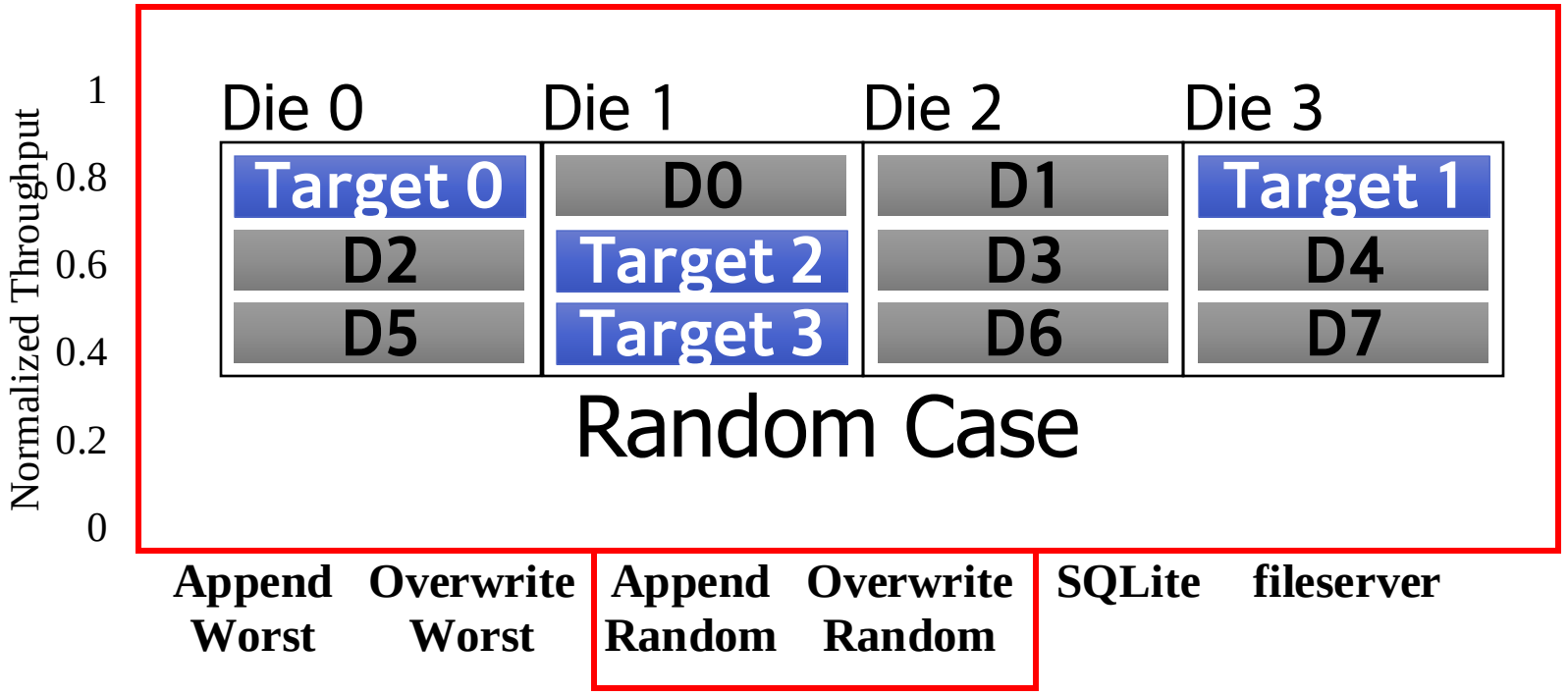
Evaluation

- Worst case of hypothetical workloads



Evaluation

- Random case of hypothetical workloads

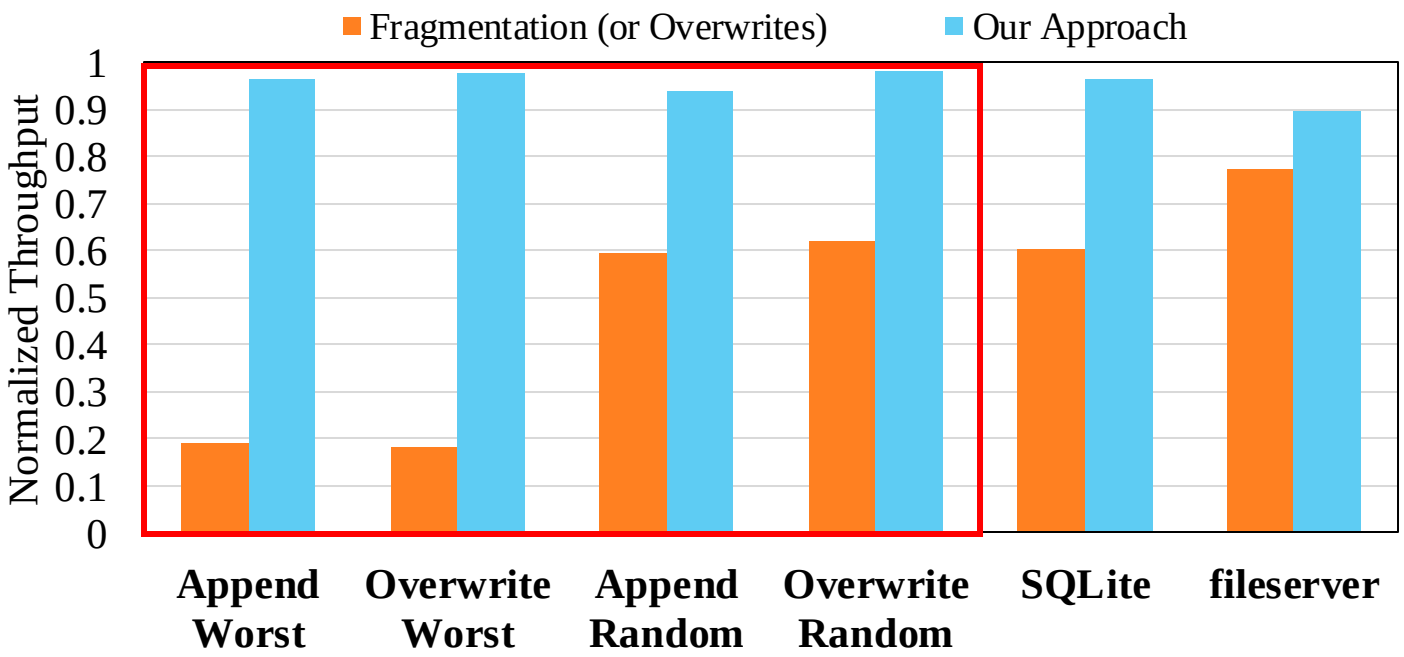


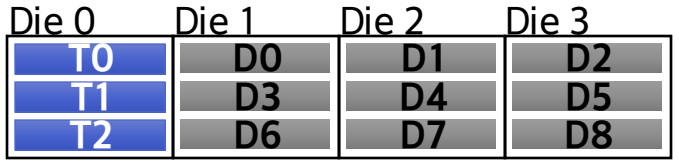
Evaluation

- In the worst case, 20% of contiguous file's
- In the random case, 60% of contiguous file's
- In SQLite, 60% of contiguous file's
- In fileserver, 77% of contiguous file's

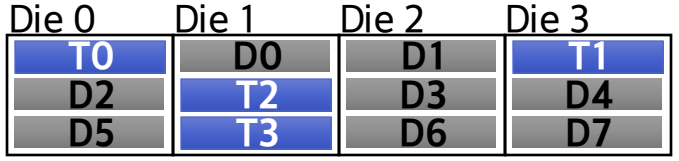
➔ Improved to within 6%.

➔ Improved to within 10%.





