Venice: Improving Solid-State Drive Parallelism at Low Cost via Conflict-Free Accesses

*Rakesh Nadig§  *Mohammad Sadrosadati§  Haiyu Mao§  Nika Mansourı Ghiasi§
Arash Tavakkol§  Jisung Park§  Hamid Sarbazi-Azad†‡  Juan Gómez Luna§  Onur Mutlu§
§ETH Zürich  †POSTECH  ‡Sharif University of Technology  ‡IPM

Abstract
The performance and capacity of solid-state drives (SSDs) are continuously improving to meet the increasing demands of modern data-intensive applications. Unfortunately, communication between the SSD controller and memory chips (e.g., 2D/3D NAND flash chips) is a critical performance bottleneck for many applications. SSDs use a multi-channel shared bus architecture where multiple memory chips connected to the same channel communicate to the SSD controller with only one path. As a result, path conflicts often occur during the servicing of multiple I/O requests, which significantly limits SSD parallelism. It is critical to handle path conflicts well to improve SSD parallelism and performance.

Our goal is to fundamentally tackle the path conflict problem by increasing the number of paths between the SSD controller and memory chips at low cost. To this end, we build on the idea of using an interconnection network to increase the path diversity between the SSD controller and memory chips. We propose Venice, a new mechanism that introduces a low-cost interconnection network between the SSD controller and memory chips and utilizes the path diversity to intelligently resolve path conflicts. Venice employs three key techniques: 1) a simple router chip added next to each memory chip without modifying the memory chip design, 2) a path reservation technique that reserves a path from the SSD controller to the target memory chip before initiating a transfer, and 3) a fully-adaptive routing algorithm that effectively utilizes the path diversity to resolve path conflicts. Our experimental results show that Venice 1) improves performance by an average of 2.65×/1.67× over a baseline performance-optimized/cost-optimized SSD design across a wide range of workloads, 2) reduces energy consumption by an average of 61% compared to a baseline performance-optimized SSD design. Venice’s benefits come at a relatively low area overhead.

CCS Concepts
• Hardware → External storage; Memory and dense storage; • Information systems → Storage architectures.

ACM Reference Format:

1 Introduction
Flash-memory-based solid-state drives (SSDs) are ubiquitous, from cloud environments to mobile devices [1–28]. The high performance, low power consumption and shock resistance of SSDs make them suitable replacements for hard disk drives (HDDs) [1, 29, 30]. The rise in the number of data-intensive applications has resulted in the widespread adoption of SSDs in computing systems, increasing the demand for higher performance and capacity in SSDs. Although SSD vendors have significantly improved both performance and capacity of SSDs (e.g., [22, 31–34]) over the years, communication within the SSD (i.e., between the SSD controller and NAND flash chips) is still a critical performance bottleneck [7, 24, 35–46] for many applications, especially workloads with a large number of random I/O requests [6, 7, 15, 17, 38, 46–50].

Commodity SSDs use a multi-channel shared bus architecture (e.g., [1, 6–15]) for communication between the SSD controller and NAND flash chips. In this architecture, the SSD controller is connected to flash chips via multiple channels (typically 4 to 16 [51–53]) with a number of flash chips (typically 4 to 32 [51, 52, 54, 55]) connected to each channel. Thus, each flash chip has only one path to communicate with the SSD controller and several flash chips share the same path. As a result, there is a high likelihood that multiple I/O requests access NAND flash chips on the same channel. These I/O requests should be transferred serially on the same channel, which significantly limits SSD parallelism. We call this problem path conflict. To quantify the effect of path conflicts on SSD performance, we compare the performance of a state-of-the-art baseline SSD with an ideal (i.e., path-conflict-free) SSD. We observe that the ideal SSD outperforms the baseline SSD by 4× on average across nineteen data-intensive real-world workloads (see §3 for more detail).

SSD vendors attempt to reduce path conflicts by increasing the number of channels in the SSD. However, this is not a scalable solution since increasing the number of channels makes the SSD controller more complex (e.g., the SSD controller needs more I/O pins to service more parallel channels), increasing the overall cost of the SSD. A recent prior work [15] attempts to address the path conflict problem by increasing the bandwidth of each SSD channel. This work proposes to utilize the control and data pins of flash...
chips for both command and data transfer, effectively providing 2× the SSD channel bandwidth. Unfortunately, such techniques 1) are expensive as they require relatively large modifications to the commodity NAND flash memory chips (e.g., 20% area overhead in each flash die [15]), and 2) alleviate but cannot effectively resolve path conflicts (as we show in §3.3).

Our goal is to fundamentally address the path conflict problem in SSDs by providing high path diversity at low cost for communication between the SSD controller and flash chips. Our key idea is to use a low-cost interconnection network to increase the path diversity between the SSD controller and flash chips. Some prior works propose the use of an interconnection network within an SSD to provide a scalable solution to increase SSD capacity [7, 38]. Such works can potentially be repurposed to tackle the path conflict problem. However, prior SSD interconnection network designs have two main weaknesses, which prevent them from effectively addressing the path conflict problem. First, prior works impose significant area (i.e., cost) overhead as they integrate a buffered router (e.g., 16KB buffer per router port) inside each flash chip. Such a design increases the area and the number of I/O pins of the flash chip. Second, prior works do not resolve path conflicts effectively because they employ a simple deterministic routing algorithm, which cannot utilize the interconnection network’s rich path diversity (as we show in §3.3).

We propose Venice, a new mechanism that introduces a low-cost interconnection network of flash chips to fundamentally address the path conflict problem while effectively addressing the two major weaknesses of prior works on SSD interconnection networks. Venice employs three key techniques. First, Venice adds a new router chip next to each flash chip without modifying the flash chip. Routers are connected in a network topology, such as a 2D mesh. Second, Venice reserves a network path for each I/O request before initiating the command and data transfer. This technique ensures that the I/O request transfer does not experience path conflicts in the network, which avoids the need for large buffers in each router. Third, to find a free path between the SSD controller and the flash chip, Venice uses a non-minimal fully-adaptive routing algorithm that effectively utilizes the interconnection network’s path diversity.

We evaluate Venice using MQSim [57, 58], a state-of-the-art SSD simulator. We use two baseline SSD configurations, performance-optimized and cost-optimized and a wide variety of I/O-intensive benchmarks (see §5). Our evaluation yields three key results that demonstrate Venice’s effectiveness. First, for the performance-optimized configuration, Venice improves performance by an average of 2.65× (up to 7.10×) and 1.92× (up to 4.30×) compared to the baseline SSD design and best-performing prior work [38] (without taking the overhead of prior work into account), respectively. For the cost-optimized configuration, Venice improves performance by an average of 1.67× (up to 3.68×) and 1.47× (up to 2.90×) compared to the baseline SSD design and the best-performing prior work [38], respectively. Second, Venice reduces energy consumption by an average of 61% and 46% compared to the baseline performance-optimized SSD design and the best-performing prior work [38], respectively. Third, Venice’s benefits come at a relatively low area overhead. Venice’s routers impose 8% area overhead to the SSD printed circuit board (PCB). Venice’s interconnection network links, in total, occupy 44% lower area compared to the baseline multi-channel shared bus architecture.

