Willow: A User-Programmable SSD

Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, *University of California, San Diego*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Willow: A User-Programmable SSD

Sudharsan Seshadri Mark Gahagan Sundaram Bhaskaran Trevor Bunker
Arup De Yanqin Jin Yang Liu Steven Swanson
Computer Science & Engineering, UC San Diego

Abstract

We explore the potential of making programmability a central feature of the SSD interface. Our prototype system, called Willow, allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The *SSD Apps* running on Willow give applications low-latency, high-bandwidth access to the SSD's contents while reducing the load that IO processing places on the host processor. The programming model for SSD Apps provides great flexibility, supports the concurrent execution of multiple SSD Apps in Willow, and supports the execution of trusted code in Willow.

We demonstrate the effectiveness and flexibility of Willow by implementing six SSD Apps and measuring their performance. We find that defining SSD semantics in software is easy and beneficial, and that Willow makes it feasible for a wide range of IO-intensive applications to benefit from a customized SSD interface.

1 Introduction

For decades, computer systems have relied on the same block-based interface to storage devices: reading and writing data from and to fixed-sized sectors. It is no accident that this interface is a perfect fit for hard disks, nor is it an accident that the interface has changed little since its creation. As other system components have gotten faster and more flexible, their interfaces have evolved to become more sophisticated and, in many cases, programmable. However, hard disk performance has remained stubbornly poor, hampering efforts to improve performance by rethinking the storage interface.

The emergence of fast, non-volatile, solid-state memories (such as NAND flash and phase-change memories) has signaled the beginning of the end for painfully slow storage systems, and this demands a fundamental rethinking of the interface between storage software and the storage device. These new memories behave very differently than disks—flash requires out-of-place updates while phase change memories (PCMs) provide byte-addressability—and those differences beg for interfaces that go beyond simple block-based access.

The scope of possible new interfaces is enormously broad and includes both general-purpose and application-specific approaches. Recent work has illustrated some of the possibilities and their potential benefits. For instance, an SSD can support complex atomic operations [10, 32, 35], native caching operations [5, 38], a large, sparse storage address space [16], delegating storage allocation decisions to the SSD [47], and offloading file system permission checks to hardware [8]. These new interfaces allow applications to leverage SSDs' low latency, ample internal bandwidth, and on-board computational resources, and they can lead to huge improvements in performance.

Although these features are useful, the current one-at-a-time approach to implementing them suffers from several limitations. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturer can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult. For example, editable atomic writes [10] were designed to support ARIES-style write-ahead logging, but not all databases take that approach.

To overcome these limitations, we propose to make programmability a central feature of the SSD interface, so ordinary programmers can safely extend their SSDs' functionality. The resulting system, called *Willow*, will allow application, file system, and operating system programmers to install customized (and potentially untrusted) *SSD Apps* that can modify and extend the SSD's behavior.

Applications will be able to exploit this kind of programmability in (at least) four different ways.

- **Data-dependent logic:** Many storage applications perform data-dependent read and write operations to manipulate on-disk data structures. Each data-dependent operation requires a round-trip between a conventional SSD and the host across the system bus (i.e., PCIe, SATA, or SAS) and through the operating system, adding latency and increasing host-side software costs.

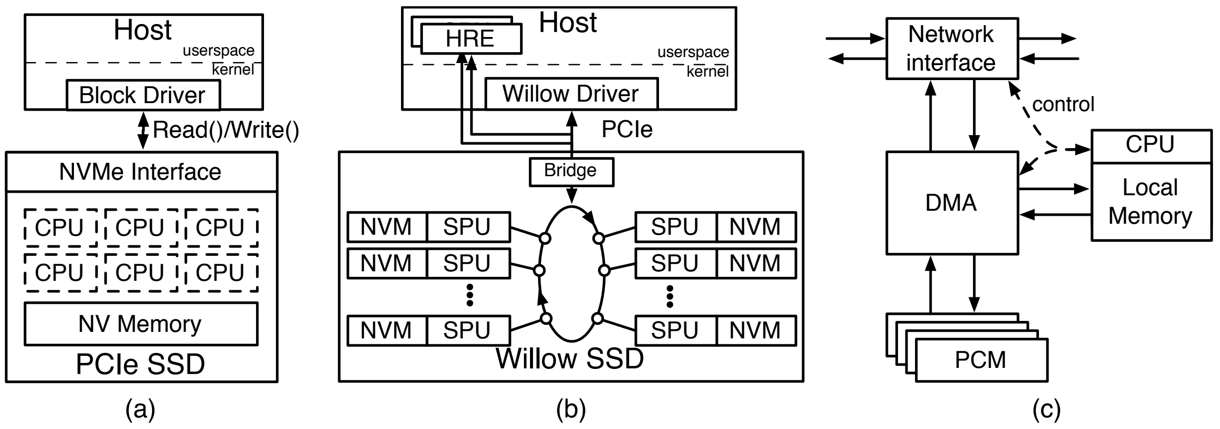


Figure 1: **A conventional SSD vs. Willow.** Although both a conventional SSD (a) and Willow (b) contain programmable components, Willow's computation resources (c) are visible to the programmer and provide a flexible programming model.

- **Semantic extensions:** Storage features like caching and logging require changes to the *semantics* of storage accesses. For instance, a write to a caching device could include setting a dirty bit for the affected blocks.

- **Privileged execution:** Executing privileged code in the SSD will allow it to take over operating and file system functions. Recent work [8] shows that issuing a request to an SSD via an OS-bypass interface is faster than a system call, so running some trusted code in the SSD would improve performance.

- **Data intensive computations:** Moving data-intensive computations to the storage system has many applications, and previous work has explored this direction in disks [37, 1, 19] and SSDs [17, 6, 43] with promising results.

Willow focuses on the first three of these use cases and demonstrates that adding generic programmability to the SSD interface can significantly reduce the cost and complexity of adding new features. We describe a prototype implementation of Willow based on emulated PCM memory that supports a wide range of applications. Then, we describe the motivation behind the design decisions we made in building the prototype. We report on our experience implementing a suite of example SSD Apps. The results show that Willow allows programmers to quickly add new features to an SSD and that applications can realize significant gains by offloading functionality to Willow.

This paper provides an overview of Willow, its programming model, and our prototype in Sections 2 and 3. Section 4 presents and evaluates six SSD Apps, Section 5 places our work in the context of other approaches to integrating programmability into storage devices. Section 6 describes some of the insights we gained from this work, and Section 7 concludes.

2 System Design

Willow revisits the interface that the storage device exposes to the rest of the system, and provides the hardware necessary to support that interface efficiently. This section describes the system from the programmer's perspective, paying particular attention to the programming model and hardware/software interface. Section 3 describes the prototype hardware in more detail.

2.1 Willow system components

Figure 1(a) depicts a conventional storage system with a high-end, PCIe-attached SSD. A host system connects to the SSD via NVM Express (NVMe) [30] over PCIe, and the operating system sends commands and receives responses over that communication channel. The commands are all storage-specific (e.g., read or write a block) and there is a point-to-point connection between the host operating system and the storage device. Modern, high-end SSDs contain several (often many) embedded, programmable processors, but that programmability is not visible to the host system or to applications.