Worst Case



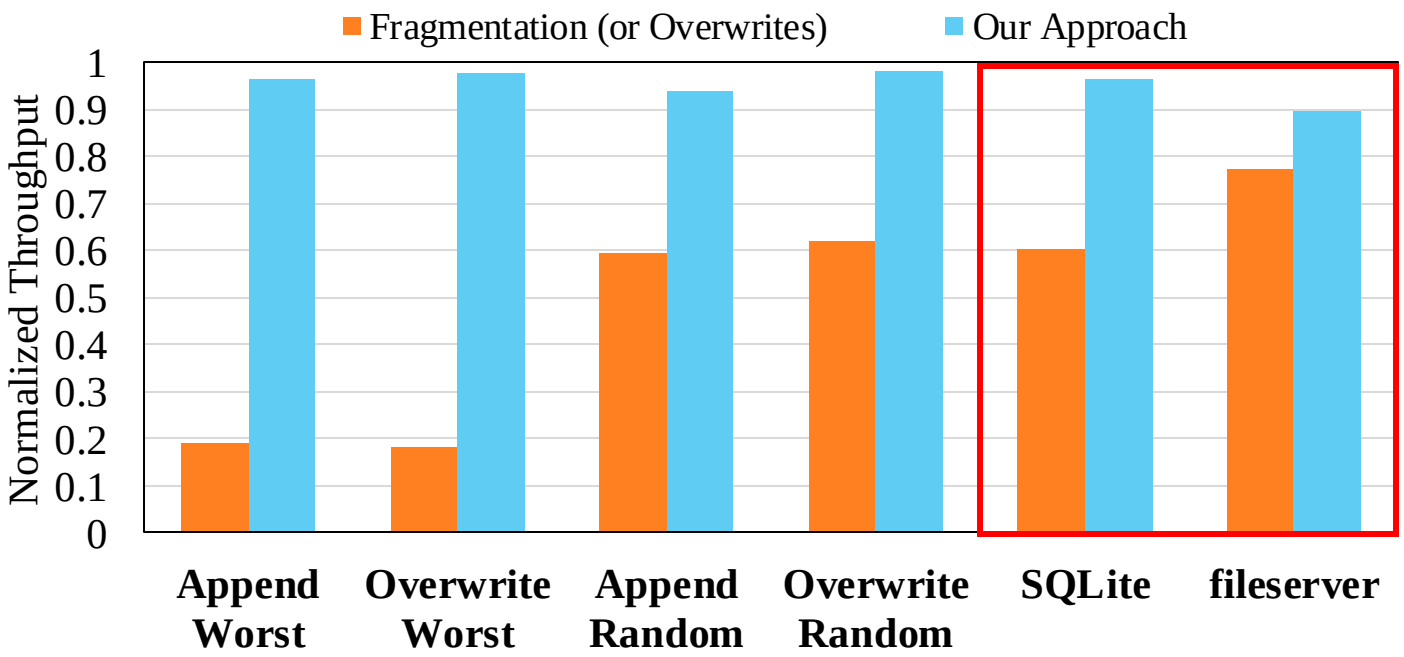
Random Case

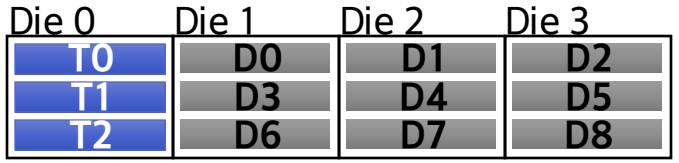
Evaluation

- In the worst case, 20% of contiguous file's
- In the random case, 60% of contiguous file's
- In SQLite, 60% of contiguous file's
- In fileserver, 77% of contiguous file's

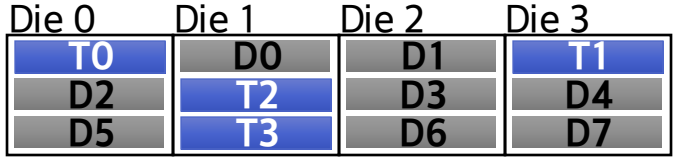
➔ Improved to within 6%.

