

Conduit

Programmer-Transparent Near-Data Processing
Using Multiple Compute-Capable Resources
in Solid State Drives

Rakesh Nadig, Vamanan Arulchelvan, Mayank Kabra,
Harshita Gupta, Rahul Bera, Nika Mansouri Ghiasi,
Nanditha Rao, Qingcai Jiang, Andreas Kosmas Kakolyris,
Yu Liang, Mohammad Sadrosadati, and Onur Mutlu

HPCA 2026

Talk Outline

Background and Motivation

Conduit

Evaluation

Summary

Talk Outline

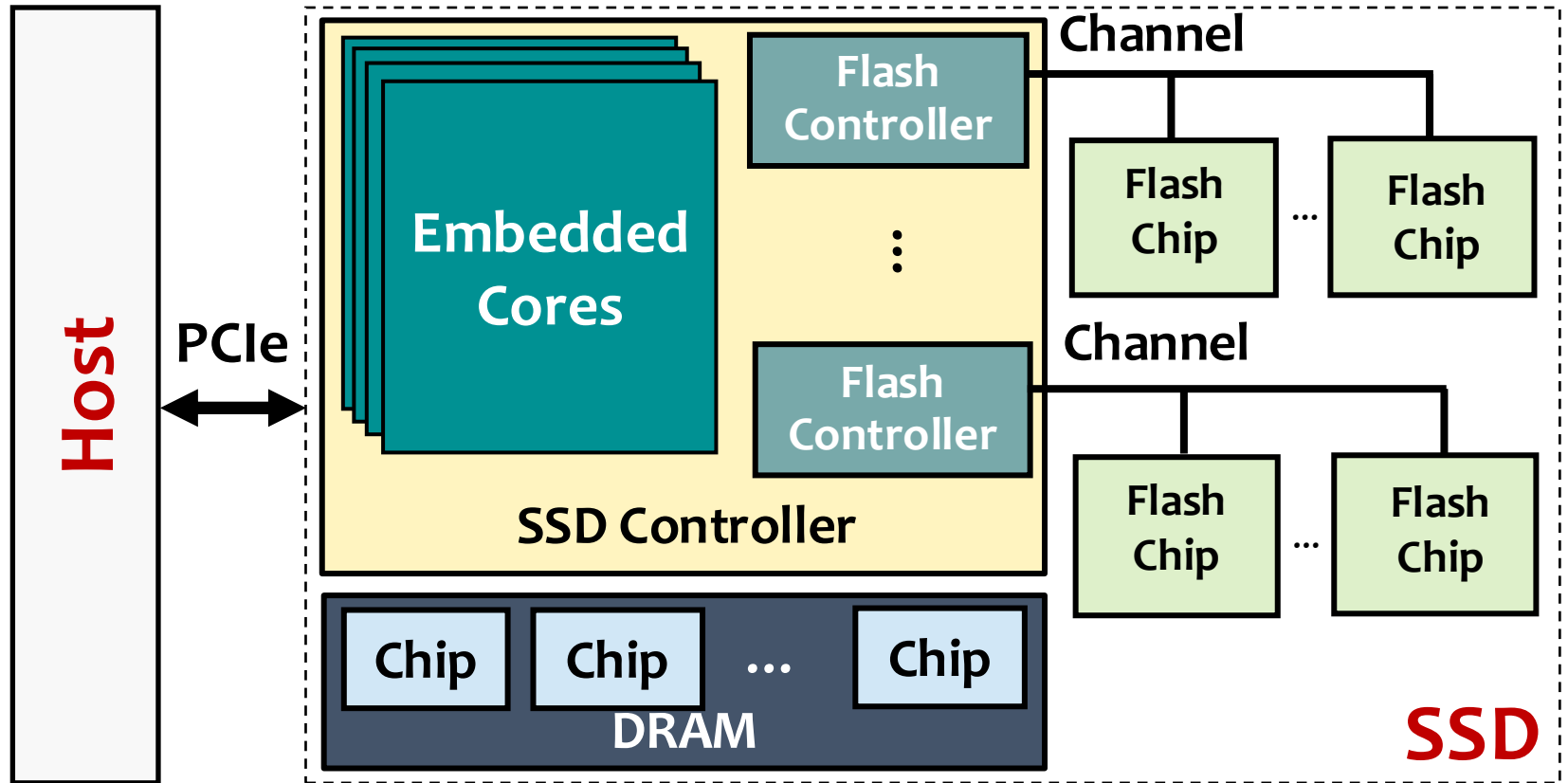
Background and Motivation

Conduit

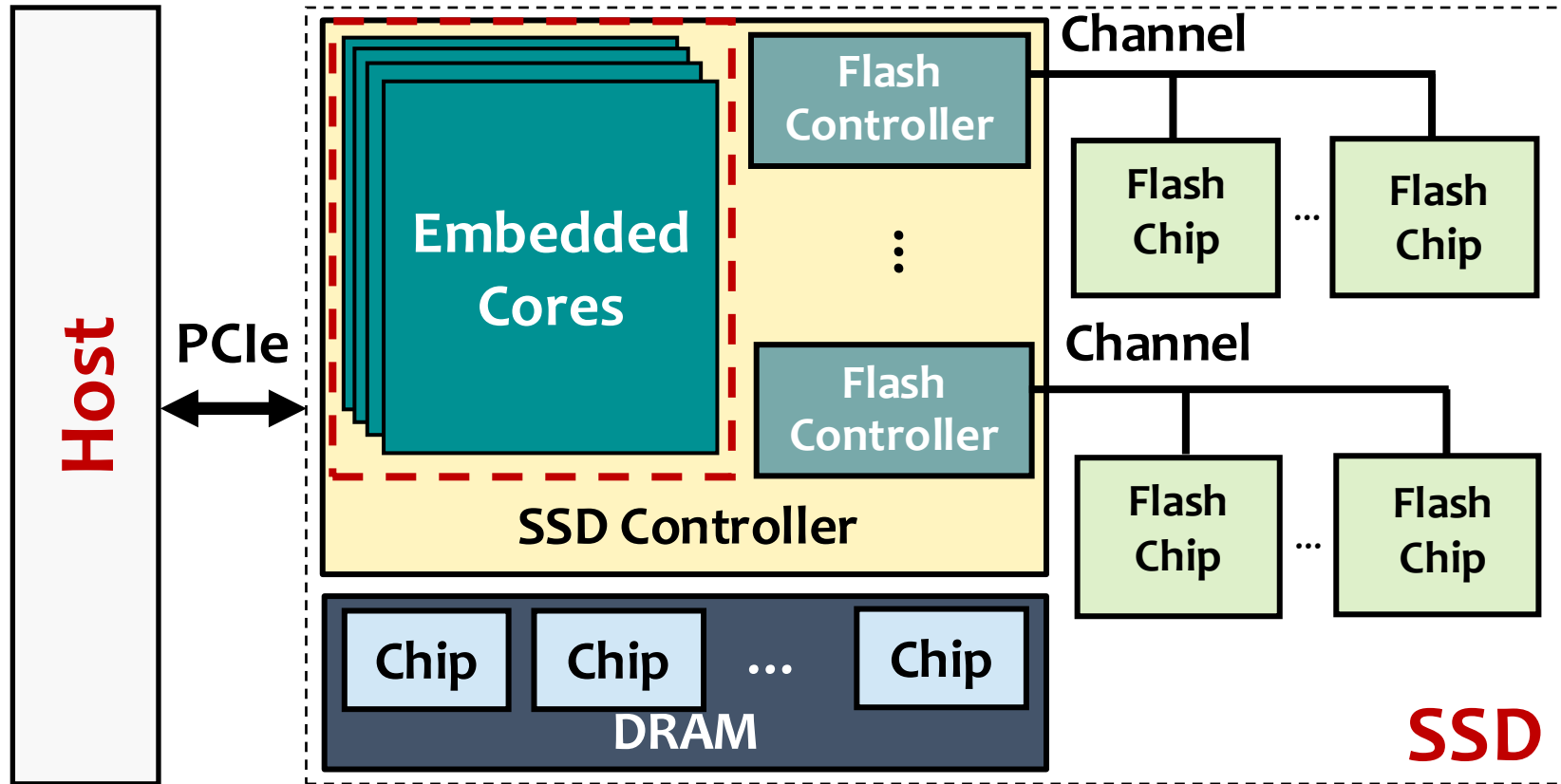
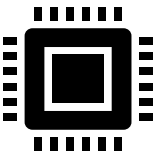
Evaluation

Summary

Overview of a Modern SSD



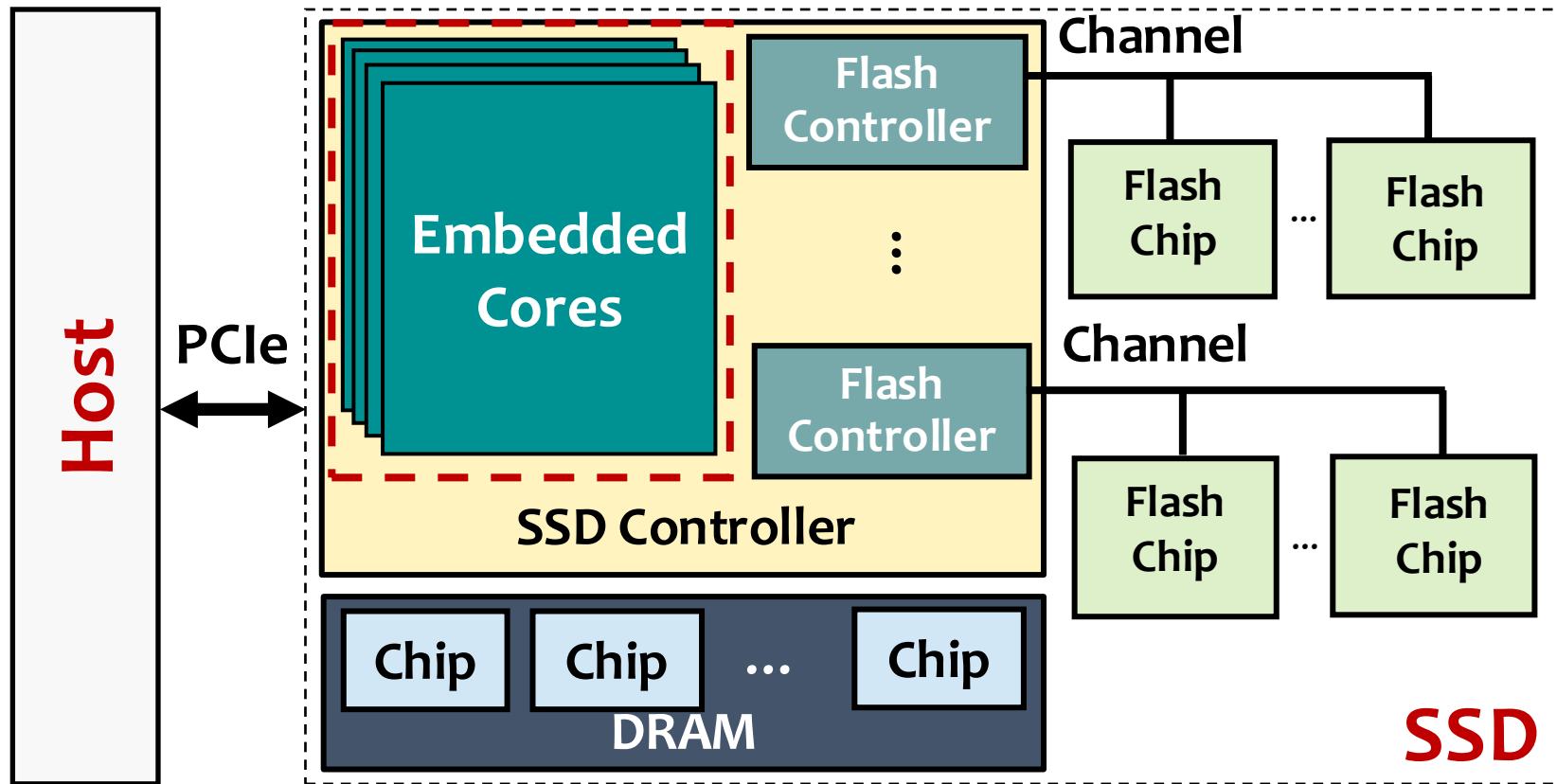
Near Data Processing in an SSD (ISP)



In-Storage Processing (ISP)

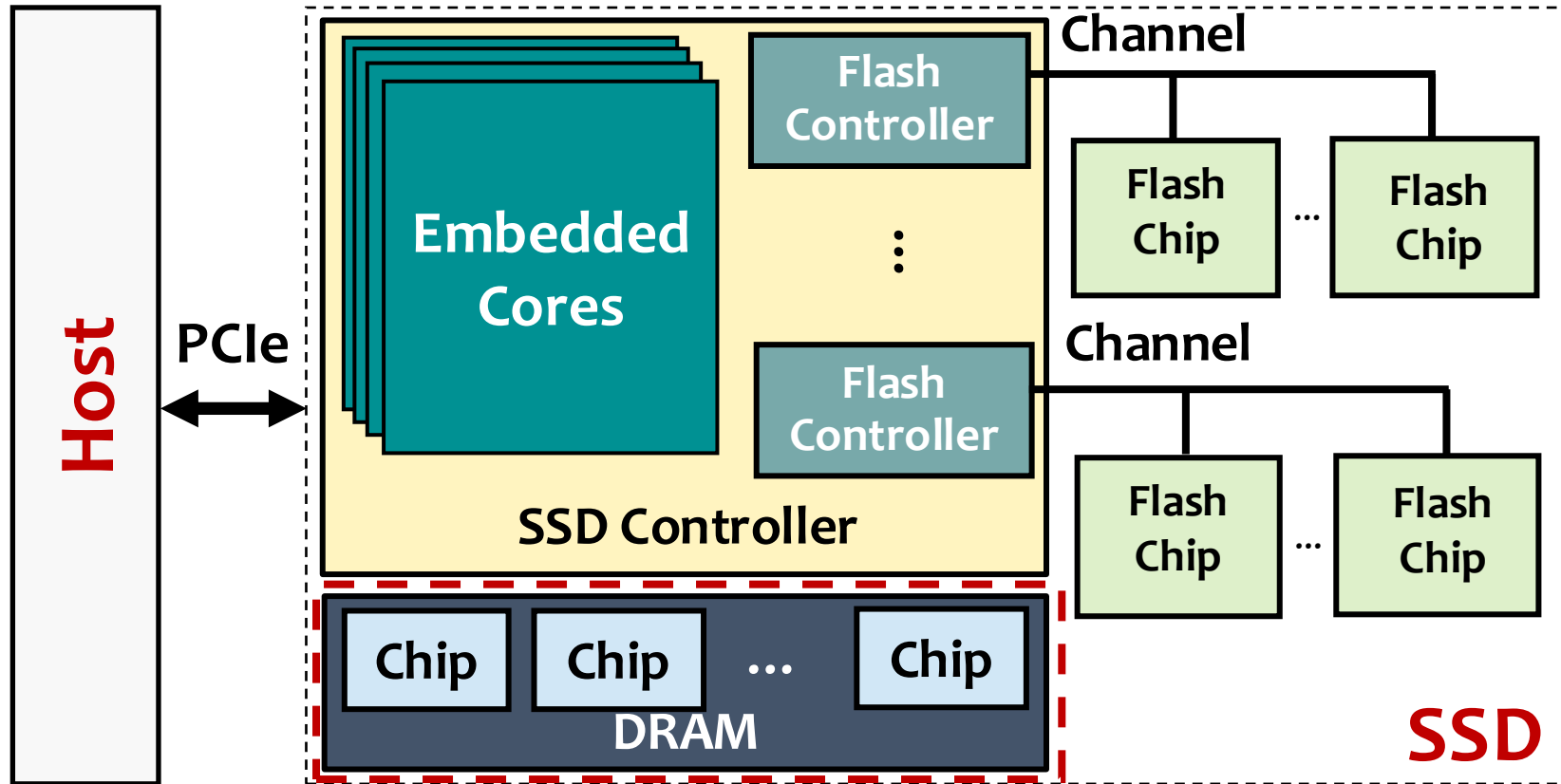
Processing using embedded cores in the SSD controller

Near Data Processing in an SSD (ISP)



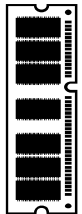
Many prior ISP works (e.g., Biscuit [Gu+, ISCA 2016], GenStore [Mansouri Ghiasi+, ASPLOS 2022], REIS [Chen+, ISCA 2025]) accelerate **complex tasks** including **filtering, aggregation, encryption, and search**

Near Data Processing in an SSD (PuD-SSD)

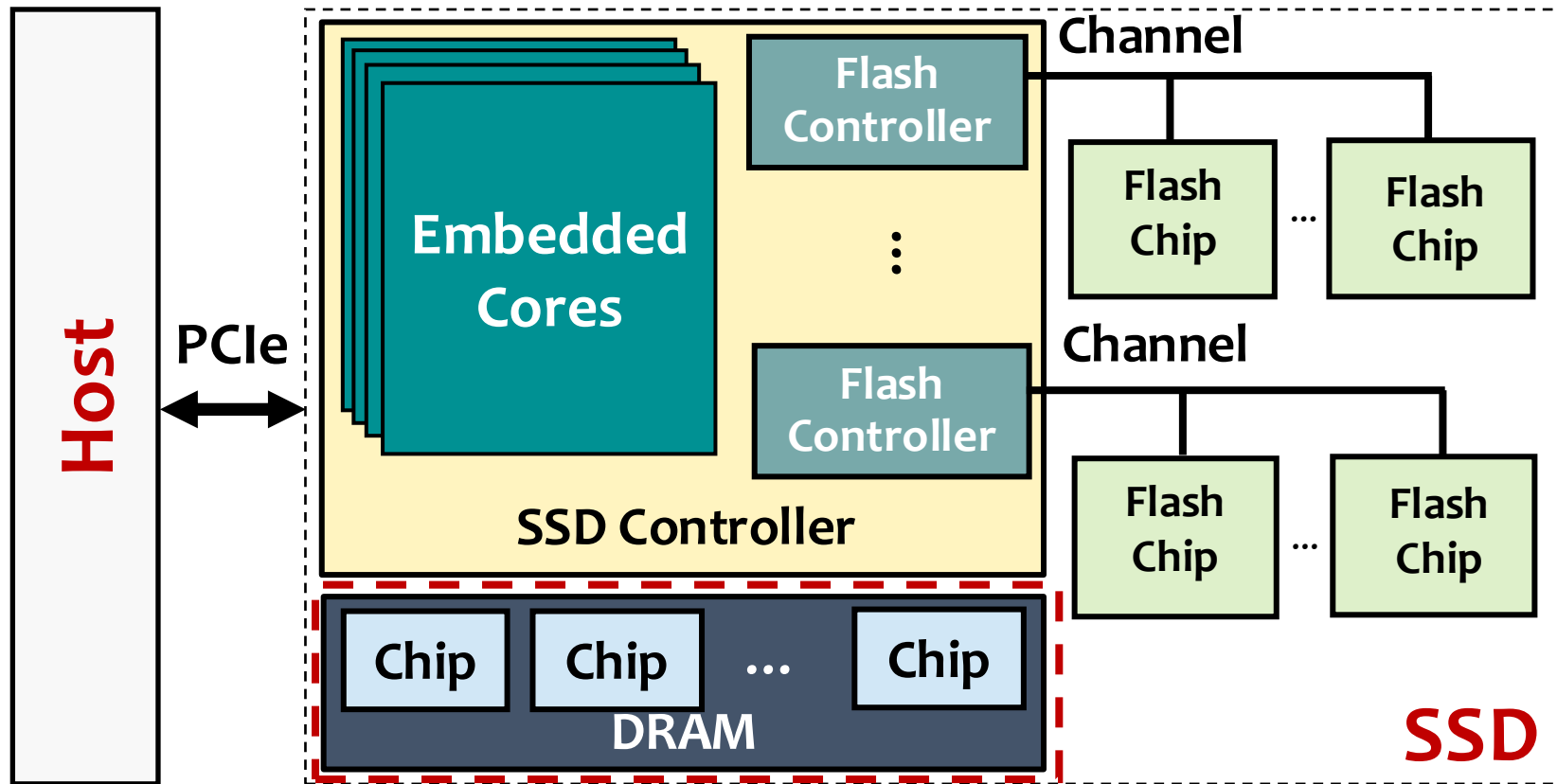


Processing-using-DRAM in SSD (PuD-SSD)

Computation using **SSD DRAM's** intrinsic operational principles

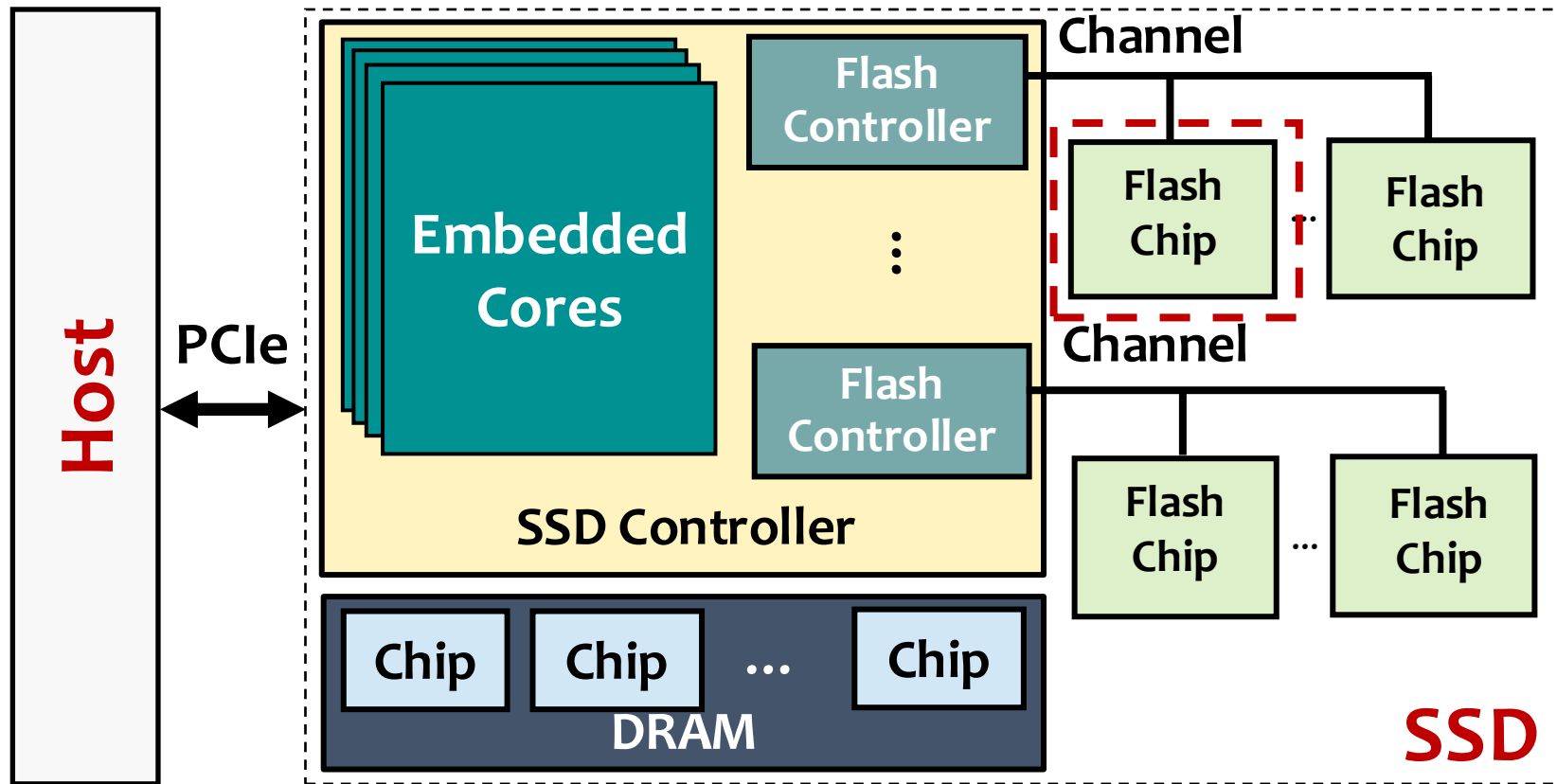
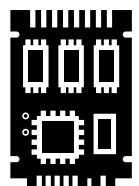