This paper makes the following contributions:

- We demonstrate the importance of the path conflict problem in modern SSD designs, and quantify its performance impact.
- We propose Venice, a new mechanism that introduces a low-cost interconnection network of flash chips to fundamentally address the path conflict problem in SSDs.
- We introduce three key techniques that enable Venice: 1) a simple router chip added next to each flash chip without modifying the flash chip itself, 2) a path reservation technique to reserve paths from the SSD controller to target flash chips, and 3) a non-minimal fully-adaptive routing algorithm to effectively utilize the path diversity in the interconnection network.
- We rigorously evaluate Venice and show that it significantly improves performance over state-of-the-art SSD designs on both performance- and cost-optimized SSD configurations.

2 Background

We provide a brief background on the baseline multi-channel shared bus SSD architecture. A typical modern SSD consists of an SSD controller and an array of flash chips. The host system uses a high-speed communication interface (e.g., PCIe [59]) to communicate with the SSD. The SSD controller communicates with the flash chips using the shared flash channels. Figure 1 shows a high-level overview of a modern SSD.

![Figure 1: High-level overview of a modern SSD](image)

2.1 Flash Chip Array

Multiple flash chips (1) [6, 22] are connected to a flash controller (2) through a shared channel in the multi-channel shared bus architecture. Each flash chip (1) contains one or more (typically 1 to 4) flash dies. Flash dies operate independently of each other. Each die (2) consists of multiple (e.g., 2 or 4) planes. A plane (3) typically contains thousands of blocks, and each block (4) consists of tens to hundreds of pages. A flash page (3) consists of a set of flash cells connected to the same wordline within a flash block. Read and write operations are typically performed at the granularity of a flash page (e.g., 16KB in size). However, erase operations happen at block granularity [19–22, 24, 27, 28]. Planes in the same die share the peripheral circuitry used to access pages; as such, they can concurrently operate only when accessing pages (or blocks) at the same offset, which are called multi-plane operations.

1Named after the network of canals in the city of Venice [56].
2.2 SSD Controller

The SSD controller (2) is responsible for managing the NAND flash chips (1) and the I/O requests sent by the host (3). The SSD controller contains an embedded microprocessor that executes firmware called the Flash Translation Layer (FTL) (4) [16, 49, 57, 68, 69]. The SSD controller stores metadata (e.g., a logical-to-physical page mapping table) used to manage the FTL functionality and caches frequently accessed pages in DRAM (5) that is part of the SSD. The SSD controller consists of multiple flash controllers (6). A flash controller is an embedded processor that interfaces with multiple flash chips using a shared channel. The flash controller selects the flash chip for a read/write operation and initiates the command and data transfer.

Host Interface Layer. Host Interface Layer (HIL) (7) [57, 70, 71] is the interface between the host system (1) and the SSD controller (6). HIL communicates with the host system using a communication protocol over the system I/O bus. HIL in a commodity SSD typically supports the Advanced Host Controller Interface (AHCI) [72] or the NVM Express (NVMe) [32] interface. AHCI builds upon the Serial ATA (SATA) [73] protocol, which is commonly used to connect the host system to the hard disk drives. AHCI and SATA interfaces provide very low throughput for SSDs because of the availability of a single I/O queue to submit I/O requests to the SSD.

To overcome the throughput bottleneck of AHCI and SATA, modern SSDs have adopted the NVMe protocol, which uses the PCIe Express (PCIe) system bus to communicate with the host (see, e.g., [23, 31, 55, 57, 74–80]). NVMe directly exposes multiple SSD I/O queues to the host, thereby enabling 1) high-bandwidth and low-latency communication between the SSD and the host, and 2) more fine-grained control of the I/O request scheduling policy by the SSD controller [23]. The host system transfers I/O requests into a Submission Queue allocated to the application in the HIL. HIL picks an I/O request from the Submission Queue and sends the request to the FTL for processing. After the completion of the I/O request, the HIL updates the Completion Queue to inform the host system.

Flash Translation Layer (3). FTL has four major responsibilities [16, 49, 57, 68, 69]. First, for each page of data, FTL manages the mapping of each logical address (i.e., the requested address in the host system’s address space) to a physical address (i.e., the actual location in the physical flash chips where the requested data resides). Before new data is written to a flash page, an entire flash block that contains the target flash page has to be erased. This is called the erase-before-write requirement. Unfortunately, the erase-before-write requirement of NAND flash memory makes in-place writes prohibitively costly in terms of performance, energy consumption, and lifetime [19, 22, 26, 27, 68]. To overcome this issue, the FTL implements an out-of-place write policy in modern SSDs [19, 22, 27]. Whenever a page of data is written to the host system to a logical page address, the FTL 1) invalidates the corresponding physical page address where the overwritten data resides, 2) writes the new page data to a different physical page address, and 3) updates the logical-to-physical page mapping metadata of the logical page.

Second, the FTL performs garbage collection (GC) [6, 19, 22, 27, 46, 57, 81–85] to recover the wasted space due to pages invalidated by the out-of-place write policy. During GC, the FTL 1) chooses a victim block with the least number of valid pages, 2) copies all valid pages in the victim block to another block, 3) updates the logical to physical address mapping metadata for pages that have already been copied, and 4) erases the victim block to use this block for future write operations. Third, the FTL implements a wear-leveling technique to distribute the writes evenly across all the flash blocks so that the flash blocks in the SSD wear out in a uniform manner [6, 16, 49, 66, 69, 86]. Having a wear-leveling mechanism in FTL is critical for SSD lifetime as the number of times a flash block can be erased and programmed is limited [22, 87, 88]. Fourth, the FTL avoids frequent lookups to the flash memory by caching frequently-accessed data (e.g., the logical-to-physical page mapping table [68]) or frequently-requested pages by the host in the DRAM (5) that is present inside the SSD.

3 Motivation

We describe the path conflict problem in a typical multi-channel shared bus SSD architecture (which we call Baseline SSD) and major approaches to mitigate path conflicts.

3.1 The Path Conflict Problem in Modern SSDs

A typical SSD (e.g., [1, 6, 8–11, 13, 14, 54, 55, 79, 80]) uses a multi-channel shared bus architecture for communication between the SSD controller and NAND flash chips. The SSD controller is connected to flash chips via a number of shared channels (typically 4 to 16 [51–53]) with multiple flash chips (typically 4 to 32 [51, 52, 54, 55]) connected to each channel. Figure 2(a) shows an example configuration in such a Baseline SSD where there are four shared channels with four flash chips connected to each shared channel. In Baseline SSD, each flash chip has only one channel (or path) to communicate with the SSD controller. Unfortunately, the flash chips connected to the same channel share the same path to the SSD controller. Path sharing causes the path conflict problem,
where an I/O request needs to wait for the path to become free, if the path is being used for another I/O request.

To demonstrate the path conflict problem, we show two examples of service timelines of ongoing read I/O requests in Figure 3. For simplicity, the figure shows only three major steps during a read request, the read command (CMD), the flash read operation (RD Operation), and read data transfer from the flash chip to the SSD controller (Transfer). The first example (Figure 3 top) shows two ongoing read requests to two different flash chips connected to the same channel (i.e., the two requests experience the path conflict problem). Unfortunately, in this case, only the second step (RD Operation), flash read operation, of the two ongoing read requests can be performed in parallel. Other steps (CMD and Transfer) should be performed one after the other (i.e., serially) because they use the same path, which increases the total service time (the total time taken for processing an I/O request within the SSD) of these two requests. The second example (Figure 3 bottom) shows two ongoing read requests to two flash chips connected to two different channels (i.e., no path conflict problem). This example shows that these two I/O requests can be serviced completely in parallel, which reduces the total service time of the two I/O requests.