Figure 1(b) shows the corresponding picture of the Willow SSD. Willow's components resemble those in a conventional SSD: it contains several *storage processor units (SPUs)*, each of which includes a microprocessor, an interface to the inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a very small operating system called SPU-OS that manages and enforces security (see Section 2.6 below).

The interface that Willow provides is very different from the interface of a conventional SSD. On the host side, the Willow driver creates and manages a set of ob-

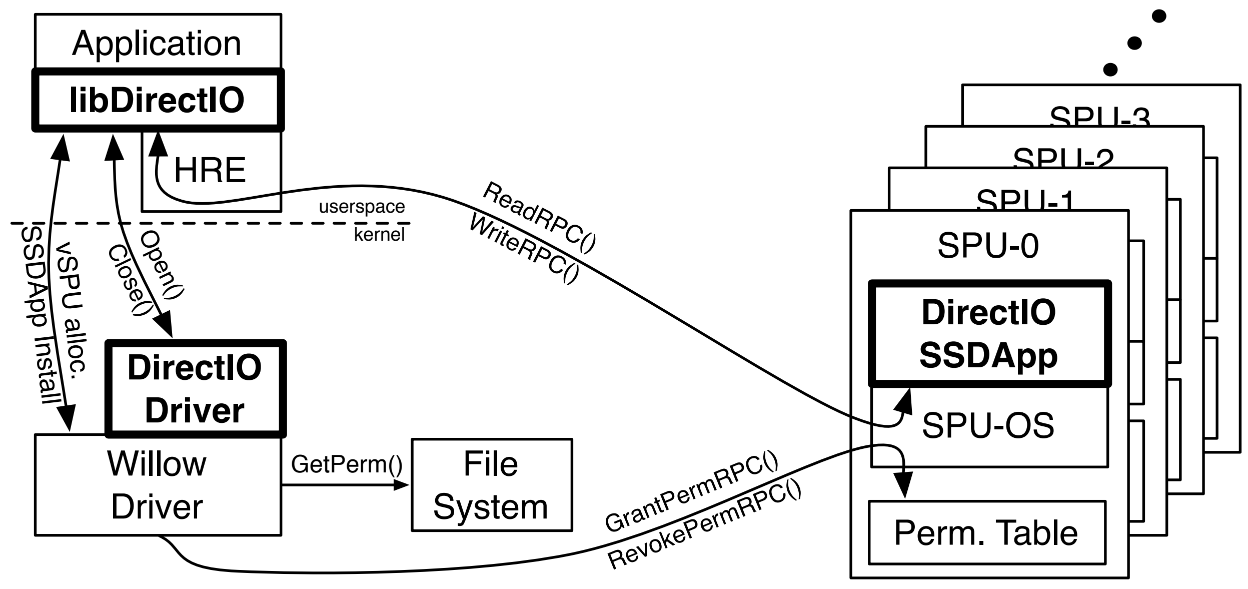


Figure 2: **The Anatomy of an SSD App.** The boldface elements depict three components of an SSD App: a userspace library, the SPU code, and an optional kernel driver. In the typical use case, a conventional file system manages the contents of Willow, and the Willow driver grants access to file extents based on file system permissions.

jects called *Host RPC Endpoints (HREs)* that allow the OS and applications to communicate with SPUs. The HRE is a data structure that the kernel creates and allocates to a process. It provides a unique identifier called the *HRE ID* for sending and receiving RPC requests and lets the process send and receive those requests via DMA transfers between userspace memory and the Willow SSD. The SPUs and HREs communicate over a flexible network using a simple, flexible RPC-based mechanism. The RPC mechanism is generic and does not provide any storage-specific functionality. SPUs can send RPCs to HREs and vice versa.

The final component of Willow is programmable functionality in the form of SSD Apps. Each SSD App consists of three elements: a set of RPC handlers that the Willow kernel driver installs at each SPU on behalf of the application, a library that an application uses to access the SSD App, and a kernel module, if the SSD App requires kernel support. Multiple SSD Apps can be active at the same time.

Below, we describe the high-level system model, the programming model, and the security model for both SPUs and HREs.

2.2 The Willow Usage Model

Willow's design can support many different usage models (e.g., a system could use it as a tightly-coupled network of "wimpy" compute nodes with associated storage). Here, however, we focus on using Willow as a conventional storage device that also provides programmability features. This model is particularly useful because it allows for incremental adoption of Willow's features and ensures that legacy applications can use Willow without modification.

In this model, Willow runs an SSD App called `Base-IO` that provides basic block device functionality (i.e., reading and writing data from and to storage locations). `Base-IO` stripes data across the SPUs (and their associated banks of non-volatile memory) in 8 kB segments. `Base-IO` (and all the other SSD Apps we present in this paper) runs identical code at each SPU. We have found it useful to organize data and computation in this way, but Willow does not require it.

A conventional file system manages the space on Willow and sets permissions that govern access to the data it holds. The file system uses the `Base-IO` block device interface to maintain metadata and provide data access to applications that do not use Willow's programmability.

To exploit Willow's programmability, an application needs to install and use an additional SSD App. Figure 2 illustrates this process for an SSD App called `Direct-IO` that provides an OS-bypass interface that avoids system call and file system overheads for common-case reads and writes (similar to [8]). The figure shows the software components that comprise `Direct-IO` in bold. To use `Direct-IO`, the application uses the `Direct-IO`'s userspace library, `libDirectIO`. The library asks the operating system to install `Direct-IO` in Willow and requests an HRE from the Willow driver to allow it to communicate with the Willow SSD.

`Direct-IO` also includes a kernel module that `libDirectIO` invokes when it needs to open a file on behalf of the application. The `Direct-IO` kernel module asks the Willow driver to grant the application permission to access the file. The driver requests the necessary permission information from the file system and issues trusted RPCs to SPU-OS to install the permission for the file extents the application needs to access in the SPU-OS permission table. Modern file systems already include the ability to query permissions from inside the kernel, so no changes to the file system are necessary.

`Base-IO` and `Direct-IO` are "standard equipment" on Willow, since they provide functions that are useful for many other SSD Apps. In particular, other SSD Apps can leverage `Direct-IO`'s functionality to implement arbitrary, untrusted operations on file data.

2.3 Building an SSD App

SSD Apps comprise interacting components running in multiple locations: in the client application (e.g., `libDirectIO`), in the host-side kernel (e.g., the `Direct-IO` kernel module), and in the Willow SSD. To minimize complexity, code in all three locations uses a common set of interfaces to implement SSD App functionality. In the host application and the kernel, the HRE library implements these interfaces, while in Willow, SPU-OS implements them. The interfaces provide the following capabilities:

1. **Send an RPC request:** SPUs and HREs can issue RPC requests to SPUs, and SPUs can issue RPCs to HREs. RPC delivery is non-reliable (due to limited buffering at the receiver), and all-or-nothing (i.e., the recipient will not receive a partial message). The sender is notified upon successful (or failed) delivery of the message. Willow supports both synchronous and asynchronous RPCs.
2. **Receive an RPC request:** RPC requests carry an *RPC ID* that specifies which SSD App they target and which handler they should invoke. When an RPC request arrives at an SPU or HRE, the runtime (i.e., the HRE library or SPU-OS) invokes the correct handler for the request.
3. **Send an RPC response:** RPC responses are short, fixed-length messages that include a result code and information about the request it responds to. RPC response delivery is reliable.
4. **Initiate a data transfer:** An RPC handler can asynchronously transfer data between the network interface, local memory, and the local non-volatile memory (for SPUs only).
5. **Allocate local memory:** SSD Apps can declare static variables to allocate space in the SPU's local data memory, but they cannot allocate SPU memory dynamically. Code on the host can allocate data statically or on the heap.
6. **General purpose computation:** SSD Apps are written in C, although the standard libraries are not available on the SPUs.

