2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing

Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego

***Abstract*—In high performance computing systems, object deserialization can become a surprisingly important bottleneck—in our test, a set of general-purpose, highly parallelized applications spends 64% of total execution time deserializing data into objects.**

This paper presents the Morpheus model, which allows applications to move such computations to a storage device. We use this model to deserialize data into application objects inside storage devices, rather than in the host CPU. Using the Morpheus model for object deserialization avoids unnecessary system overheads, frees up scarce CPU and main memory resources for compute-intensive workloads, saves I/O bandwidth, and reduces power consumption. In heterogeneous, co-processor-equipped systems, Morpheus allows application objects to be sent directly from a storage device to a co-processor (e.g., a GPU) by peer-to-peer transfer, further improving application performance as well as reducing the CPU and main memory utilizations.

This paper implements Morpheus-SSD, an SSD supporting the Morpheus model. Morpheus-SSD improves the performance of object deserialization by 1.66×, reduces power consumption by 7%, uses 42% less energy, and speeds up the total execution time by 1.32×. By using NVMe-P2P that realizes peer-to-peer communication between Morpheus-SSD and a GPU, Morpheus-SSD can speed up the total execution time by 1.39× in a heterogeneous computing platform.

I. Introduction

In the past decade, advances in storage technologies and parallel/heterogeneous architectures have significantly improved the access bandwidth of storage devices, reduced the I/O time for accessing files, and shrunk execution times in computation kernels. However, as input data sizes have grown, the process of deserializing application objects — of creating application data structures from files — has become a worsening bottleneck in applications. For a set of benchmark applications using text-based data interchange formats, deserialization accounts for 64% of execution time.

In conventional computation models, the application relies on the CPU to handle the task of deserializing file contents into objects. This approach requires the application to first load raw data into the host main memory buffer from the storage device. Then, the host CPU parses and transforms the file data to objects in other main memory locations for the rest of computation in the application.

In a modern machine setup, this CPU-centric approach becomes inefficient for several reasons: (1) The code for object deserialization can perform poorly on modern CPUs and suffer considerable overhead in the host system. (2) This model intensifies the bandwidth demand of both the I/O interconnect and the CPU-memory bus. (3) It leads to additional system overheads in a multiprogrammed environment. (4) It prevents applications from using emerging system optimizations, such as PCIe peer-to-peer (P2P) communication between a solid state drive (SSD) and a Graphics Processing Unit (GPU), in heterogeneous computing platforms.

This paper presents *Morpheus*, a model that makes the computing facilities inside storage devices available to applications. In contrast to the conventional computation model in which the host computer can only fetch raw file data from the storage device, the Morpheus model can perform operations such as deserialization on file data in the storage device without burdening the host CPU. Therefore, the storage device supporting the Morpheus model can transform the same file into different kinds of data structures according to the demand of applications.

The Morpheus model is especially effective for creating application objects in modern computing platforms, both parallel and heterogeneous, as this model brings several benefits to the computer system: (1) The Morpheus model uses the simpler and more energy-efficient processors found inside storage devices, which frees up scarce CPU resources that can either do more useful work or be left idle to save energy. (2) In multiprogrammed environments, the Morpheus model offloads object deserialization to storage devices, reducing the host operating system overhead. (3) It consumes less bandwidth than the conventional model, as the storage device delivers only those objects that are useful to host applications. This model eliminates superfluous memory accesses between host processors and the memory hierarchy. (4) It allows applications to utilize new architectural optimization. For example, the SSD can directly send application objects to other peripherals (e.g. NICs, FPGAs and GPUs) in the system, bypassing CPU and the main memory.

To support the Morpheus model, we enrich the semantics of storage devices used to access data so that the application can describe the desired computation to perform. We design and implement Morpheus-SSD, a Morpheus-compliant SSD that understands these extended semantics on a commercially available SSD. We utilize the processors inside the SSD controller to perform the desired computation, for example, transforming files into application objects. We extend the NVMe standard [1] to allow the SSD to interact with the host application using the new semantics. As the Morpheus model enables the opportunity of streaming application objects directly from the SSD to the GPU, we also implement NVMe-P2P (an extension of Donard [2]) that provides peer-to-peer data exchange between Morpheus-SSD and the GPU.