To understand how much the path conflict problem can increase the total service time, we use the latency numbers for CMD, RD Operation, and Transfer from a performance-optimized SSD configuration (see §5). In a performance-optimized SSD configuration, CMD, RD Operation, and Transfer take 10ns, 3μs, and 4μs, respectively [31, 99]. The total service time for the two read requests that experience the path conflict problem (as depicted in Figure 3 top) is 11.01μs (i.e., CMD + RD Operation + Transfer + Transfer = 11.01μs). In contrast, ideally (i.e., without a path conflict, as depicted in Figure 3 bottom), the total service time of the two requests is 7.01μs (i.e., CMD + RD Operation + Transfer = 7.01μs). Thus, in this simple example, the path conflict problem increases the average I/O access latency by 57%, which in turn results in lower SSD throughput. The performance overhead of path conflicts can be even higher when (1) more than two I/O requests experience path conflicts, and (2) the data transfer size of each request is larger (e.g., a multi-plane operation; see §2).

The path conflict problem affects the performance of read requests more than that of write requests [9, 11, 12, 40–42]. This is because data transfer time for a read request is comparable to or longer than the flash read latency [31, 100, 101], while the flash write latency (e.g., 100μs for a performance-optimized SSD configuration [31, 99]) dominates the total service time of a write request.

We conclude that the path conflict problem can significantly increase the total service time of I/O requests and limit SSD throughput, especially for read-intensive workloads.

3.2 Approaches to Mitigate Path Conflicts

We describe two major prior approaches to address path conflict problem and their limitations. We quantitatively analyze the effectiveness of these approaches at mitigating path conflicts in §3.3.

Increasing Flash Channel Bandwidth. A recent work by Kim et al. [15] proposes the Packetized SSD (pSSD) (Figure 2(b)), a technique to increase the flash channel bandwidth to 2X the channel bandwidth of the Baseline SSD. This technique 1) utilizes control and data pins of the flash chip to transfer both commands and data, thus increasing the channel bandwidth, and 2) integrates an on-die controller inside each flash chip to enable packetization between the flash controller and the flash chip. While pSSD can reduce the performance overhead of path conflicts by reducing the I/O transfer latency, pSSD imposes significant area overhead (i.e., 20% [15]) in each flash chip.

Increasing Path Diversity. Prior works [7, 15, 38] propose techniques to mitigate path conflicts by increasing the number of paths through which the SSD controller can access a flash chip (i.e., these approaches increase path diversity).

Kim et al. [15] propose the Packetized Network SSD (pnSSD) (Figure 2(c)), a technique that provides two paths to access each flash chip, which reduces the performance overhead of path conflicts. pnSSD introduces an interconnection network similar to the 2D mesh topology [102], except in each dimension, flash chips are connected using a shared bus. As a result, an NxN flash chip array has N horizontal and N vertical channels. In pnSSD, a flash chip can be accessed using either a horizontal channel or a vertical channel.

Tavakkol et al. [7, 38] propose Network-on-SSD (NoSSD) (Figure 2(d)), which replaces the multi-channel shared bus architecture with a 2D mesh interconnection network of flash chips. NoSSD significantly increases the path diversity compared to the Baseline SSD and pnSSD. However, NoSSD has two main weaknesses that
limit its effectiveness at mitigating the path conflict problem. First, NoSSD imposes significant area and cost overhead due to 1) the integration of a buffered router (e.g., with a 16KB buffer per router port) inside each flash chip, and 2) 4x increase in the number of I/O pins compared to a commodity flash chip. Second, NoSSD does not utilize the path diversity effectively as NoSSD employs simple deterministic routing (i.e., the dimension-order routing algorithm [102]) that cannot adapt to the availability of multiple free paths between the flash controller and target flash chip.

### 3.3 Effectiveness of Prior Approaches

#### Methodology. We study the effectiveness of prior approaches at mitigating path conflicts using a state-of-the-art SSD simulator, MQSim [57, 58], across nineteen real-world data-intensive workloads (see §5 for our methodology). To this end, we measure the speedup of pSSD, pnSSD, and NoSSD over the Baseline SSD in a performance-optimized SSD configuration (see §5). We compare the speedup results with the speedup of the ideal (i.e., path-conflict-free) SSD. In the path-conflict-free SSD, we assume that each flash chip has a direct separate channel to communicate with the SSD controller; therefore, no path conflict can happen. An I/O request does not experience path conflicts in the path-conflict-free SSD, but it can still be delayed if the target flash chip is busy.

#### Performance Results. Figure 4 shows the performance of pSSD, pnSSD, NoSSD and path-conflict-free SSD compared to the Baseline SSD. We make five major observations. First, the path-conflict-free SSD provides an average of 4x (up to 11.74x) the performance of the Baseline SSD since it does not suffer from any path conflict. Second, pSSD shows an average performance improvement of 27% over Baseline SSD due to its increased channel bandwidth. Third, pnSSD provides an average performance improvement of 30% over Baseline SSD due to its increased path diversity. Fourth, NoSSD outperforms Baseline SSD by 35% on average due to the significantly increased path diversity provided by the interconnection network of flash chips. Fifth, although NoSSD outperforms pSSD and pnSSD, NoSSD’s speedup is still greatly lower than the path-conflict-free SSD’s speedup (4x). The main reason is that NoSSD does not utilize the path diversity effectively.

We conclude that while prior approaches improve the performance of the SSD at a large cost overhead, none of them effectively mitigate the path conflict problem, and a large potential remains between their performance and the performance of an SSD that does not suffer from path conflicts.

### 3.4 Our Goal

Based on our observations and analyses in §3.1, §3.2 and §3.3, we conclude that 1) the path conflict problem significantly limits the performance of modern SSDs, and 2) none of the prior approaches (i.e., pSSD, pnSSD, and NoSSD) effectively mitigate the path conflict problem even though they come with significant area overheads and cost overheads.

Our goal is to fundamentally address the path conflict problem in SSDs by 1) providing path diversity inside the SSD at low cost, and 2) effectively utilizing the increased path diversity for communication between the SSD controller and flash chips.

## 4 Venice

#### Overview. We design Venice, a new mechanism that fundamentally addresses the path conflict problem in modern SSDs. Venice 1) provides rich path diversity between the SSD controller and flash chips by introducing a low-cost interconnection network of flash chips, and 2) utilizes the path diversity to identify and reserve a conflict-free path for an I/O request. Venice’s design is based on three key techniques: (1) a low-cost interconnection network of flash chips in the SSD (§4.1), (2) reservation of a path between the flash controller and the flash chip for each I/O request (§4.2), and (3) a non-minimal fully-adaptive routing algorithm to utilize the path diversity provided by the interconnection network of flash chips (§4.3).

### 4.1 Low-Cost Interconnection Network of Flash Chips

We want to provide rich path diversity between the SSD controller and flash chips at low cost. Venice can utilize the rich path diversity to eliminate path conflicts. To this end, we connect the flash chips using a low-cost interconnection network. The key design decision that enables our approach to be low cost is the separation of the router from the flash chip such that the flash chip is not modified.

