USENIX ATC 2024

ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs

Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li
Chenxi Wang, Bo Mao, Myoungsoo Jung, Jie Zhang

Why We Need All-Flash Array (AFA)?

DELL EMC

DELL EMC VMAX

PURESTORAGE

PureStorage FlashArray

AFA → Superb **performance** & **reliability**!

Datacenters

Supercomputers

FUJITSU

FUJITSU ETERNUS

NetApp

NetApp AFF

All-flash array are widely adopted in diverse domains.

Background: Evolvement of SSD and AFA

Continual advancement in SSD performance.

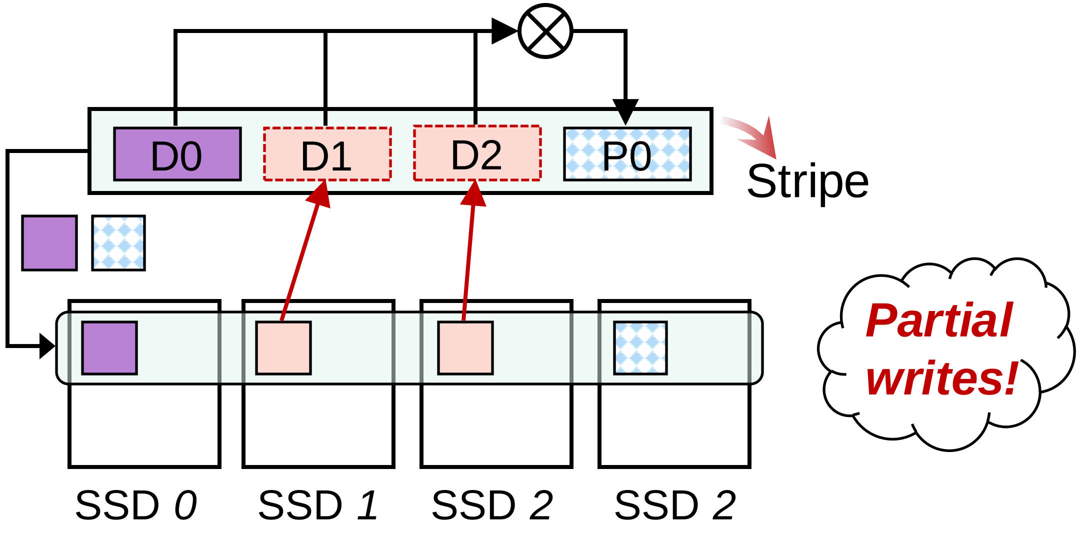
SATA SSD: 500 MB/s

PCIe5 SSD: 13 GB/s

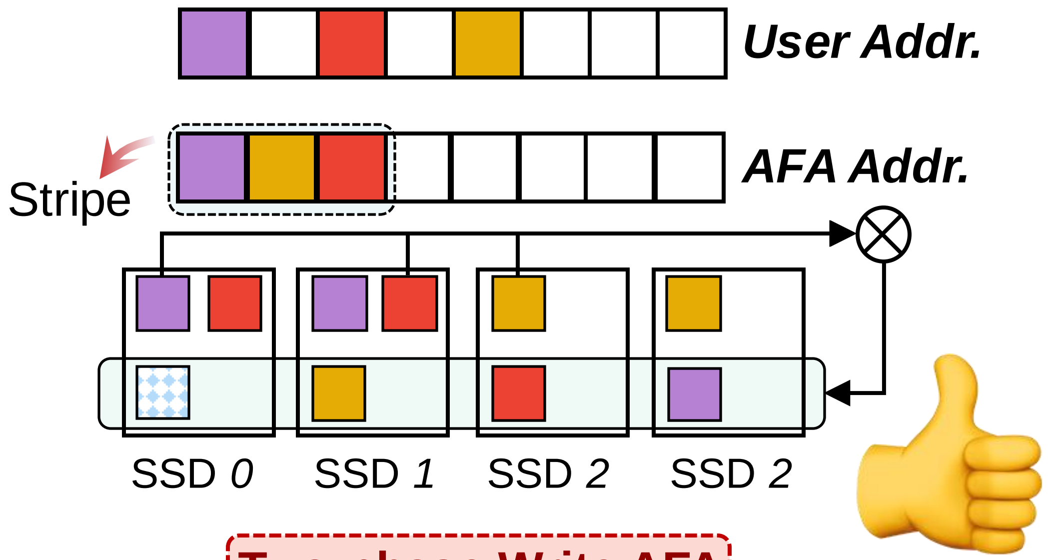
How can AFA fully leverage high-performance SSDs?

Background: Evolvement of SSD and AFA

- **Stripe write AFA:** mdraid [Linux], ScalaRAID [HotStorage'22], stRAID [ATC'22]
  - Partial writes issue: read-construct-write → significantly delay the I/O completion time
- **Two-phase write AFA:** LDM [TOS'16], FusionRAID [FAST'21]
  - Replication as the prelude of striping, out-of-place update