The Morpheus programming model is simple. Programmers write code in C or C++ to perform computations such as deserialization in the storage device. The Morpheus compiler generates binaries for both the host computer and the storage device and inserts code to allow these two types of binaries to interact with each other.

Our initial implementation of Morpheus-SSD improves object deserialization from text files by 66%, leading to a 32% improvement in overall application performance. Because Morpheus-SSD does not rely on CPUs to convert data into objects, Morpheus-SSD reduces the CPU load, eliminates 97% of context switches, and saves 7% of total system power or 42% of energy consumption during object deserialization. With Morpheus-SSD, applications can enjoy the benefit of P2P data transfer between an SSD and a GPU: this increases application performance gain to 39% in a heterogeneous computing platform. The performance gain of using Morpheus-SSD is more significant in slower servers—Morpheus-SSD can speed up applications by 2.19×.

Although this paper only demonstrates using the Morpheus-SSD model for deserialization objects from text files, we can apply this model to other input formats (e.g. binary inputs) as well as other kinds of interactions between memory objects and file data (e.g. serialization or emitting key-value pairs from flash-based key-value store [3]).

This paper makes the following contributions: (1) It identifies object deserialization as a useful application for computing inside modern, high-performance storage devices. (2) It presents the Morpheus model, which provides a flexible general-purpose programming interface for object deserialization in storage. (3) It demonstrates that in-storage processing model, like the Morpheus model, enables new opportunities of architectural optimizations (e.g. PCIe P2P communications) for applications in heterogeneous computing platforms. (4) It describes and evaluates Morpheus-SSD, a prototype implementation of the Morpheus model, made using commercially available components.

The rest of this paper is organized as follows: Section II describes the current object deserialization model and the corresponding performance issues. Section III provides an overview of the Morpheus execution model. Section IV introduces the architecture of Morpheus-SSD. Section V depicts the programming model of Morpheus. Section VI describes our experimental platform. Section VII presents our results. Section VIII provides a summary of related work to put this project in context, and Section IX concludes the paper.

II. Deserializing application objects

A typical computer system contains one or more processors with DRAM-based main memory and stores the bulk data as files in storage devices. Because of SSDs' high-speed, low-power, and shock-resistance features, the system may use flash-based SSDs as the file storage. The system may also include GPU accelerators that contain thousands of Arithmetic Logic Units (ALUs) to provide vectorized parallelism. These SSDs and GPUs connect with the host computer system through the Peripheral Component Interconnect Express (PCIe) [4] interconnect. These SSDs communicate using standards including Serial Advanced Technology Attachment (Serial ATA or SATA) [5] or NVM Express (NVMe) [6] [1]. These standards allow the host computer to issue requests to the storage device, including reads, writes, erases, and some administrative operations.

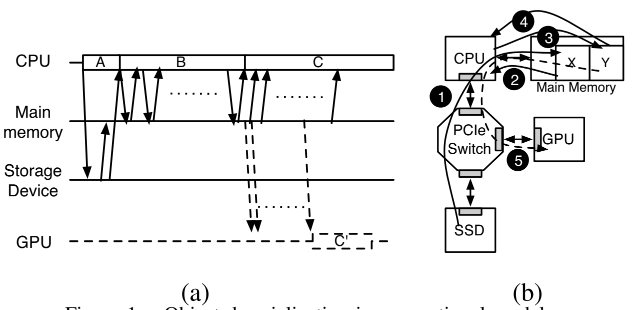

Figure 1. Object deserialization in conventional models

To exchange, transmit, or store data, some applications use data interchange formats that serialize memory objects into ASCII or Unicode text encoding (e.g. XML, CSV, JSON, TXT, YAML). Using these text-based encodings for data interchange brings several benefits. For one, it allows machines with different architectures (e.g. little endian vs. big endian) to exchange data with each other. These file formats allow applications to create their own data objects that better fit the computation kernels without requiring knowledge of the memory layout of other applications. These text-based formats also allow users to easily manage (e.g. compare and search) the files without using specific tools. However, using data from these data interchange formats also results in the overhead of deserializing application objects, since it requires the computer to convert data strings into machine binary representations before generating in-memory data structures from these binary representations.