In addition to these interfaces, the host-side HRE library also provides facilities to request HREs from the Willow driver and install SSD Apps.

This set of interfaces has proved sufficient to implement a wide range of different applications (see Section 4), and we have found them flexible and easy to use. However, as we gain more experience building SSD Apps, we expect that opportunities for optimization, new capabilities, and bug-preventing restrictions on SSD Apps will become apparent.

2.4 The SPU Architecture

In modern SSDs (and in our prototype), the embedded processor that runs the SSD's firmware offers only modest performance and limited local memory capacity compared to the bandwidth that non-volatile memory and the SSD's internal interconnect can deliver.

In addition, concerns about power consumption (which argue for lower clock speeds) and cost (which argue for simple processors) suggest this situation will persist, especially as memory bandwidths continue to grow. These constraints shape both the Willow hardware we propose and the details of the RPC mechanism we provide.

The SPU has four hardware components we use to implement the SSD App toolkit (Figure 1(c)):

1. **SPU processor:** The processor provides modest performance (perhaps 100s of MIPS) and kilobytes of per-SPU instruction and data memory.
2. **Local non-volatile memory:** The array of non-volatile memory can read or write data at over 1 GB/s.
3. **Network interface:** The network provides gigabytes-per-second of bandwidth to match the bandwidth of the local non-volatile memory array and the link bandwidth to the host system.
4. **Programmable DMA controller:** The DMA controller routes data between non-volatile memory, the network port, and the processor's local data memory. It can handle the full bandwidth of the network and local non-volatile memory.

The DMA controller is central to the design of both the SPU and the RPC mechanism, since it allows the modestly powerful processor to handle high-bandwidth streams of data. We describe the RPC interface in the following section.

The SPU runs a simple operating system (SPU-OS) that provides simple multi-threading, works with the Willow host-side driver to manage SPU memory resources, implements protection mechanisms that allow multiple SSD Apps to be active at once, and enforces the

```
void Read_Handler (RPCHdr_t *request_hdr) {        // RPCHdr_t part of the RPC interface
    // Parse the incoming RPC
    BaseIOCmd_t cmd;
    RPCReceiveBytes(&cmd, sizeof(BaseIOCmd_t));    // DMA the IO command header
    RPCResp_t response_hdr;                        // Allocate response
    RPCCreateResponse(request_hdr,                 // populate the response
                      &response_hdr,
                      RPC_SUCCESS);
    RPCSendResponse(response_hdr);                 // Send the response

    // Send the read data back via a second RPC
    CPUID_t dst = request_hdr->src;
    RPCStartRequest(dst,                           // Destination PU
                    sizeof(IOCmd_t) + cmd.length,  // Request body length
                    READ_COMPLETE_HANDLER);        // Read completion RPC ID
    RPCAppendRequest(LOCAL_MEMORY_PORT,            // Source DMA port
                     sizeof(BaseIOCmd_t),          // IO command header size
                     &cmd);                        // IO command header address
    RPCAppendRequest(NV_MEMORY_PORT,               // Source DMA Port
                     cmd.length,                   // Bytes to read
                     cmd.addr);                    // Read address
    RPCFinishRequest();                            // Complete the request
}
```

Figure 3: **READ() implementation for `Base-IO`.** Handling a READ() requires parsing the header on the RPC request and then sending requested data from non-volatile memory back to host via another RPC.

file system's protection policy for non-volatile storage. Section 2.6 describes the protection facilities in more detail.

2.5 The RPC Interface

The RPC mechanism's design reflects the constraints of the hardware described above. Given the modest performance of the SPU processor and its limited local memory, buffering entire RPC messages at the SPU processor is not practical. Instead, the RPC library parses and assembles RPC requests in stages. The code in Figure 3 illustrates how this works for a simplified version of the READ() RPC from `Base-IO`.

When an RPC arrives, SPU-OS copies the RPC header into a local buffer using DMA and passes the buffer to the appropriate handler (`Read_Handler`). That handler uses the DMA controller to transfer the RPC parameters into the SPU processor's local memory (`RPCReceiveBytes`). The header contains generic information (e.g., the source of the RPC request and its size), while the parameters include command-specific values (e.g., the read or write address). The handler uses one or more DMA requests to process the remainder of the request. This can include moving part of the request to the processor's local memory for examination or performing bulk transfers between the network port and the non-volatile memory bank (e.g., to implement a write). In the example, no additional DMA transfers are needed.

The handler sends a fixed-sized response to the RPC request (`RPCCreateResponse` and `RPCSendResponse`). Willow guarantees the reliable delivery of fixed-size responses (acks or nacks) by guaranteeing space to receive them when the RPC is sent. If the SSD App needs to send a response that is longer than 32 bits (e.g., to return the data for a read), it must issue an RPC to the sender. If there is insufficient buffer space at the receiver, the inter-SPU communication network can drop packets. In practice, however, dropped packets are exceedingly rare.

The process of issuing an RPC to return the data follows a similar sequence of steps. The SPU gives the network port the destination and length of the message (`RPCStartRequest`). Then it prepares any headers in local memory and uses the DMA controller to transfer them to the network interface (`RPCAppendRequest`). Further DMA requests can transfer data from non-volatile memory or processor memory to the network interface to complete the request. In this case, the SSD App transfers the read data from the non-volatile memory. Finally, it makes a call to signal the end of the message (`RPCFinishRequest`).

2.6 Protection and sharing in Willow

Willow has several features that make it easy for users to build and deploy useful SSD Apps: Willow supports untrusted SSD Apps, protects against malicious SSD Apps (assuming the host-side kernel is not compromised), allows multiple SSD Apps to be active simultaneously, and allows one SSD App to leverage functionality that another provides. Together these four features allow a user to build and use an SSD App without the permission of a system administrator and to focus on the functionality specific to his or her particular application.

Providing these features requires a suite of four protection mechanisms. First, it must be clear which host-side process is responsible for the execution of code at the SPU, so SPU-OS can enforce the correct set of protection policies. Second, the SPU must allow an SSD App to access data stored in Willow only if the process that initiated the current RPC has access rights to that data. Third, the SPU must restrict an SSD App to accessing only its own memory and executing only its own code. Finally, it must allow some control transfers between SSD Apps so the user can compose SSD Apps. We address each of these below.

Tracking responsibility: The host system is responsible for setting protection policy for Willow, and it does so by associating permissions with operating system processes. To correctly enforce the operating system's policies, SPU-OS must be able to determine which process is responsible for the RPC handler that is currently running.