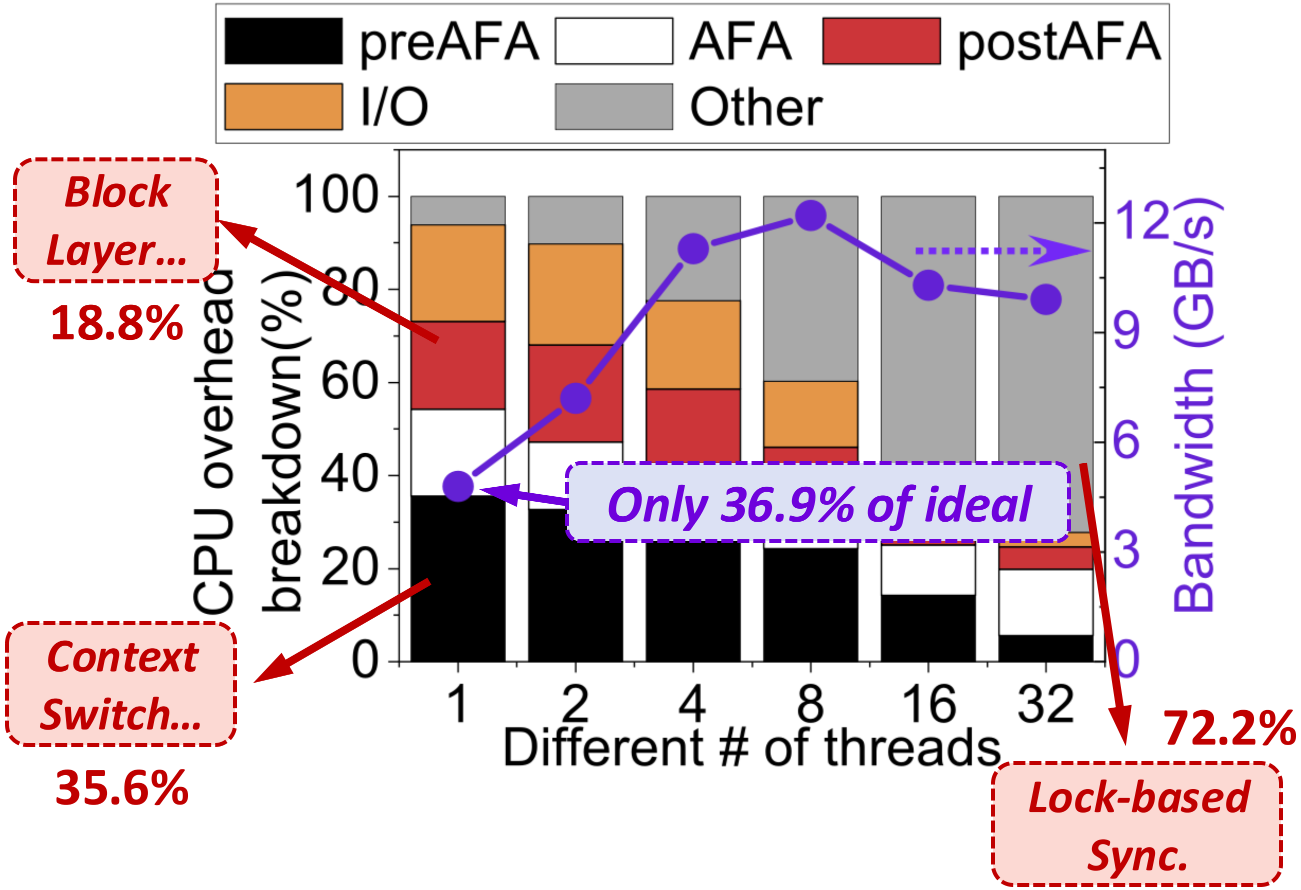
Stripe Write AFA



Two-phase Write AFA

Challenge in Replication Phase

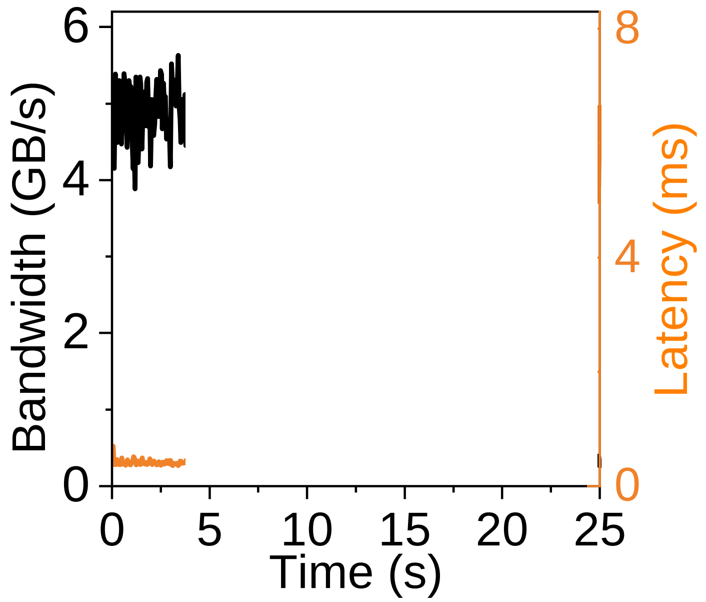
- **Only 36.9% of ideal with 1 thread**
  - I/O only accounts for 20.8%
  - block layer: 18.8%, context switch: 35.6%
- **Not scalable with more threads**
  - Lock issue: 72.2%
  - Real AFA: CPU:SSD ≈ 1:3



Challenge 1

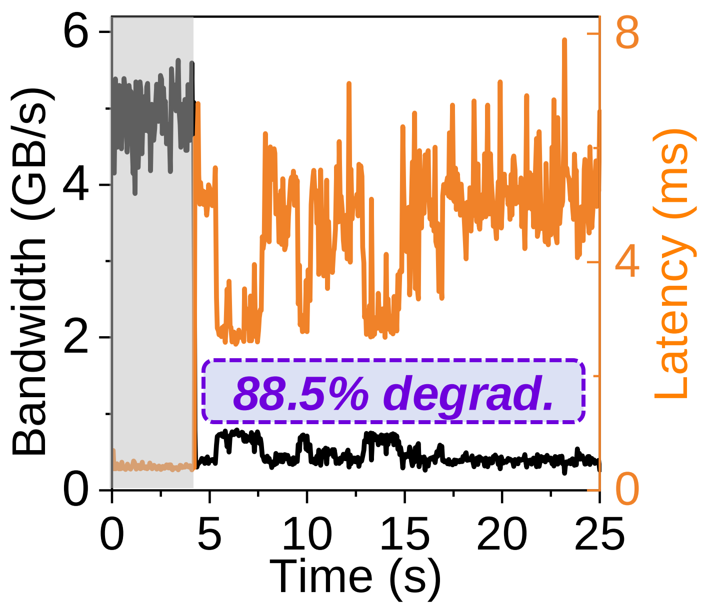
Software overhead is the bottleneck for achieving high performance.

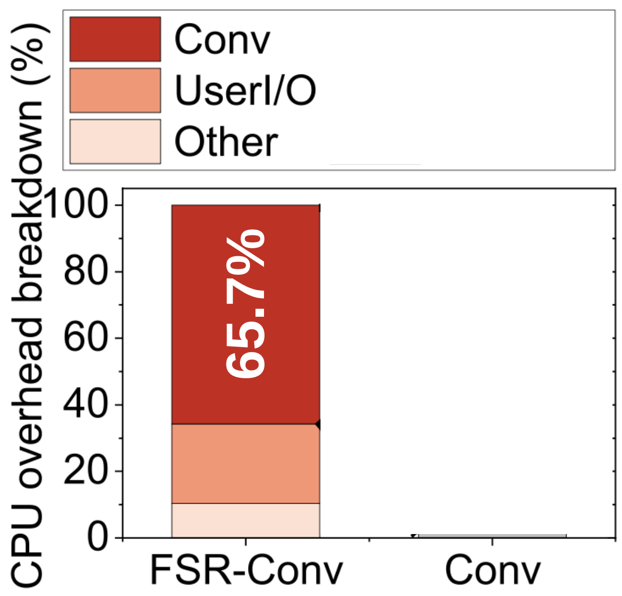
Challenge in Conversion Phase



Challenge in Conversion Phase

88.5% throughput ↓ & 13.7x latency ↑ in conversion phase. ☹



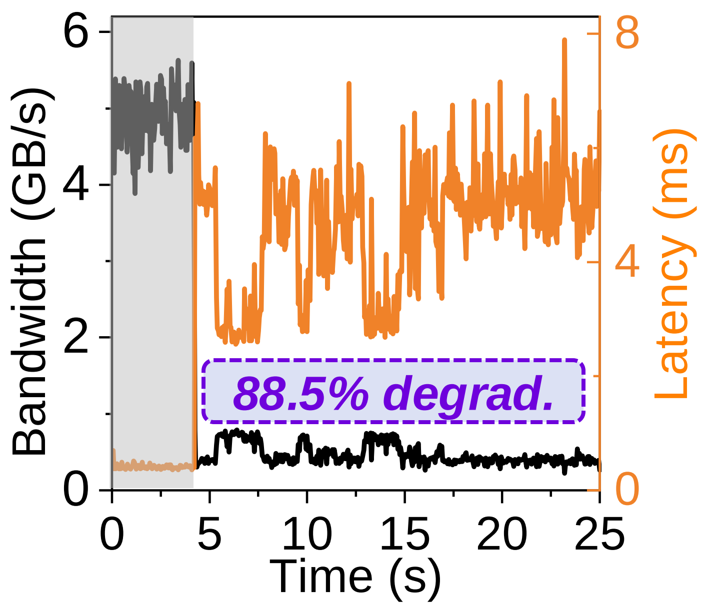


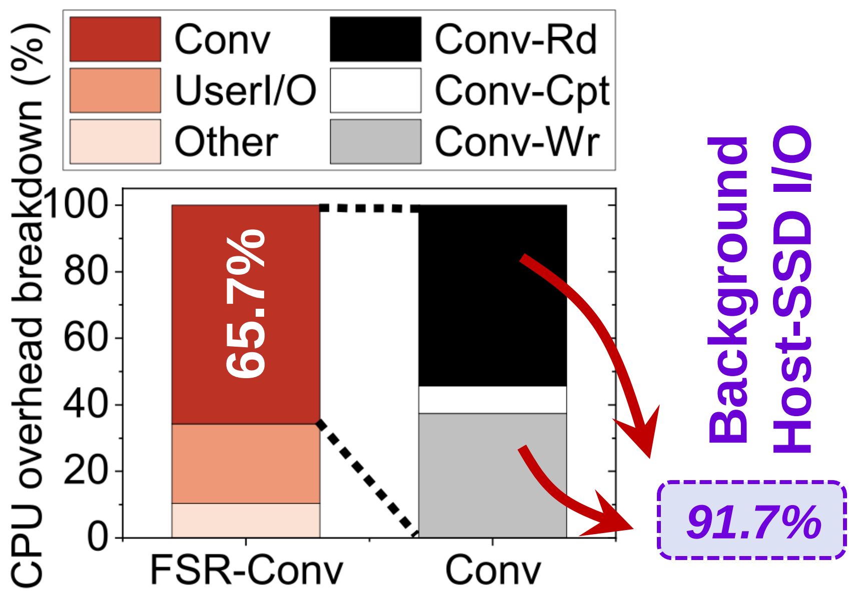
Challenge in Conversion Phase

88.5% throughput ↓ & 13.7x latency ↑ in conversion phase.