We introduce a new building block, called flash node, which consists of a flash chip and a separate router chip. Figure 5(a) shows a flash node. In each flash node, a flash chip communicates with a router chip using its I/O data pins (i.e., injection/ejection ports) that are otherwise used for connecting the flash chip to the shared channel. Our design connects the flash nodes using an interconnection network topology. Figure 5(b) shows an example interconnection network of flash nodes using the 2D mesh topology. The router chip in each flash node is connected to the router chips in the neighboring flash nodes using bidirectional links.

### 4.2 Path Reservation

#### Key Idea. To ensure that the I/O request transfer does not experience path conflicts in the network, Venice reserves a conflict-free path between the flash controller and the target flash chip for each I/O request before starting the transfer. This technique avoids the need for large buffers in each router that are otherwise required to store the data of each I/O request that experiences a path conflict.

#### Implementation. Venice identifies and reserves a path by sending a special packet called scout packet. Figure 6 shows the structure of a scout packet for an SSD with 64 flash chips and 8 flash controllers.
The router reservation table keeps track of 1) the packet ID, which has \( \log(n) \) bits to denote one of the \( n \) flash controllers, which allows up to \( n \) scout packets to be sent simultaneously. In our example interconnection network configuration with 8 flash controllers, we need 3 bits for the packet ID. The entry port 2 and exit port 3 information in the router reservation table each contains 2 bits 4 to denote one of the four ports in the router.

When the scout packet arrives at the destination flash chip, Venice has already reserved the conflict-free forward and backward paths. The router connected to the destination flash chip uses the backward path to send the scout packet back to the source flash controller. Once the source flash controller receives back the scout packet, it schedules the I/O request transfer using the reserved path.

If a scout packet is unable to find a free link at a router during the path reservation process, the router enables the cancel mode in the scout packet, which cancels the reservation by removing its entry in the router reservation table. The scout packet backtracks along its path to a previously traversed router (i.e., upstream router). Depending on the routing algorithm’s adaptivity, the scout packet may either try a different free output link in the upstream router or backtrack further (i.e., to the upstream router of the upstream router). In case the scout packet is unable to find a free output link during backtracking, the scout packet can arrive back at the flash controller without reserving a path. When the source flash controller receives the scout packet in cancel mode, it retries the path reservation process immediately by sending a new scout packet. 5

The scout packet consists of two 8-bit scout flits, a header flit 1 and a tail flit 2. Each scout flit contains a 2-bit type information, whose most significant bit denotes whether the flit is the header flit or the tail flit, and 2) least significant bit denotes if the flit is in reserve mode to reserve a link in the path or cancel mode to cancel a reservation. The destination flash chip ID is stored in the last 6 bits of the header flit (6 bits are required to represent 64 flash chips). In the tail flit, 3 bits are used to denote the source flash controller, and the other 3 bits are unused. The source flash controller ID is the same as the scout packet ID.

![Figure 5: Venice’s low-cost interconnection network](image_url)

![Figure 6: Structure of the scout packet for an SSD with 64 flash chips and 8 flash controllers](image_url)

For a given I/O request, Venice checks if the closest flash controller to the target flash chip is available. If so, Venice selects the flash controller to handle the I/O request. Otherwise, Venice uses the nearest free flash controller. The source flash controller sends a scout packet in reserve mode to identify and reserve a path to the destination chip. Venice uses a routing algorithm (e.g., the non-minimal fully-adaptive routing algorithm as described in §4.3) to route a scout packet from the source flash controller to the destination flash chip. Venice reserves the interconnection network’s links that a scout packet takes to reach the destination node. Each reserved link is bidirectional, which enables data transfer 1) from the forward path to the flash controller to the flash chip (e.g., a write request) using the forward path, and 2) from the flash chip to the flash controller (e.g., a read request) using the backward path. To this end, we introduce a table, called router reservation table, to each router chip. Figure 7 shows the structure of Venice’s router 2 and router reservation table 4. The router reservation table keeps track of 1) the packet ID 5, which is the same as the source flash controller ID from which the packet was sent, and 2) which two ports (i.e., entry 6 and exit 7 ports) are connected bidirectionally (based on reservations that were made). Each row in the router reservation table has a valid bit 8 that shows whether the entry is valid. The packet ID

![Figure 7: Structure of a router in Venice’s interconnection network of flash nodes (assuming 8 flash controllers)](image_url)

### 4.3 Utilizing Path Diversity

**Key Idea.** To effectively utilize the interconnection network’s path diversity, Venice uses a non-minimal fully-adaptive routing algorithm for routing a scout packet (during the path reservation process) through the interconnection network of flash nodes. Venice’s non-minimal fully-adaptive routing algorithm dynamically identifies a conflict-free path between the flash controller and the flash chip. This algorithm effectively utilizes the idle links in the interconnection network to find a non-minimal path when a minimal path is unavailable.

Figure 8 illustrates how a non-minimal fully-adaptive routing algorithm helps to mitigate the path conflict problem via an example. In this example, a new I/O request \( R \) has \( F_2 \) as its destination flash chip. In the network, there are three paths already reserved for other I/O requests (marked in red in Figure 8): \( FC_0 \rightarrow F_0 \rightarrow F_1 \rightarrow F_6, FC_1 \rightarrow F_5 \rightarrow F_6 \rightarrow F_7 \rightarrow F_8, \) and \( FC_2 \rightarrow F_{10} \rightarrow F_{11} \rightarrow F_{12} \rightarrow F_7. \) The only free flash controller, \( FC_3, \) is assigned to request \( R. \) Each

---

5We study more optimizations, including when to resend the scout packet, in the path reservation process in the extended version of our paper [103].
minimal path from \( FC_3 \) to \( F_2 \) has at least one busy link, and thus, is not path-conflict-free. However, there are a number of non-minimal paths from the \( FC_3 \) to \( F_2 \) that are path-conflict-free. An example is \( FC_3 \rightarrow F_{15} \rightarrow F_{16} \rightarrow F_{17} \rightarrow F_{18} \rightarrow F_{13} \rightarrow F_8 \rightarrow F_3 \rightarrow F_2 \) (shown in blue in Figure 8). Venice uses a non-minimal fully-adaptive routing algorithm (described in Algorithm 1) during the path reservation process to increase its ability to mitigate the path conflict problem.

![Figure 8: Example demonstrating how Venice’s non-minimal routing algorithm finds a conflict-free path in the interconnection network of flash nodes](image)

To effectively use a non-minimal fully-adaptive routing algorithm during the path reservation process in Venice, we should address two key challenges: 1) performance overhead of exercising a non-minimal path to service the I/O request, and 2) the need to avoid deadlock/livelock. We describe these two key challenges and Venice’s techniques to address them.