To facilitate this, Willow tracks the *originating HRE* for each RPC. An HRE is the originating HRE for any RPCs it makes and for any RPCs that an SPU makes as a result of that RPC and any subsequent RPCs. The PCIe interface hardware in the Willow SSD sets the originating HRE for the initial RPC, and SPU hardware and SPU-OS propagate it within the SSD. As a result, the originating HRE ID is unforgeable and serves as a capability [23].

To reduce cache coherence traffic, it is useful to give each thread in a process its own HRE. The Willow driver allocates HREs so that the high-order bits of the HRE ID are the same for every HRE belonging to a single process.

Non-volatile storage protection: To limit access to data in the non-volatile memory banks, SPU-OS maintains a set of permissions for each process at each SPU. Every time the SSD App uses the DMA controller to move data to or from non-volatile memory, SPU-OS checks that the permissions for the originating HRE (and therefore the originating process) allow it. The worst-case permission check latency is 2 μs.

The host-side kernel driver installs extent-based permission entries on behalf of a process by issuing privileged RPCs to SPU-OS. The SPU stores the permissions for each process as a splay tree to minimize permission check time. Since the SPU-OS permission table is fixed size, it may evict permissions if space runs short. If a request needs an evicted permission entry, a "permission miss" occurs, and the DMA transfer will fail. In response, SPU-OS issues an RPC to the kernel. The kernel forwards the request to the SSD App's kernel module (if it has one), and that kernel module is responsible for resolving the miss. Most of our SSD Apps use the `Direct-IO` kernel module to manage permissions, and it will re-install the permission entry as needed.

Code and Data Protection: To limit access to the code and data in the SPU processor's local memory, the SPU processor provides segment registers and disallows access outside the current segment. Each SSD App has its own data and instruction segments that define the base address and length of the instruction and data memory regions it may access. Accesses outside the SSD App's segment raise an exception and cause SPU-OS to notify the kernel via an RPC, and the kernel, in turn, notifies the applications that the SSD App is no longer available. SPU-OS provides a trusted RPC dispatch mechanism for incoming messages. This mechanism sets the segment registers according to the SSD App that the RPC targets.

The host-side kernel is in charge of managing and statically allocating SPU instruction and data memory to the active SSD Apps. Overlays could extend the effective instruction and data memory size (and are common in commercial SSD controller firmware), but we have not implemented them in our prototype.

Limiting access to RPCs: A combination of hardware and software restricts access to some RPCs. This allows safe composition of SSD Apps and allows SSD Apps to create RPCs that can be issued only from the host-side kernel.

To support composition, SPU-OS provides a mechanism for changing segments as part of a function call from one SSD App to another. An *SSD App-intercall table* in each SPU controls which SSD Apps are allowed to invoke one another and which function calls are allowed. A similar mechanism restricts which RPCs one SSD App can issue to another.

To implement kernel-only RPCs, we use the convention that a zero in the high-order bit of the HRE ID means the HRE belongs to the kernel. RPC implementations can check the ID and return failure when a non-kernel HRE invokes a protected RPC.

SSD Apps can use this mechanism to bootstrap more complex protection schemes as needed. For example, they could require the SSD App's kernel module to grant access to userspace HREs via a kernel-only RPC.

3 The Willow Prototype

We have constructed a prototype Willow SSD that implements all of the functionality described in the previous section. This section provides details about the design.

The prototype has eight SPUs and a total storage capacity of 64 GB. It is implemented using a BEE3 FPGA-based prototyping system [4]. The BEE3 connects to a host system over a PCIe 1.1x8. The link provides 2 GB/s of full-duplex bandwidth.

Each of the four FPGAs that make up a BEE3 hosts

Description	Name	LOC (C)	Devel. Time (Person-months)
Simple IO operations [7]	`Base-IO`	1500	1
Virtualized SSD interface with OS bypass and permission checking [8]	`Direct-IO`	1524	1.2
Atomic writes tailored for scalable database systems based on [10]	`Atomic-Write`	901	1
Direct-access caching device with hardware support for dirty data tracking [5]	`Caching`	728	1
SSD acceleration for MemcacheDB [9]	`Key-Value`	834	1
Offload file appends to the SSD	`Append`	1588	1

Table 1: **SSD Apps.** Implementing and testing each SSD App required no more than five weeks and less than 1600 lines of code.

two SPUs, each attached to an 8 GB bank of DDR2 DRAM. We use the DRAM combined with a customized memory controller to emulate phase change memory with a read latency of 48 ns and a write latency of 150 ns. The memory controller implements start-gap wear-leveling [36].

The SPU processor is a 125 MHz RISC processor with a MIPS-like instruction set. It executes nearly one instruction per cycle, on average. We use the MIPS version of gcc to generate executable code for it. For debugging, it provides a virtual serial port and a rich set of performance counters and status registers to the host. The processor has 32 kB of local data memory and 32 kB of local instruction memory.

The kernel driver statically allocates space in the SPU memory to SSD Apps, which constrains the number and size of SSD Apps that can run at once. SPU-OS maintains a permission table in the local data memory that can hold 768 entries and occupies 20 kB of data memory.

The ring in Willow uses round-robin, token-based arbitration, so only one SPU may be sending a message at any time. To send a message, the SPU's network interface waits for the token to arrive, takes possession of it, and transmits its data. To receive a message, the interface watches the header of messages on the ring to identify messages it should remove from the ring. The ring is 128 bits wide and runs at 250 MHz for a total of 3.7 GB/s of bisection bandwidth.

For communication with the HREs on the host, a bridge connects the ring to the PCIe link. The bridge serves as a hardware proxy for the HREs. For each of the HREs, the bridge maintains an upstream (host-bound) and downstream (Willow-bound) queue. This queue-based interface is similar to the scheme that NVMExpress [30] uses to issue and complete IO requests. The bridge in our prototype Willow supports up to 1024 queue pairs, so it can support 1024 HREs on the host.

The bridge also helps enforce security in Willow. Messages from HREs to SPUs travel over the bridge, and the bridge sets the originating HRE fields on those messages depending on which HRE queue they came in on. Since processes can send messages only via the queues for the HREs they control, processes cannot send forged RPC requests.

4 Case Studies

Willow makes it easy for storage system engineers to improve performance by incorporating new capabilities into a storage device. We have evaluated Willow's effectiveness in this regard by implementing six different SSD Apps and comparing their performance to implementations that use a conventional storage interface.

The six applications are: basic IO, IO with OS bypass, atomic-writes, caching, a key-value store, and appending data to a file in the Ext4 filesystem. Table 1 briefly describes all six apps and provides some statistics about their implementations. We discuss each in detail below.

4.1 Basic IO

The first SSD App is `Base-IO`, the SSD App we described briefly in Section 2 that provides basic SSD functionality: READ(), WRITE(), and a few utility operations (e.g., querying the size of the device) that the operating system requires to recognize Willow as a block device.

A Willow SSD with `Base-IO` approximates a conventional SSD, since the SSD's firmware would implement the same functions that `Base-IO` provides. We compare to `Base-IO` throughout this section to understand the performance impact of Willow's programmability features.

Figure 4 plots the performance of `Base-IO`. We collected the data by running XDD [46] on top of XFS. `Base-IO` is able to utilize 78% and 73% of the PCIe bandwidth for read and write, respectively, and can sustain up to 388K read IOPs for small accesses. This level of PCIe utilization is comparable to what we have seen in commercial high-end PCIe SSDs.