Background parity generation, especially host-SSD I/O.

Challenge 2





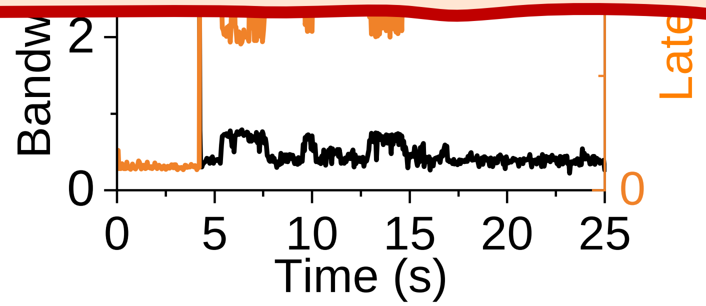
Challenge in Conversion Phase

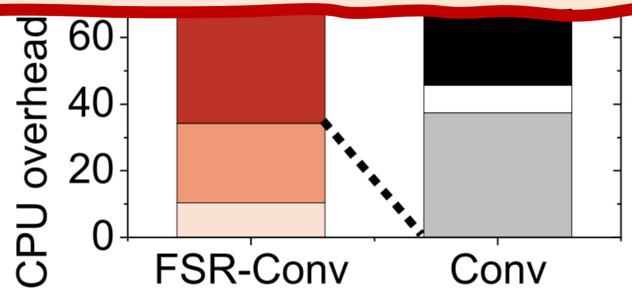
88.5% **throughput ↓ &** **13.7x** **latency ↑ in conversion phase.**

Other intrinsic issues of two-phase write

Challenge 3 **Out of Place → Mapping → Crash Consist. Cost**

Challenge 4 **Replicating + Striping → Write Amp. → Lifetime↓**





Our Solution: ScalaAFA

- ✓ **Embracing user-space storage stack to lighten software overhead**
  - Adopt SPDK to take advantage of its high-performance storage stack
  - Enable lock-free multi-thread access with message-passing mechanism

  Challenge 1

- ✓ **Harnessing SSD-internal hardware resources for parity generation**
  - Employ a novel data placement policy to curtails background I/O
  - Leverage the SSD built-in XOR engine to calculate the parity codes in situ

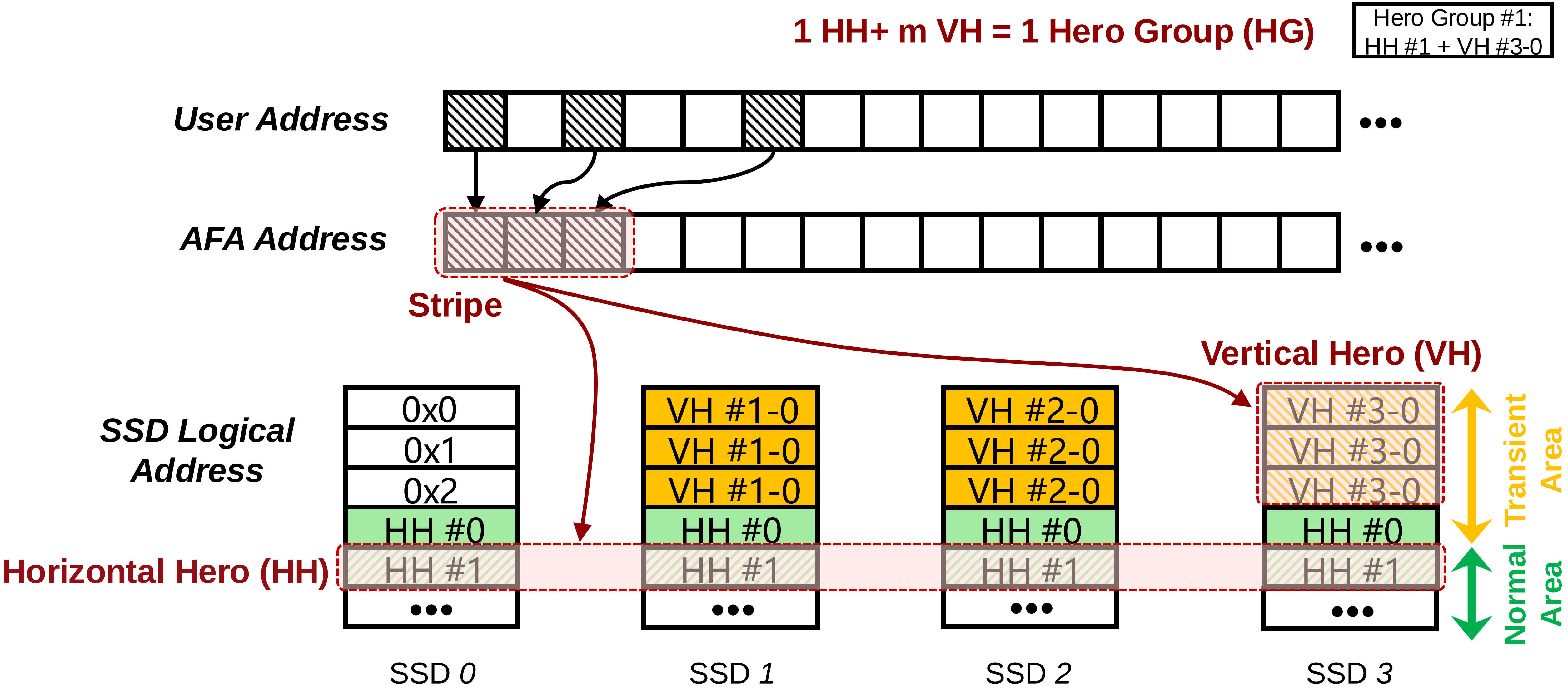
  Challenge 2

- ✓ **Enjoying SSD architectural innovations to tackle the intrinsic issues**
  - Store sliced mapping tables in SSD OOB for low-cost crash consistency
  - Avoid flushing transient replicas from SLC to the vulnerable MLC blocks*

  Challenge 3&4

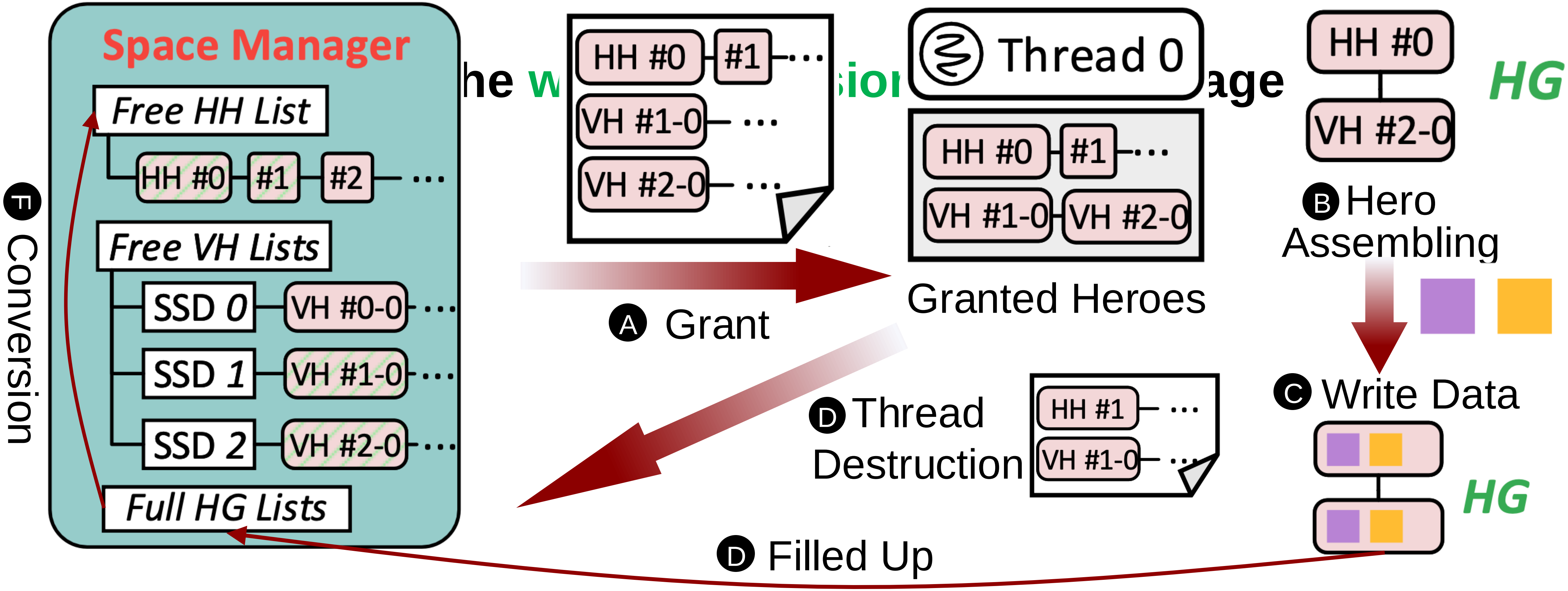
*Please refer to our paper for more details.