➔ Improved to within 10%.





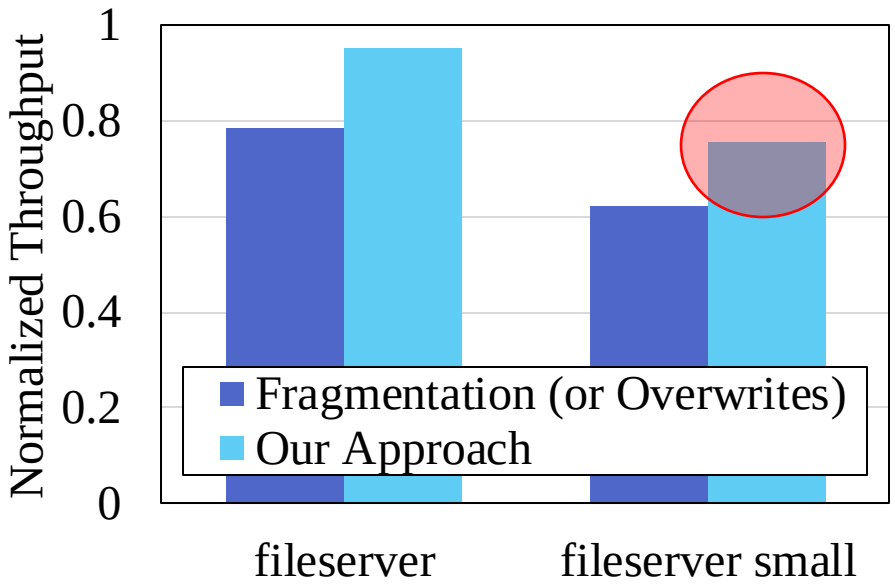
Worst Case



Random Case

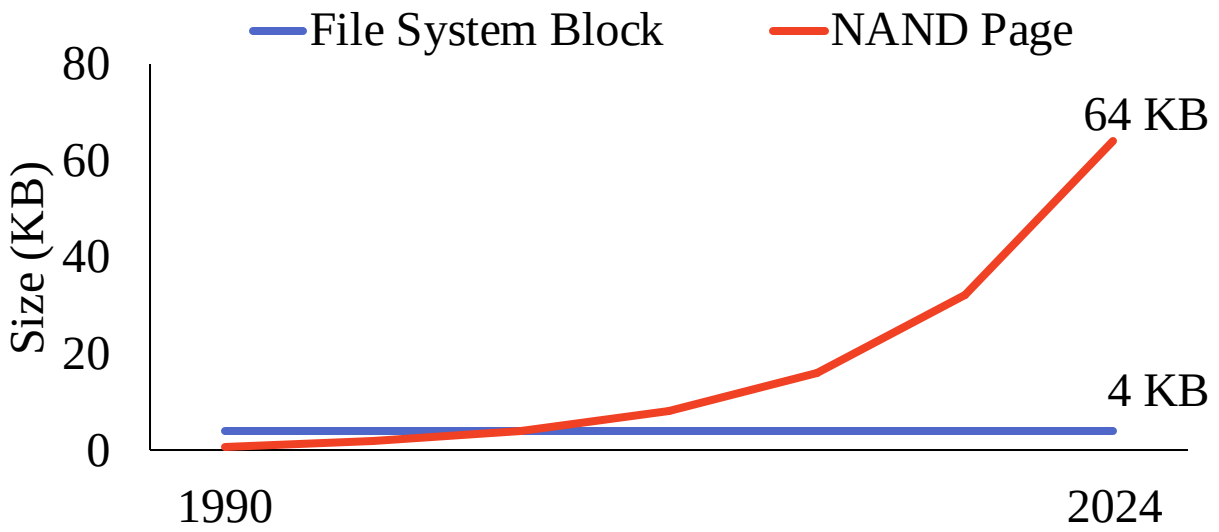
Issues Occurring When Write Sizes are Small