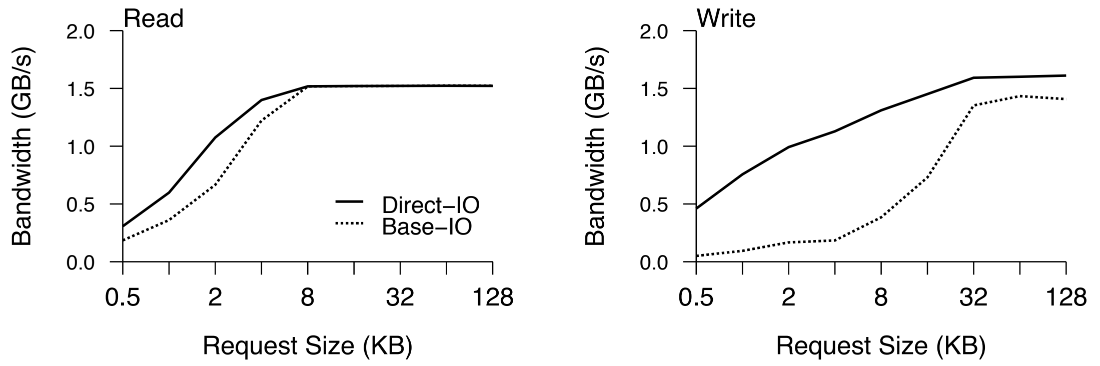


Figure 4: **Bandwidth comparison between `Direct-IO` and `Base-IO`.** Bypassing the kernel with a virtualized SSD interface and software permission checks improves the performance by up to 66% for reads, and 8× for writes, relative to Base-IO.

4.2 Direct-IO

The second SSD App is `Direct-IO`, the OS-bypass interface that allows applications to perform READ() and WRITE() operations without operating system intervention. We described `Direct-IO` in Section 2. `Direct-IO` is similar to the work in [8] and, like that work, `Direct-IO` relies on a userspace library to implement a POSIX-compliant interface for applications.

Figure 4 compares the performance of `Direct-IO` and `Base-IO` running under XFS. `Direct-IO` outperforms `Base-IO` by up to 66% for small reads and 8× for small writes by avoiding system call and file system overheads. The performance gain for writes is larger than for reads because writes require one RPC round trip while reads require two: an RPC from the host to the SSD to send the request and an RPC from the SSD to the host to return the data. `Direct-IO` reduces the cost of the first RPC, but not the second.

Figure 5 breaks down the read latency for 4 kB accesses on three different configurations. All of them share the same hardware (DMA and NVM access) and host-side (command issue, memory copy and software) latencies, but `Direct-IO` saves almost 35% of access latency by avoiding the operating system. The final bar (based on projections) shows that running the SPU at 1 GHz would almost eliminate the impact of SPU software overheads on overall latency, although it would increase power consumption. Such a processor would be easy to implement in a custom silicon version of Willow.

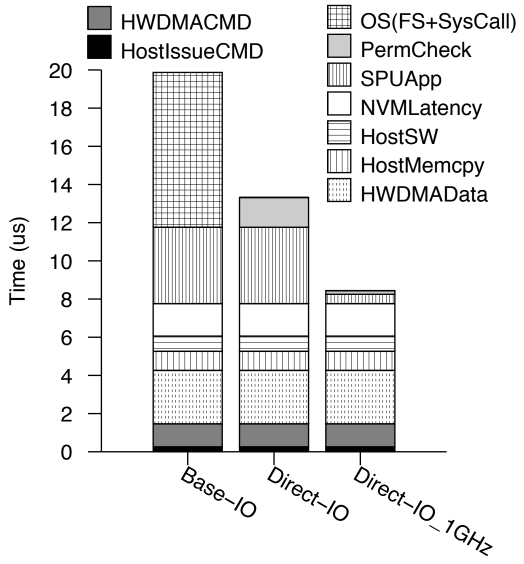


Figure 5: **Read Latency Breakdown.** The bars give the component latencies for `Base-IO` with a file system, for `Direct-IO` on the current SPU processor, and for `Direct-IO` on a hypothetical version of Willow with a 1 GHz processor.

4.3 Atomic Writes

Many storage applications (e.g., file systems and databases) use write-ahead logging (WAL) to enforce strict consistency guarantees on persistent data structures. WAL schemes range from relatively simple journaling mechanisms for file system metadata to the complex ARIES scheme for implementing scalable transactions in databases [27]. Recently, researchers and industry have developed several SSDs with built-in support for multi-part atomic writes [32, 35], including a scheme called MARS [10] that aims to replace ARIES in databases.

MARS relies on a WAL primitive called editable atomic writes (EAW). EAW provides the application with detailed control over where logging information resides inside the SSD and allows it to edit log records prior to committing the atomic operations.

We have implemented EAWs as an SSD App called `Atomic-Writes`. `Atomic-Writes` implements four RPCs—LOGWRITE(), COMMIT(), LOGWRITECOMMIT(), and ABORT()—summarized in Table 2. `Atomic-Writes` makes use of the `Direct-IO` functionality as well.

RPC	Description
LOGWRITE()	Start a new atomic operation and/or add a write to an existing atomic operation.
COMMIT()	Commit an atomic operation.
LOGWRITECOMMIT()	Create and commit an atomic operation comprised of single write.
ABORT()	Abort an atomic operation.

Table 2: **RPCs for `Atomic-Writes`.** The `Atomic-Write` SSD App allows applications to combine multiple writes into a single atomic operation and commit or abort them.

The implementations of LOGWRITE() and COMMIT() illustrate the flexible programmability of Willow's RPC interface. Each SPU maintains the redo-log as a complex persistent data structure for each active transaction. An array of log metadata entries resides in a reserved area of non-volatile memory with each entry pointing to a log record, the data to be written, and the location where it should be written. LOGWRITE() appends an entry to this array and initializes it to add the new entry to the log.

COMMIT() uses a two-phase commit protocol among the SPUs to achieve atomicity. The host library tracks which SPUs are participating in the transaction and selects one of them as the coordinator. In Phase 1, the coordinator broadcasts a "prepare" request to all the SPUs participating in this transaction (including itself). Each participant decides whether to commit or abort and reports back to the coordinator. In Phase 2, if any participant decides to abort, the coordinator instructs all participants to abort. Otherwise the coordinator broadcasts a "commit" request so that each participant plays its local portion of the log and notifies the coordinator when it finishes.

We have modified the Shore-MT [40] storage manager to use MARS and EAW to implement transaction processing. We also fine-tuned EAWs to match how Shore-MT manages transactions, something that would not be possible in the "black box," one-size-fits-all implementation of EAWs that a non-programmable SSD might include. Figure 6 shows the performance difference between MARS and ARIES for TPC-B [44]. MARS scales better than ARIES when increasing thread count and outperforms ARIES by up to 1.5×. These gains are ultimately due to the rich semantics that `Atomic-Writes` provides.

4.4 Caching

SSDs are more expensive and less dense than disks. A cost-effective option for integrating them into storage systems is to utilize high performance SSDs as caches for larger conventional backing stores. Traditional SSD caching systems such as FlashCache [41] and bcache [3] implement cache look-up and management operations as a software component in the operating system. Several groups [5, 38] have proposed adding caching-specific interfaces to SSDs in order to improve the performance of the storage system.