Storage Space Abstraction of ScalaAFA



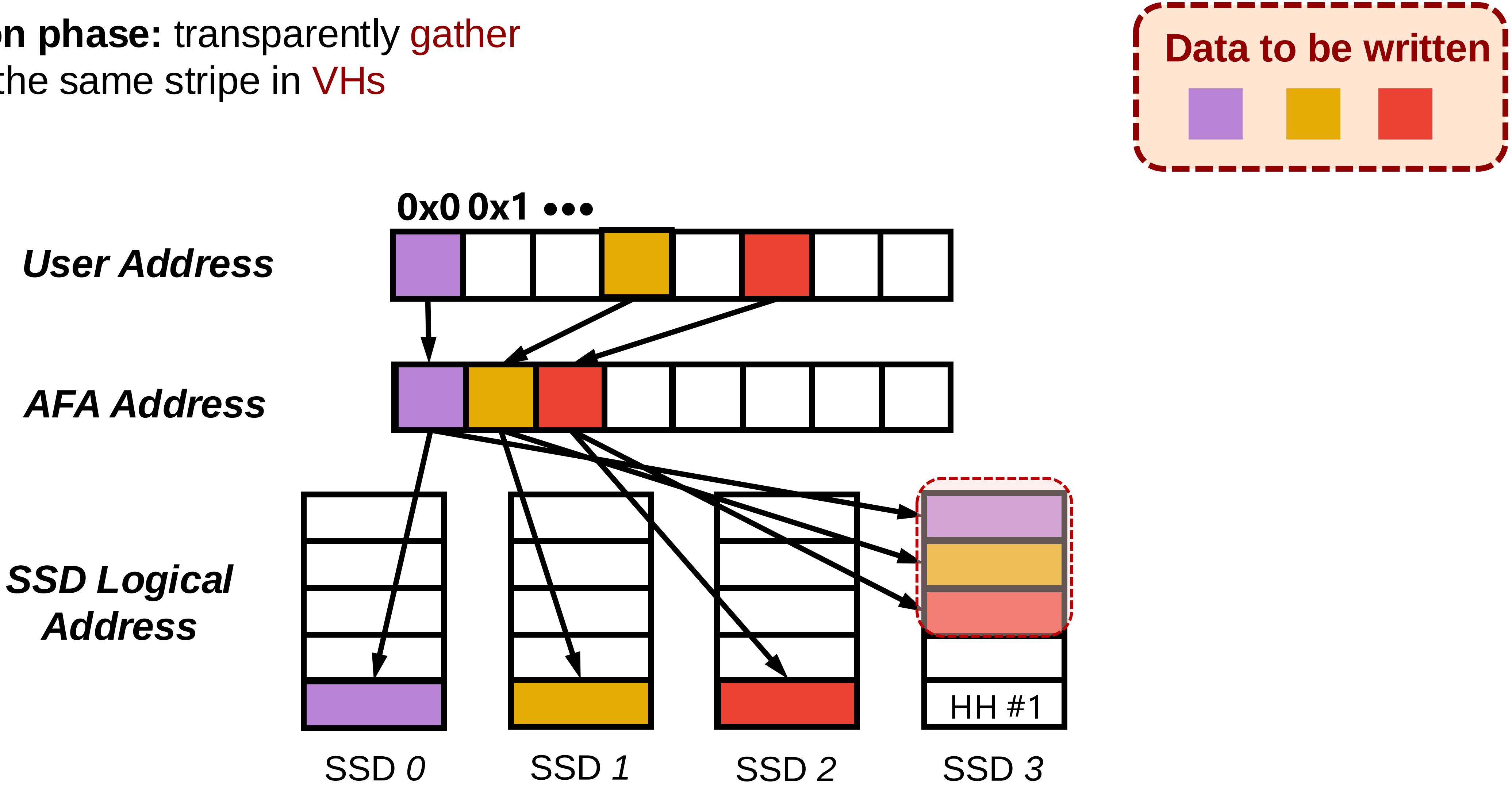
Enable Lock-free Multi-Thread Access

Prohibit threads from placing data on the same SSD address.



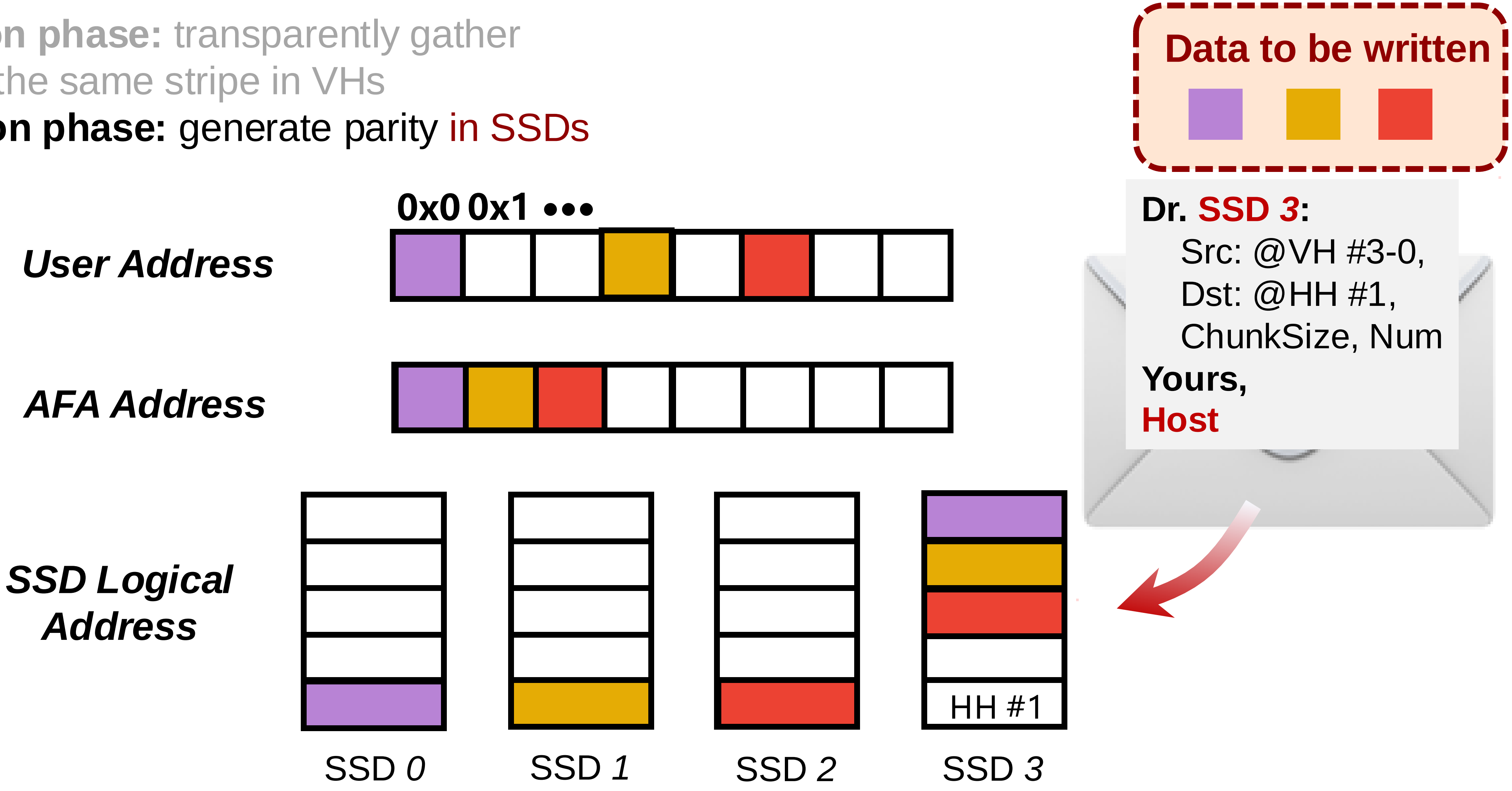
Evolve the Write Path: Data Placement Policy