Figure 1 illustrates the conventional model (Figure 1(a)) and the corresponding data movements (Figure 1(b)) for deserializing application objects. The application begins object deserialization by requesting the input data from the storage device as in phase A of Figure 1(a). After the storage device finishes reading the content of the input file, the storage device sends the raw data to the main memory through the I/O interconnect, resulting in the data transfer from the SSD to the main memory buffer X as in arrow (1) of Figure 1(b). In application phase B of Figure 1(a), the application loads the input data from the main memory (data transfer (2) in Figure 1(b)), converts the strings into binary representations, and stores the resulting object back to another location in the main memory (data transfer (3) from the CPU to memory location Y in Figure 1(b)). The program may repeat phases A and B several times before retrieving all necessary data.

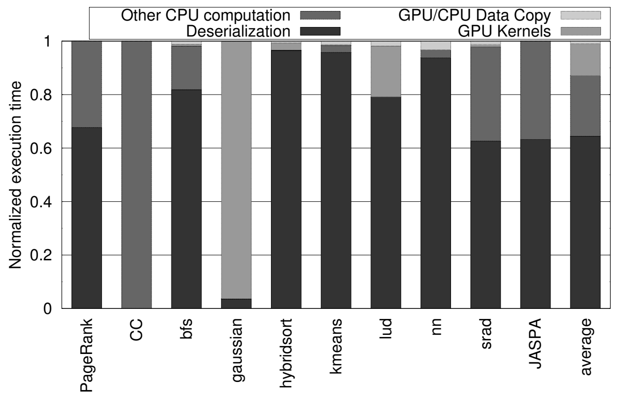

Figure 2. The overhead of object deserialization in selected applications

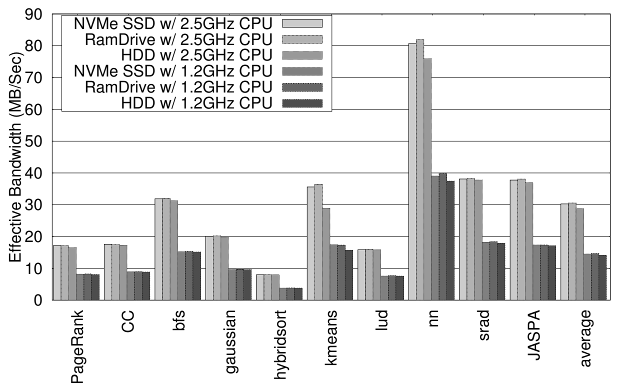

Figure 3. The effective bandwidth of object deserialization in selected applications using different storage devices under different CPU frequencies

Finally, the CPU or the graphics accelerator on an APU (Accelerated Processing Unit (APU) [7] can start executing the computation kernel which consumes the application objects in the main memory as in phase C (data transfer (4) loading data from memory location Y in Figure 1(b)). If the application executes the computation kernel on a discrete GPU (use phase C' instead of C), the GPU needs to load all objects from the main memory before the GPU kernel can begin (hidden lines of Figure 1).

Because of the growing size of input data and intensive optimizations in computation kernels, deserializing application objects has become a significant performance overhead. Figure 2 breaks down the execution time of a set of applications that use text-based encoding as their input data. (Section VI provides more details about these applications and the testbed.) With a high-speed NVMe SSD, these applications still spend 64% of their execution time deserializing objects. To figure out the potential for improving the conventional object deserialization model, we perform detailed experiments in these benchmark applications and obtain the following observations:

Object deserialization is CPU-bound. To examine the correlation between object deserialization and the speed of storage devices, we execute the same set of applications and load the input data from a RAM drive, an NVMe SSD, or a hard drive. The NVMe SSD can sustain more than 2 GB/sec bandwidth and the hard drive provides 158 MB/sec bandwidth. We create a 16GB RAM drive using the host system DRAM attached to a DDR3 memory bus that theoretically can offer up to 12.8 GB/sec bandwidth.