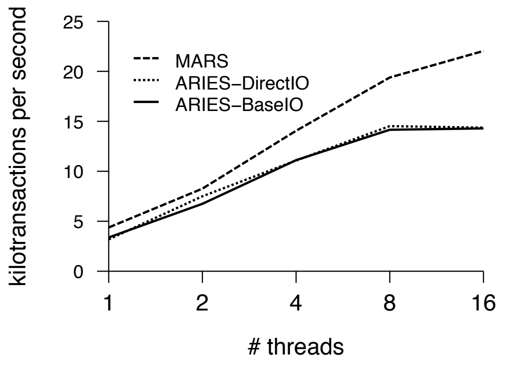


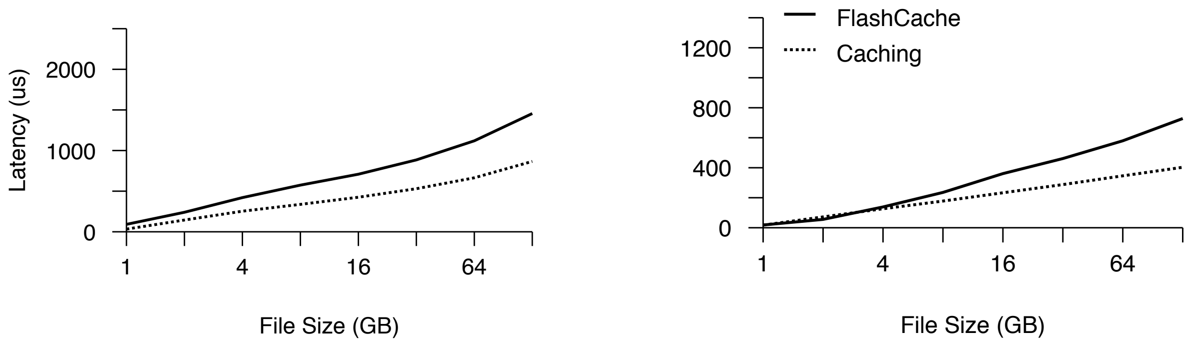
Figure 6: **TPC-B Throughput.** MARS using `Atomic-Writes` yields up to 1.5× throughput gain compared to ARIES using `Base-IO` and `Direct-IO`.

We have implemented an SSD App called `Caching` that turns Willow into a caching SSD. `Caching` tracks which data in the cache are dirty, provides support for recovery after failures, and tracks statistics about which data is "hot" to guide replacement policies. It services cache hits directly from user space using `Direct-IO`'s OS-bypass interface. For misses, `Caching` invokes a kernel-based cache manager. Its design is based on Bankshot [5].

`Caching` transforms Willow into a specialized caching SSD rather than providing application-specific features on top of normal cache access. Instead of using the file system's extent-based protection policy, `Caching` uses a specialized permission mechanism based on large, fixed-size cache chunks (or groups of blocks) that make more efficient use of the SPU's limited local memory. `Caching`'s kernel module uses a privileged kernel-only RPC to install the specialized permission entries and to manage the cache's contents.

To measure `Caching`'s performance we use the Flexible IO Tester (Fio) [14]. We configure Fio to generate Zipf-distributed [2] accesses such that 90% of accesses are to 10% of the data. We vary the file size from 1 GB to 128 GB. We use a 1 GB cache and report average latency after the cache is warm. The backing store is a hard disk.

Figure 7 shows the average read and write latency for 4 kB accesses to FlashCache and `Caching`. Because it is a kernel module, FlashCache uses the `Base-IO` rather than `Direct-IO`. `Caching`'s fully associative



(a) Read Latency (b) Write Latency

Figure 7: **Latency vs. working set size.** `Caching` offers improved average latency for 4 kB reads (a) and writes (b) compared to FlashCache. As the file sizes grow beyond the cache size of 1 GB, latency approaches that of the backing disk.

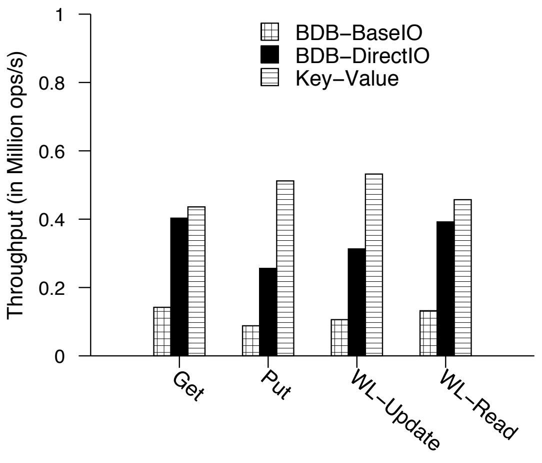


Figure 8: **MemcacheDB performance.** `Key-Value` improves performance of GET() and PUT() operations by 8% and 2× respectively, compared to Berkeley DB running on `Direct-IO`. It improves performance by the same operation by 2× and 4.8× respectively compared to Berkeley DB running on `Base-IO`.

allocation policy allows for more efficient use of cache space, and its ability to allow direct user space access reduces software overheads. `Caching` reduces the miss rate by 6%-23% and improves the cache hit latency for write by 61.8% and read by 36.3%. Combined, these improve read latency by between 1.7 and 2.8× and writes by up to 1.8×.

4.5 Key-Value Store

Key-value stores have proved a very useful tool in implementing a wide range of applications, from smart phone apps to large scale cloud services. Persistent key-value stores such as BerkeleyDB [31], Cassandra [21], and MongoDB [34] rely on complex in-storage data structures (e.g., BTrees or hash tables) to store their data. Traversing those data structures using conventional IO operations results in multiple dependent accesses that consume host CPU cycles and require multiple crossings of the system interconnect (i.e., PCIe). Offloading those dependent accesses to the SSD eliminates much of that latency.

We implement support for key-value operations in an SSD App called `Key-Value`. It provides three RPC functions: PUT() to insert or update a key-value pair, GET() to retrieve the value corresponding to a key, and DELETE() to remove a key-value pair. `Key-Value` stores pairs in a hash table using open chaining to avoid collisions.

`Key-Value` computes the hash of the key on the host and uses the hash value to distribute hash buckets across the SPUs in Willow. For calls to GET() and DELETE(), it passes the hash value and the key (so the SPU can detect matches). For PUT(), it includes the value in addition to the key. All three RPC calls operate on an array of buckets, each containing a linked list of key-value pairs with matching hashes. The SPU code traverses the linked list with a sequence of short DMA requests.

We used MemcacheDB [9] to evaluate `Key-Value`. MemcacheDB [9] combines memcached [26], the popular distributed key-value store, with BerkeleyDB [31], to build a persistent key-value store. MemcacheDB has a client-server architecture, and for this experiment we run it on a single computer that acts both as client (using a 16 thread configuration) and server.

We compare three configurations of MemcacheDB. The first two configurations use BerkeleyDB [31] running on top of `Base-IO` and `Direct-IO` separately to store the key-value pairs. The third replaces BerkeleyDB with a `Key-Value`-based implementation.