1. **Replication phase:** transparently gather chunks of the same stripe in VHs



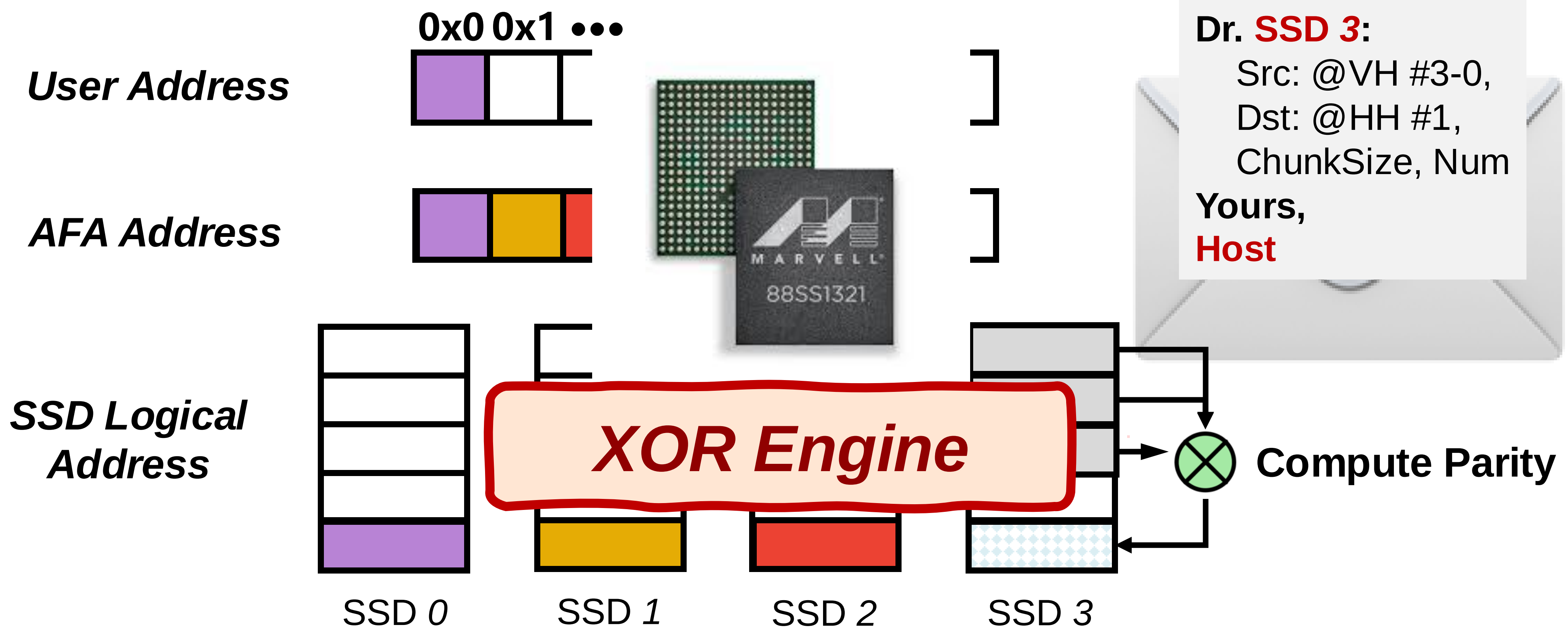
Evolve the Write Path: Conversion Offloading

1. **Replication phase:** transparently gather chunks of the same stripe in VHs
2. **Conversion phase:** generate parity in SSDs



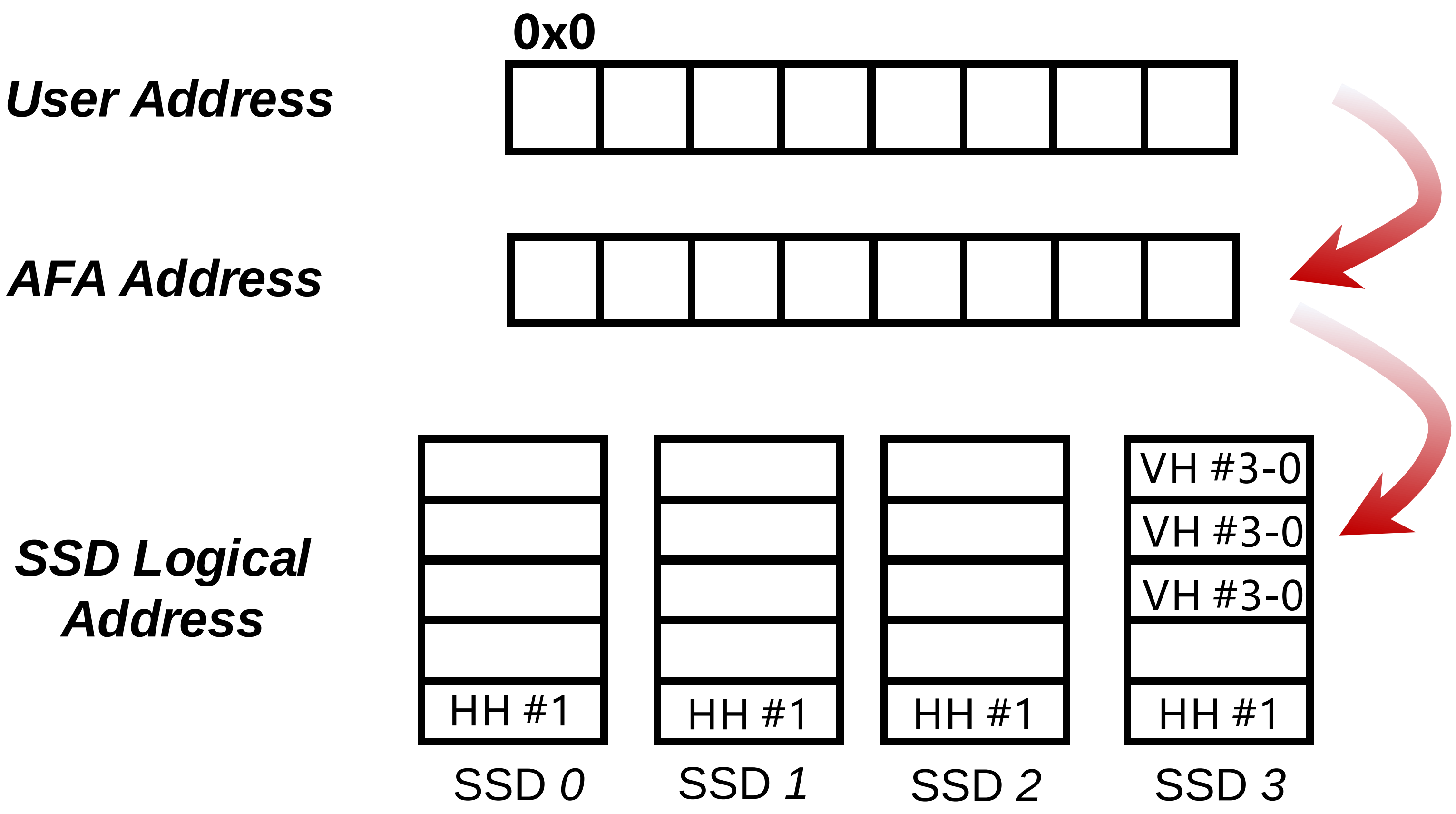
Evolve the Write Path: Conversion Offloading

1. **Replication phase:** transparently gather chunks of the same stripe in VHs
2. **Conversion phase:** generate parity in SSDs



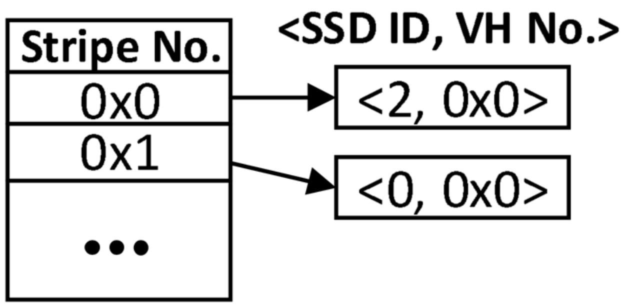
Persist the Mapping Table

How to persist mapping tables with low cost?