Near Data Processing in an SSD (PuD-SSD)



PuD works (e.g., Ambit [Seshadri+, MICRO 2017], SIMDRAM [Hajinazar+, ASPLOS 2021], MIMDRAM [Oliveira+, HPCA 2024]) enable **bulk-bitwise** and **complex arithmetic operations** entirely within the DRAM chips

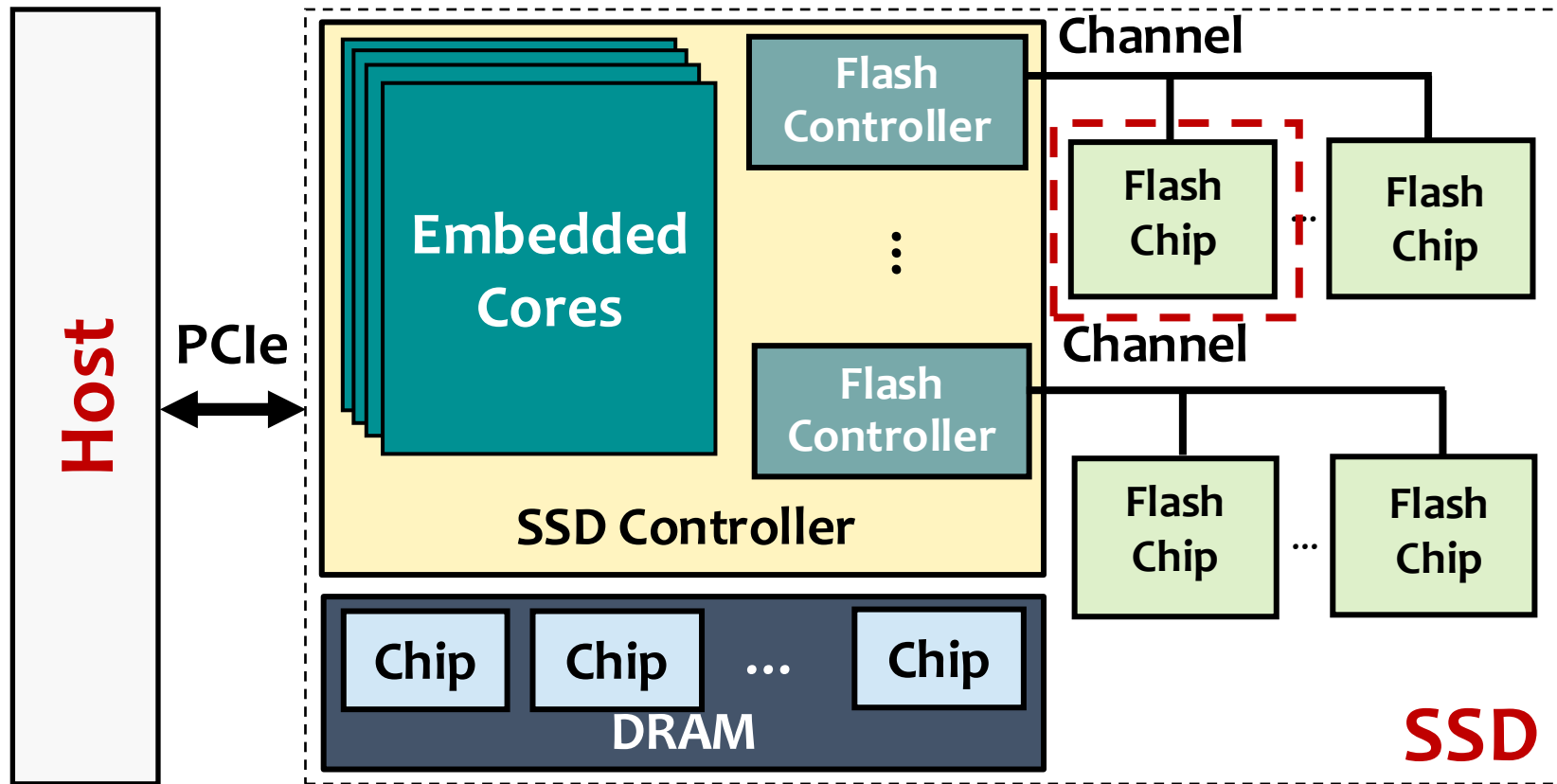
Near Data Processing in an SSD (IFP)



In-Flash Processing (IFP)

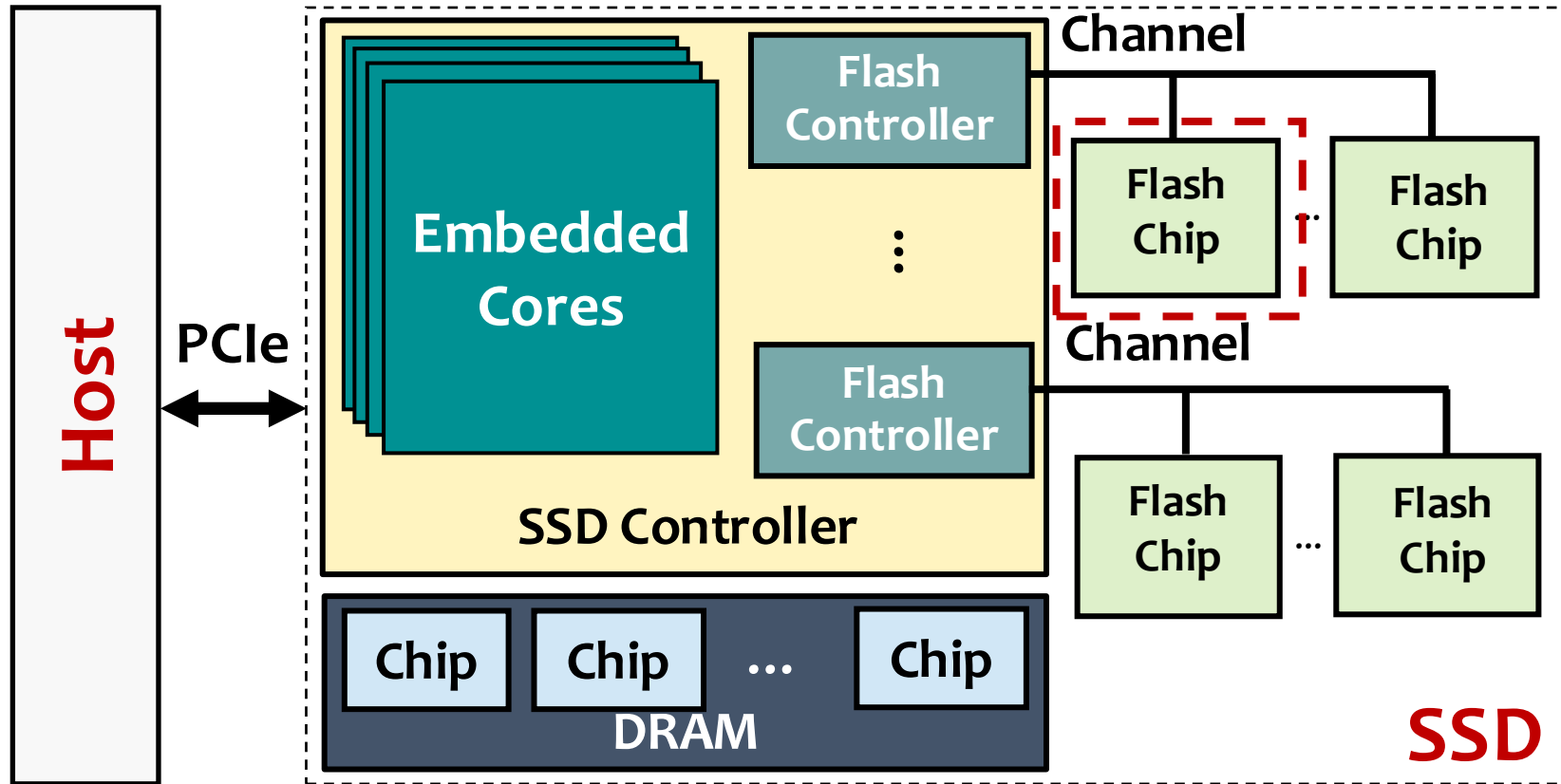
Computation by exploiting the flash chip's structure

Near Data Processing in an SSD (IFP)



Several IFP techniques (e.g., Ares-Flash [Chen+, MICRO 2024]) enable **arithmetic operations** inside NAND flash chips by **controlling the latching circuit of the page buffer**

Near Data Processing in an SSD (IFP)



Flash-Cosmos [Park+, MICRO 2022] enables **bulk-bitwise operations** within NAND flash chips by **simultaneously reading multiple pages**

Near Data Processing in an SSD: Tradeoffs

NDP Paradigm		
ISP	Supports general-purpose computations	Wimpy embedded cores provide limited performance
PuD-SSD	Computations in a highly parallel manner within the DRAM chips	Requires transferring operands from flash chips via bandwidth-limited flash channels
IFP	Eliminates data movement overhead and provides high computation parallelism	Limited operation set

NDP in an SSD: Research Questions

?

Is a single NDP paradigm good for all workloads?

?

Do we need to combine these NDP paradigms?

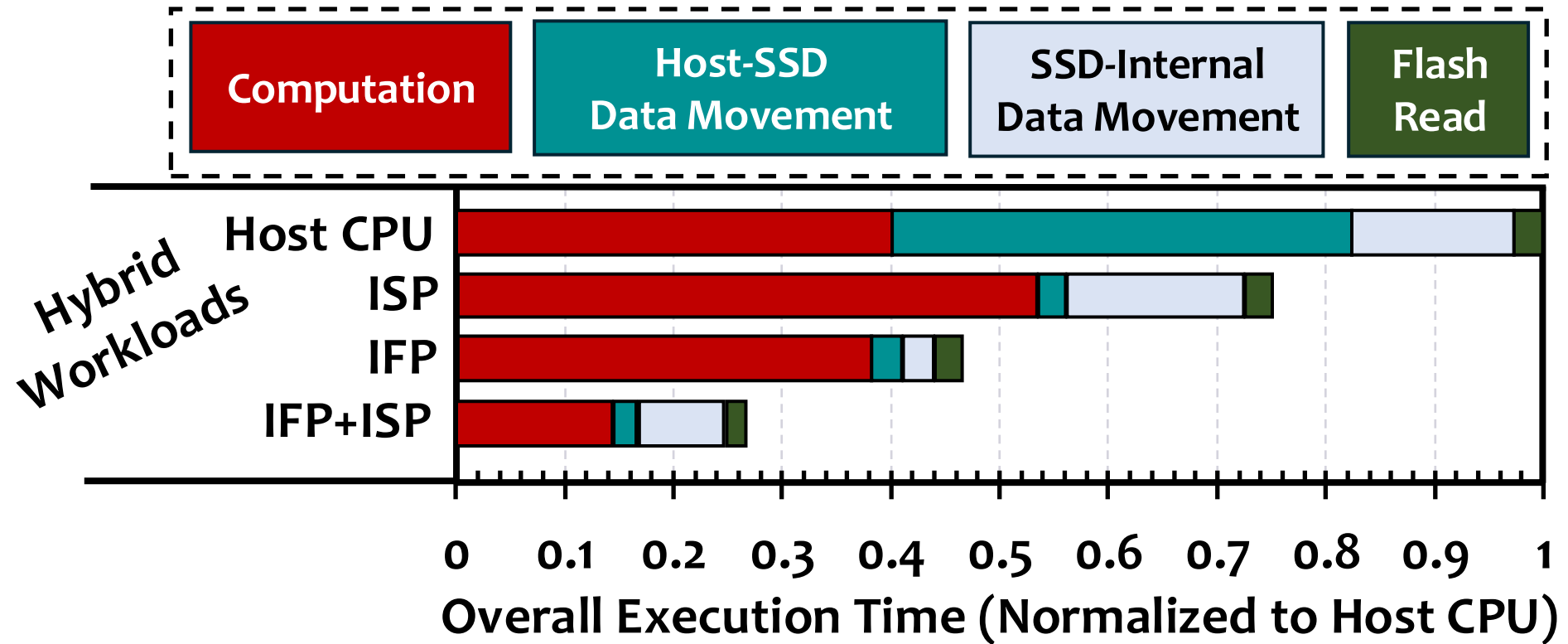
?

Which is the most suitable NDP paradigm for each computation?

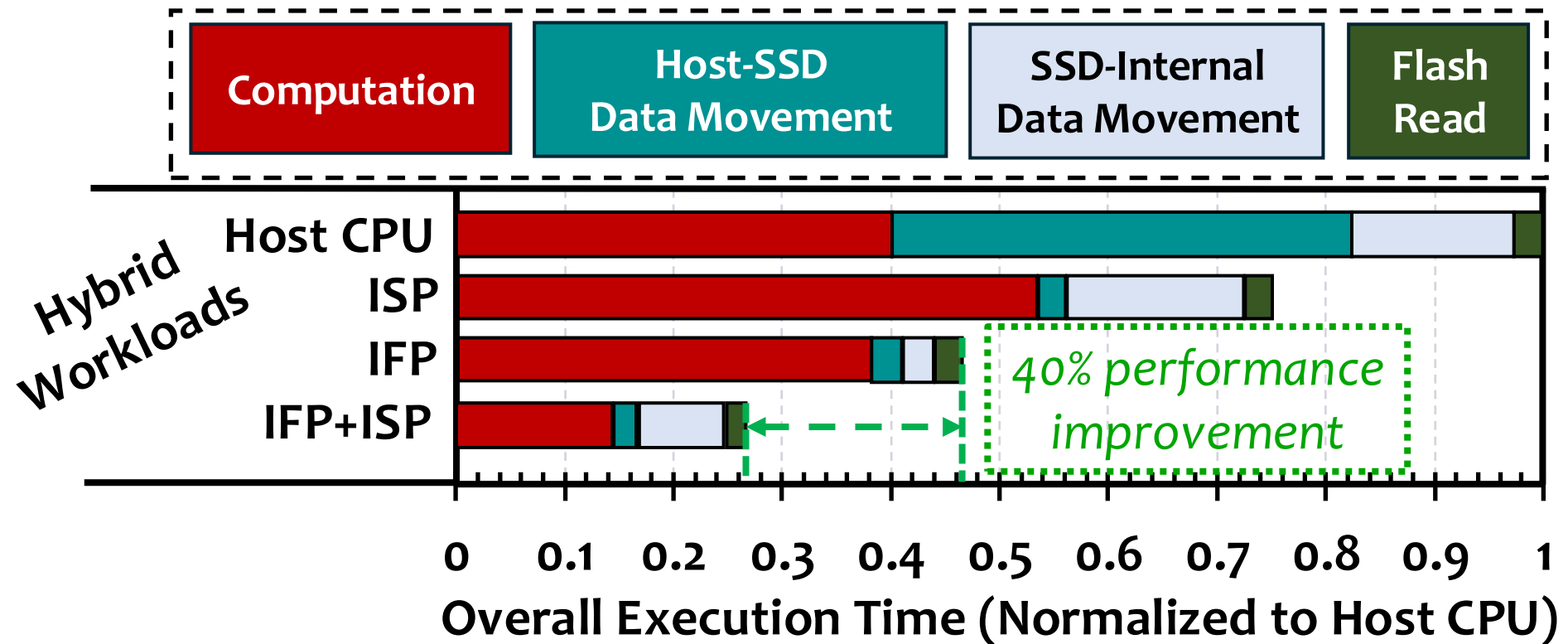
Case Study on Combining NDP Paradigms (1/5)

- We examine the **performance bottlenecks** by **hand-mapping** parts of the workload to **multiple SSD computation resources**
- **Evaluated Policies**
 - Host CPU
 - In-storage processing (ISP)
 - In-flash processing (IFP)
 - Offloading to both ISP and IFP (ISP + IFP)
- **Evaluated Workloads**
 - **Hybrid workloads**: Workloads that involve both computation and data movement (e.g., aggregation, sorting)
 - **I/O-Intensive workloads**: Workloads dominated by data movement (e.g., databases, bitmap indices)
 - **More in the paper**

Case Study on Combining NDP Paradigms (2/5)

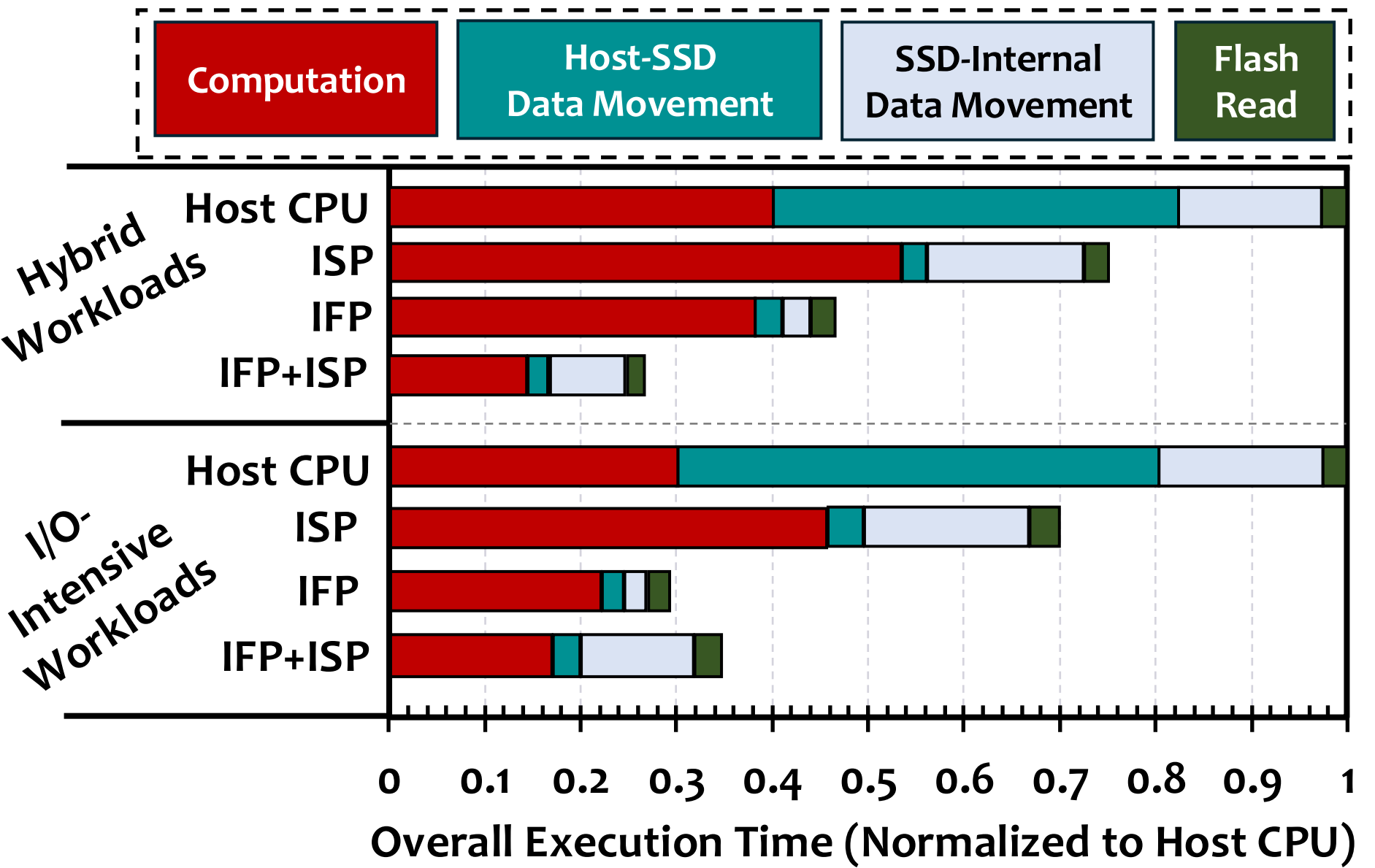


Case Study on Combining NDP Paradigms (3/5)



Offloading to ISP and IFP improves performance by 40% over IFP by reducing data movement and mapping computations to the most suitable computation resource

Case Study on Combining NDP Paradigms (4/5)



Case Study on Combining NDP Paradigms (5/5)