**Performance Overhead of Exercising a Non-Minimal Path.**

The increased path length of the non-minimal route can cause two issues. First, a non-minimal path may lead to an increase in the command/data transfer time and thus the overall latency for servicing the I/O request. After reserving a free path, the transfer latency \( T_{transfer} \) in seconds can be calculated using Equation 1:

\[
T_{transfer} = \frac{distance + \frac{transfer_size}{link_width}}{link_lat} \times link_width. \tag{1}
\]

where \( distance \), \( transfer_size \), \( link_width \), and \( link_lat \) are the number of links between the flash controller and flash chip (i.e., hops), the command/data transfer size in terms of the number of bytes, the link width in terms of the number of bytes, and the latency of a single transfer (of size \( link_width \)) on the link in seconds, respectively. A non-minimal path has a longer distance compared to a minimal path. We study the latency overhead of a non-minimal path for I/O data and flash command transfer. For the I/O data transfer, the performance overhead of a longer path is negligible. This is because \( transfer_size \) dominates \( T_{transfer} \) as I/O data transfers are large in size (e.g., 16KB). Flash commands, on the other hand, have a small \( transfer_size \) (i.e., only a few bytes), and thus, a longer path can significantly increase their transfer latency. As discussed in §3.1, the service time of an I/O request consists of flash command transfer time, flash operation latency and I/O data transfer time. The total service time is dominated by the I/O data transfer time and the flash operation latency, and thus, the longer command transfer time due to a non-minimal path has a negligible effect on the total service time of the I/O request.

Second, a non-minimal path occupies more links in the interconnection network compared to a minimal path. If a minimal path is used instead of a non-minimal path, the extra links of a non-minimal path can potentially be used for transferring other ongoing requests, which increases the effectiveness of Venice. To this end, Venice attempts to find path-conflict-free minimal paths during the path reservation process as much as possible (as described in Algorithm 1).

**Need to Avoid Deadlock/Livelock.**

A non-minimal fully-adaptive routing algorithm can potentially cause (1) deadlock in the interconnection network [102, 104–114], where multiple network packets cannot move forward as they circularly depend on each other to free up resources (e.g., channels, buffers), and (2) livelock [102, 104–109, 114–117], where at least one packet keeps traversing the network without reaching its destination. Venice’s interconnection network can experience deadlock and livelock only during the path reservation process where the scout packets are routed using the non-minimal fully-adaptive routing algorithm.⁴

Venice handles deadlock by using backtracking of a scout packet. When a scout packet experiences path conflict during the path reservation process, it backtracks along its path to the previously visited router (i.e., the upstream router) in order to choose a different path. As a result, a scout packet is never blocked due to resource unavailability in the network and deadlock does not happen.

Venice handles livelock by restricting the number of times a scout packet can visit the same router. A scout packet can reserve each output port of a router only once and hence, the scout packet may revisit the same router at most three times⁵ in an interconnection network with a 2D mesh topology. When a scout packet revisits the same router three times, the scout packet traces its path back to the upstream router (using the router reservation table) and attempts to reserve a different output port in the upstream router. In the worst case, when a scout packet fails to reserve a path to the destination after visiting all the routers at most three times, it will return to the source flash controller. The flash controller immediately sends a new scout packet to retry the path reservation.

**Implementation.**

Algorithm 1 shows the pseudocode for the non-minimal fully-adaptive routing algorithm in Venice. The inputs to the algorithm are: 1) scout packet ID, 2) current router ID, 3) scout packet’s destination router ID, 4) input port of the router through which the scout packet has arrived, 5) the status (free or busy) of the output ports in the router, and 6) the interconnection network structure in terms of number of rows and columns (assuming a 2D mesh topology). The algorithm returns the output port in which the scout packet should traverse to the downstream router.

To find an appropriate output port, the algorithm first attempts to find a free output port that leads to a minimal path (lines 2-32). To this end, the algorithm compares the current router ID with the scout packet’s destination router ID in both X (horizontal) and Y (vertical) dimensions. Based on the comparison, it switches among nine cases (lines 5-26). In each case, the algorithm checks the status of the corresponding output port and adds the output port to the output list if the corresponding output port is free (e.g., lines 6-10).

The algorithm checks the number of output ports added to the output list (line 27). In a 2D mesh topology, the size of the output

---

⁴Once a path between the flash controller and the destination flash chip is reserved, there is no 1) deadlock, as there is no path conflict during I/O data or flash command transfer, or 2) livelock, as the path from the source flash controller and the destination flash chip is deterministically set as a circuit.

⁵The number of times a scout packet can revisit a router is four minus one, i.e., number of ports in a router minus the entry port of the scout packet.
Algorithm 1 Venice’s Non-Minimal Fully-Adaptive Routing Alg.

Input: scout packet ID \( P_{sc} \), current router ID \( ID_{curr} \), scout packet’s destination router ID \( ID_{dst} \), the input port, the output ports’ status, and network structure: \( N_{r} \) rows and \( N_{c} \) columns

Output: Output port

1: procedure FIND OUTPUT PORT
2: \( \text{Diff}_{fx} = ID_{fx} \& N_{r} - ID_{fx} \& N_{c} \)
3: \( \text{Diff}_{fy} = ID_{fy} \& N_{r} - ID_{fy} \& N_{c} \)
4: \( \text{Output}_{tx} = \emptyset \)
5: \( \text{Switch}(\text{Diff}_{fx} \text{ and } \text{Diff}_{fy}) \) // Nine cases in total as \( \text{Diff}_{fx} \) and \( \text{Diff}_{fy} \) can each be a positive, zero, or negative value
6: \( \text{Case } 1: \text{Diff}_{fx} > 0 \& \text{Diff}_{fy} > 0 \)
7: if \( \text{Right}.\text{status}()==\text{free} \) then
8: \( \text{Output}_{tx} = \text{Right} \)
9: if \( \text{Up}.\text{status}()==\text{free} \) then
10: \( \text{Output}_{tx} = \text{Up} \)
11: Case 2: \( \text{Diff}_{fx} > 0 \& \text{Diff}_{fy} < 0 \)
12: if \( \text{Right}.\text{status}()==\text{free} \) then
13: \( \text{Output}_{tx} = \text{Right} \)
14: if \( \text{Down}.\text{status}()==\text{free} \) then
15: \( \text{Output}_{tx} = \text{Down} \)
16: Case 3: \( \text{Diff}_{fx} > 0 \& \text{Diff}_{fy} = 0 \)
17: if \( \text{Right}.\text{status}()==\text{free} \) then
18: \( \text{Output}_{tx} = \text{Right} \)
19: Case 4: \( \text{Diff}_{fx} < 0 \& \text{Diff}_{fy} > 0 \)
20: Case 5: \( \text{Diff}_{fx} < 0 \& \text{Diff}_{fy} < 0 \)
21: Case 6: \( \text{Diff}_{fx} < 0 \& \text{Diff}_{fy} = 0 \)
22: Case 7: \( \text{Diff}_{fx} = 0 \& \text{Diff}_{fy} > 0 \)
23: Case 8: \( \text{Diff}_{fx} = 0 \& \text{Diff}_{fy} < 0 \)
24: Case 9: \( \text{Diff}_{fx} = 0 \& \text{Diff}_{fy} = 0 \)
25: \( \text{Output}_{tx} = \text{Reserved}() \)
26: end