Figure 3 reports the effective bandwidth of object deserialization for each I/O thread in these applications. In this paper, we define the effective bandwidth as the memory size of application objects that the system can produce within a second. Compared to the traditional magnetic hard drive, the NVMe SSD delivers 5% higher effective bandwidth when using a 2.5 GHz Xeon processor. However, the performance of the RAM drive is essentially no better than the NVMe SSD. If we under-clock the CPU to 1.2 GHz, we observed significant performance degradation. However, the performance differences among different storage devices remain marginal.

This result indicates that CPU code is the bottleneck when systems with high-speed storage devices deserialize objects from files. In other words, if we use the conventional model for object deserialization in these applications, the application does not take advantage of higher-speed storage devices, even those as fast as a RAM drive.

Executing object deserialization on the CPU is expensive. To figure out the source of the inefficiency, we profile the code used to parse a file of ASCII-encoded strings into an array of integers. The profiling result shows that the CPU spent only 15% of its time executing the code of converting strings to integers. The CPU spent the most of the rest of its time handling file system operations including locking files and providing POSIX guarantees.

The conventional approach to object deserialization also results in increased context switch overhead. In the conventional approach, applications must access the storage device and the memory many times when deserializing objects. As a result, the system must frequently perform context switches between kernel spaces and different processes since fetching data from the storage device or accessing memory data can lead to system calls or latency operations. For example, if a memory access in phase B of Figure 1(a) misses in the last-level cache or if the memory buffer in phase B needs to fetch file content, the system may switch to the kernel space to handle the request from the application.

Object deserialization performs poorly on CPUs. To further investigate the potential of optimizing the object deserialization code, we implemented a function that maintains the same interface as the original primitive but bypasses these overheads. Eliminating these overheads speeds up file parsing by 1.74 $\times$. However, the instruction-per-cycle (IPC) of the remaining code, which examines the byte arrays storing the input strings and accumulates the resulting values, is only 1.2. This demonstrates that decoding ASCII strings does not make wise use of the rich instruction-level parallelism inside a CPU core.

Deserializing objects using CPUs wastes bandwidth and

memory. The conventional object deserialization model also creates unnecessary traffic on the CPU-memory bus and intensifies main memory pressure in a multiprogrammed environment.

Deserializing objects from data inputs requires memory access to the input data buffer and the resulting object locations as shown in phases A and B of Figure 1(a) and steps (2)–(3) in Figure 1(b). When the computation kernel begins (phase C of Figure 1(a)), the CPU or a heterogeneous computing unit (e.g. the GPU unit of an APU) needs to access deserialized objects. This results in memory accesses to these objects again. Since the computation kernel does not work directly on the input data strings, storing the raw data in the memory creates additional overhead in a multiprogrammed environment. For APU-based heterogeneous computing platforms in which heterogeneous types of processors share the same processor-memory bus, these memory accesses can lead to contentions and degrade performance of bandwidth hungry GPU applications [8] [9].

Since text-based encoding usually requires more bytes than binary representation, the conventional model may consume larger bandwidth to transport the same amount of information compared to binary-encoded objects. For example, the number "12345678" requires at least 8 bytes in ASCII code, but only needs 4 bytes using 32-bit binary representation.

Conventional object deserialization prevents applications from using emerging optimizations in heterogeneous computing platforms. Many systems support P2P communication between two endpoints in the same PCIe interconnect, bypassing the CPU and the main memory overhead [4][10][11][12][13][14][15][16][17][18][19][20][21][22][2]. the conventional model restricts the usage of this emerging technique in applications. Applications that still rely on the CPU-centric object deserialization model to generate vectors for GPU kernels must go through every step in Figure 1(b), but cannot directly send data from the SSD to the GPU. This model also creates huge traffic in the system interconnect and CPU-memory bus as it exchanges application objects between heterogeneous computing resources.

In the rest of the paper, we will present and evaluate the Morpheus model that addresses the above drawbacks of deserializing objects in the conventional computation model.

III. The MORPHEUS model

In contrast to the conventional model, in which the CPU program retrieves raw file data from the storage device to create application objects, the Morpheus model can transform files into application objects and send the result directly to the host computer from the storage device and minimizes the intervention from the CPU.