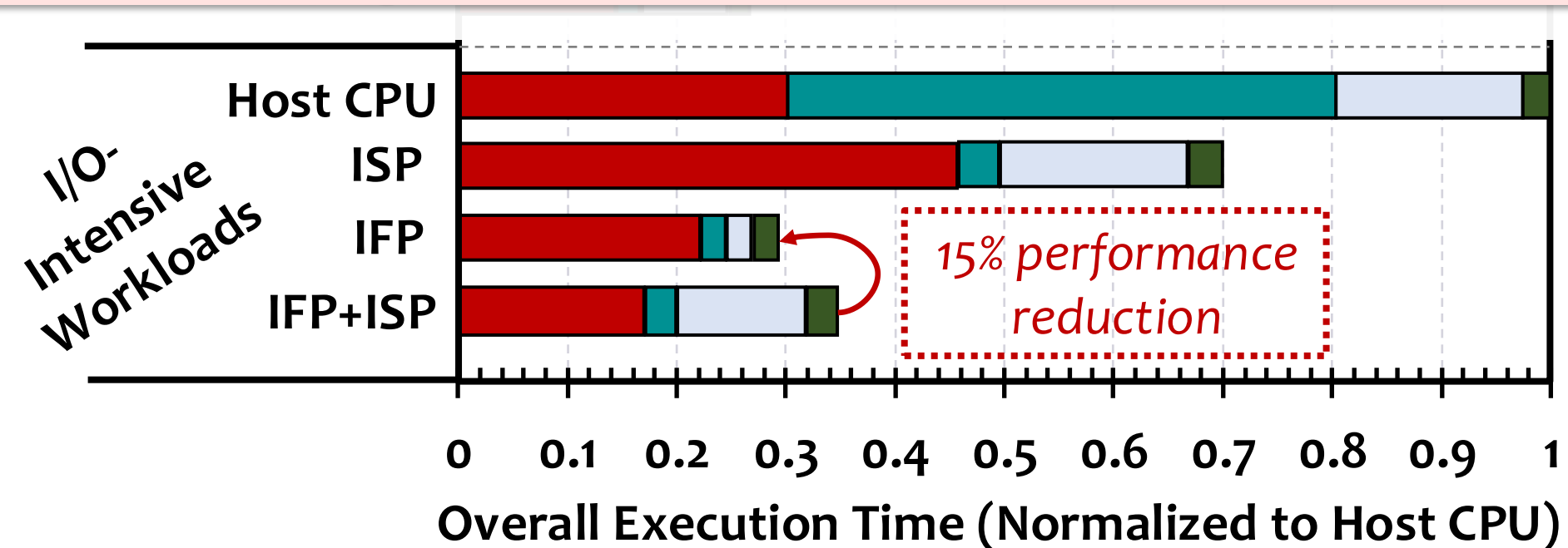
Computation

Host-SSD
Data Movement

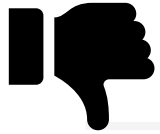
SSD-Internal
Data Movement

Flash
Read

Naively combining multiple computation resources in I/O-intensive workloads negates offloading benefits due to data movement between computation resources



Case Study: Key Takeaways



No single computation resource consistently performs well across different workloads



Performance bottlenecks are workload-dependent and vary across different execution phases of the same workload



Offloading to multiple SSD computation resources provides benefits, but only when done judiciously

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Overall Execution Time (Normalized to Host CPU)

Limitations of Prior SSD-Based NDP Techniques

- Prior SSD-based NDP techniques



Map computations only to one or two NDP paradigms and fail to exploit the SSD's full computational potential



Lack generality across a wide range of applications



Are not programmer-transparent

NDP Offloading Techniques For Main Memory

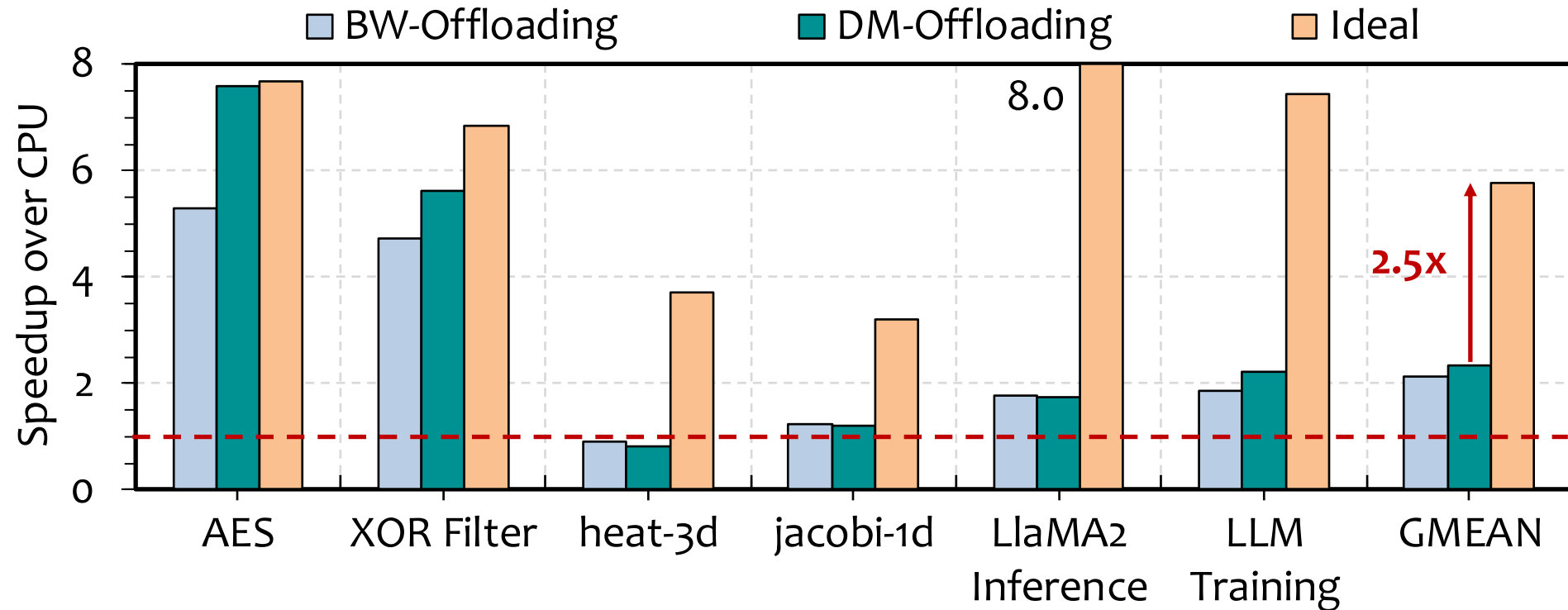
- **No prior work** proposes **application-transparent offloading** of computations across multiple SSD computation resources
- Several NDP offloading techniques explore **partitioning and mapping application code segments** between **host and NDP units near main memory**

BW-Offloading: Makes offloading decisions **based on bandwidth utilization** of the host and NDP units (e.g., TOM [Hsieh+, ISCA 2016], λ -IO [Yang+, FAST 2023])

DM-Offloading: Makes offloading decisions to **minimize data movement** between the host and NDP units (e.g., ALP [Mansouri Ghiasi+, TETC 2023], PIMProf [Wei+, DATE 2022])

- We **adapt these offloading models** for SSD-based NDP

Effectiveness of Prior Offloading Techniques



DM-Offloading falls short of the Ideal's performance by **2.5x** due to its **narrow offloading model**

DM-Offloading prefers a single computation resource to reduce data movement, causing **queueing delays and resource contention**

Our Goal

To enable **programmer-transparent NDP** in an SSD that

- effectively utilizes multiple SSD computation resources to perform computations
- makes **workload- and system-aware offloading** decisions
- **improves performance and energy efficiency** of a wide range of applications

Talk Outline

Background and Motivation

Conduit

Evaluation

Summary

Our Proposal



Conduit

A general-purpose programmer-transparent NDP framework that dynamically offloads computations at instruction granularity to multiple heterogeneous SSD computation resources

Conduit: Key Steps



Compile-time pre-processing step to identify offloadable code regions in the application and transform them to wide vector operations



Runtime offloading of vectorized instructions to the most suitable SSD computation resource based on application and system characteristics

Conduit: Key Steps



Compile-time pre-processing step to identify offloadable code regions in the application and transform them to wide vector operations



Runtime offloading of vectorized instructions to the most suitable SSD computation resource based on application and system characteristics

Conduit: Compile-Time Preprocessing (1/3)

Application Code

```
for (j = 0; j < nb; ++j)
{
    ....
    if ((ret[1 + j * 4] >> 8) == 1)
    {
        ret[1 + j * 4] ^= 283;
    }
    ....
}
```

Host

Intermediate Representation (IR)

```
...
%38 = load <4096 x i32>, ptr %37, align 4
%39 = xor <4096 x i32> %38, %34
%40 = xor <4096 x i32> %39, %21
%41 = xor <4096 x i32> %40, %22
%42 = and <4096 x i32> %20, <i32 283 ... i32>
...
```

Loop
Auto-Vectorization

Conduit runs a **custom compiler (e.g., LLVM) pass** to identify offloadable code regions

Loop auto-vectorization transforms **scalar instructions** into **wide vector operations** that match the **SSD-internal parallelism**

Conduit leverages **partial vectorization** so that **partially vectorizable code regions** benefit from parallel execution in an SSD

Conduit: Compile-Time Preprocessing (2/3)

Application Code

```
for (j = 0; j < nb; ++j)
{
    ....
    if ((ret[1 + j * 4] >> 8) == 1)
    {
        ret[1 + j * 4] ^= 283;
    }
    ....
}
```

Host

Intermediate Representation (IR)

```
...
%38 = load <4096 x i32>, ptr %37, align 4
%39 = xor <4096 x i32> %38, %34
%40 = xor <4096 x i32> %39, %21
%41 = xor <4096 x i32> %40, %22
%42 = and <4096 x i32> %20, <i32 283 ... i32>
...
```

Loop
Auto-Vectorization

Metadata

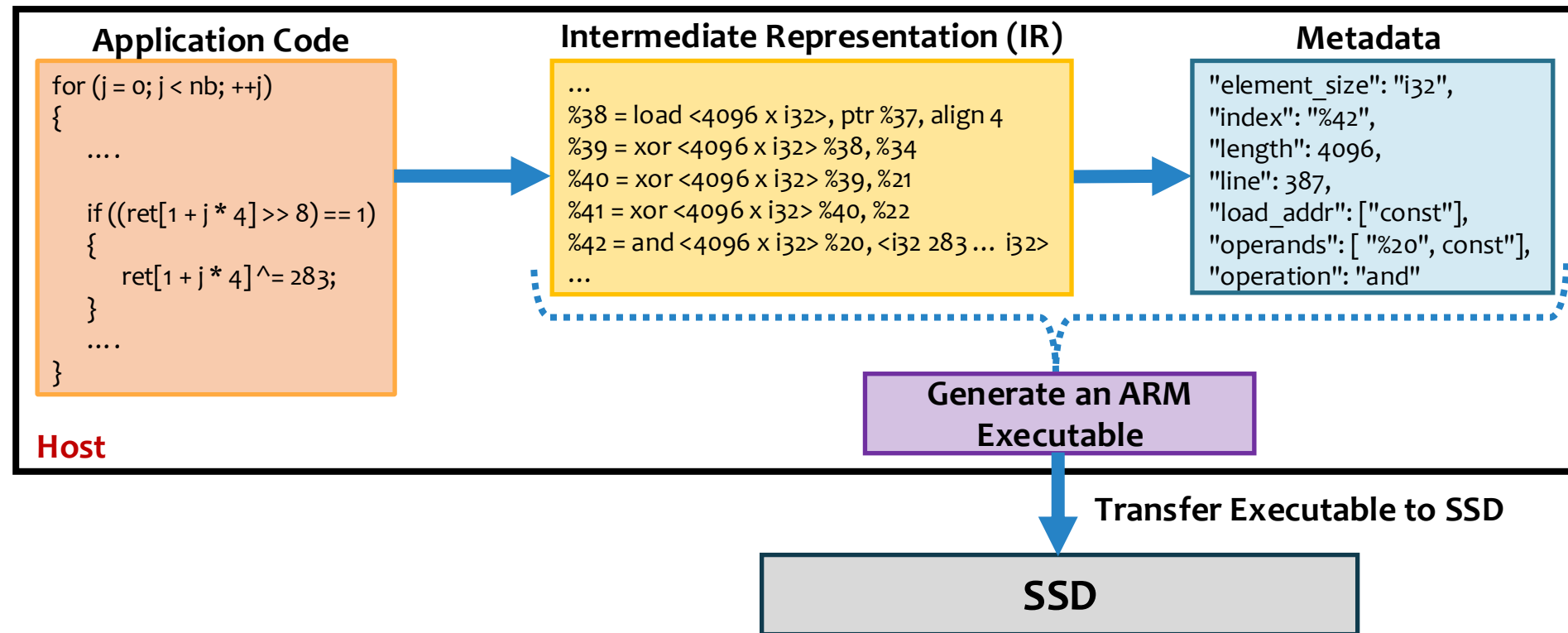
```
"element_size": "i32",
"index": "%42",
"length": 4096,
"line": 387,
"load_addr": ["const"],
"operands": [ "%20", const"],
"operation": "and"
```

Extract Metadata

Conduit extracts **lightweight metadata** (e.g., operation type, operand pointers, element sizes, vector length) **for each vector operation**

Conduit **embeds the metadata with IR** to reduce runtime scheduling latency and **preserve programmer transparency**

Conduit: Compile-Time Preprocessing (3/3)



Conduit **converts the IR with metadata** to an **executable** capable of running on ARM cores

Conduit **transfers the ARM executable** to the **SSD** by repurposing existing **NVMe admin commands** for firmware update

Conduit: Key Steps

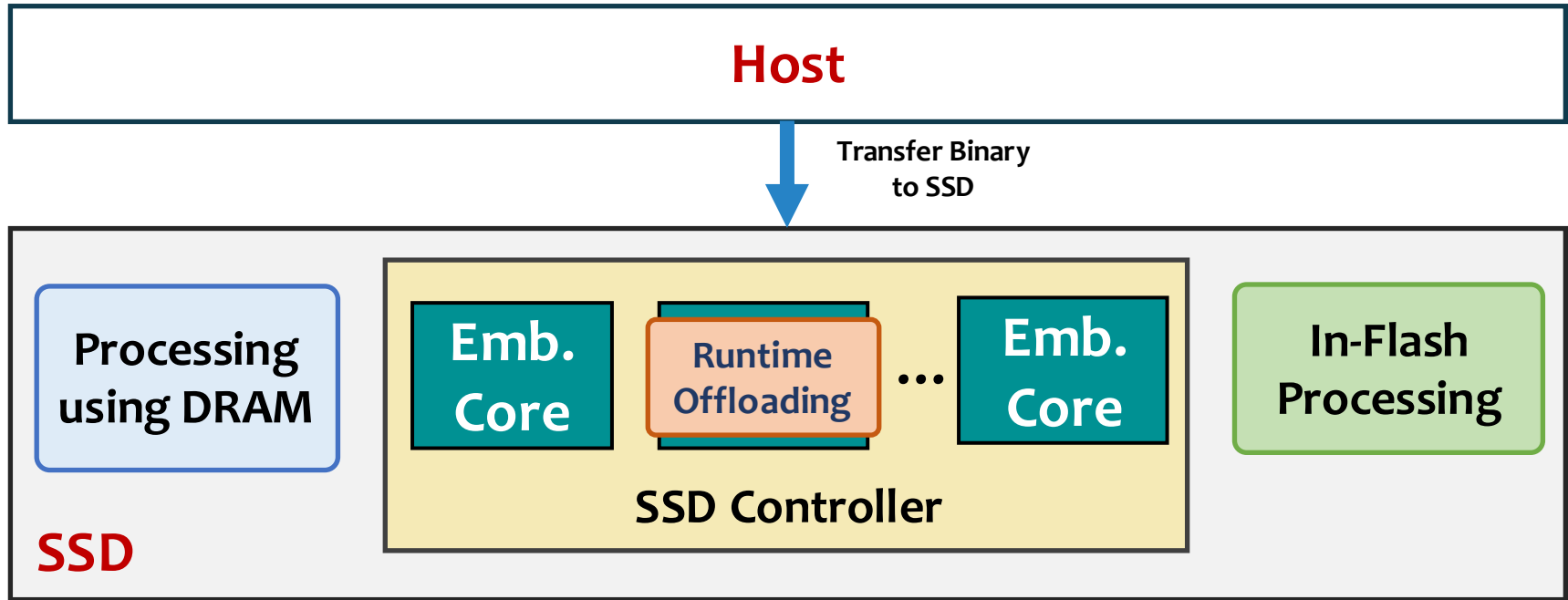


Compile-time pre-processing step to identify offloadable code regions in the application and transform them to wide vector operations



Runtime offloading of vectorized instructions to the most suitable SSD computation resource based on application and system characteristics

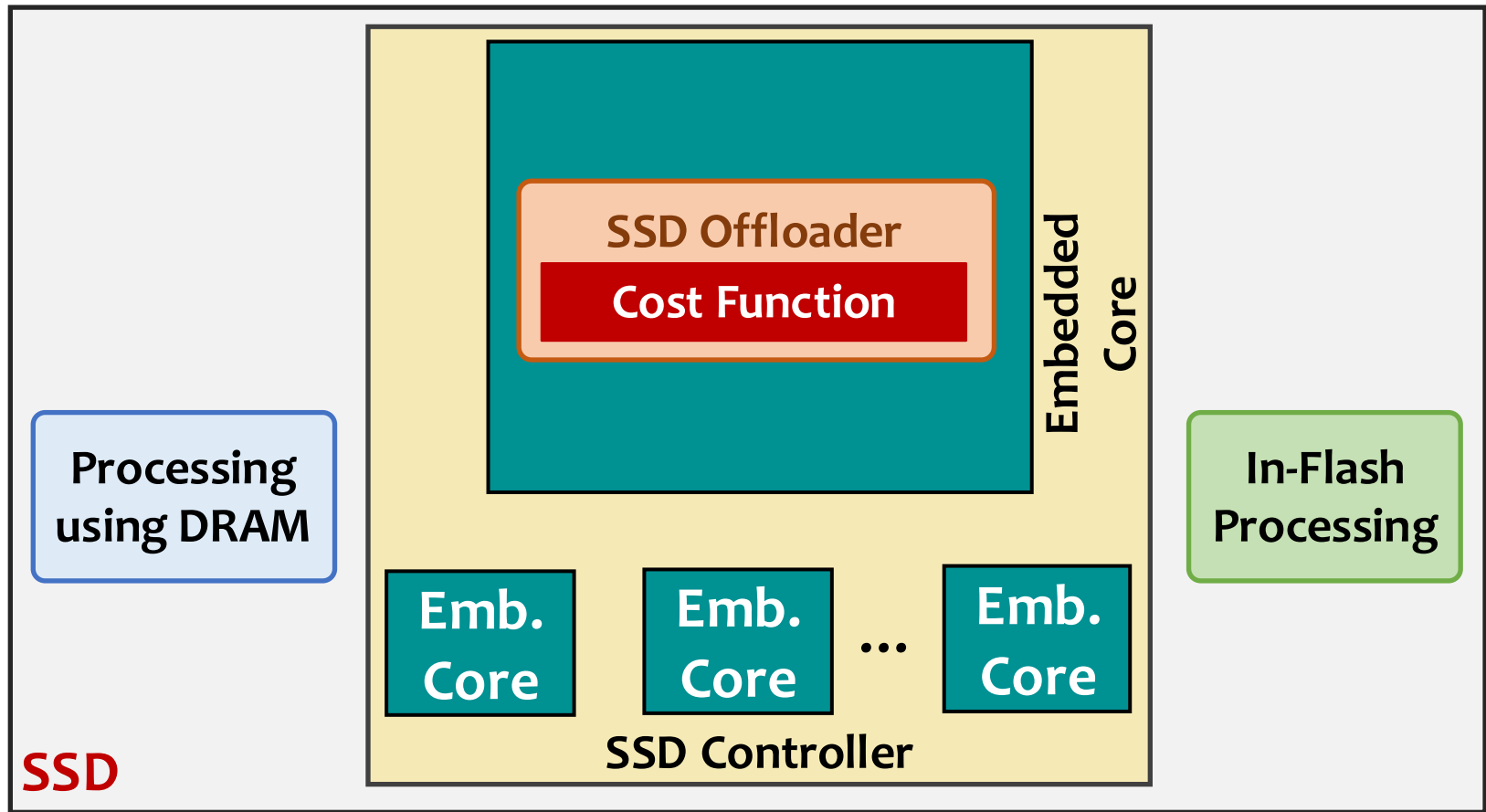
Conduit: Runtime Offloading (1/3)



Conduit performs **runtime offloading** in the SSD controller to **preserve programmer transparency** and **avoid changes to the storage stack**

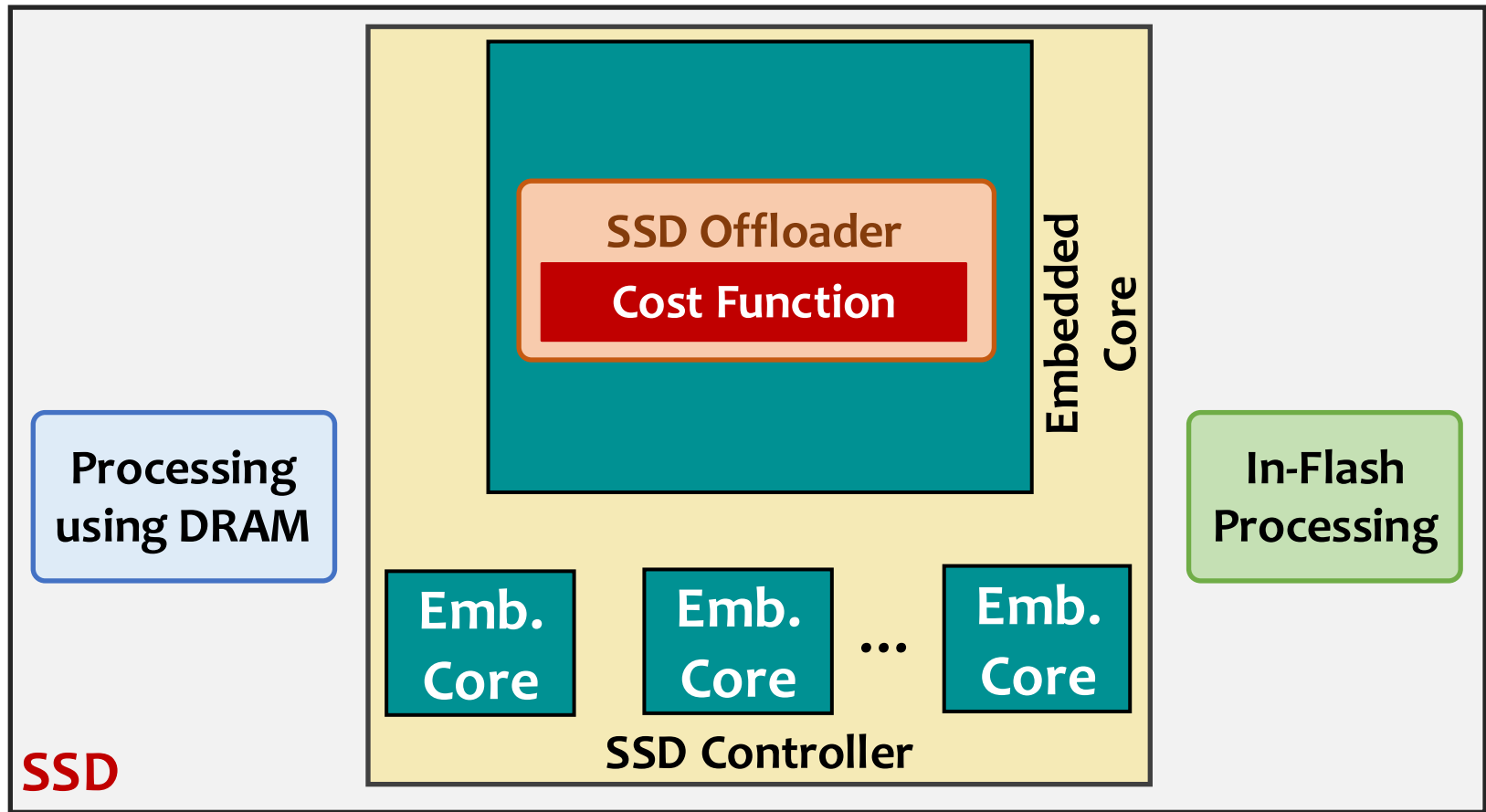
The SSD controller has **real-time knowledge** of the **current state** (e.g., resource utilization, contention in shared channels) of all SSD computation resources