27: \( \text{if } (\text{Output}_{tx} \neq \emptyset) \) then
28: \( \text{Output}_{port} = \text{randomly select one output port from Output}_{tx} \)
29: Routing Reservation Table.insert \((P_{sc}, \text{Input}_{port}, \text{Output}_{port})\)
30: else if \( (\text{Output}_{tx} \neq \emptyset) \) then
31: \( \text{Output}_{port} = \text{Output}_{tx} \)
32: Routing Reservation Table.insert \((P_{sc}, \text{Input}_{port}, \text{Output}_{port})\)
33: else
34: \( \text{Nonminimal Output}_{tx} = \emptyset \)
35: if \( (\text{Up}.\text{status}()==\text{free} \& \text{Up} \neq \text{Input}_{link}) \) then
36: \( \text{Nonminimal Output}_{tx} = \text{up} \)
37: if \( (\text{Down}.\text{status}()==\text{free} \& \text{Down} \neq \text{Input}_{link}) \) then
38: \( \text{Nonminimal Output}_{tx} = \text{down} \)
39: if \( (\text{Right}.\text{status}()==\text{free} \& \text{Right} \neq \text{Input}_{link}) \) then
40: \( \text{Nonminimal Output}_{tx} = \text{right} \)
41: if \( (\text{Left}.\text{status}()==\text{free} \& \text{Left} \neq \text{Input}_{link}) \) then
42: \( \text{Nonminimal Output}_{tx} = \text{left} \)
43: if \( \text{Nonminimal Output}_{tx} > 0 \) then
44: \( \text{Output}_{port} = \text{randomly select one output port from Nonminimal Output}_{tx} \)
45: Routing Reservation Table.insert \((P_{sc}, \text{Input}_{port}, \text{Output}_{port})\)
46: else
47: \( \text{Output}_{port} = \text{Input}_{port} \) // traverse back to the upstream router
48: end
49: end procedure
Venice: Improving Solid-State Drive Parallelism
at Low Cost via Conflict-Free Accesses

a technique that increases path diversity by introducing vertical flash channels; (4) NoSSD [7, 38], a state-of-the-art proposal on interconnection network of flash chips that uses a deterministic minimal routing policy to route I/O requests. We compare Venice and the four prior approaches with an ideal path-conflict-free SSD. In a path-conflict-free SSD, we assume that each flash chip has a direct separate channel to communicate with the SSD controller, which eliminates path conflicts.

Workloads. We select nineteen data-intensive storage workloads from MSR Cambridge traces [122], Yahoo! Cloud Serving Benchmark (YCSB) suite [123], Slacker [124], SYSTOR ‘17 [125] and YCSB RocksDB traces [126] that are collected from real enterprise and datacenter workloads. These workloads are chosen to represent diverse I/O access patterns, with different read and write ratios, I/O request sizes, and inter-request arrival times. Table 2 reports the characteristics of the workloads chosen for our evaluation.

Table 2: Characteristics of the evaluated I/O traces

<table>
<thead>
<tr>
<th>Traces</th>
<th>Read %</th>
<th>Avg. Request Size (KB)</th>
<th>Avg. Inter-request Arrival Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR Cambridge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bm_0</td>
<td>76</td>
<td>3.8</td>
<td>38</td>
</tr>
<tr>
<td>mix4</td>
<td>38</td>
<td>7.8</td>
<td>46</td>
</tr>
<tr>
<td>mix5</td>
<td>39</td>
<td>9.6</td>
<td>129</td>
</tr>
<tr>
<td>mix6</td>
<td>40</td>
<td>22.8</td>
<td>96</td>
</tr>
<tr>
<td>mix7</td>
<td>41</td>
<td>32.7</td>
<td>102</td>
</tr>
<tr>
<td>mix8</td>
<td>42</td>
<td>37.6</td>
<td>107</td>
</tr>
<tr>
<td>mix9</td>
<td>43</td>
<td>39.2</td>
<td>75</td>
</tr>
<tr>
<td>mix10</td>
<td>44</td>
<td>49.2</td>
<td>103</td>
</tr>
<tr>
<td>mix11</td>
<td>45</td>
<td>52.8</td>
<td>91</td>
</tr>
<tr>
<td>mix12</td>
<td>46</td>
<td>65.7</td>
<td>87</td>
</tr>
<tr>
<td>mix13</td>
<td>47</td>
<td>93.9</td>
<td>15</td>
</tr>
<tr>
<td>mix14</td>
<td>48</td>
<td>13.3</td>
<td>15</td>
</tr>
<tr>
<td>mix15</td>
<td>49</td>
<td>16.5</td>
<td>2</td>
</tr>
<tr>
<td>mix16</td>
<td>50</td>
<td>18.8</td>
<td>2</td>
</tr>
<tr>
<td>mix17</td>
<td>51</td>
<td>20.4</td>
<td>2</td>
</tr>
<tr>
<td>mix18</td>
<td>52</td>
<td>22.8</td>
<td>2</td>
</tr>
<tr>
<td>mix19</td>
<td>53</td>
<td>24.2</td>
<td>2</td>
</tr>
</tbody>
</table>

To evaluate Venice under real-world scenarios, where multiple workloads access the same SSD, we create mixed workloads by combining two or three independent storage workloads. Table 3 shows six mixed workloads and their different characteristics.

Mixed workloads usually have a higher intensity of I/O requests (i.e., lower inter-request arrival time between I/O requests), which likely exacerbates the path conflict problem in the SSD.

Table 3: Characteristics of mixed workloads

<table>
<thead>
<tr>
<th>Mix</th>
<th>Constituent Workloads [122, 123]</th>
<th>Description</th>
<th>Avg. Inter-request Arrival Time (μs)</th>
</tr>
</thead>
</table>
| mix1  | mix1, mix2 and mix3             | Both workloads are read-intensive | 5.8
| mix2  | mix2, mix3 and mix4             | All three workloads are read-intensive | 4.4
| mix3  | mix3, mix4 and mix5             | Both workloads are write-intensive | 9.3
| mix4  | mix4, mix5 and mix6             | All three workloads are read-intensive | 3.6
| mix5  | mix5, mix6 and mix2             | prxy_0 is write-intensive and mix2_1 is read-intensive | 5
| mix6  | mix6, mix7 and mix8             | mix7 is read-intensive and mix2_1 is read-intensive | 3

Metrics. To compare Venice with prior systems, we report the following metrics in our experimental results (see §6) for each workload: 1) performance in terms of speedup in overall execution time over Baseline SSD, 2) SSD throughput in IOPS (i.e., number of I/O operations per second), 3) tail latency in the 99th percentile of I/O requests, 4) SSD power/energy consumption, and 5) power and area overheads.

6 Evaluation

6.1 Performance Analysis

Execution Time. Figures 9(a) and 9(b) show the performance improvement of pSSD, pnSSD, NoSSD, Venice and path-conflict-free SSD over Baseline SSD in terms of speedup in overall execution time of each workload in a performance-optimized SSD and a cost-optimized SSD, respectively.

We make three key observations. First, Venice consistently outperforms all the prior approaches across all workloads in both SSD configurations. In the performance-optimized SSD configuration, Venice outperforms Baseline SSD/pSSD/pnSSD/NoSSD by an average of 2.65×/2.10×/2.00×/1.92× across all workloads. In the cost-optimized SSD configuration, Venice shows an average performance speedup of 1.67×/1.52×/1.55×/1.47× over Baseline SSD.