Figure 4(a) depicts the Morpheus execution model. To begin deserializing objects for the host application, the application provides the location of the source file to the Morpheus-compliant storage device (in this paper, the Morpheus-SSD) and invokes a *StorageApp*, a user-defined program that the underlying storage device can execute. After the storage device fetches the raw file from its storage medium, the storage device does not send it to the host main memory as it would in the conventional model. Instead, the storage device executes the StorageApp to create binary objects as the host application requests. Finally, the storage device sends these binary objects to the host main memory (or the device memory of heterogeneous computing resources (e.g. the GPU)), and the computation kernel running on the CPU or GPU (phase C or phase C') can use these objects without further processing.

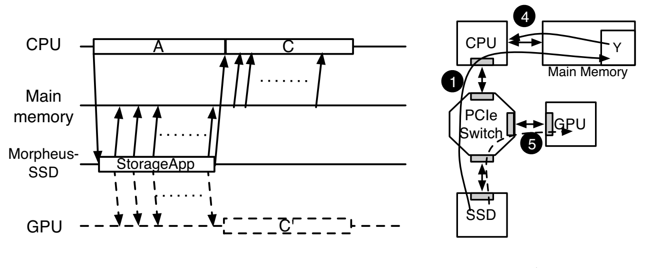

Figure 4. Object deserialization in using the Morpheus model

Figure 4(b) illustrates the data movements using the Morpheus model. Because the system leverages the storage device for object deserialization, the storage device only needs to deliver the resulting application objects to the system main memory as in step (1), eliminating the memory buffer (location X in Figure 1(b)) and avoiding the CPU–memory round-trips of steps (2)–(3) in Figure 1(b).

By removing the CPU from object deserialization, this model achieves several benefits for the system and the application:

Improving power, reducing energy, allowing more efficient use of system resources: The Morpheus model allows applications to use more energy-efficient embedded processors inside the storage device, rather than more power-hungry high-end processors. This reduces the energy consumption required for object deserialization. The Morpheus model improves resource utilization in the CPU by eliminating the low-IPC object deserialization code and allowing the CPU to devote its resources to other, higher-IPC processes and applications. In addition, if the system does not have more meaningful workloads, the host processor can operate in low-power model to further reduce power.

Bypassing system overhead: The Morpheus model executes StorageApp directly inside the storage device. Therefore, StorageApp is not affected by the system overheads of running applications on the host CPU, including locking, buffer management, and file system operations. In addition, due to the low-ILP nature of object deserialization, even with the embedded cores inside a modern SSD, the Morpheus model can still deliver compelling performance.

Mitigating system overheads in multiprogrammed en-

vironments: In addition to freeing up CPU resources for other workloads, this model can also mitigate context switch overheads since it does not require the CPU to perform any operation until the end of a StorageApp. By eliminating the memory buffers for raw input data (memory location X in Figure 1(b)), the Morpheus model also relieves pressure on main memory and reduces the number of page faults in multiprogrammed environments.

Reducing traffic in system interconnects: Since the Morpheus model removes the host-side code of object deserialization, this model also eliminates the memory operations that load raw data and store the converted objects to the main memory in the conventional model (phase B in Figure 1(a) and steps (2)–(3) in Figure 1(b)). As a result, the Morpheus model can reduce the traffic on the CPU–memory bus. This is especially helpful for APU-based systems, in which heterogeneous computing resources compete for space and bandwidth for the same main memory.

In the interconnect that moves data among peripherals, the Morpheus-compliant storage device can send application objects to the rest of the system that are more condensed than text strings. Therefore, the Morpheus model also reduces the amount of data transferred over the I/O interconnect. If the final destination of objects is the main memory, we can further reduce the size of data going through the memory bus.

Enabling more efficient P2P communication for heterogeneous computing applications: The PCIe interconnect allows P2P communication between two peripherals; SSDs can support this mechanism through re-engineering the system as NVMMU [21], GPUDrive [22], Donard [2], or (in this paper) NVMe-P2P. However, if the application needs to generate application objects using the CPU and stores these objects in the main memory, supporting P2P communication between the SSD and the GPU will not help application performance.