Conduit: Runtime Offloading (2/3)



Conduit's **SSD offloader** maps **each vectorized instruction** in the ARM executable to the **most-suitable SSD computation resource** based on a **cost function**

Conduit: Runtime Offloading (3/3)



Conduit's **SSD offloader** maps **each vectorized instruction** in the ARM executable to the **most-suitable SSD computation resource** based on a **cost function**

Conduit: Cost Function

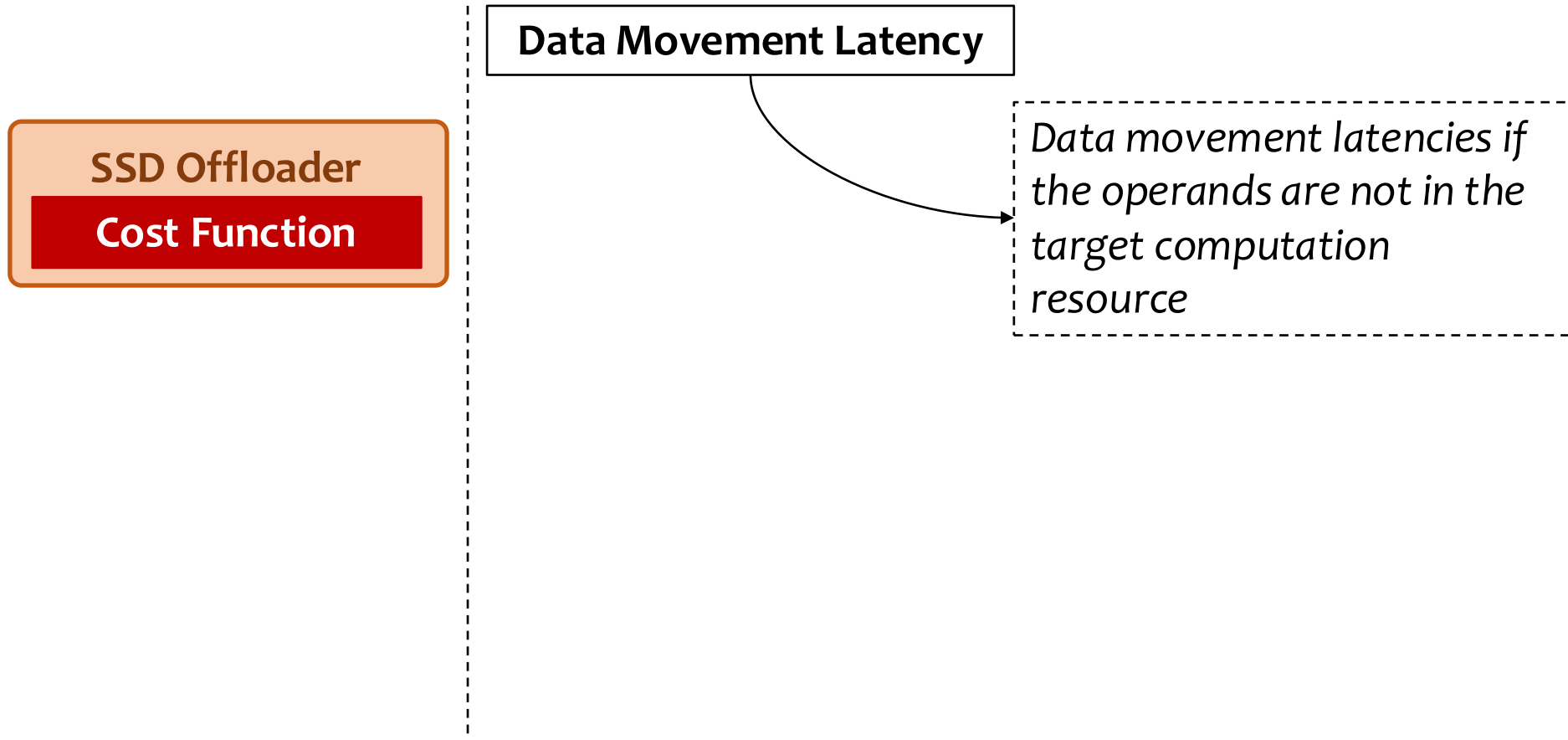
SSD Offloader

Cost Function

SSD Offloader

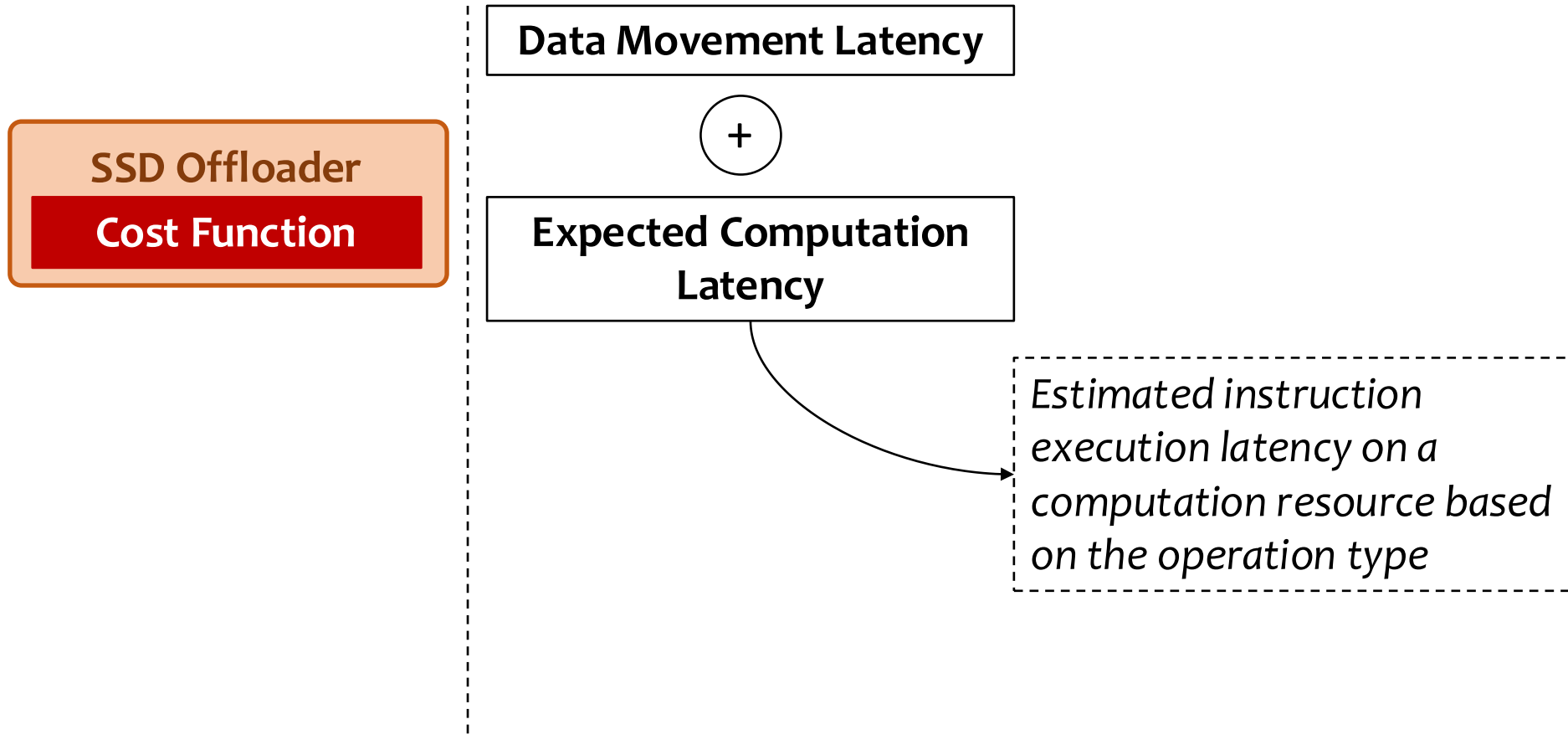
Cost Function

Conduit: Cost Function



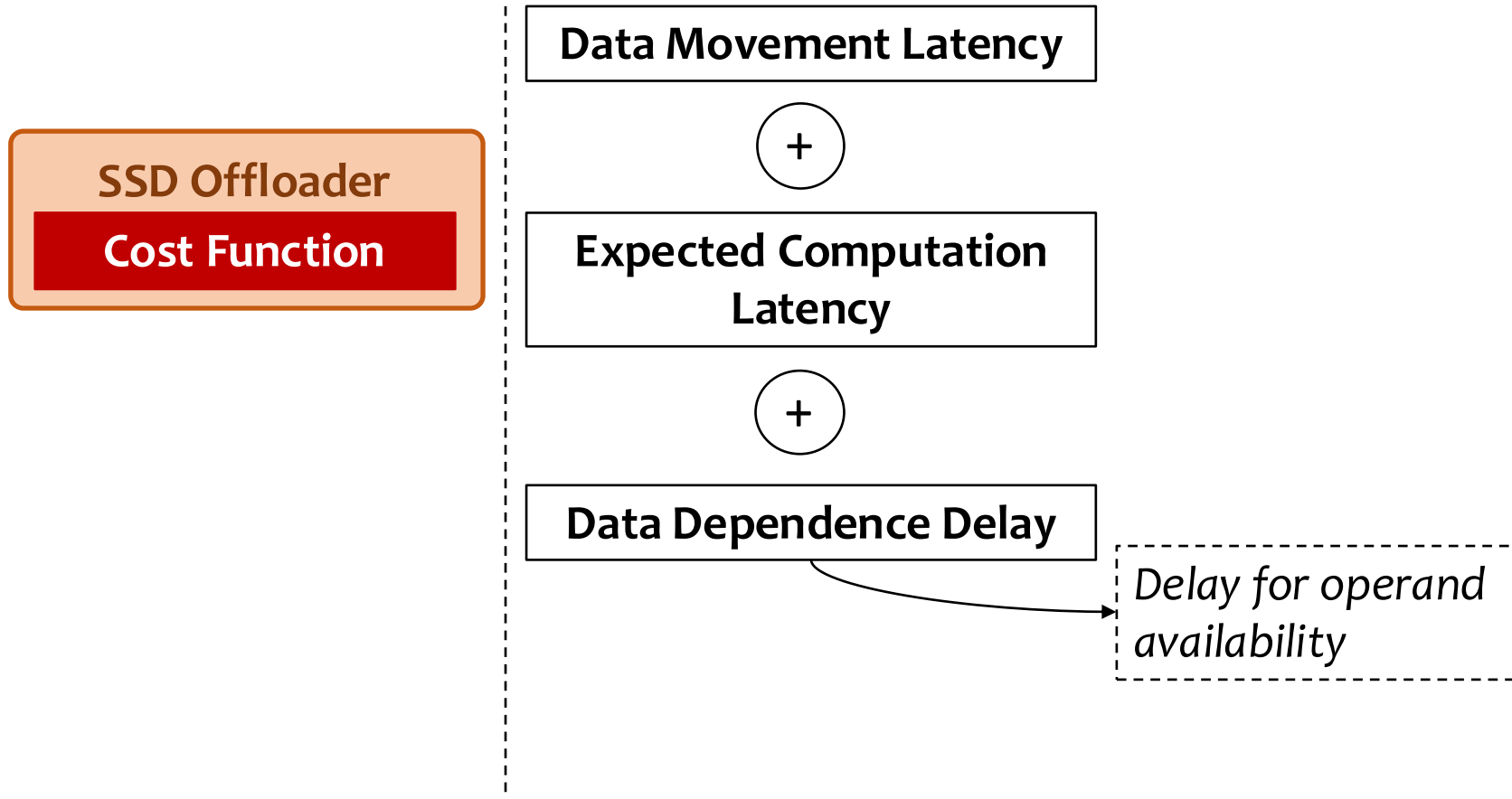
Conduit tracks **both** application characteristics and current system conditions to evaluate the **cost of offloading an instruction** to **each SSD computation resource**

Conduit: Cost Function



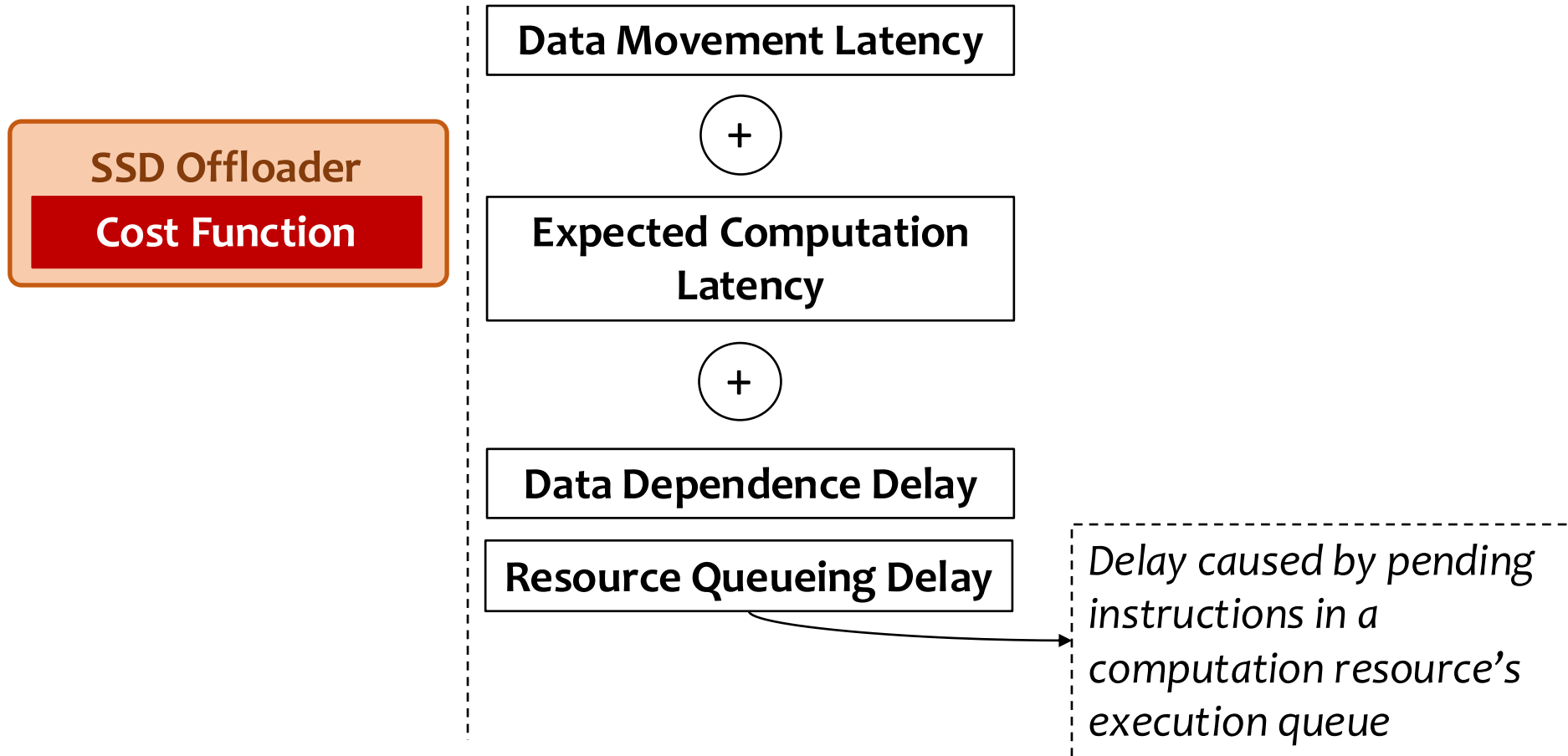
Conduit tracks **both application characteristics and current system conditions** to evaluate the **cost of offloading an instruction to each SSD computation resource**

Conduit: Cost Function



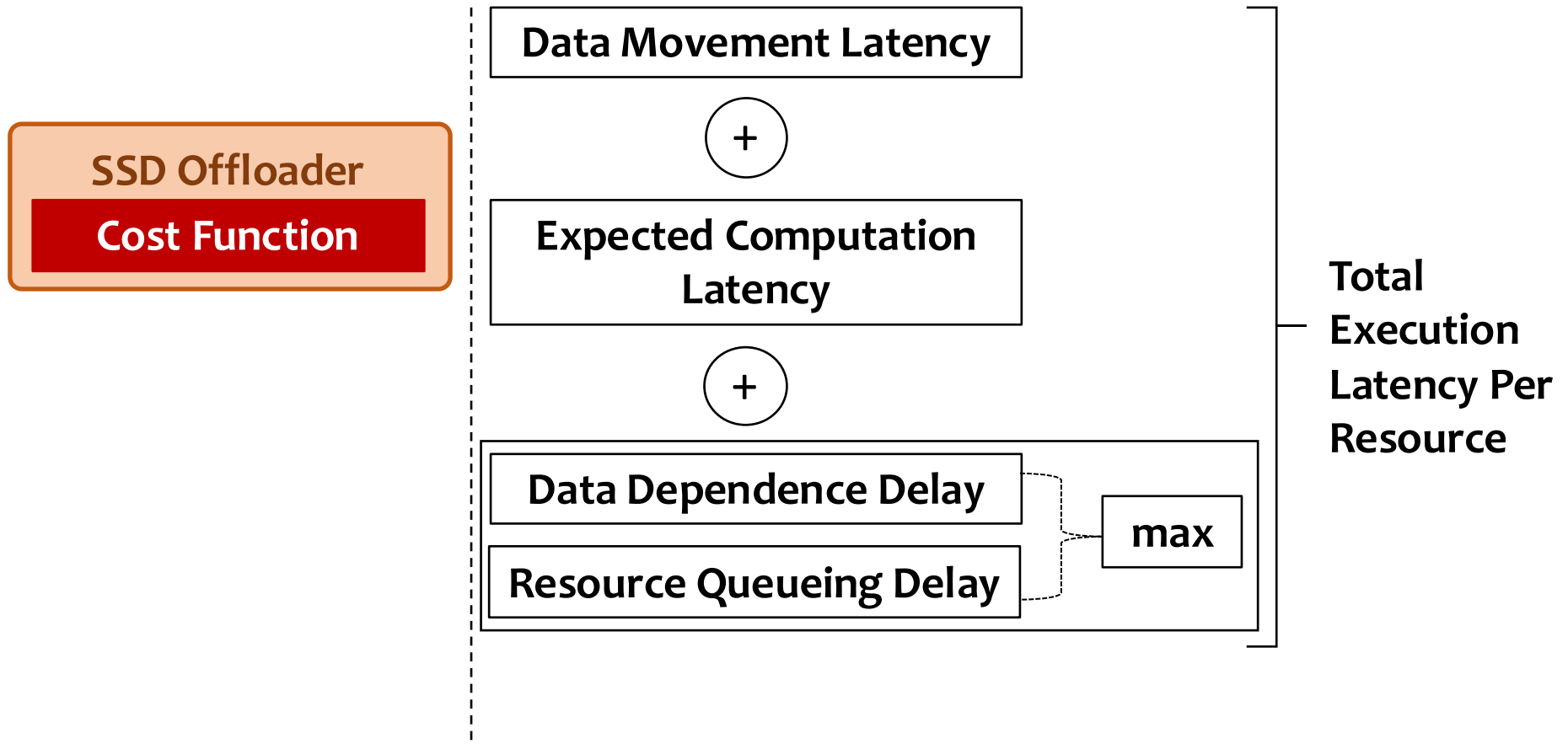
Conduit tracks **both** application characteristics and current system conditions to evaluate the **cost of offloading an instruction** to **each SSD computation resource**

Conduit: Cost Function



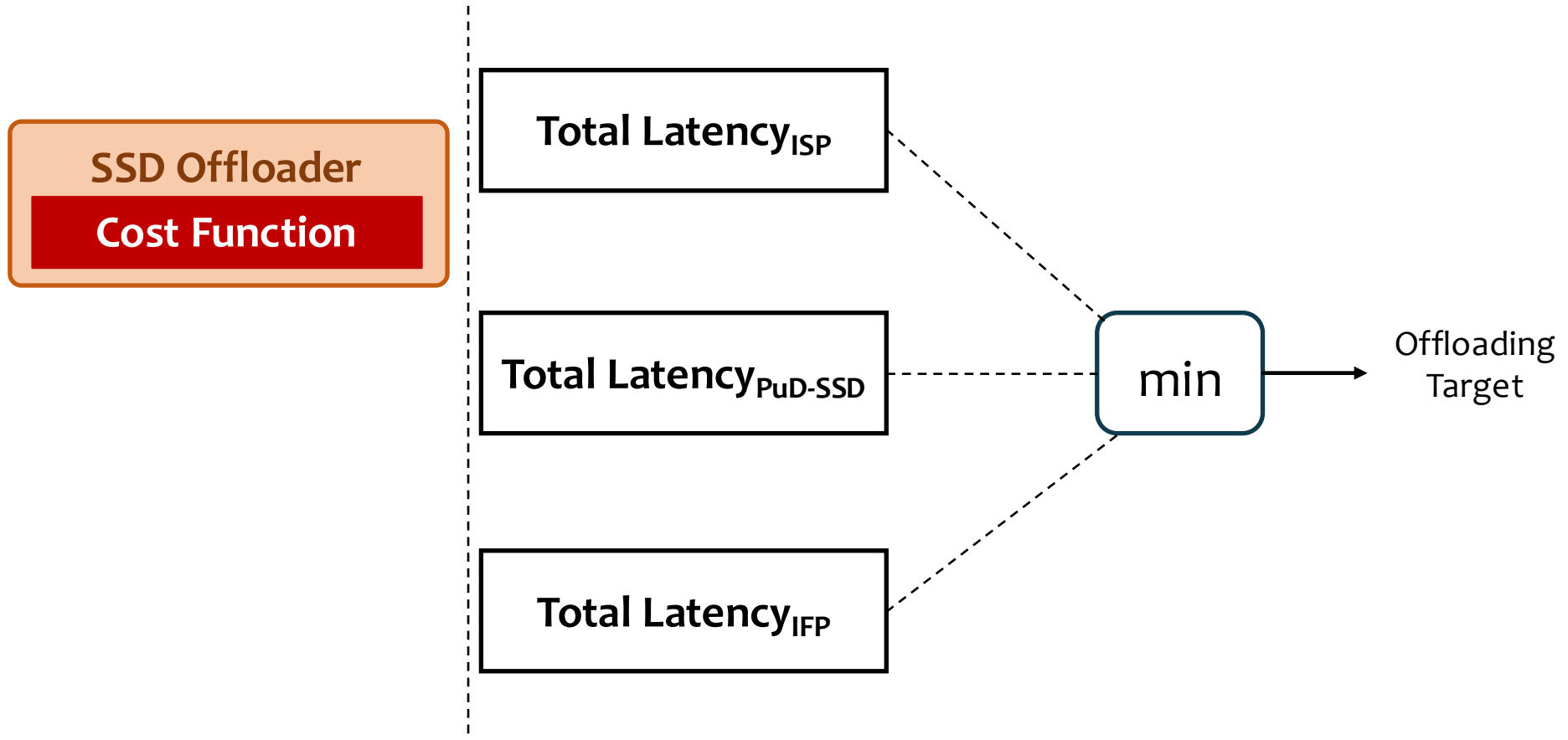
Conduit tracks **both application characteristics and current system conditions** to evaluate the **cost of offloading an instruction to each SSD computation resource**