User Chunk No.	AFA Chunk No.
0x0	0x0
0x1	0x1
0x2	NULL
0x3	0x2
•••	•••

AFA Mapping Table



Hero Group Mapping Table

Persist the Mapping Table: SSD OOB

- **Key insight:** flash page and its OOB can be written with one program operation
  1. Convert mapping tables to a segmentable data structure
  2. Slice persistent mapping table based on SSD LPN

User Chunk No.	AFA Chunk No.
0x0	0x0
0x1	0x1
0x2	NULL
0x3	0x2
•••	•••

AFA Mapping Table

Stripe No.	<SSD ID, VH No.>
0x0	<2, 0x0>
0x1	<0, 0x0>
•••	

Hero Group Mapping Table

User Chunk No.	Slot	SSD LPN
0x0	0	0x20
0x1	1	0x20
0x3	0	0x40
•••	•••	•••

Persistent Mapping Table

Persist the Mapping Table: SSD OOB

- **Key insight:** flash page and its OOB can be written with one program operation
  1. Convert mapping tables to a segmentable data structure
  2. Slice persistent mapping table based on SSD LPN
  3. Piggyback the sliced metadata in write requests
  4. Persists the metadata to OOB via the same program operation

User Chunk No.	AFA Chunk No.
0x0	0x0
0x1	0x1
0x2	NULL
0x3	0x2
•••	•••

AFA Mapping Table

Stripe No.	<SSD ID, VH No.>
0x0	<2, 0x0>
0x1	<0, 0x0>
•••	

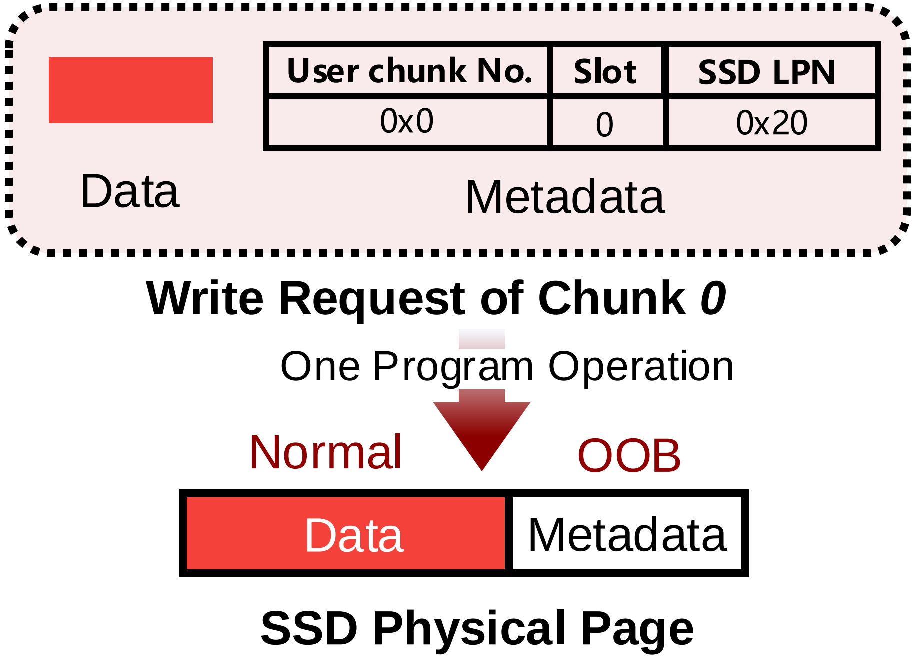
Hero Group Mapping Table

User chunk No.	Slot	SSD LPN
0x0	0	0x20

User chunk No.	Slot	SSD LPN
0x0	1	0x20

User chunk No.	Slot	SSD LPN
0x3	0	0x40

Sliced Metadata



Prototype and Testbed Setup

Component	LOC
SPDK v22.05	6K
FEMU Emulator	1K

Implementation Complexity

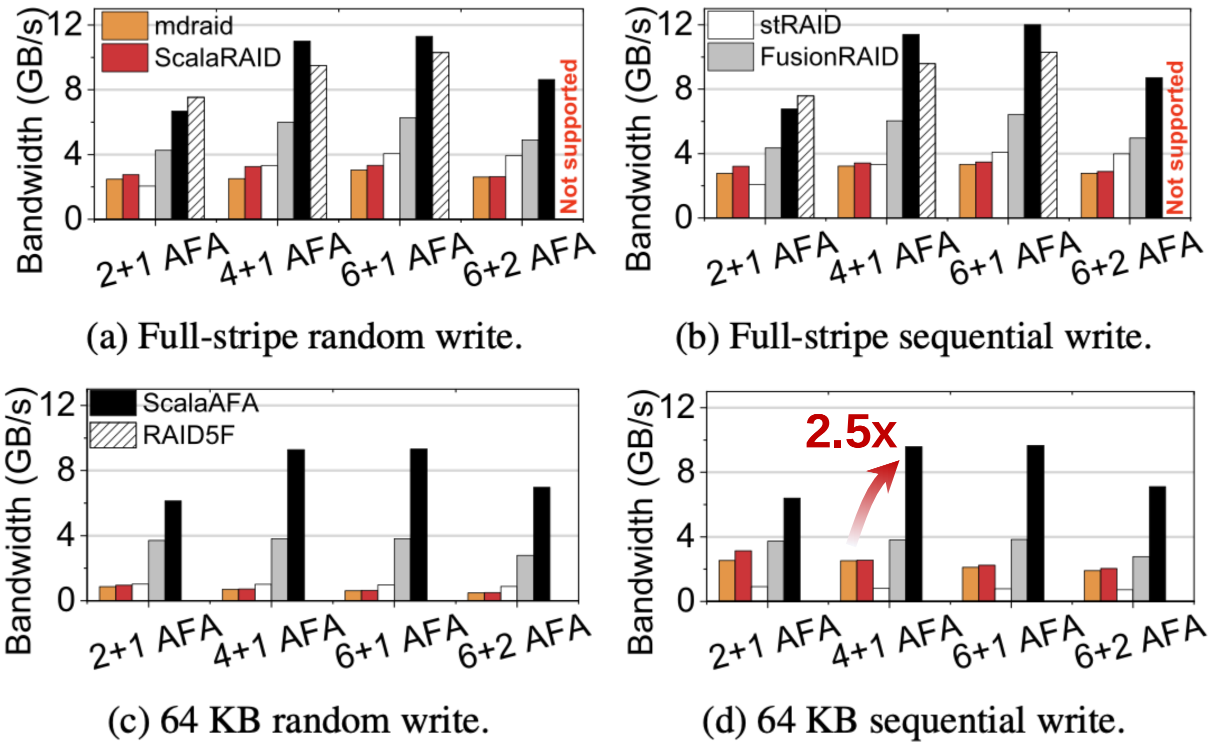
Name	Description
mdraid	Default stripe write AFA of Linux kernel.
ScalaRAID	Mitigates lock overheads of mdraid.
stRAID	Alleviates sync. overheads in mdraid.
RAID5F	Ideal, only serve RAID5 full-stripe I/O.
FusionRAID	SOTA two-phase write AFA engine.