With the Morpheus model, the storage device can generate application objects using the StorageApp. It can then directly send these objects (as in Step (5) of Figure 4(b)), bypassing the CPU and the main memory overhead. Therefore, the Morpheus model allows more opportunities for applications to reduce traffic in system interconnects, as well as to reduce CPU loads, main memory usage, and energy consumption. This is not possible in the conventional model as the application cannot bypass the CPU.

IV. The system architecture

As the Morpheus model changes the role of storage devices in applications—deserializing objects in addition to conventional data access operations—we need to re-engineer the hardware and system software stacks. On the hardware side, the storage device needs to support new semantics that allow its own processing capability to be tapped for deserializing application objects. The system software also needs to interface with the application and the hardware to efficiently utilize the Morpheus model.

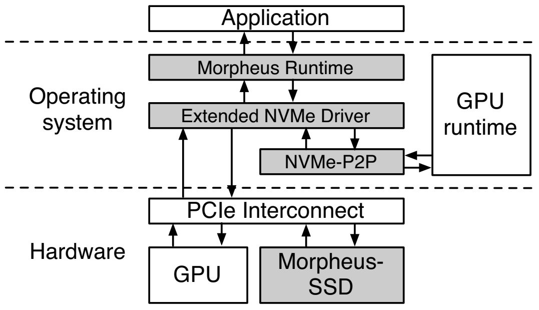

Figure 5. The system architecture of Morpheus-SSD

Figure 5 depicts the system architecture that supports the Morpheus model. We implement Morpheus-SSD, an SSD that provides new NVMe commands to execute StorageApps, by extending a commercially available NVMe SSD. For software to utilize Morpheus-SSD, we extend the NVMe driver and develop a runtime system that provides an interface for applications. With the runtime system translating application demands into NVMe commands, the extended NVMe driver can set up the host system resource and communicate the Morpheus-SSD through the PCIe interconnect. As the Morpheus model eliminates the host CPU from object deserialization, this model also allows for a more efficient data transfer mechanism. To demonstrate this benefit, the system also implements NVMe-P2P, allowing Morpheus-SSD to directly exchange data with a GPU without going through the host CPU and the main memory in a heterogeneous computing platform.

In the following paragraphs, we will describe the set of NVMe extensions, the design of Morpheus-SSD and the implementation of NVMe-P2P in detail.

A. NVMe extensions

NVMe is a standard that defines the interaction between a storage device and the host computer. NVMe provides better support for contemporary high-speed storage using non-volatile memory than conventional standards designed for mechanical storage devices (e.g. SATA). NVMe contains a set of I/O commands to access the data and admin commands to manage I/O requests. NVMe encodes commands into 64-byte packets and uses one byte inside the command packet to store the opcode. The latest NVMe standard defines only 14 admin commands and 11 I/O commands, allowing Morpheus-SSD to add new commands in this one-byte opcode space.

To support the Morpheus model, we define four new NVMe commands. These new commands enrich the semantics of an NVMe SSD by allowing the host application to execute a StorageApp and transfer the application object to/from an NVMe storage device. These new commands are:

MInit: This command initializes the execution of a StorageApp in a Morpheus-SSD. The MInit command contains a pointer, the length of the StorageApp code, and a

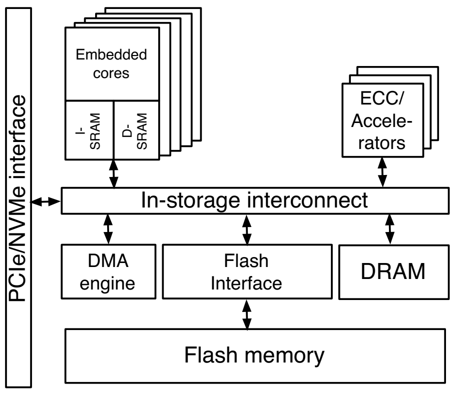

Figure 6. The architecture of an NVMe SSD

list of arguments from the host application. This command also carries an *instance ID* that allows the Morpheus-SSD to differentiate StorageApp requests from different host system threads.

MREAD: This command acts like a conventional NVMe read operation, except that it reads data from the SSD to the embedded core and then uses a StorageApp (selected according to the instance ID specified by the command packet) to process that data and send it back to the host.