Conduit: Cost Function



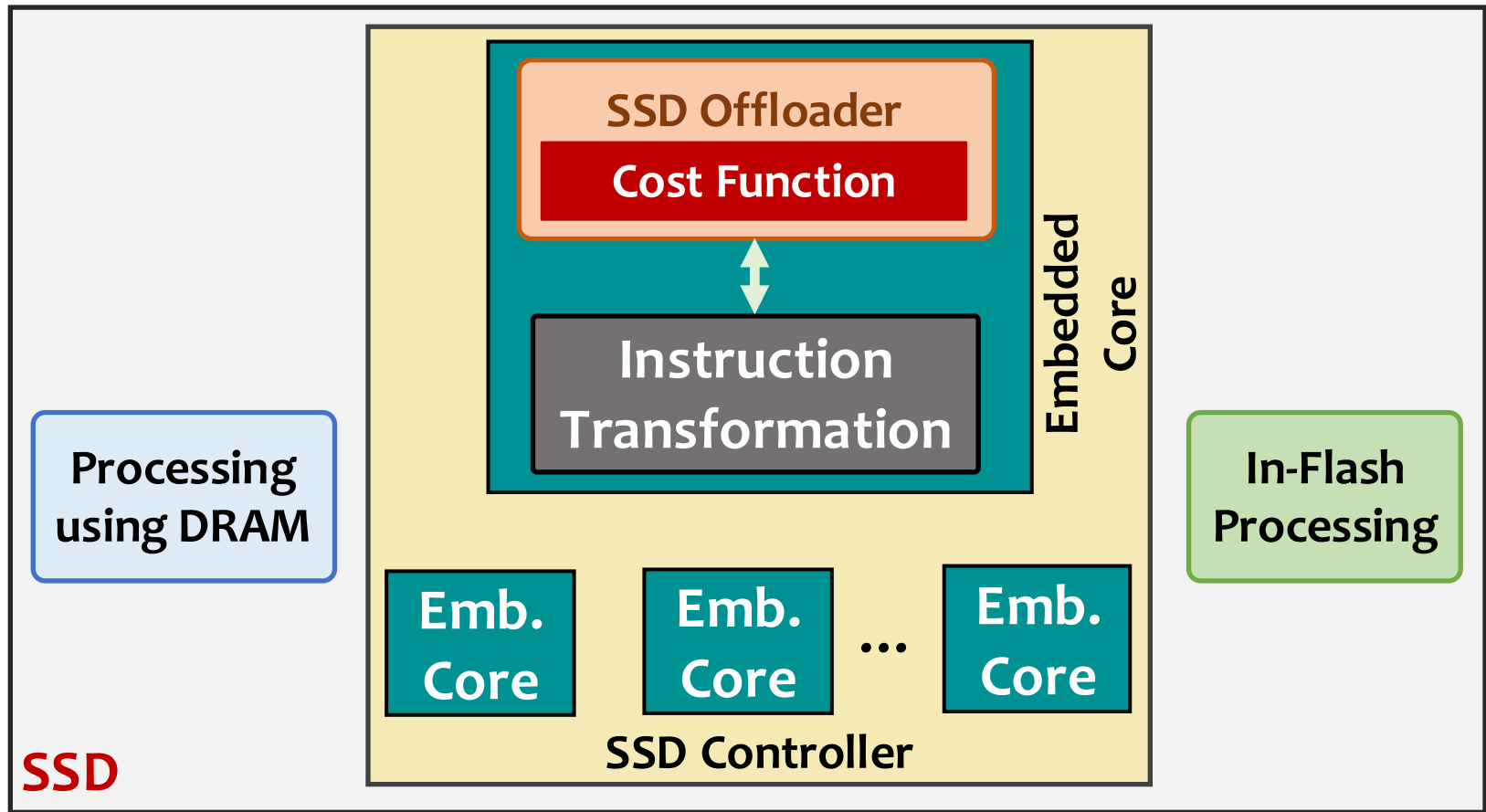
Conduit tracks **both application characteristics and current system conditions** to evaluate the **cost of offloading an instruction to each SSD computation resource**

Conduit: Cost Function



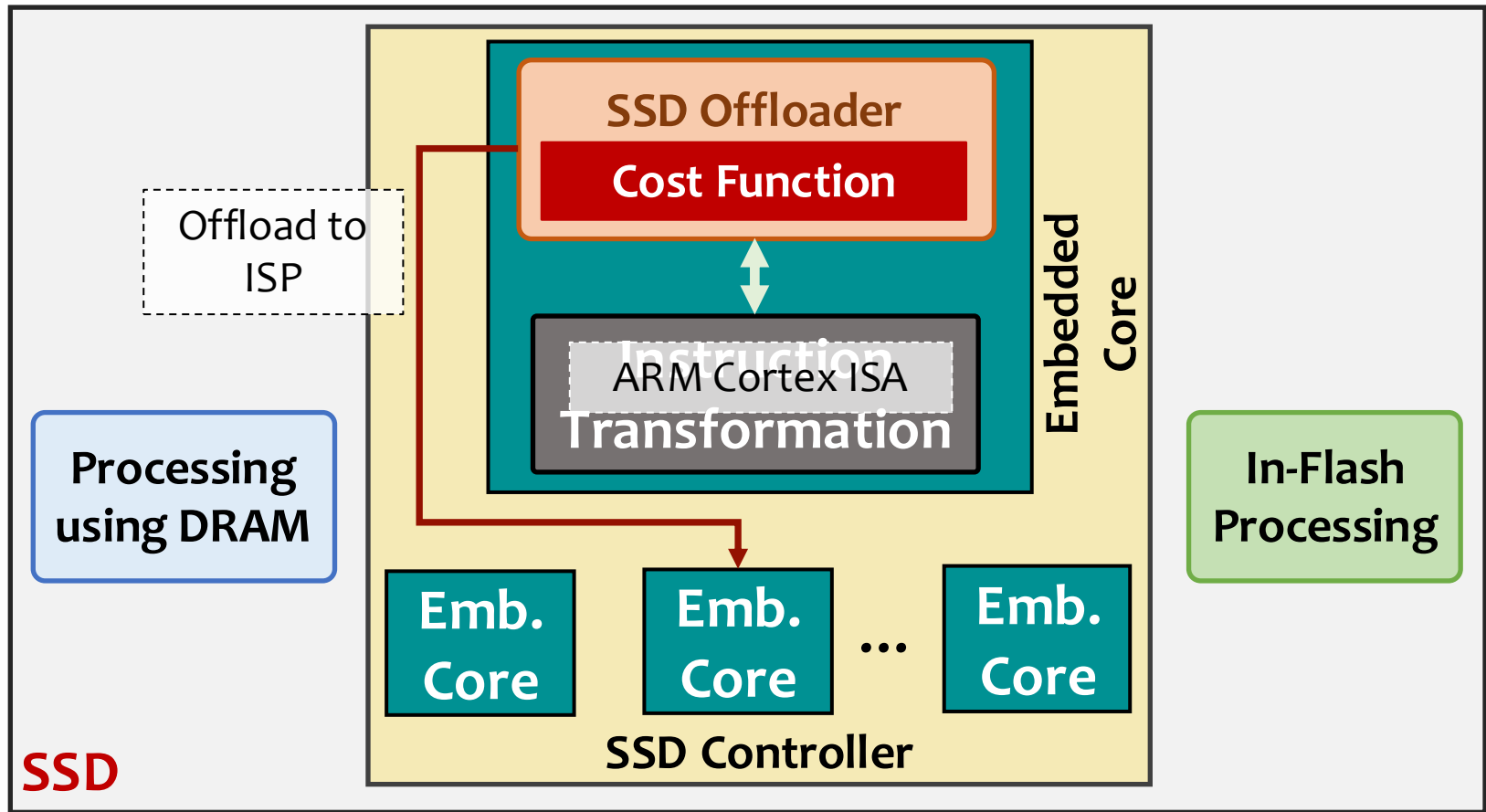
Conduit calculates the **latency of executing each instruction on each computation resource**, and **selects the computation resource with the lowest execution latency**

Conduit: Instruction Transformation (1/4)



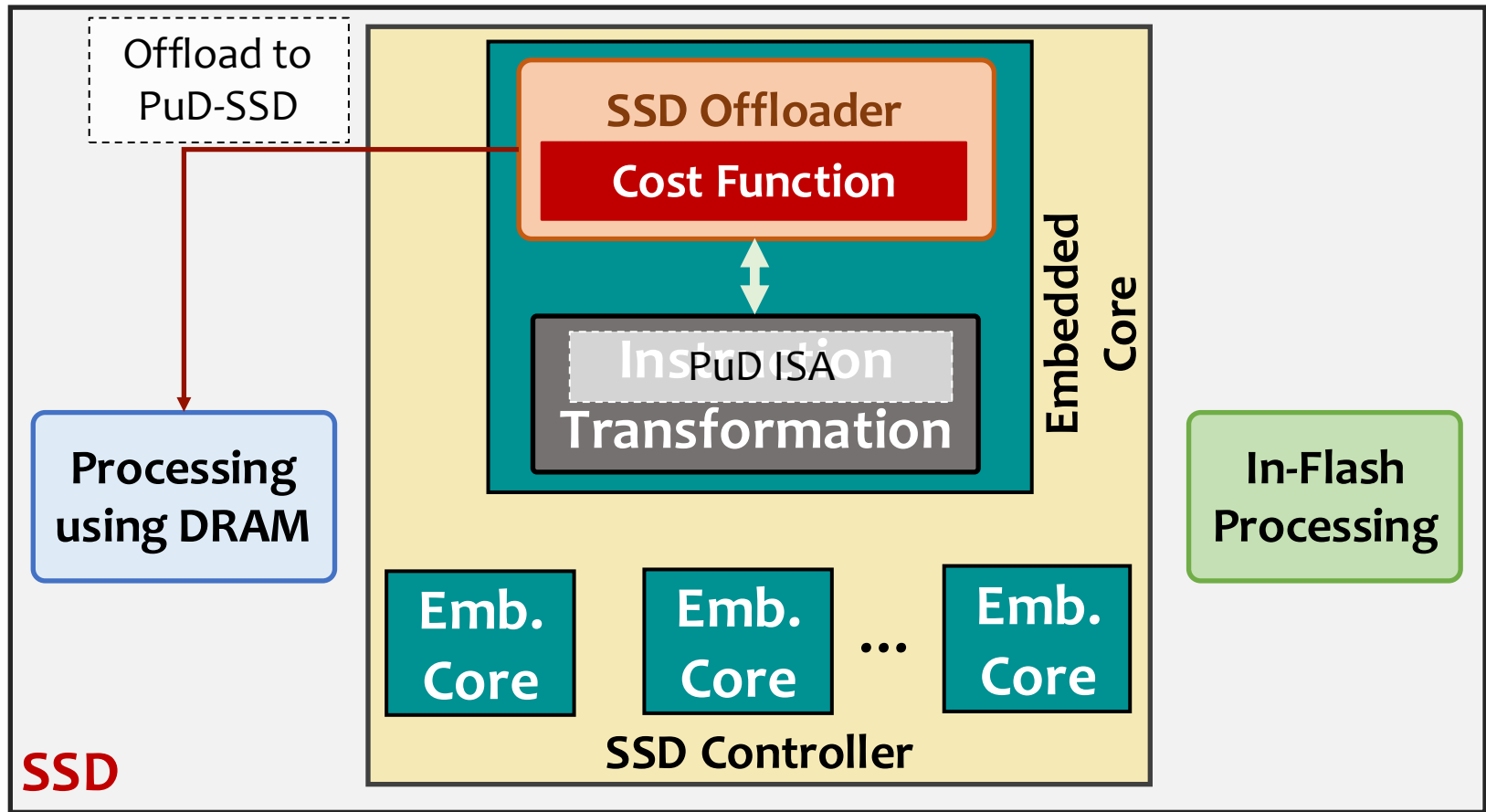
Conduit's instruction transformation unit transforms each vectorized instruction to the **native ISA of the chosen SSD computation resource**

Conduit: Instruction Transformation (2/4)



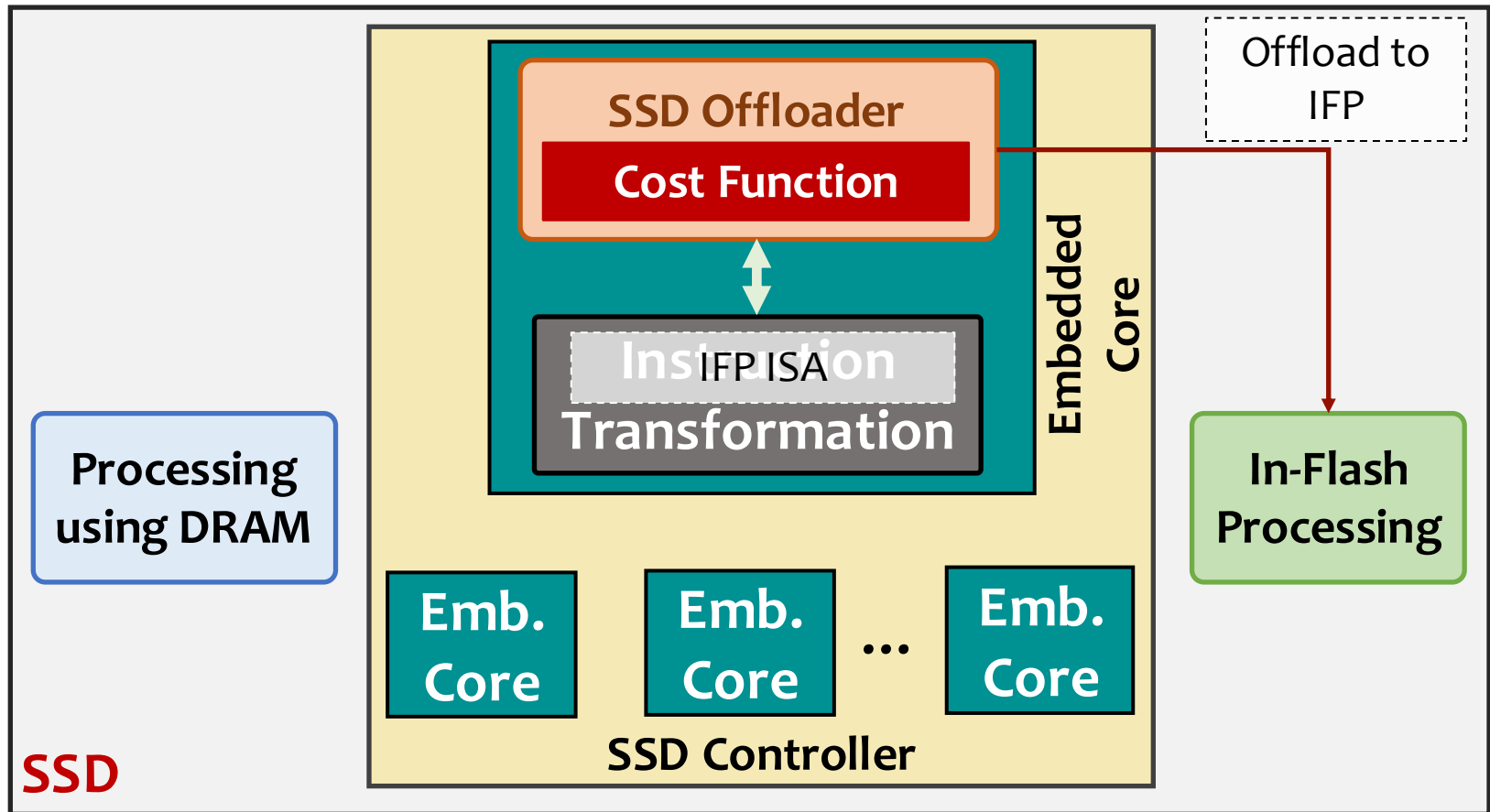
For ISP, Conduit transforms each vector operation to ARM Cortex ISA, and schedules the instruction for execution

Conduit: Instruction Transformation (3/4)



Conduit transforms each vector operation to **ISA extensions** (e.g., `bbop_op` for 2-input arithmetic) from **prior PuD works** (e.g., SIMDGRAM [Hajinazar and Oliveira+, ASPLOS 2021])

Conduit: Instruction Transformation (4/4)



For IFP, Conduit leverages the *MWS* primitives from Flash-Cosmos [Park+, MICRO 2022] and arithmetic operations from Ares-Flash [Chen+, MICRO 2024] for instruction transformation

More in the paper

- Data coherence across SSD computation resources
 - Conduit employs a lazy coherence mechanism
- Data mapping and SSD-internal data movement
- Host-SSD communication
 - Regular I/O mode
 - Computation mode
- Failure handling
- Conduit's extensibility
- Limitations of auto-vectorization
- Conduit's applicability to irregular workloads
- Detailed background on the three NDP paradigms
- Detailed description of binary transfer from the host to the SSD

Conduit: Programmer-Transparent Near-Data Processing Using Multiple Compute-Capable Resources in Solid State Drives

Rakesh Nadig[†] Vamanan Arulchelvan[†] Mayank Kabra[†] Harshita Gupta[†]
Rahul Bera[†] Nika Mansouri Ghiasi[†] Nanditha Rao[†] Qingcai Jiang[†]
Andreas Kosmas Kakolyris[†] Yu Liang^{†‡} Mohammad Sadrosadati[†] Onur Mutlu[†]

[†]*ETH Zürich* [‡]*Inria, Paris*



<https://arxiv.org/pdf/2601.17633>

Talk Outline

Background and Motivation

Conduit

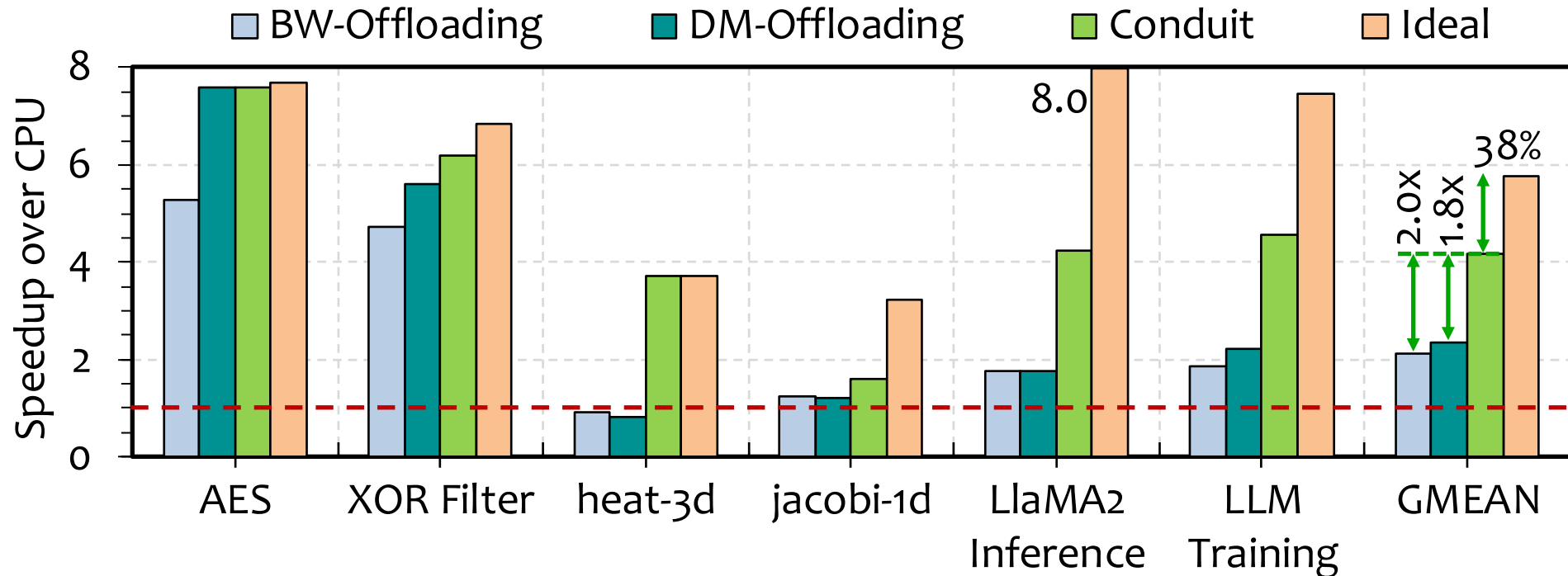
Evaluation

Summary

Evaluation Methodology

- Using an **in-house event-driven SSD simulator** that faithfully models:
 - Core SSD functionalities (e.g., address translation, transaction scheduling) based on **MQSim [Tavakkol+, FAST 2018]**
 - DRAM modeling based on **Ramulator [Kim+, CAL 2015]**
 - Shared resource contention
 - Performance models of three NDP paradigms
- SSD configuration based on **Samsung 980 Pro SSD**
- Six **data-intensive workloads** from
 - **AES, XOR Filter, heat-3d, jacobi-1d, LLaMA2 training and inference**
- Evaluated Policies
 - **Host CPU**
 - **BW-Offloading**: Offloading based on bandwidth utilization (based on TOM [Hsieh+, ISCA 2016], λ -IO [Yang+, FAST 2023])
 - **DM-Offloading**: Offloading based on data movement cost (based on ALP [Mansouri Ghiasi+, TETC 2023], PIMProf [Wei+, DATE 2022])
 - **Conduit**
 - **Ideal Offloading**: No resource contention, resource selection based on lowest computation latency and zero data movement