- Discovered a performance issue when the write chunk size is small



Page Bloating Caused by Small Writes

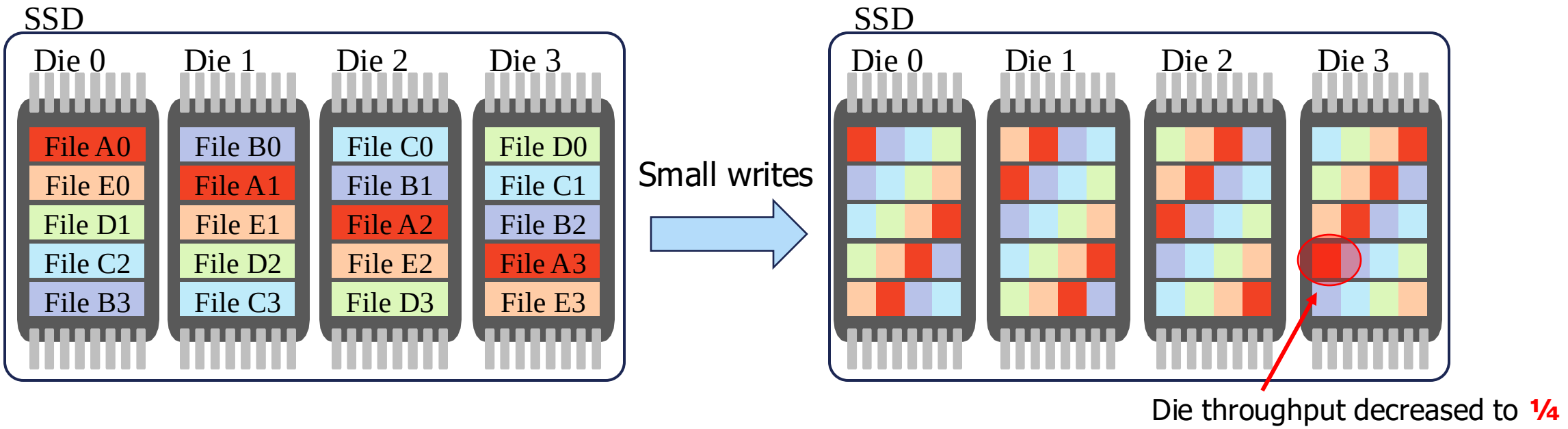
- Due to cost and performance considerations, the size of NAND pages is increasing
  - Originally functioning at 512 bytes[1], they are now up to 64 KB[2] and continue to grow
- In contrast, the file system block size has not increased in over 20 years[3]



1. Suh *et al*. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. IEEE Journal of Solid-State Circuits, 1995
2. Kim *et al*. A 1Tb 3b/Cell 8th-generation 3D-NAND flash memory with 164MB/s write throughput and a 2.4Gb/s interface. IEEE International Solid-State Circuits Conference, 2022
3. Tweedie *et al*. Journaling the linux ext2fs filesystem. In Proceedings of The Fourth Annual Linux Expo. Durham, North Carolina, 1998.

Page Bloating Caused by Small Writes

- Writing multiple files smaller than NAND page size simultaneously can cause bloating across several NAND pages
- Inefficiency results from size mismatch between SSD's page and file system's block



Conclusion

- We identify the true cause of performance degradation due to file fragmentation
  - Request splitting overhead is concealed in a multi-queue environment
  - Primary cause is *read collisions due to misaligned die allocation*

- We proposed an approach to mitigate the misalignment
  - By providing filesystem information to the SSD,
    it maintains the proper die allocations even under adverse conditions
  - Addressing not only append write cases, but also overwrite cases

Thank you

Q&A

SUNGKYUNKWAN UNIVERSITY 성균관대학교

성균관대학교