Counterparts

Component	Configuration
CPU	Intel Xeon Gold 5320, 26 cores 2.2 GHz with hyper-threading
Memory	DDR4 3200 MT/s, 8 × 64GB
Real SSD	Up to 8 × Samsung 980 Pro Read/Write : 7000/5200 MB/s
VM	32 CPU cores & 32 GB DRAM
FEMU SSD	8 Channel / 12 Die / 1 Plane 352 Block / 512 Page / 4 KB Read / Write : 7500 / 4890 MB/s
XOR Cost	20 us / 64 KB, 16 mW DP
OS	Ubuntu 20.04 LTS, Linux v5.11.0
Software	fio v3.30, perf v5.11, mdadm v4.1

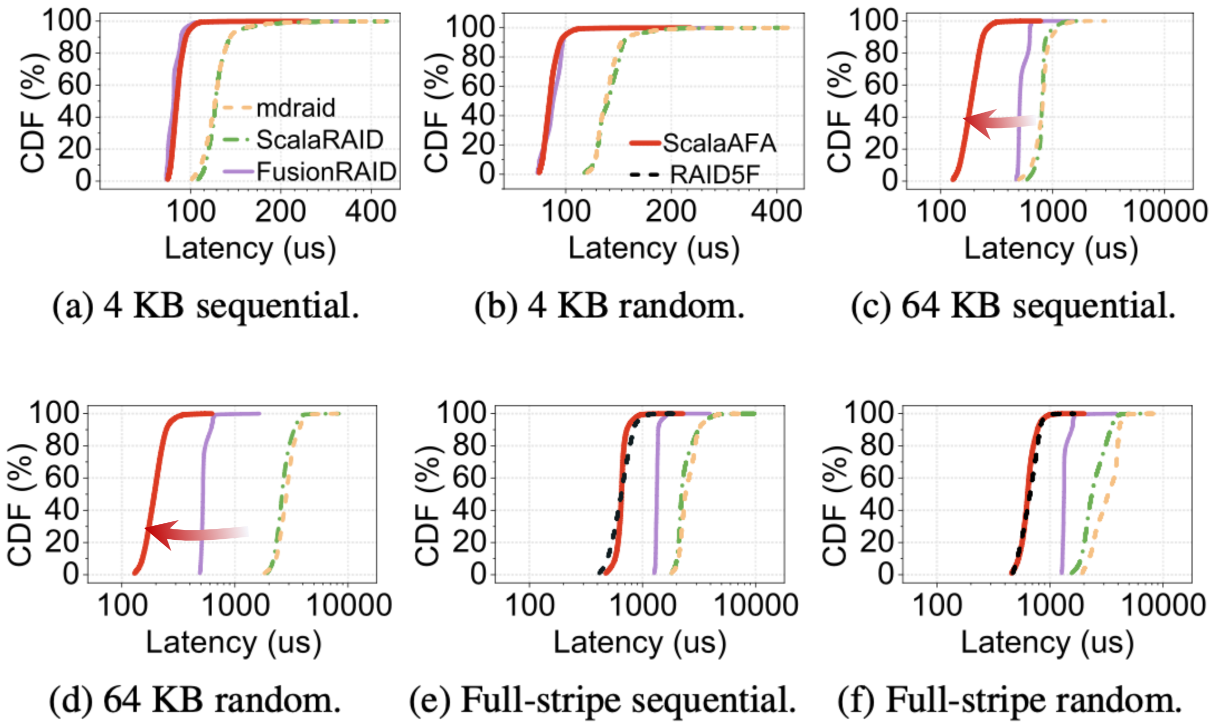
Testbed Configuration

Microbenchmark



(a) Full-stripe random write. (b) Full-stripe sequential write.

(c) 64 KB random write. (d) 64 KB sequential write.



(a) 4 KB sequential. (b) 4 KB random. (c) 64 KB sequential.

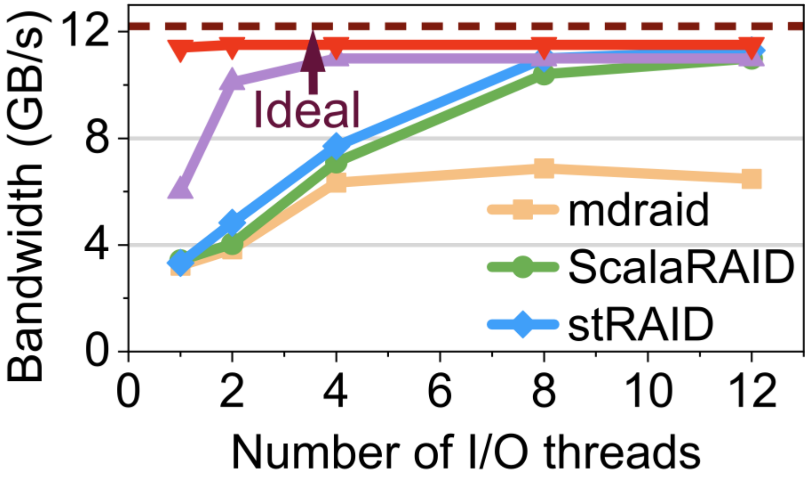
(d) 64 KB random. (e) Full-stripe sequential. (f) Full-stripe random.

Bandwidth

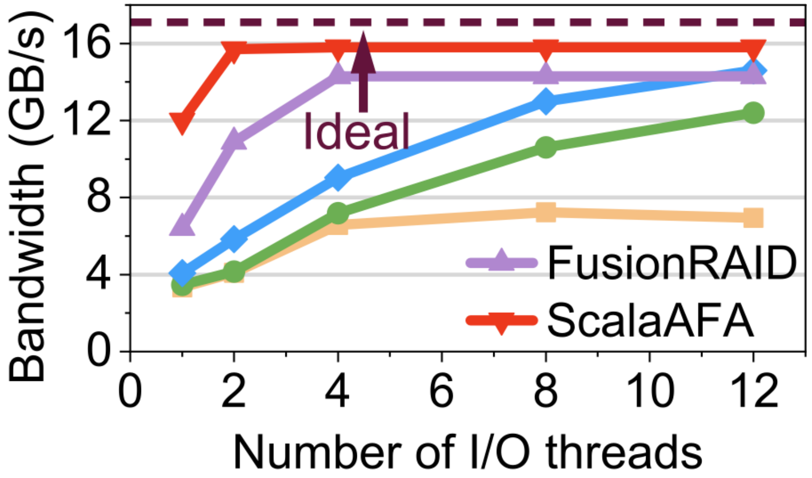
Latency

ScalaAFA improves bandwidth by 2.5x while decreasing average latency by 52.7%!