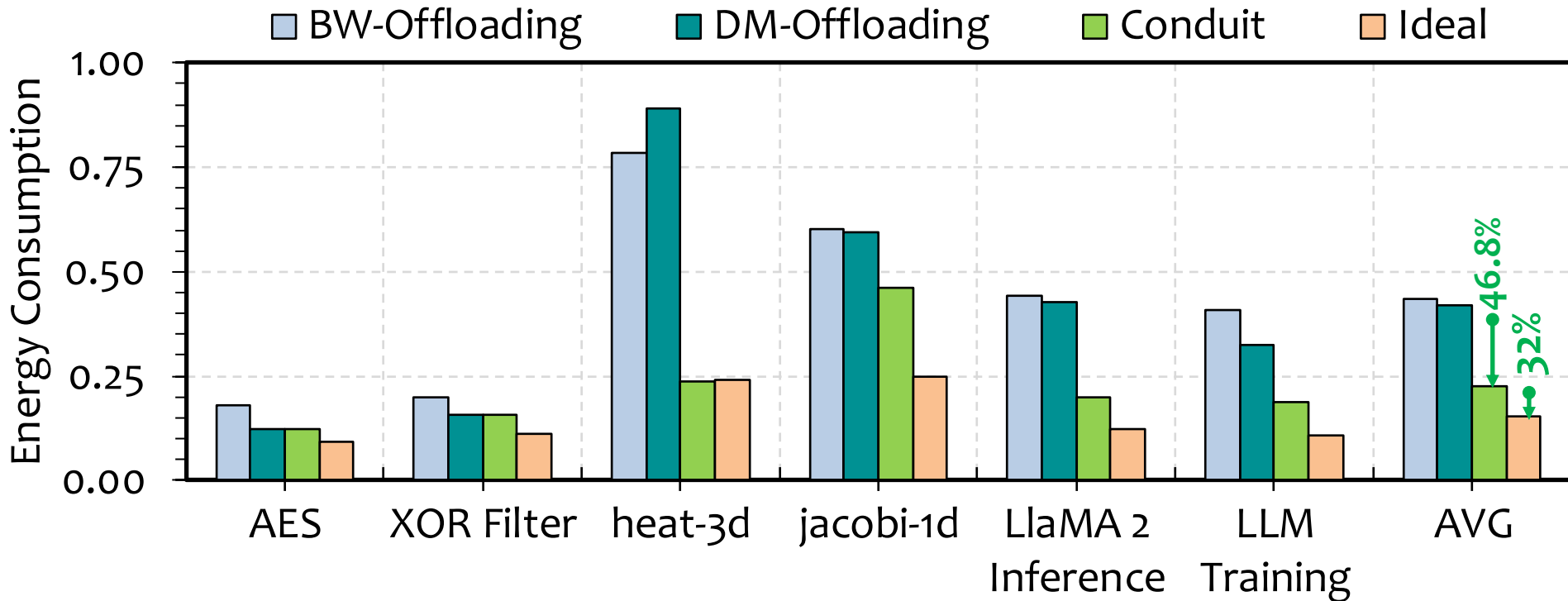
Results: Performance Analysis



Conduit **outperforms** the **best-performing prior approach** by **1.8x** on average by making **holistic offloading decisions**

Conduit **provides 62%** of the **Ideal offloading policy's** performance without the **ideal approach's unrealistic assumptions**

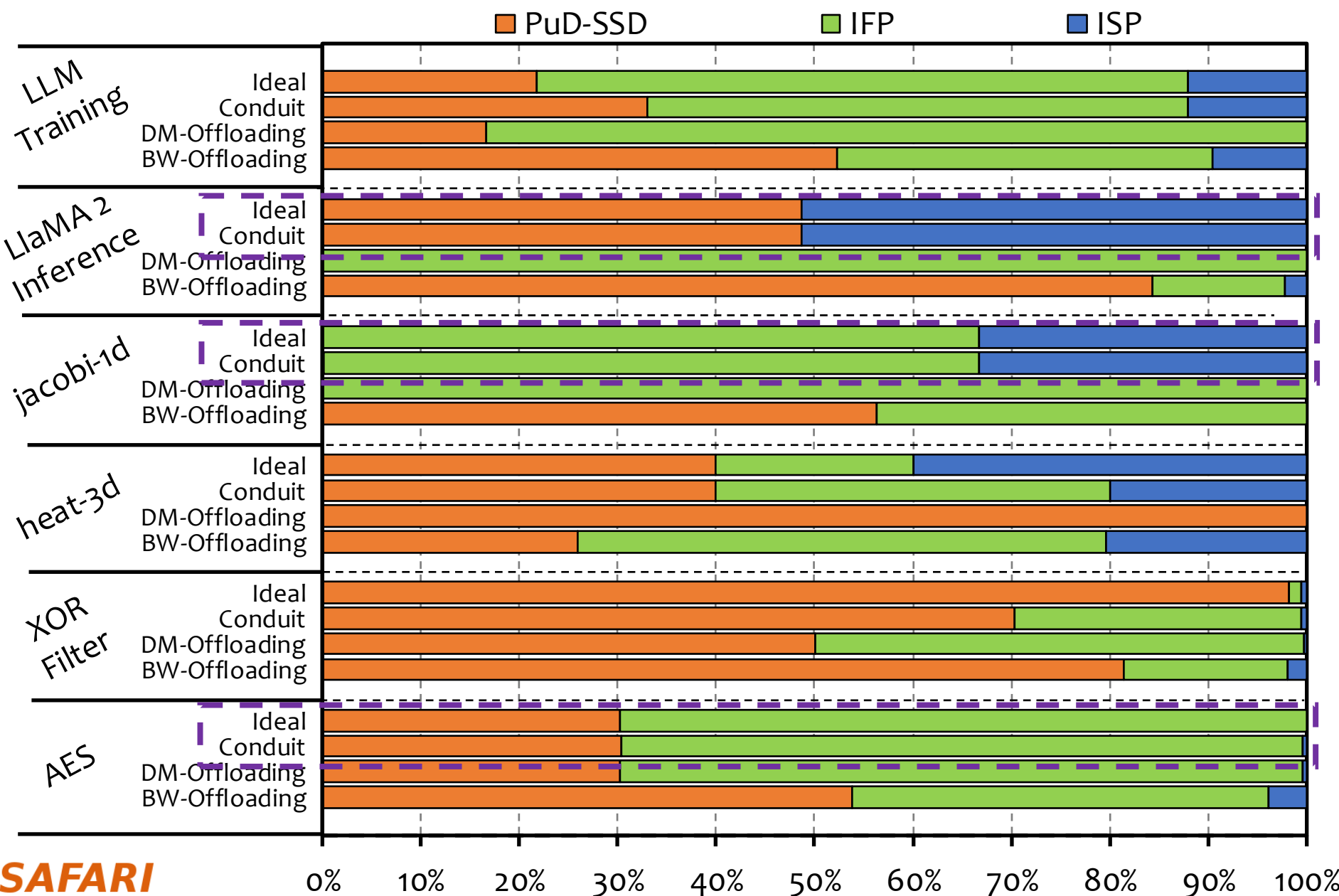
Results: Energy Consumption



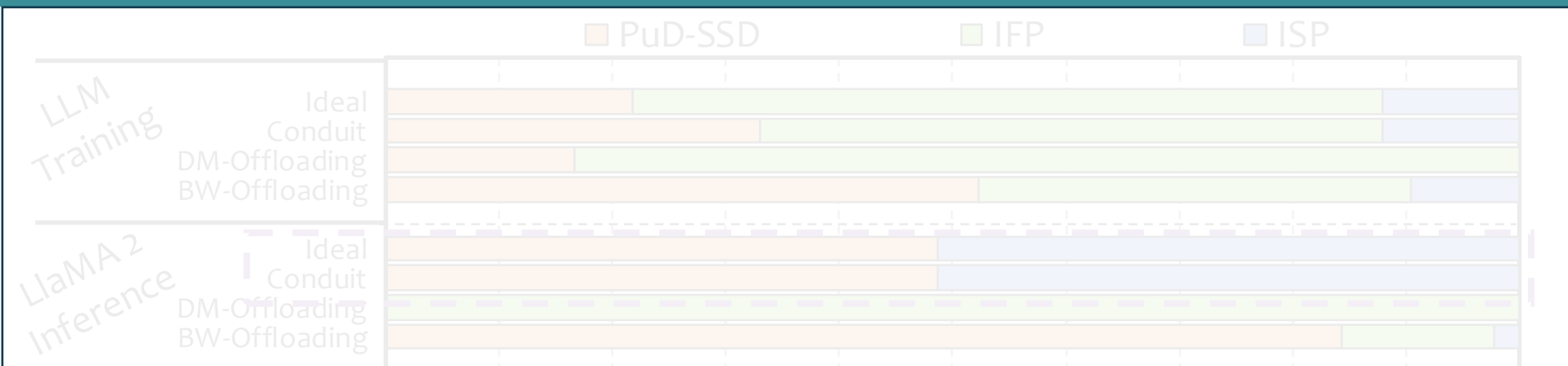
Conduit reduces energy consumption by 46.8% on average compared to DM-Offloading by exploiting all SSD computation resources and reducing resource queueing delays

Conduit provides 68% of the energy efficiency of the Ideal approach

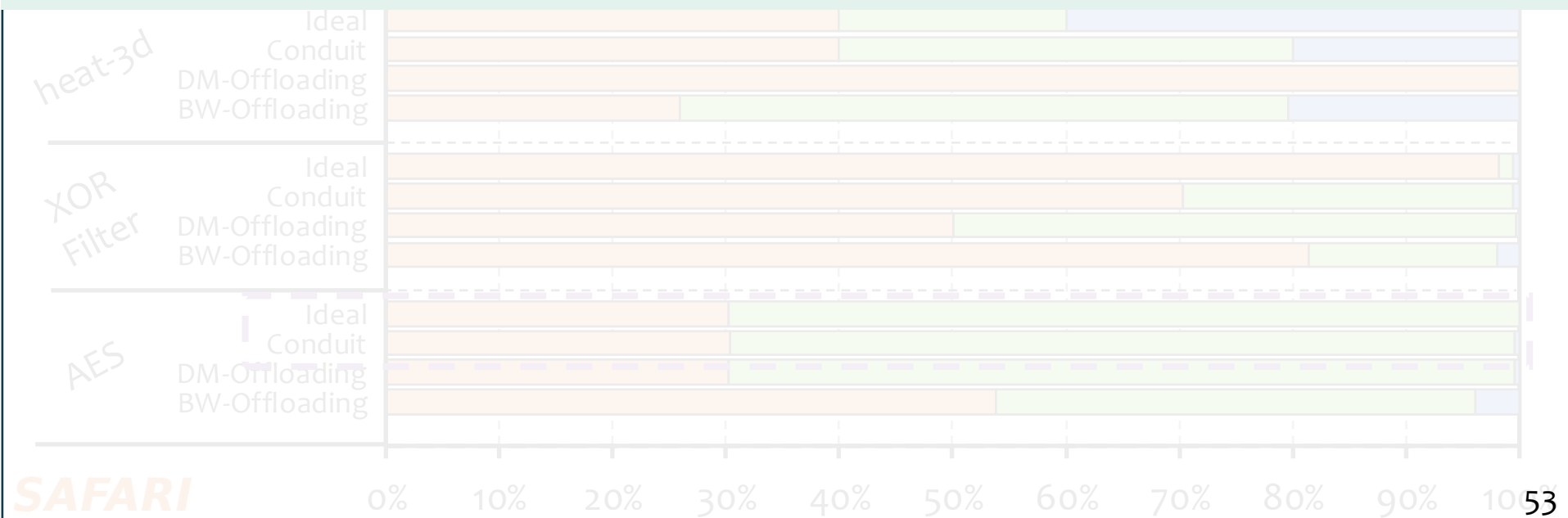
Offloading Decisions



Offloading Decisions



Conduit makes similar offloading decisions as the Ideal offloading approach without the Ideal's unrealistic assumptions



Overhead Analysis

- **Storage Overhead**

17 bytes for storing cost function features in the SSD DRAM

1.5 KiB for storing the instruction transformation mapping of vectorized instructions to the native ISA of all computation resources

- **Runtime Latency Overhead**

3.8 μ s on average for cost function feature collection and instruction transformation for each instruction

- **Area Overhead**

Conduit does not add any area overhead beyond that of the prior techniques it leverages

More in the Paper

- Tail latency evaluation
 - In LLaMA2 inference, compared to DM-Offloading, Conduit reduces 99th percentile latency by 5.6x, and 99.99th percentile latency by 22.3x
- Interaction between the workload and SSD computation resources
- Detailed description of overhead analysis
- Detailed workload description and characteristics
- Detailed description of SSD modeling and NDP extensions

Conduit: Programmer-Transparent Near-Data Processing Using Multiple Compute-Capable Resources in Solid State Drives

Rakesh Nadig[†] Vamanan Arulchelvan[†] Mayank Kabra[†] Harshita Gupta[†]
Rahul Bera[†] Nika Mansouri Ghiasi[†] Nanditha Rao[†] Qingcai Jiang[†]
Andreas Kosmas Kakolyris[†] Yu Liang^{†‡} Mohammad Sadrosadati[†] Onur Mutlu[†]

[†]*ETH Zürich* [‡]*Inria, Paris*



<https://arxiv.org/pdf/2601.17633>

Talk Outline

Background and Motivation

Conduit

Evaluation

Summary

Conduit: Summary



First **general-purpose programmer-transparent NDP framework** that enables **fine-grained computation offloading across multiple SSD computation resources**



Improves performance by 1.8x on average over the **best-performing prior offloading approach**



Reduces energy consumption by 46.8% on average over the **most efficient prior work**



Low latency and storage overheads



Conduit

Programmer-Transparent Near-Data Processing
Using Multiple Compute-Capable Resources
in Solid State Drives

Rakesh Nadig, Vamanan Arulchelvan, Mayank Kabra,
Harshita Gupta, Rahul Bera, Nika Mansouri Ghiasi,
Nanditha Rao, Qingcai Jiang, Andreas Kosmas Kakolyris,
Yu Liang, Mohammad Sadrosadati, and Onur Mutlu

HPCA 2026

Backup Slides

FAQ (1/2)

- Executive Summary
- PuD-SSD
- IFP
- Extended Case Study (With More Compute-Intensive Workloads)
- Compile-Time Preprocessing
- Runtime Offloading
- System Integration

- Evaluated Configurations
- Workload Characteristics
- Results
 - Extended Performance Results
 - Extended Energy Consumption
 - Tail Latency
 - Workload-Computation Resource Interaction



- **Background:** A solid-state drive enables three near-data processing paradigms: processing using embedded cores, processing using DRAM, and in-flash processing
- **Problem:**
 - Prior SSD-based NDP techniques (1) do not exploit the SSD's full computational potential, (2) are application-specific, and (3) are not programmer-transparent
 - Prior main-memory-based NDP offloading techniques provide limited benefits for an SSD because they optimize for limited factors (e.g., bandwidth utilization)
- **Goal:** To enable a programmer-transparent NDP framework in SSDs that (1) offloads computations across multiple SSD computation resources, and (2) makes offloading decisions that are both workload- and system-aware
- **Conduit:** A general-purpose programmer-transparent NDP framework that consists of:
 - compile-time preprocessing to identify offloadable code regions and transform them to wide vector operations,
 - runtime offloading that maps computations to the most suitable SSD computation resource based on workload and system factors
- **Key Results:** We evaluate Conduit and two prior NDP offloading techniques using an in-house event-driven SSD simulator on six data-intensive workloads
 - Compared to the best prior offloading policy, Conduit improves performance by 1.8x on average and reduces energy consumption by 46% on average



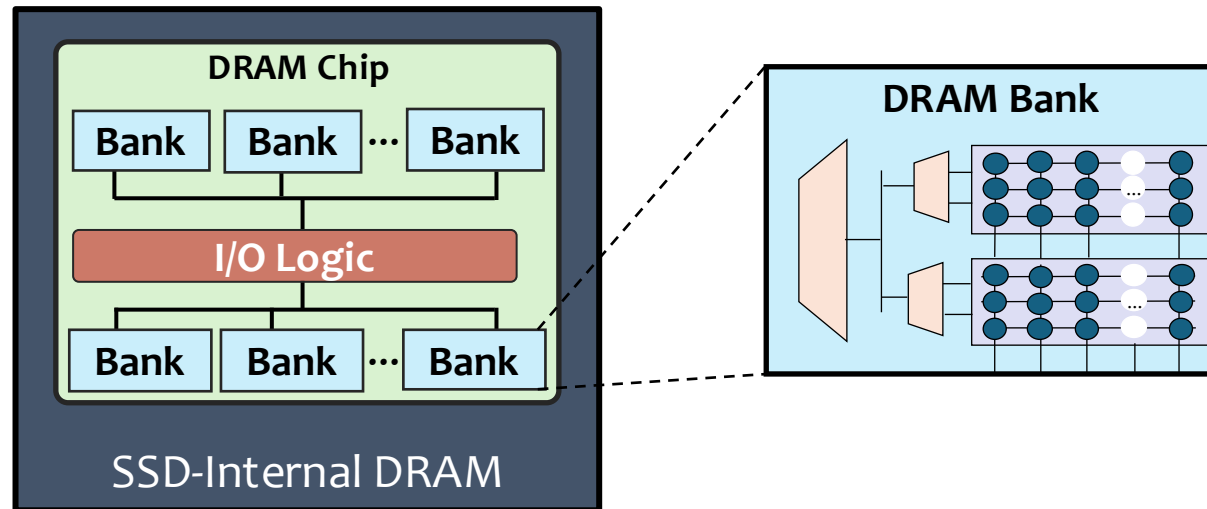
The **embedded cores** in the SSD controller typically execute **FTL firmware** and handle **host communication**

The embedded cores can be **repurposed** to perform **general-purpose computations**

Many prior works show that ISP accelerates various tasks including **filtering, aggregation, encryption, and search**

Limited SIMD parallelism (e.g., 32-bit registers in ARM Cortex R5)

Processing-Using-DRAM in SSD (PuD-SSD)



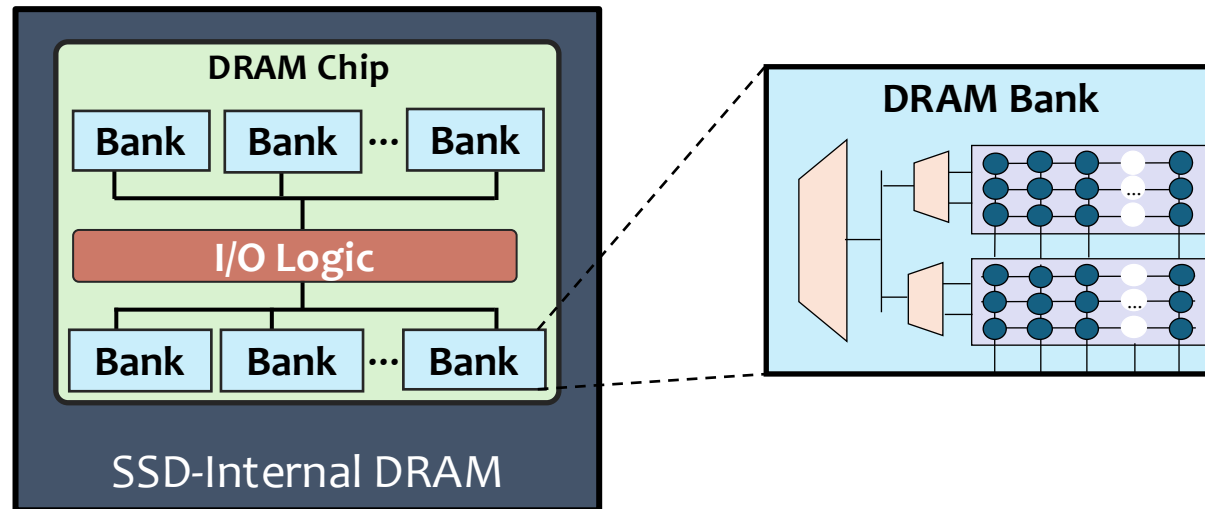
PuD techniques leverage **DRAM's operational principles** to perform computations within the memory array

Prior works (e.g., SIMDGRAM^[1], MIMDRAM^[2]) enable **bulk-bitwise** and **complex arithmetic operations** entirely within the DRAM chips

[1] Hajinazar+, "SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM," In ASPLOS, 2021

[2] Oliveira+, "MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing," In HPCA, 2024

Processing-Using-DRAM in SSD (PuD-SSD)