We evaluate the performance of GET() and PUT() operations and then measure the overall performance for both update-heavy (50% PUT() / 50% GET()) and read-heavy (5% PUT() / 95% GET()) workloads. Both workloads use random 16-byte keys and 1024-byte values.

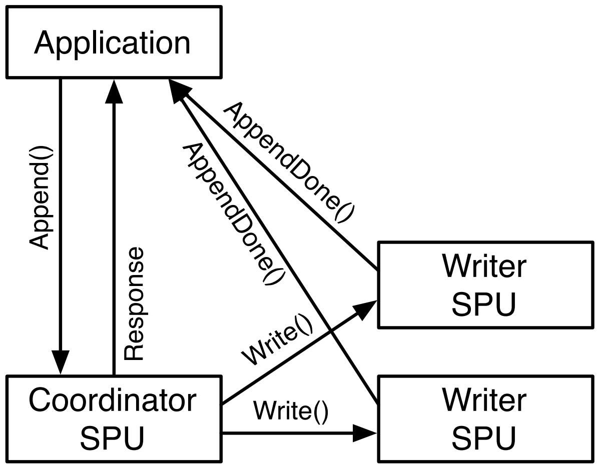


Figure 9: **RPCs to implement APPEND().** The coordinator SPU delegates writing the appended data to the SPUs that host the affected memory banks. Those SPUs notify the host on completion.

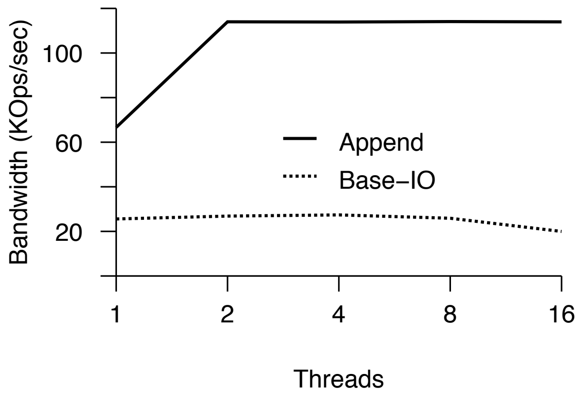


Figure 10: **File append in Willow.** `Append` provides better performance than relying on the operating system to append data to a file.

Figure 8 shows the performance comparison between different MemcacheDB implementations. For GET() and PUT() operations `Key-Value` outperforms the `Direct-IO` configuration by 8.2% and 100% respectively, and improves over the `Base-IO` configuration by 2× and 4.8×. Results for the update- and read-heavy workloads show a similar trend, with `Key-Value` improving performance by between 17% and 70% over the `Direct-IO` configuration and between 2.5× and 4× over the `Base-IO` configuration.

4.6 File system offload

File systems present several opportunities for offloading functionality to Willow to improve performance. We have created an SSD App called `Append` that exploits one of these opportunities, allowing `Direct-IO` to append data to a file (and update the appropriate metadata) from userspace.

`Direct-IO` reduces overheads for most read and write operations by allowing them to bypass the operating system, but it cannot do the same for append operations, since appends require updates to file system metadata. We can extend the OS bypass interface to include appends by building a trusted SSD App that can coordinate with the file system to maintain the correct file length.

`Append` builds upon `Direct-IO` (and `libDirectIO`) and works with a modified version of the Ext4 file system to manage file lengths. The first time an application tries to append to a file, it asks the file system to delegate control of the file's length to `Append`. In response, the file system uses a trusted RPC to tell `Append` where the last extent in the file resides. The file system also sets a flag in the inode to indicate that `Append` has ownership of the file's *logical* length. Since the file system allocates space in 4 kB blocks and may pre-allocate space for the file, the *physical* length of the file is often much longer than the logical length. The physical length remains under the file system's control.

After that, the application can send APPEND() RPCs directly to Willow. Figure 9 illustrates the sequence of RPCs involved. The APPEND() RPCs include the file's inode number and the data to append. The application sends the RPC to the SPU whose ID is the inode number modulo the number of SPUs in Willow.

When the SPU receives an APPEND() RPC, it checks to see whether the application has permissions to append to the file and whether appended data will fit in the physical length of the file. If the permission exists and the data will fit, `Append` issues a special WRITE() to the SPUs that manage the memory that is the target of the append (there may be more than one depending on the size and alignment of the update). While the writes are underway, APPEND() logs the updated length to persistent storage (for crash recovery), and sends a response to the application.

This response does not signal the completion of the APPEND(). Instead, it contains the number of WRITE()s that the coordinating SPU issued and the starting address of the append operation. The WRITE()s for the append notify the host-side application (rather than the coordinating SPU) when they are complete via an APPENDDONE() RPC. When the application has received all of the APPENDDONE() RPCs, it knows the APPEND() is complete. If any of the writes fail, the application needs to re-issue the write using `Direct-IO`.

If the append data will not fit in the physical length of the file, `Append` sends an "insufficient space" response to the host-side application. The host-side application then invokes the file system to allocate physical space for the file and notify the SPU.

If the file system needs to access the file's length, it retrieves it from the SSD and updates its in-memory data structures.

Figure 10 compares the performance of file appends using `Append` and using `Base-IO`. For `Base-IO` we open the file with O_DSYNC, which provides the same durability guarantees as `Append`. The appends are 1 kB. We modify Ext4 to pre-allocate 64 MB of physical extents. `Append` improves append latency by 2.5× and bandwidth by between 4× and 5.7× with multiple threads.

5 Related Work

Many projects (and some commercial products) have integrated compute capabilities into storage devices, but most of them focus on offloading bulk computation to an active hard drive or (more recently) an SSD.

In the 1970s and 1980s, many advocates of specialized database machines pressed for custom hardware, including processor-per-track or processor-per-head hard disks to achieve processing at storage device. None of these approaches turned out to be successful due to high design complexity and manufacturing cost.

Several systems, including CASSM [42], RAP [33], and RARES [24] provided a processor for each disk track. However, the extra logic required to enable processing ability on each track limited storage density, drove up costs and prevented processor-per-track from finding wide use.

Processor-per-head techniques followed, with the goal of reducing costs by associating processing logic with each read/write head of a moving head hard disk. The Ohio State Data Base Computer (DBC)[18] and SURE [22] each took this approach. These systems demonstrated good performance for simple search tasks, but could not handle more complex computation such as joins or aggregation.

Two different projects, each named Active Disks, continued the trend toward fewer processors, providing just one CPU per disk. The first [37] focused on multimedia, database, and other scan-based operations, and their analysis mainly addressed performance considerations. The second [1] provided a more complete system architecture but supported only stream-based computations called disklets.

Several systems [39, 11] targeted (or have been applied to) databases with programmable in-storage processing resources and some integrated FPGAs [28, 29]. IDisk [19] focused on decision support databases and considered several different software organizations, ranging from running a full-fledged database on each disk to just executing data-intensive kernels (e.g., scans and joins). Willow resembles the more general-purpose programming models for IDisks.

Recently researchers have extended these ideas to SSDs [13, 20], and several groups have proposed offloading bulk computation to SSDs. The work in [17] implements Map-Reduce [12]-style computations in an SSD, and two groups [6, 43] have proposed offloading data analysis for HPC applications to the SSD's processor. Samsung is shipping an SSD with a key-value interface.