![Figure 9: Performance of pSSD, pnSSD, NoSSD, Venice and path-conflict-free SSD on performance-optimized (top) and cost-optimized (bottom) SSD configurations. Performance is shown in terms of speedup in overall execution time over Baseline SSD.](image-url)
SSD/pSSD/pnSSD/NoSSD. Second, Venice results in higher performance improvements in the performance-optimized SSD configuration. This is because the performance-optimized SSD uses faster flash chips (with significantly lower read/write latencies), and thus the I/O data transfer time within the SSD dominates the I/O service time. As a result, improving I/O data transfer performance in the performance-optimized SSD provides a higher improvement in workload execution time. Third, Venice performs within 45% and 25% of the path-conflict-free SSD in the performance-optimized and cost-optimized SSD configuration, respectively. We conclude that Venice significantly improves workload execution time by mitigating the path conflict problem in modern SSDs.

SSD Throughput. Figures 10(a) and 10(b) show the SSD throughput (in IOPS) of Baseline SSD, pSSD, pnSSD, NoSSD and Venice on performance-optimized (top) and cost-optimized (bottom) SSD configurations. Throughput values (in IOPS) are normalized to the path-conflict-free SSD.

We make two key observations. First, Venice improves SSD throughput over Baseline SSD/pSSD/pnSSD/NoSSD by 176%/120%/113%/102% in the performance-optimized SSD and 76%/58%/61%/51% in the cost-optimized SSD configuration. Second, Venice’s SSD throughput is within 30% and 10% of the path-conflict-free SSD’s throughput in the performance-optimized and cost-optimized SSD configuration, respectively. We conclude that Venice significantly improves SSD throughput by mitigating the path conflict problem.

Tail Latency. Path conflicts can cause some I/O requests to experience significantly long access latencies. Figures 11(a) and 11(b) show the 99th percentile of I/O request latencies (i.e., tail latency) in the path-conflict-free SSD, Venice, NoSSD, pnSSD, pSSD and Baseline SSD in the form of a cumulative density function (CDF) for two representative workloads src1_0 and hm_0, respectively. We use only the performance-optimized SSD configuration for this experiment. We make the key observation that Venice significantly reduces the tail latency compared to prior systems. For src1_0, Venice reduces the tail latency by 32%/31%/30%/27% over Baseline SSD/pSSD/pnSSD/NoSSD. For hm_0, Venice reduces the tail latency by 22%/21%/18%/17% over Baseline SSD/pSSD/pnSSD/NoSSD.

6.2 Mixed Workloads
To evaluate the effectiveness of Venice at improving SSD performance in real-world scenarios where multiple workloads access the SSD, we compare SSD performance using different systems under six mixed workloads. Figure 12 shows the speedup of pSSD, pnSSD, NoSSD, Venice and path-conflict-free SSD over Baseline SSD for six mixed workloads. We report results only for the performance-optimized SSD configuration.

We make two key observations. First, across all mixed workloads, Venice provides an average speedup of 1.83×/1.81×/1.80×/1.63×/1.49× over Baseline SSD.
SSD/pSSD/pnSSD/NoSSD. Second, Venice’s performance improvement is higher in mix6. In mix6, the average inter-request arrival time is 90% lower than its constituent workloads, leading to increased path conflicts in Baseline SSD. Venice is able to schedule I/O requests on conflict-free paths using its non-minimal fully-adaptive routing algorithm. We conclude that Venice outperforms prior approaches on high-intensity mixed workloads by effectively mitigating path conflicts.

6.3 Path Conflict Analysis

To show the effectiveness of Venice at mitigating the path conflict problem, we measure the percentage of I/O requests in each workload that experience path conflicts using different systems. Figure 13 shows the results for the Baseline SSD, pSSD, pnSSD, NoSSD and Venice in the performance-optimized SSD configuration.

We make the key observation that Venice significantly mitigates the path conflict problem. Our experimental results show that Venice provides conflict-free paths (on the first try) for 99.98% of I/O requests, on average, across all workloads, while Baseline SSD/pSSD/pnSSD/NoSSD provides conflict-free paths for 76.40%/78.47%/77.88%/80.65% of I/O requests. For a small number of I/O requests (i.e., 0.02% of I/O requests, on average), Venice fails to find conflict-free paths on the first try, and thus, the corresponding I/O requests should wait for a longer amount of time until Venice successfully reserves conflict-free paths. Venice’s path reservation process can fail due to two major reasons. First, if all flash controllers are busy processing ongoing I/O requests, Venice cannot start the path reservation process for a new I/O request until a flash controller becomes idle. Second, during the path reservation process, a scout packet (see §4.2) cannot reserve a path if all the links leading to the destination flash chip are reserved. We conclude that Venice effectively eliminates the path conflict problem via path reservation and effective utilization of path diversity in the SSD interconnection network.

6.4 Power and Energy Consumption

We study the impact of Venice and prior approaches on the SSD power and energy consumption.

**Average SSD Power Consumption.** Figure 14(a) shows the average power consumption for pSSD, pnSSD, NoSSD and Venice on a performance-optimized SSD configuration. Average power consumption values are normalized to the average power consumption of the Baseline SSD. We make four key observations. First, Venice reduces SSD average power consumption by 4% compared to the Baseline SSD. This is mainly because a link in the interconnection network consumes significantly lower power than the shared channel (see §6.6). Second, Venice consumes slightly less power (around 1%) than NoSSD due to Venice’s simple router design. Third, the impact of Venice and prior systems on SSD power consumption is small since the SSD power consumption is dominated by flash operations (i.e., read, program, and erase). The number of flash operations remains the same in Venice and all prior systems.

**SSD Energy Consumption.** To calculate the energy consumption of Venice and prior approaches, we multiply the average power consumption by the overall execution time of each workload. Figure 14(b) shows the energy consumption for pSSD, pnSSD, NoSSD and Venice on a performance-optimized SSD configuration. Energy consumption values are normalized to the Baseline SSD.

6.5 Sensitivity to Interconnection Network Configurations

We study the effect of the interconnection network configuration on Venice’s performance improvement. To this end, we compare the performance of Venice and prior works using three systems that use 4, 8, and 16 flash controllers in the performance-optimized SSD configuration. We keep the total number of flash chips in the SSD constant across the three systems. Figure 15 shows the average speedup of pSSD, NoSSD, Venice, and path-conflict-free SSD over the Baseline SSD, across all workloads. The X-axis shows three systems, 4×16, 8×8, and 16×4. 4×16, for example, denotes four flash controllers with 16 flash chips in each row of the flash array.

We make two key observations from our sensitivity analysis. First, Venice provides significant speedup over prior approaches across all three systems with different numbers of flash controllers. Venice outperforms Baseline SSD/pSSD/NoSSD by 1) 2×/1.7×/1.5×.

*Note that we omit pnSSD from this study because pnSSD requires an N×N flash array configuration where N is the number of flash controllers as well as the number of flash chips in each row in the flash array. Hence, 4×16 and 16×4 configurations are not supported by pnSSD.

---

**Figure 13:** Percentage of I/O requests in Baseline SSD, pSSD, pnSSD, NoSSD and Venice that experience path conflicts in each workload on a performance-optimized SSD configuration. The y-axis is shown in two parts in order to display the negligible number of I/O requests that suffer from path conflicts in Venice.
in 4×16, 2) 2.6×/2×/1.9× in 8×8, and 3) 1.9×/1.8×/1.7× in 16×4. Second, Venice’s performance improvement is higher for the 8×8 flash array configuration compared to both 4×16 and 16×4 configurations. In the system with 4 flash controllers, Venice can reserve conflict-free paths for up to four ongoing I/O requests in the system with 8 flash controllers. As a result, Venice has a lower ability to eliminate the path conflict problem in the system with 4 flash controllers, which results in lower performance improvements for Venice. On the other hand, the system with 16 flash controllers has a lower number of path conflicts compared to systems with 4 and 8 flash controllers, and thus, Venice provides lower performance improvement compared to those in the other two systems. We conclude that Venice is effective for different SSD interconnection network configurations.