CPU Overhead



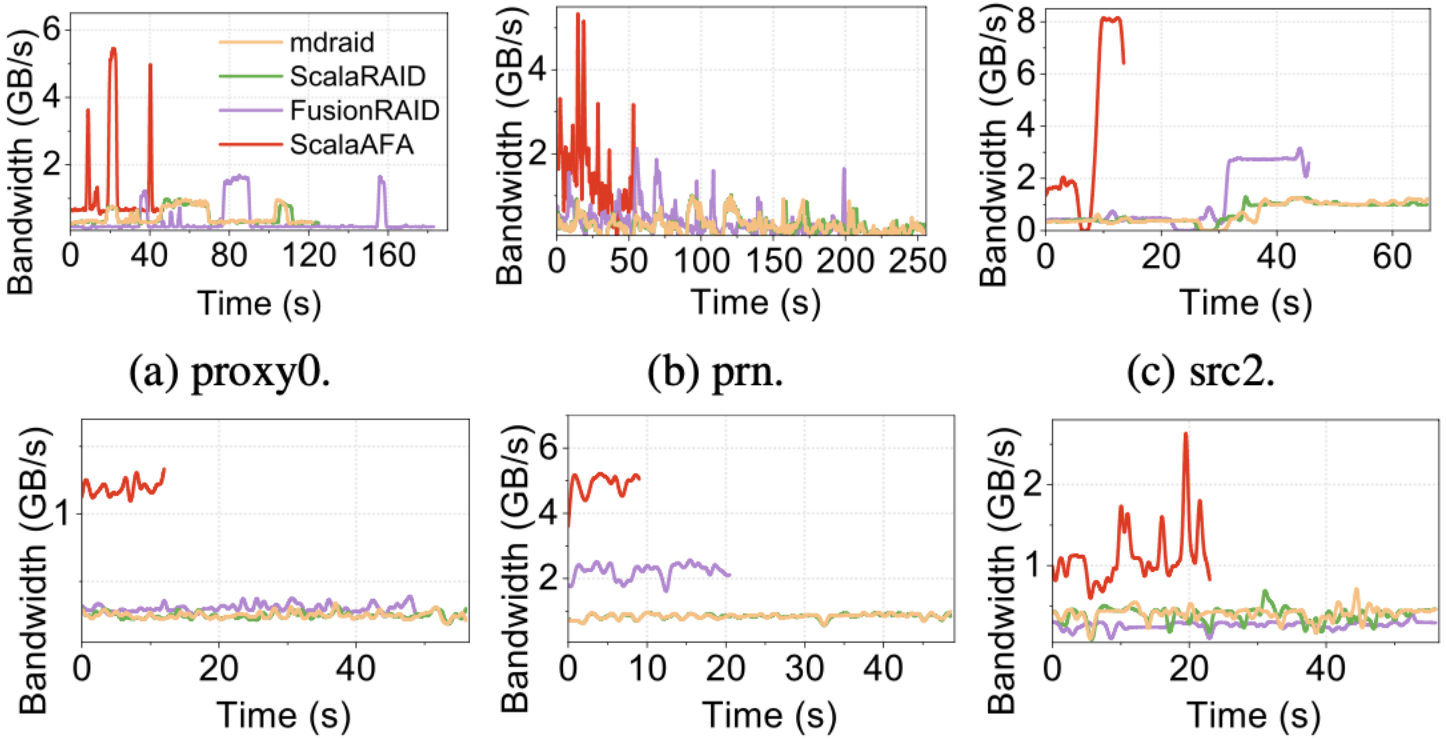
(a) 4+1 AFA.



(b) 6+1 AFA.

ScalaAFA achieves almost the ideal performance with 1/2 threads for 4+1/6+1 AFA (Thread:SSD=1:3)!

Macrobenchmark

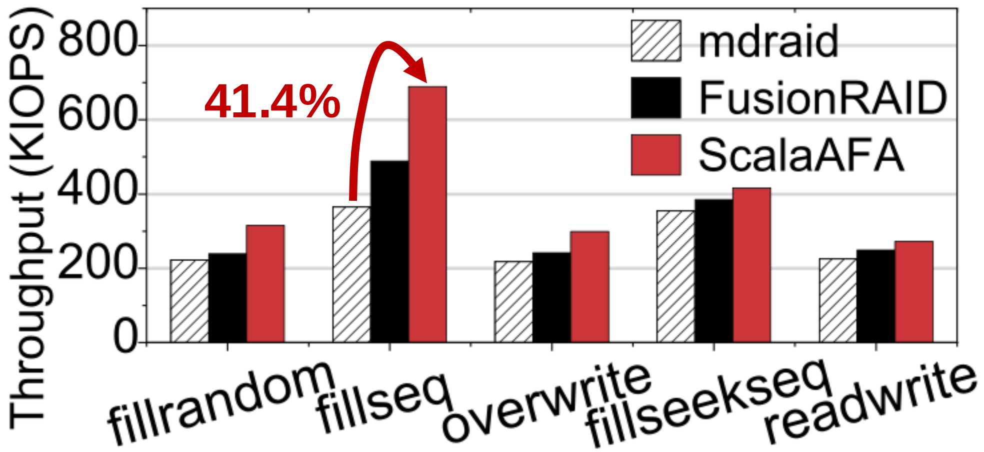


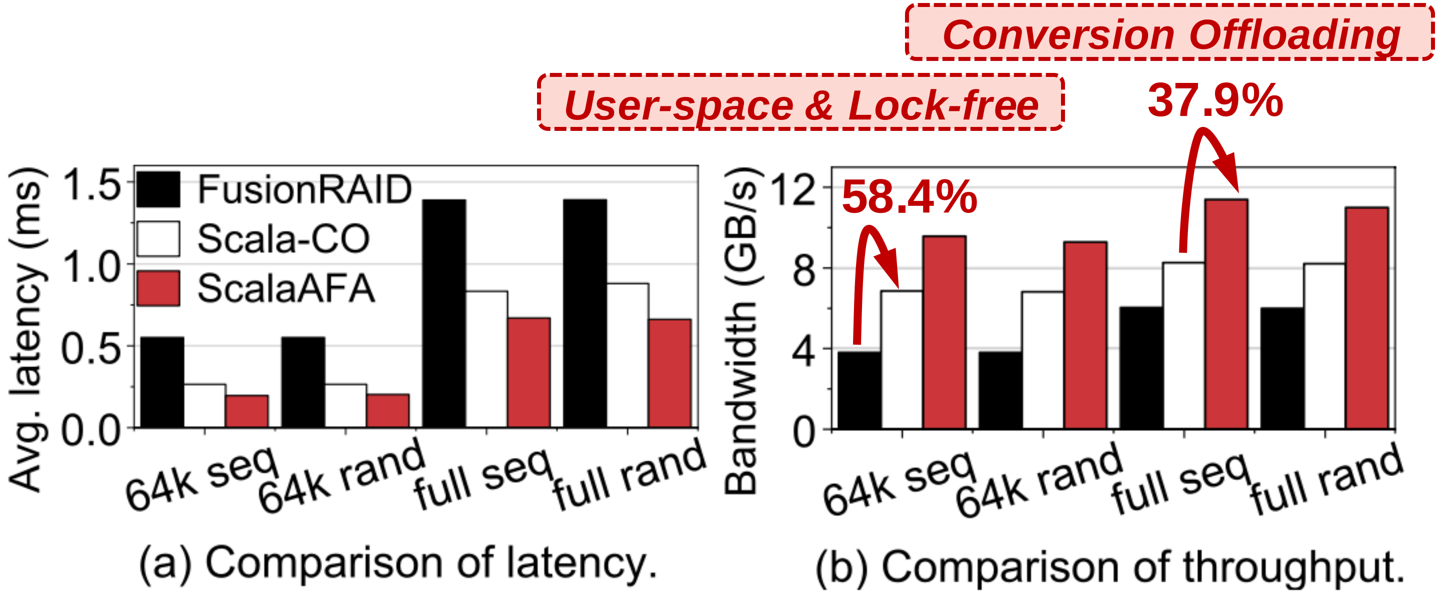
(a) proxy0. (b) prn. (c) src2.

(d) CFS. (e) DAP. (f) webmail.

ScalaAFA shortens the runtime by 2.8x!

Application & Incremental Tests





Throughput on RocksDB

Incremental Tests

Conclusion

- **Existing AFA engines fail to adopt high-performance SSDs**
  - Software overhead and AFA internal tasks
- ***ScalaAFA:* deliver high performance at low CPU costs**
  - Key insight: embracing user space & harnessing SSD built-in resources
  - Lock-free permission management for concurrent access *(Design 1)*
  - Offloading conversion tasks to SSDs with novel placement policy *(Design 2)*
  - Store sliced metadata in SSD OOB for low-cost crash consistency *(Design 3)*
  - Avoid flushing transient replicas to the vulnerable MLC blocks *(Design 4)**
- **Significantly improves write throughput and reduces latency**

*Please refer to our paper for more details.

Thanks & QA

ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs

https://github.com/ChaseLab-PKU/ScalaAFA

Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li

Chenxi Wang, Bo Mao, Myoungsoo Jung, Jie Zhang

CHASELab Computer Hardware And System Evolution Laboratory

PEKING UNIVERSITY

ZGC LAB

UNIVERSITAS AMOIENSIS

Alibaba

CHINESE ACADEMY OF SCIENCES

KAIST