Projects that place general computation power into other hardware components, such as programmable NICs, have also been proposed [15, 45, 25]. These devices allow for application-specific code to be placed within the NIC in order to offload network-related computation. This in turn reduces the load of the host OS and CPU in a similar manner to Willow.

Most of these projects focus on bulk computation, and we see that as a reasonable use case for Willow as well, although it would require a faster processor. However, Willow goes beyond bulk processing to include modifying the semantics of the device and allowing programmers to implement complex, control-intensive operations in the SSD itself. Some programmable NICs have taken this approach. Many projects [10, 32, 35, 5, 38, 16, 47, 8, 9] have shown that moving these operations to the SSD is valuable, and making the SSD programmable will open up many new opportunities for performance improvement for both application and operating system code.

6 Discussion

Willow's goal is to expose programmability as a first-class feature of the SSD interface and to make it easier to add new, application-specific functionality to a storage device. Our six example SSD Apps demonstrate that Willow is flexible enough to implement a wide range of SSD Apps, and our experience programming Willow demonstrates that building, debugging, and refining SSD Apps is relatively easy.

`Atomic-Writes` serves as a useful case study in this regard. During its development we noticed that our Willow-aware version of ShoreMT was issuing transactions that comprised several small updates in quick succession. The overhead for sending these LOGWRITE() RPCs was hurting performance. To reduce this overhead, we implemented a new RPC, VECTORLOGWRITE(), that sent multiple IO requests to Willow in a single RPC. Adding this new operation to match ShoreMT's needs took only a couple of days.

Several aspects of Willow's design proved especially helpful. Providing a uniform, generic, and simple programming interface for both HREs and SPUs made Willow easier to use and implement. The RPC mechanism is generic and familiar enough to let us implement most applications in an intuitive way. The simplicity meant that SPU-OS could be both compact and efficient, critical advantages in the Willow SSD's performance- and memory-constrained environment.

SSD Apps' composability was also useful. First, reusing code allowed Willow to make more efficient use of the the available instruction memory. Second, it made developing SSD Apps easier. For instance, most of our SSD App relied on `Direct-IO` to manage basic file access and permissions. Even better, doing so frees the developer from needing to write a custom kernel module and convincing the system administrator to install it.

Willow has the flexibility to implement a wide range of SSD Apps, and the architecture of the Willow SSD provides scalable capacity and supports a great deal of parallelism. However, some trade-offs made in the design present challenges for SSD App developers. We discuss several of these below.

First, striping memory across SPUs provides scalable memory bandwidth, but it also makes it more difficult to implement RPCs that need to make changes across multiple memory banks. The `Append` would have been much simpler if the coordinating SPU had been able to directly access all the file's data.

Second, the instruction memory available at each SPU limits the complexity of SSD Apps, the number of SSD Apps that can execute simultaneously, and the number of permission entries that can reside in the Willow SSD at once. While moving to a custom silicon-based (rather than FPGA-based) controller would help, these resource restrictions would likely remain stringent.

Third, the bandwidth of Willow SSD's ring-based interconnect is much lower than the aggregate bandwidth of the memory banks at the SPUs. This is not a problem for applications that make large transfers mostly between the host and the SSD, since the ring bandwidth is higher than the PCIe link bandwidth. However, it would limit the performance of applications that require large, simultaneous transfers between SPUs.

7 Conclusion

Solid state storage technologies offer dramatic increases in flexibility compared to conventional disk-based storage, and the interface that we use to communicate with storage needs to be equally flexible. Willow offers programmers the ability to implement customized SSD features to support particular applications. The programming interface is simple and general enough to enable a wide range of SSD Apps that can improve performance on a wide range of applications.

Acknowledgements

We would like to thank the reviewers, and especially Ed Nightingale, our shepherd, for their helpful suggestions. We also owe a debt of gratitude to Isabella Furth for her excellent copyediting skills. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also supported in part by NSF award 1219125.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[2] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[3] Bcache. http://bcache.evilpiepirate.org/.

[4] http://www.beecube.com/platform.html.

[5] M. S. Bhaskaran, J. Xu, and S. Swanson. BankShot: Caching slow storage in fast non-volatile memory. In *First Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '13, 2013.

[6] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[8] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, March 2012. ACM.

[9] S. Chu. Memcachedb. http://memcachedb.org/.

[10] J. Coburn, T. Bunker, M. Shwarz, R. K. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation solid-state drives. In *Proceedings of the 24th International Symposium on Operating Systems Principles (SOSP)*, 2013.

[11] G. P. Copeland, Jr., G. J. Lipovski, and S. Y. Su. The architecture of CASSM: A cellular system for non-numeric processing. In *Proceedings of the First Annual Symposium on Computer Architecture*, ISCA '73, pages 121–128, New York, NY, USA, 1973. ACM.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[13] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIG-*

MOD International Conference on Management of Data, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[14] Flexible I/O Tester. http://freecode.com/projects/fio.

[15] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: A safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, EW 8, pages 7–12, New York, NY, USA, 1998. ACM.

[16] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, Sept. 2010.

[17] Y. Kang, Y. Kee, E. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12, 2013.

[18] K. Kannan. The design of a mass memory for a database computer. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 44–51, New York, NY, USA, 1978. ACM.

[19] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, Sept. 1998.

[20] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee. Fast, energy efficient scan inside flash memory SSDs. In *Proceedings of ADMS 2011*, 2011.

[21] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[22] H.-O. Leilich, G. Stiege, and H. C. Zeidler. A search processor for data base management systems. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 280–287. VLDB Endowment, 1978.

[23] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[24] C. S. Lin, D. C. P. Smith, and J. M. Smith. The design of a rotating associative memory for relational database applications. *ACM Trans. Database Syst.*, 1(1):53–65, Mar. 1976.

[25] A. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable NIC. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 67–74, 2002.

[26] http://memcached.org/.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[28] R. Mueller and J. Teubner. FPGA: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[29] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[30] NVMHCI Work Group. NVM Express. http://nvmexpress.org.

[31] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB.

[32] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[33] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik. Performance evaluation of a relational associative processor. *ACM Trans. Database Syst.*, 2(2):175–195, June 1977.

[34] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkeley, CA, USA, 1st edition, 2010.

[35] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[36] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[37] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[38] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[39] S. Schuster, H. B. Nguyen, E. Ozkarahan, and K. Smith. RAP: An associative processor for databases and its applications. *Computers, IEEE Transactions on*, C-28(6):446–458, 1979.

[40] Shore-MT. http://research.cs.wisc.edu/shore-mt/.

[41] M. Srinivasan. FlashCache: A Write Back Block Cache for Linux. https://github.com/facebook/flashcache.

[42] S. Y. W. Su and G. J. Lipovski. CASSM: A cellular system for very large data bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 456–472, New York, NY, USA, 1975. ACM.

[43] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. Reducing data movement costs using energy efficient, active computation on SSD. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower '12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[44] TPC-B. http://www.tpc.org/tpcb/.

[45] P. Willmann, H. Kim, S. Rixner, and V. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 96–107, Feb 2005.

[46] XDD version 6.5. http://www.ioperformance.com/.

[47] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.