6.6 Power and Area Overhead Analysis

Power. As discussed in §6.4, Venice reduces average power and energy consumption. This section studies the power consumption of Venice’s interconnection network router and links. We analyze the power consumption of the router by implementing its hardware description language (HDL) model and synthesizing it for the UMC 65nm technology node [120]. We observe that each router consumes 0.241 mW. We measure the power consumption of each network link using ORION 3.0 [121] power model tool and observe that each link consumes about 1.08 mW for a 4KB NAND flash page transfer, which is 90% less power consumption than that of a shared channel bus. Each network link consumes significantly lower power than a shared channel bus due to its lower capacitance load. Link capacitance is lower than bus capacitance since (1) it is shorter and thinner than a shared bus and (2) it has only two drivers compared to several (e.g., 8 in an 8×8 flash array configuration) drivers in a shared bus. Table 4 (3rd column) summarizes the power consumption of Venice’s components. We have already observed in §6.4 that Venice reduces the average power consumption by 4% over Baseline SSD.

<table>
<thead>
<tr>
<th>Component</th>
<th># of Instances</th>
<th>Avg. Power [mW] for 4KB page transfer</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Router</td>
<td>1 per flash node</td>
<td>0.241</td>
<td>8% of flash chip area</td>
</tr>
<tr>
<td>Link</td>
<td>Up to 4 per flash node</td>
<td>1.08</td>
<td>0.84× flash channel area</td>
</tr>
</tbody>
</table>

Area. Venice does not impose area overhead on NAND flash chip design as it does not integrate the router inside the NAND flash chip (i.e., no pins are added to the commodity flash chips). Router chips and the links connecting them can impose area overhead on SSD printed circuit board (PCB) design. To estimate this overhead, we model the area overhead of the interconnection network’s routers and links.

We estimate the area overhead of Venice’s routers using the HDL model of the router. Each router in Venice has an area of 614 μm². However, each router occupies a higher area on the PCB due to its I/O pad overheads. Each router has 40 I/O pins. Considering I/O pad sizes (about 0.2 mm) and the safety distance between two I/O pads (about 0.2 mm), we expect that each router occupies about 8 mm², which is 8% of a typical 100 mm² NAND flash chip [127].

We estimate the area overhead of Venice’s links using ORION 3.0 [121]. Assuming an 8×8 2D mesh topology connecting 64 NAND flash chips, Venice requires 112 network links (instead of the eight shared channels required in the baseline). Note that we do not count injection/ejection links as they are the same as flash chips’ connectors to the shared channel bus. Our experimental results show that each link’s area is roughly 0.04× of the shared channel.

Figure 14: Power consumption (top) and energy consumption (bottom) for pSSD, pnSSD, NoSSD and Venice on the performance-optimized SSD configuration. Power and energy consumption values are normalized to the Baseline SSD.

Figure 15: Performance speedup of pSSD, NoSSD, Venice and path-conflict-free SSD in the performance-optimized SSD configuration. The X-axis shows three systems with 4, 8 and 16 flash controllers respectively. For example, 4×16 denotes 4 flash controllers and 16 flash chips connected in each row of the flash array. The speedup in overall execution time over Baseline SSD is averaged across all workloads.
area. As a result, in total, Venice’s interconnect links occupy 44% lower area compared to the baseline multi-channel shared bus architecture.7 A network link occupies significantly smaller space than a bus for two main reasons. First, links are shorter than buses (e.g., by 8x in our case); thus, link wires can be 8x thinner to ensure the same impedance as the bus. Second, as the link wires are thinner, they require lower pitch sizes, reducing the overall area required by the links. We summarize the area overhead of Venice’s components in Table 4 (4th column). We conclude that Venice’s benefits come at relatively low area overhead.

7We measure the total area overhead of Venice’s interconnect links compared to a baseline multi-channel shared bus architecture using this equation:
\[ \frac{1 \times \text{(total area of interconnection network links)}}{\text{(total area of channels in multi-channel shared bus architecture), i.e., } 1 \times (\text{#Links} \times \text{Link area})}{\text{(Channels} \times \text{Channel area})} = \frac{(112 \times 0.04)}{(8 \times 1)} = 0.44.\]

7 We propose Venice to improve SSD performance by mitigating the path conflict problem in SSDs. Venice can be extended to improve SSD and system performance in other ways. We discuss other use cases of Venice in this section.

7 Applicability of Venice to Near-Data Processing (NDP). NDP is a computing paradigm that moves the computation closer to where the data resides (e.g., [2, 24, 25, 135–160]). Venice’s improved parallelism can facilitate NDP inside the SSD by efficiently co-locating different operands required by the NDP operations. Prior proposals [24, 139] that perform in-flash bulk bitwise operations have data location constraints where the operands must be moved to a single flash chip before the computation is performed. Path conflicts can impact this data movement, which can significantly reduce the performance benefits of in-flash processing. Venice can leverage its improved flash-array parallelism to efficiently gather operand data from different flash chips to the target flash chip that performs the NDP computation.

7 Improving Garbage Collection. The garbage collection (GC) process [6, 19, 22, 27, 46, 57, 81–85] in NAND flash-based systems is critical to reduce fragmentation and maintain free blocks for write operations. During GC, the SSD controller reads a large number of valid pages from victim blocks. These pages are written to new blocks in the same flash chip or a different flash chip. This data movement can interfere with I/O requests [15, 23, 57, 89, 161–163] and cause path conflicts. Venice can leverage its improved path diversity to efficiently schedule both host I/O requests and GC-related requests in parallel.

9 Conclusion
We propose Venice, a new mechanism that introduces a low-cost interconnection network of flash chips and utilizes the path diversity efficiently to fundamentally address the path conflict problem in SSDs. Venice mitigates path conflicts and improves SSD parallelism using three key techniques: (1) a simple router chip placed next to each flash chip without modifying the flash chip itself, (2) a path reservation technique to reserve a path for each I/O request from the SSD controller to the target flash chip, and (3) a non-minimal fully-adaptive routing algorithm to effectively utilize the path diversity in the interconnection network. Our evaluation shows that Venice significantly improves performance over state-of-the-art prior approaches on both performance-optimized and cost-optimized SSD configurations for a wide range of real-world data-intensive workloads, by effectively eliminating path conflicts. As the demand for performance and scalability of SSDs increases, we hope that Venice inspires future work in several directions to mitigate path conflicts and improve parallelism within the SSD.

Acknowledgments
We thank the anonymous reviewers of ISCA 2023 for their feedback and comments. We thank the SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts of our industrial partners, especially Google, Huawei, Intel, Microsoft and VMware. This research was partially supported by the Semiconductor Research Corporation, the Swiss National Science Foundation, and the ETH Future Computing Laboratory.
References


Venice: Improving Solid-State Drive Parallelism at Low Cost via Conflict-Free Accesses

ISCA ’23, June 17–21, 2023, Orlando, FL, USA