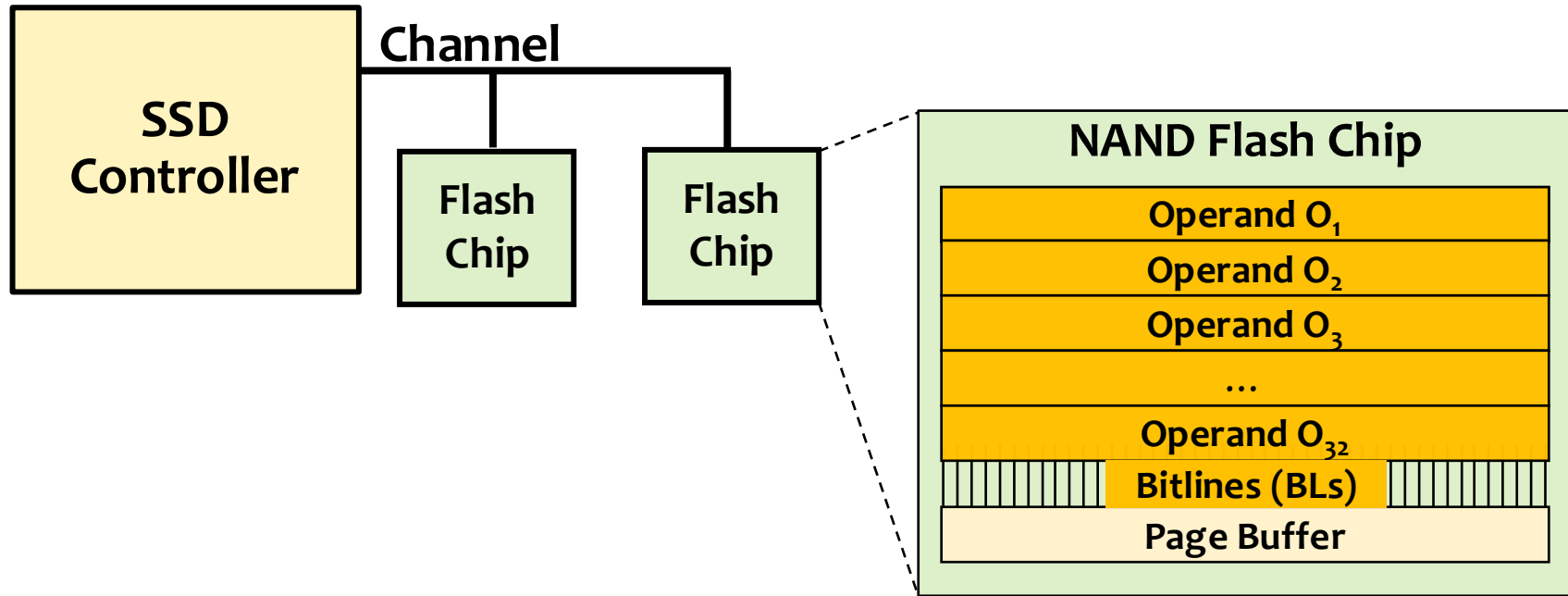
PuD-SSD requires transferring operands from flash chips to the DRAM via bandwidth-limited flash channels

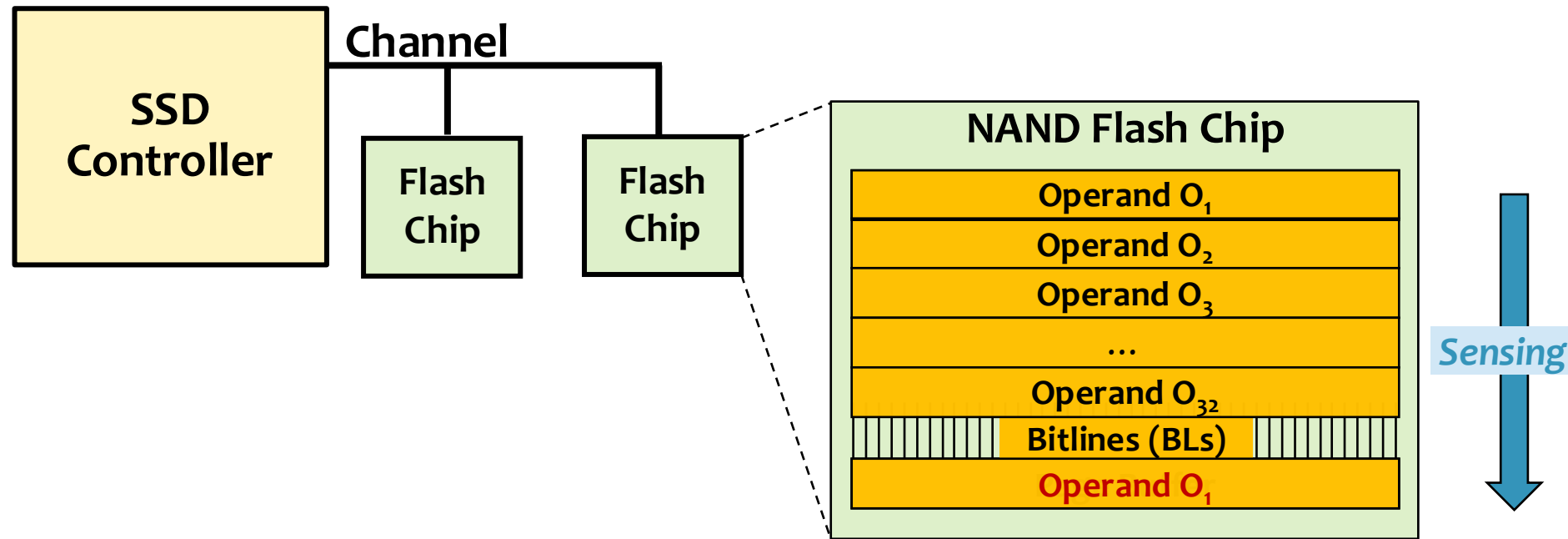
These data transfers incur latency and energy overheads, and cause contention in flash channels

[1] Hajinazar+, "SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM," In ASPLOS, 2021

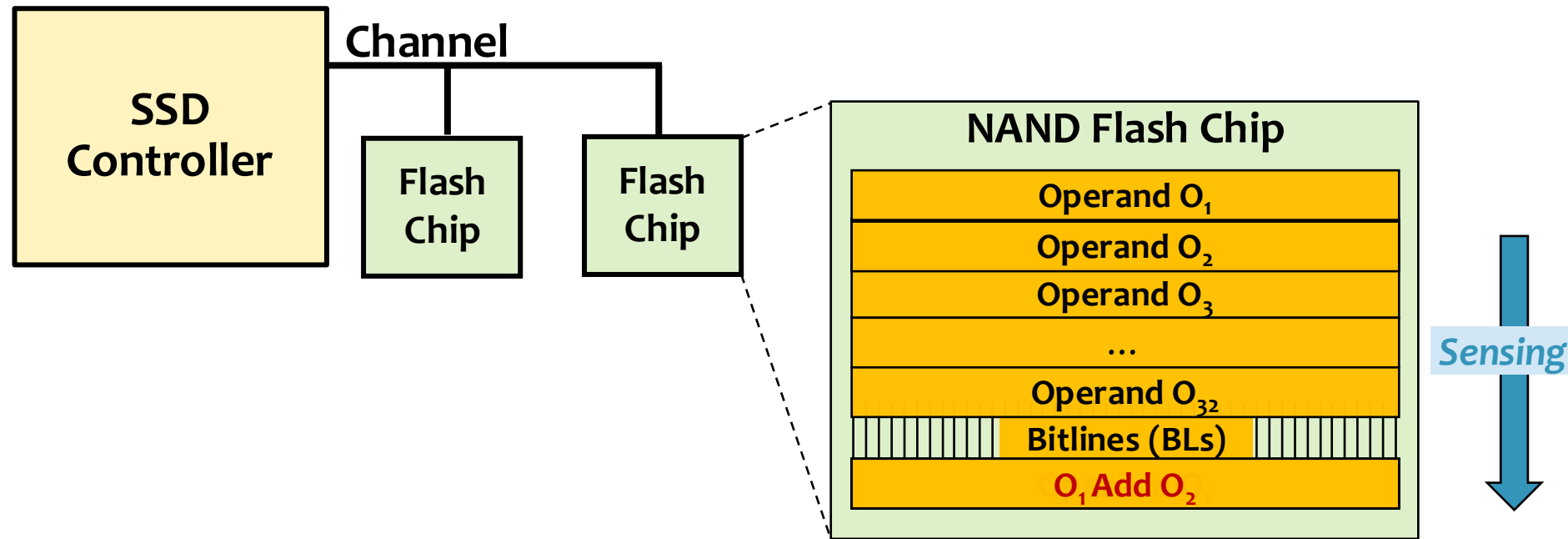
[2] Oliveira+, "MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing," In HPCA, 2024

In-Flash Processing (IFP) (1/2)



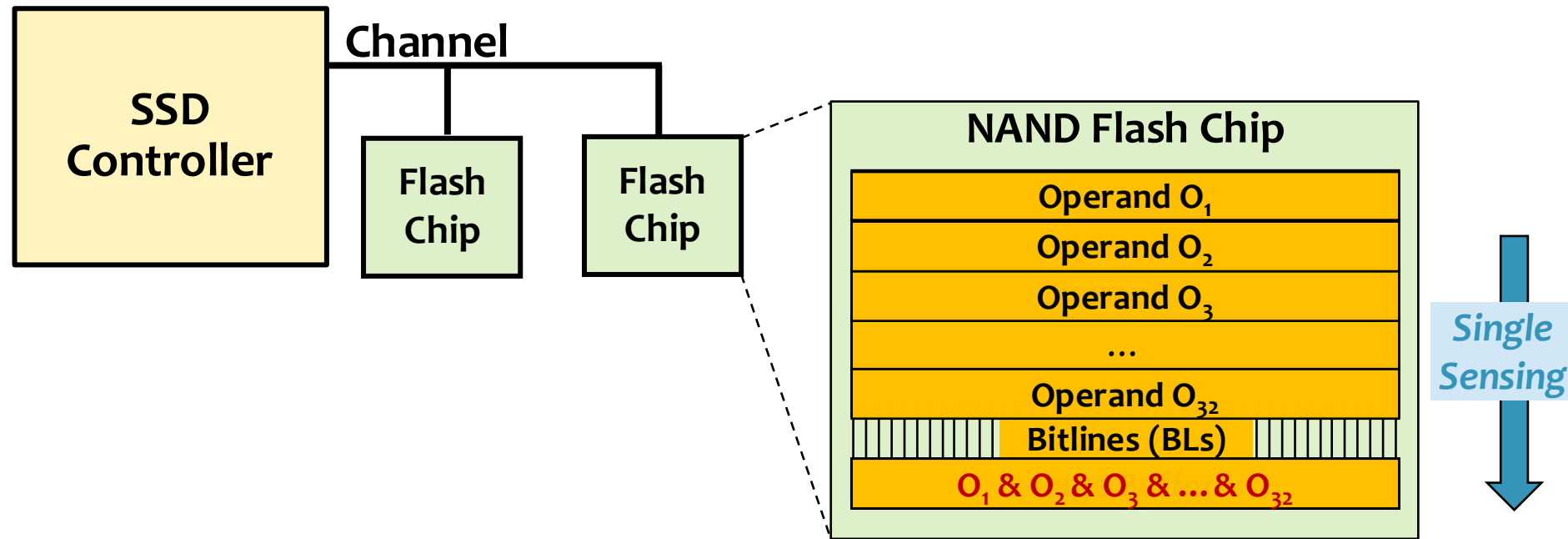


Several IFP techniques (e.g., Ares-Flash^[3]) enable arithmetic operations inside NAND flash chips by controlling the latching circuit of the page buffer



Several IFP techniques (e.g., Ares-Flash^[3]) enable arithmetic operations inside NAND flash chips by controlling the latching circuit of the page buffer

In-Flash Processing (IFP) (2/2)



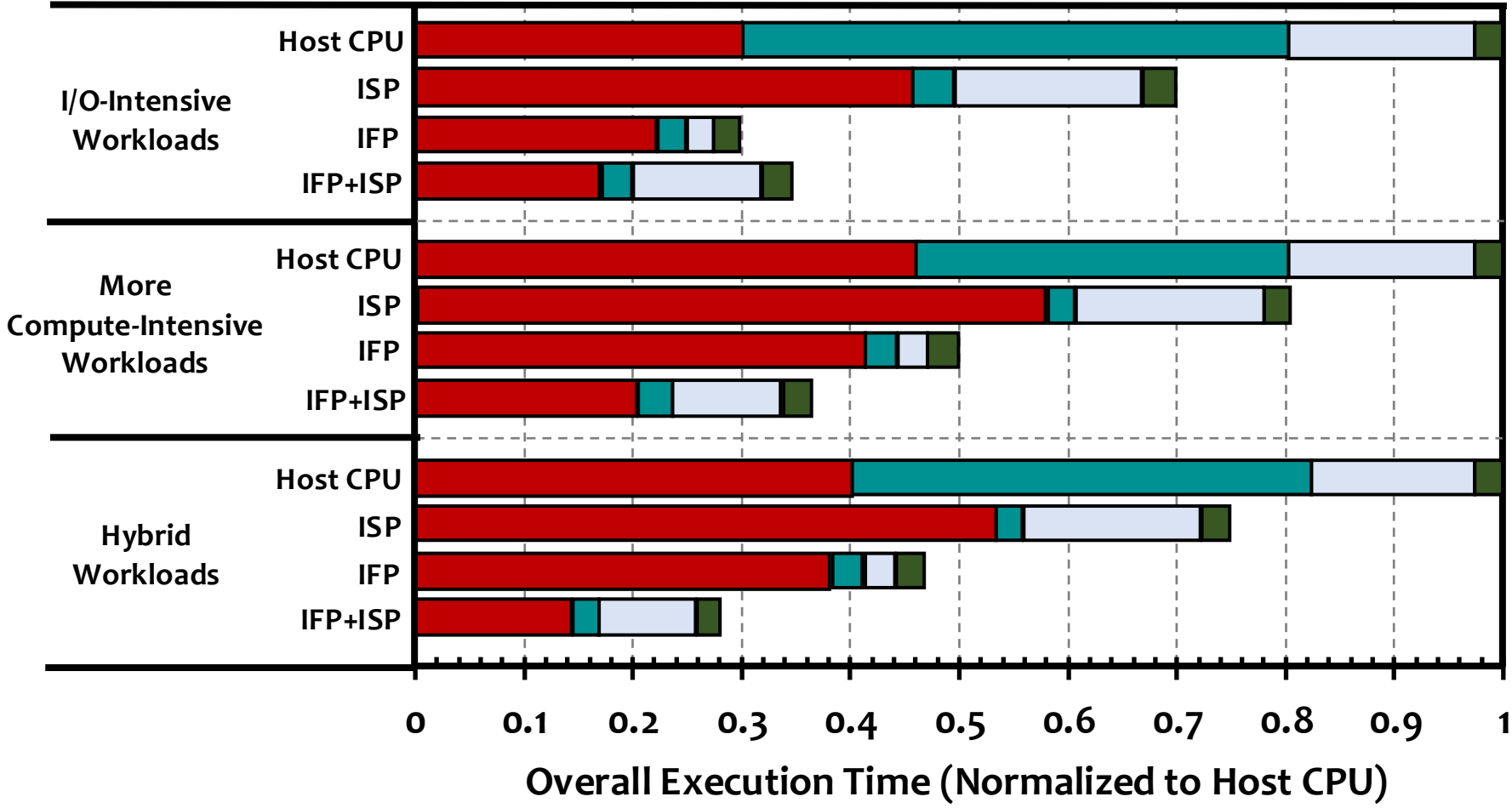
Flash-Cosmos^[4] enables bulk-bitwise operations inside NAND flash chips by simultaneously reading multiple pages within the flash chips

IFP techniques support limited set of operations and are less suitable for complex workloads

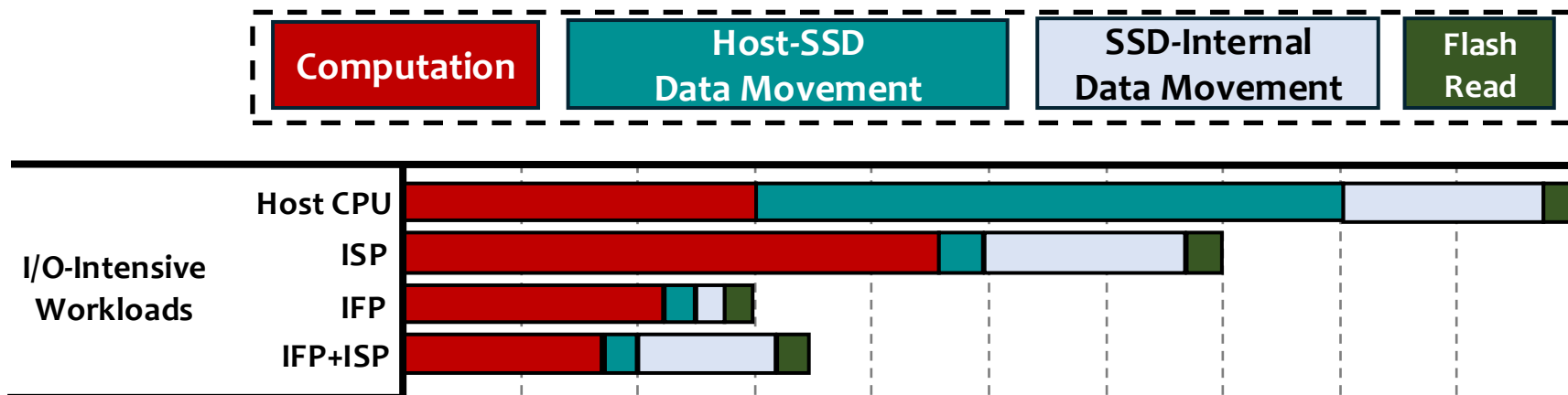


- We examine the **performance bottlenecks** when **offloading computations** to **multiple SSD computation resources**
- **Evaluated Policies**
 - Host CPU
 - In-storage processing (ISP)
 - In-flash processing (IFP)
 - Offloading to both ISP and IFP (ISP + IFP)
- **Evaluated Workloads**
 - **I/O-Intensive workloads**: Workloads dominated by data transfer (e.g., databases, bitmap indices)
 - **More compute-intensive workloads**: Workloads dominated by computations (e.g., encryption, matrix multiplication)
 - **Hybrid workloads**: Workloads that involve both computation and data movement (e.g., aggregation, sorting)

Extended Case Study (2/5)

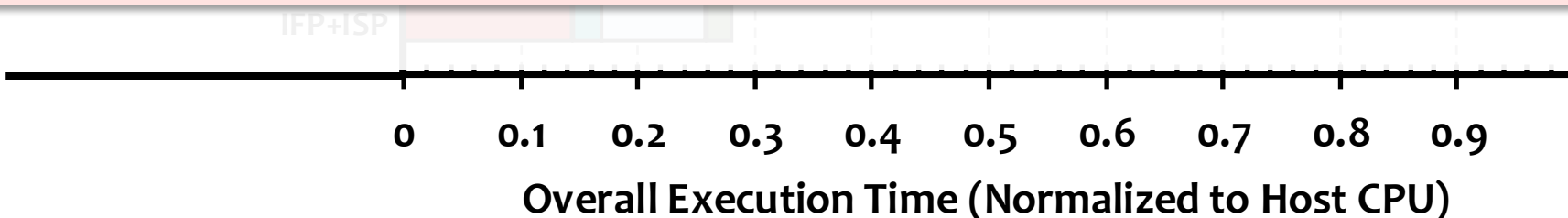


Extended Case Study (3/5)

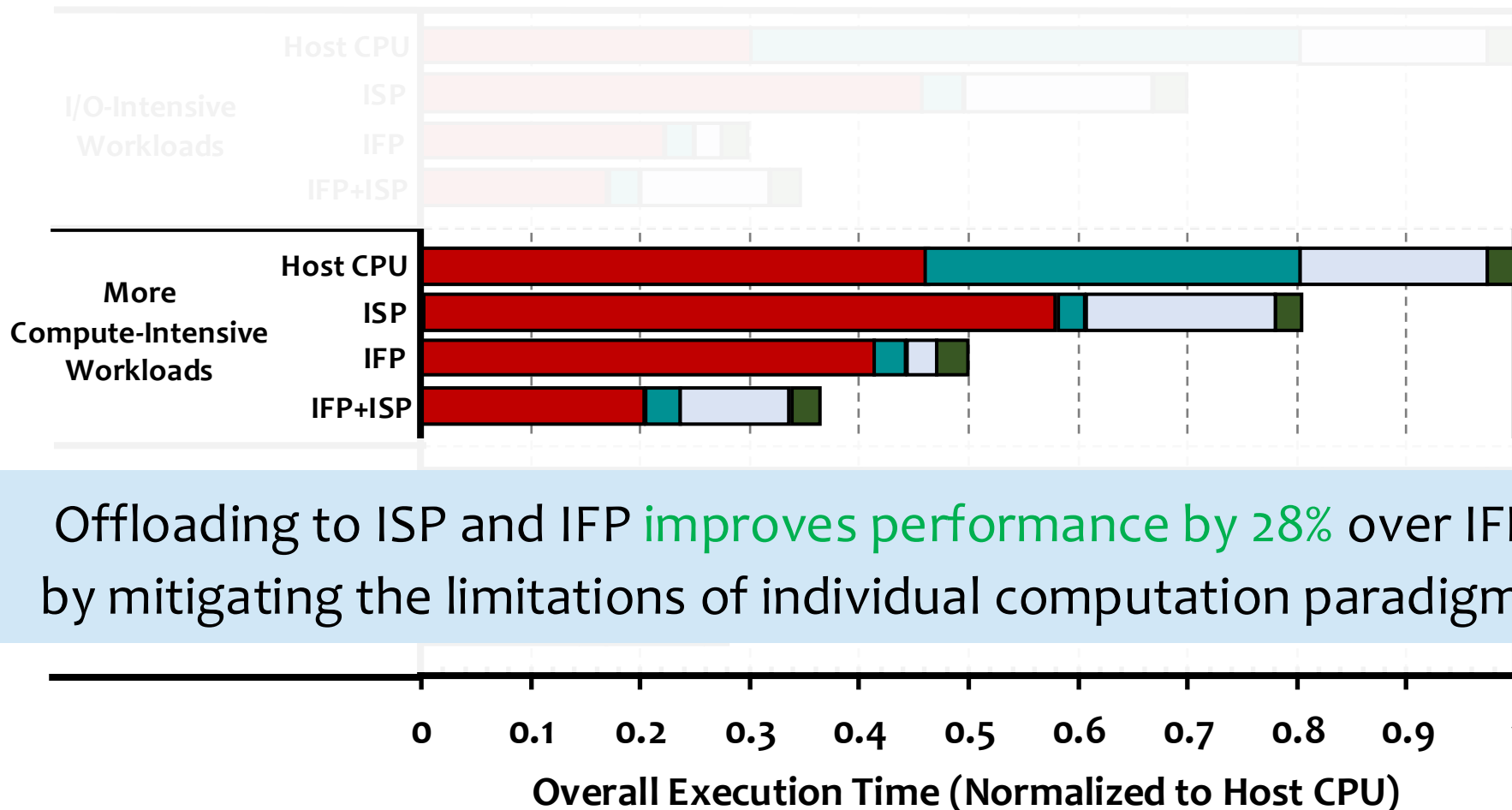


Offloading to ISP and IFP in I/O-intensive workloads
reduces performance by 15% compared to IFP

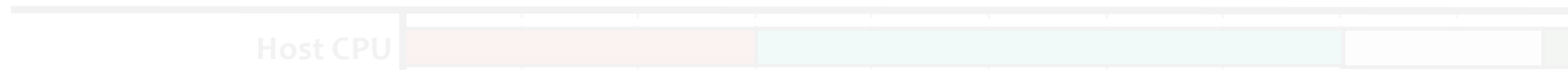
Offloading to multiple computation resources incurs
additional data movement between flash chips and SSD controller



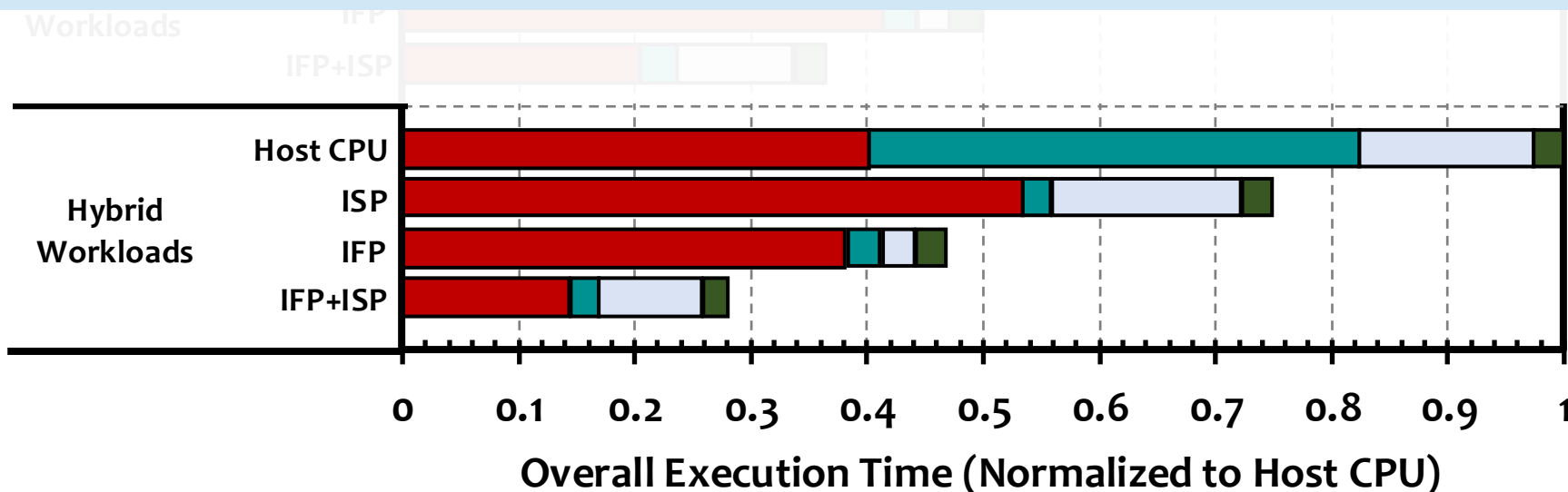
Extended Case Study (4/5)



Extended Case Study (5/5)



Offloading to ISP and IFP improves performance by 40% over IFP by reducing data movement and mapping computations to the most suitable computation resource





- **Compiler Flags**

- `-O3 -g -mllvm -force-vector-width=4096 -force-vector-interleave=1 -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize`
- `-force-vector-width=4096` - Aligns each vector to a typical NAND flash page (vector width of 4096 for 32-bit operands)
 - Simplifies data movement across computation resources
 - Avoids misaligned accesses
 - Matches L2P mapping granularity
- `-force-vector-interleave=1` flag controls loop unrolling to enable instruction-granularity offloading



Feature	Description
Operation Type	Type of operation (e.g., bulk-bitwise, arithmetic, predication)
Operand Location	Current location of the operand (flash chips or SSD DRAM)
Data Dependence Delay ($delay_{dd}$)	Delay for operand availability
Resource Queueing Delay ($delay_{queue}$)	Delay caused by pending instructions in a computation resource's execution queue
Data Movement Latency ($latency_{dm}$)	Data movement latencies if the operands are not in the target computation unit
Expected Computation Latency ($latency_{comp}$)	Estimated instruction execution latency on a computation resource

Table 1: Features Used by the Cost Function to Select the SSD Computation Resource.



- SSD controller has **real-time knowledge** of all computation resources and shared resource contention
- We explored **offloading in the host CPU**
- **Limitations** of offloading in the host CPU
 - **No real-time status** of the SSD internals in the host
 - **Significant traffic** if the SSD needs to transfer information to the host frequently
 - We need to **think of a better way** to do this in the future



- **Transferring the binary**
 - Repurpose fw-download and fw-commit commands
 - Extend these commands with a new flag to identify Conduit's executable
- **Data mapping and SSD-internal data movement**
 - Conduit adheres to the data layout requirements of prior techniques
- **Host-SSD communication**
 - *Regular I/O mode*: Host I/O and FTL operations are executed
 - *Computation mode*: Host I/O is suspended while performing maintenance tasks such as garbage collection and wear-leveling



- **Data Coherence between SSD computation resources**
 - Conduit employs a **lazy coherence mechanism**
 - Conduit synchronizes data across SSD computation resources only when
 - **Another computation resource requests data**
 - Computation result must be **transferred to the host**
 - Data must be **evicted from SSD DRAM, page buffers or SSD controller registers**
 - FTL initiates **garbage collection**
 - SSD undergoes **power cycle**



- **Data Coherence between SSD computation resources**
 - Conduit maintains coherence at page-granularity by extending the logical-to-physical mapping table to store:
 - *Owner* - SSD computation resource that holds the latest version of the page
 - *State* – page's modification status (i.e., clean or dirty)
 - *Version* – 1-byte monotonically-increasing counter to order updates and detect stale copies



- **Failure Handling**

- For transient failures (e.g., instruction abort, ECC correction failures), the SSD controller maintains completion and status flags, and replays the instruction in a different computation resource
- For permanent faults (e.g., bad blocks), Conduit relies on existing FTL recovery mechanisms such as bad-block remapping, garbage collection and wear-leveling



Host CPU (Real System)	Intel(R) Xeon(R) Gold 5118
	x86-64 [187], 6 cores, out-of-order, 3.2 GHz
	<i>L1 Data + Inst. Private Cache</i> : 32kB, 8-way, 64B line
	<i>L2 Private Cache</i> : 256kB, 4-way, 64B line
	<i>L3 Shared Cache</i> : 8MB, 16-way, 64B line
Host GPU (Real System)	NVIDIA A100 GPU [188], [189], NVIDIA Ampere Architecture [190] (7nm)
	108 Streaming Multiprocessors, 1.4 GHz base clock;
	<i>L2 Cache</i> : 40 MB; Main Memory: 40GB HBM2 [191], [192]
Host Main Memory (Real System)	32GB DDR4-2400, 4 channels, 1 rank, 16 banks;
	Peak throughput: 19.2 GB/s;
	Latency : T_{bbop} : 49 ns; Energy : E_{bbop} : 0.864 nJ; where <i>bbop</i> is bulk bitwise operation
SSD [87] (Simulated)	48-WL-layer 3D TLC NAND flash-based SSD; 2 TB
	External Bandwidth : PCIe 4.0, 8GB/s
	NAND Config : 8 channels; 8 dies/channel; 2 planes/die; 2,048 blocks/plane; 196 (4×48) WLs/block; 4 KiB/page
	Flash Channel Bandwidth : 1.2 GB/s
	Latency : T_{read} (SLC mode): 22.5 μ s [10]; $T_{AND/OR}$: 20 ns [14]; $T_{latchtransfer}$: 20 ns [14]; T_{XOR} : 30 ns [10]; T_{DMA} : 3.3 μ s;
	Energy : E_{read} (SLC mode): 20.5 μ J/channel [10]; $E_{AND/OR}$: 10nJ/KB [14]; $E_{latchtransfer}$: 10nJ/KB [14]; E_{XOR} : 20nJ/KB [10]; E_{DMA} : 7.656 μ J/channel;
SSD DRAM (Simulated)	2GB LPDDR4-1866 DRAM cache; 1 channel, 1 rank, 8 banks
	Latency : T_{bbop} : 49 ns; Energy : E_{bbop} : 0.864 nJ; where <i>bbop</i> is bulk bitwise operation
SSD Controller Cores	ARM Cortex-R8 series @1.5GHz; 5 Cores [94]



- **AES**
 - 256-bit encryption and decryption algorithm with **high data reuse** and **low-latency bitwise operations**
- **XOR Filter**
 - A probabilistic data structure for fast membership tests
 - Dominated by **arithmetic and predication operations**
- **heat-3d**
 - **3D stencil computation** that repeatedly updates each grid point using its neighbors
 - **Moderate-to-high arithmetic intensity** with high data reuse
- **jacobi-1d**
 - **1D stencil computation solver** that updates each element using values from its immediate neighbors
 - **Moderate-to-high latency operations**



- **LlaMA2 Inference**

- INT8 inference of a 7B-parameter LlaMA2 model
- High and medium latency operations

- **LLM Training**

- Bandwidth-intensive INT8 training of 7B-parameter model characterized by frequent weight updates, data movement, and a combination of arithmetic and control-intensive operations
- Quantization enables complete execution of all workloads within the SSD
- Memory footprint of each workload exceeds the SSD capacity by 2x
 - Induces resource contention and data movement scenarios



Workload	Avg. Reuse	Low Latency Operations	Medium Latency Operations	High Latency Operations	Units
AES	15.2	87%	13%	0%	Single
XOR-Filter	2.0	1%	98%	1%	Single
heat-3D	16	0%	60%	40%	Multiple
jacobi-1D	3	0%	67%	33%	Multiple
LlaMA2 Inference	1.8	0%	53%	47%	Multiple
LLM Training	5.2	0%	88%	12%	Multiple

- **LLM Inference**

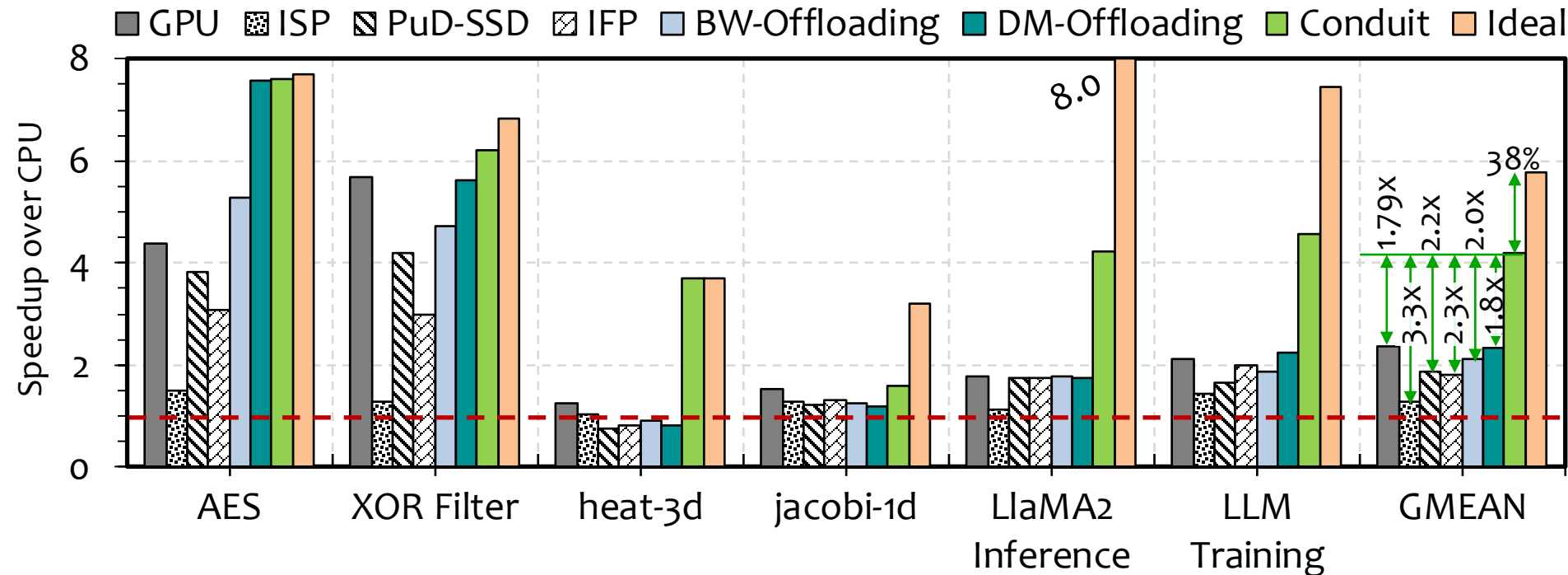
- Linear layers (Q, K, V, FFN)
 - Matrix multiplications, high arithmetic intensity
- Attention-Score computation
 - Dot products
- Element-wise operations: scaling, residual add...
- Layer normalization
 - Accumulation, scale and shift
- Partially vectorizable – reduction, exponentiation, normalization
- Not vectorizable
 - Token selection and sampling
 - Branch-heavy, data-dependent



- **LLM Training**

- Forward Pass
 - Vectorizable
- Backpropagation
 - Gradient computation: Dense matrix computations, high reuse, arithmetic-heavy
- Weight updates
 - Element-wise arithmetic (SIMD friendly)
- Optimizer logic
 - Conditionals, state updates and control flow

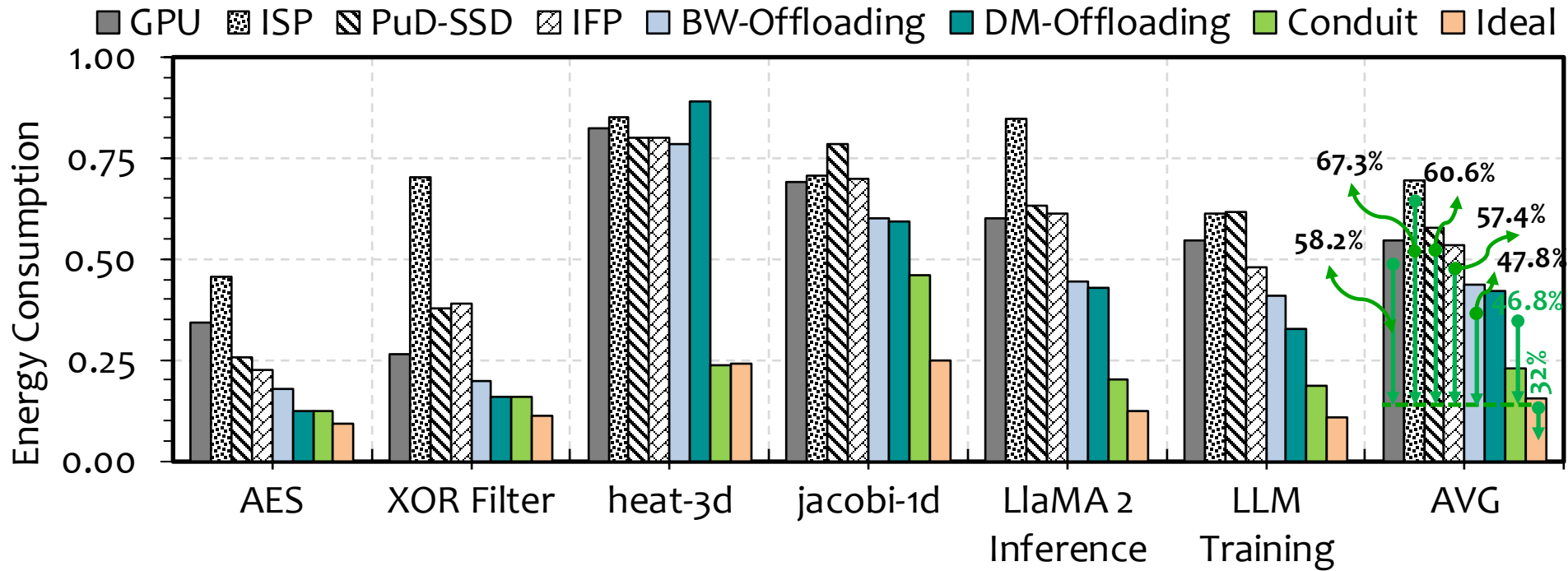
Extended Results: Performance Analysis



Conduit **outperforms** the **best-performing prior approach** by **1.8x** on average by making **holistic offloading decisions**

Conduit **provides 62%** of the **Ideal offloading policy's** performance without the **ideal approach's unrealistic assumptions**

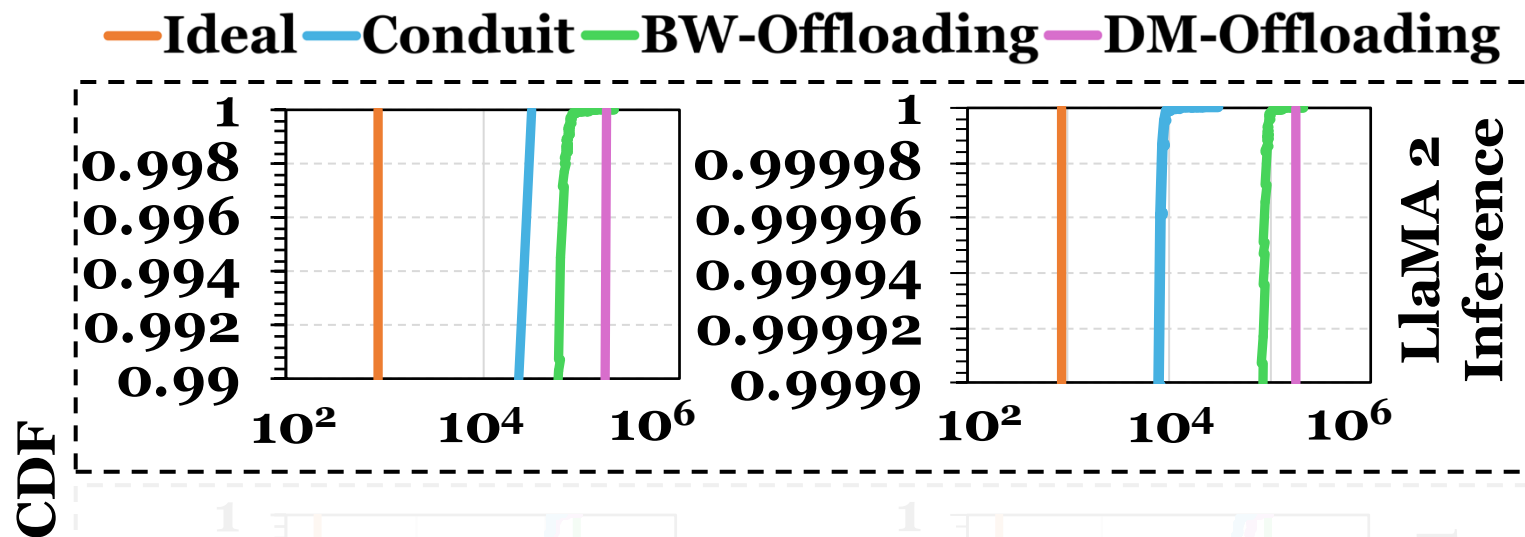
Extended Results: Energy Consumption



Conduit reduces energy consumption by 46.8% on average compared to DM-Offloading by reducing resource queueing delays and exploiting all SSD computation resources

Conduit provides 68% of the energy efficiency of the Ideal approach

Tail Latency Evaluation (1/2)



In LlaMA2 inference, compared to DM-Offloading, Conduit reduces **99th percentile latency by 5.6x**, and **99.99th percentile latency by 22.3x**

(a) 99th Percentile Access Latency

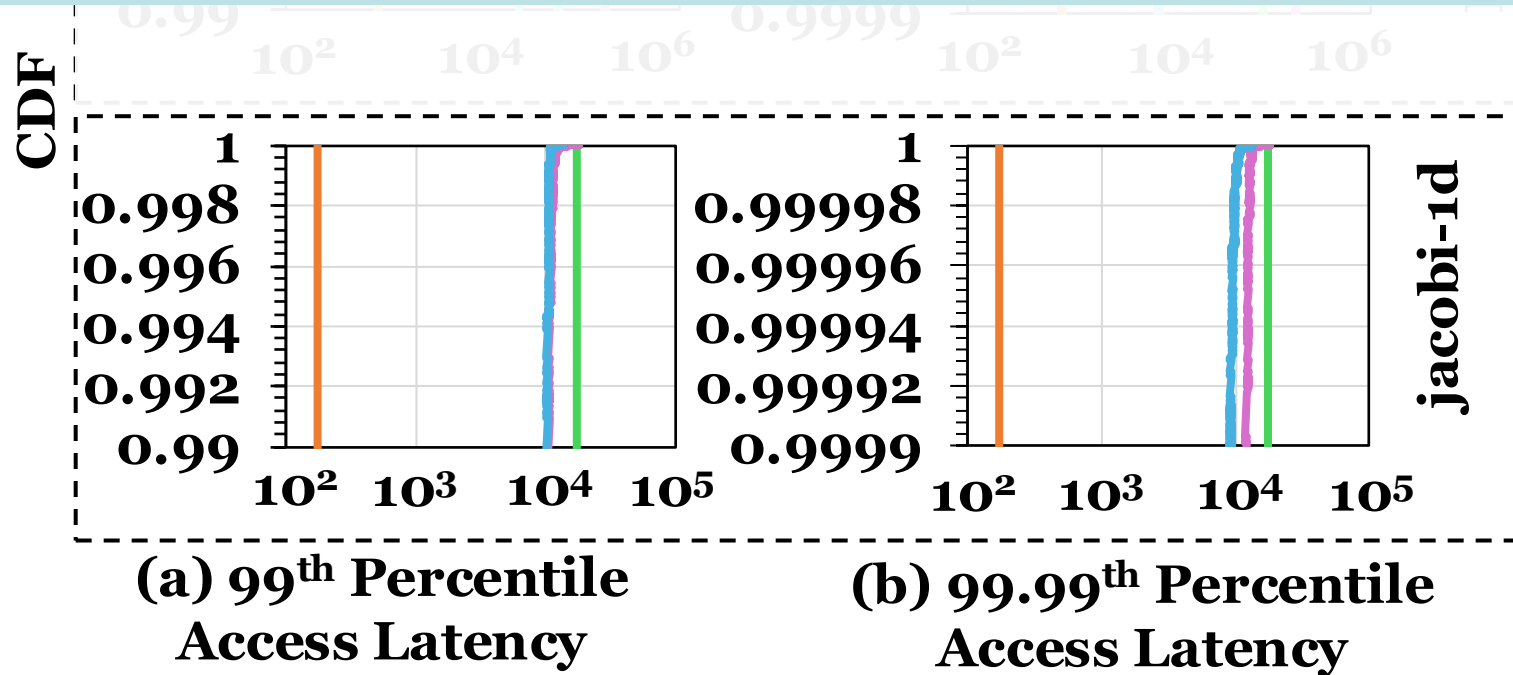
(b) 99.99th Percentile Access Latency

Tail Latency Evaluation (2/2)



— Ideal — Conduit — BW-Offloading — DM-Offloading

In jacobi-1d, compared to DM-Offloading, Conduit reduces 99th percentile latency by 1.1x, and 99.99th percentile latency by 1.3x



Workload-Computation Resource Interaction