MWRITE: This command is similar to the MREAD, but works for writing data to the SSD, again using a StorageApp (selected according the instance ID specified by the command packet) to process the writing data and send it back to the host.

MDEINIT: This command completes the execution of a StorageApp. Upon receiving this command, the Morpheus-SSD releases SSD memory of the corresponding StorageApp instance. The StorageApp can use the completion message to send a return value to the host application.

These commands follow the same packet format as conventional NVMe commands: each command uses 40 bytes for the header and 24 bytes for the payload.

B. The design of Morpheus-SSD

We build the Morpheus-SSD, a Morpheus-compliant SSD, by modifying a commercially available NVMe SSD. This section describes our extension to the NVMe SSD and the firmware operations.

Figure 6 demonstrates the hardware architecture of the baseline NVMe SSD. This SSD contains an NVMe/PCIe interface that transmits/receives packets through the PCIe I/O interconnect. The NVMe/PCIe forwards these command packets to the corresponding units in the SSD to complete the requests. The SSD also includes several GB of DRAM for data buffering and uses a DMA engine to transfer data between the SSD and other device memory through the NVMe/PCIe interface.

Because of the write-once, bulk-erase nature of flash memory chips, the SSD executes firmware programs using the embedded cores based on ARM [23] or MIPS architectures [24] and the DRAM to maintain the flash translation layer (FTL) that maps logical block addresses to physical chip addresses. The firmware programs that maintain the FTL also periodically reclaim free space for later data updates. Each embedded core contains instruction SRAM (I-SRAM) and data SRAM (D-SRAM) to provide instruction and data storage for the running program. The SSD uses the flash interface to interact with NAND flash chip arrays when accessing data.

To handle these new NVMe commands and execute the user-defined StorageApps, we re-engineer the NVMe/PCIe interface and firmware programs running on the embedded cores inside the Morpheus-SSD. Because these new NVMe commands share the same format with existing NVMe commands, the modifications to the NVMe interface are minor. We only need to let the NVMe interface recognize the opcode and instance ID of these new commands and deliver the command to the corresponding embedded cores. The current Morpheus-SSD implementation delivers all packets with the same instance ID to the same core.

The process of executing a StorageApp on the designated embedded core goes as follows. After receiving a MINIT command, the firmware program first ensures that the StorageApp code resides in the I-SRAM. The running StorageApp works on the D-SRAM data and moves data in or out of D-SRAM upon the requests of subsequent MREAD or MWRITE commands. As with conventional read/write commands, the StorageApp uses the in-SSD DRAM to buffer the DMA data between Morpheus-SSD and other devices.

The Morpheus-SSD leverages the existing read/write process and the FTL of the baseline SSD to manage the flash storage, except when processing data after fetching data from the flash array or other devices. Morpheus-SSD also uses the fast locking mechanism from the SSD hardware support to ensure data integrity. These firmware programs maintain and support existing features for conventional NVMe commands without sacrificing performance or guarantees. Since the Morpheus model does not alter the content of storage data, Morpheus-SSD performs no changes to the FTL of the baseline SSD.

C. NVMe-P2P

To provide efficient data communication schemes between Morpheus-SSD and other devices, we develop NVMe-P2P and place it in the system stack. NVMe-P2P allows Morpheus-SSD to directly exchange application objects with GPUs or other devices, bypassing the host CPU and main memory overhead as well as eliminating redundant data copies and movements in the system. The current implementation of NVMe-P2P focuses on the link between Morpheus-SSD and the GPU.

The conventional approach of establishing P2P communication over PCIe interconnect requires two devices to each map their own device memory to the PCIe switch by programming the base address registers (BARs) that the PCIe switch reserves for each connecting peripheral. As the PCIe switch examines the destination addresses of each data packet from DMA requests, it can directly deliver the data packets to the desired devices without going through the system main memory. However, this approach does not work for NVMe SSDs since, as a block device, NVMe uses a doorbell model for PCIe communication and does not map device memory for data accesses.

NVMe-P2P overcomes this limitation using a similar approach as the NVMMU [21] and Donard [2]. We extend the Morpheus-SSD NVMe driver code for MREAD/MWRITE commands. If the driver receives MREAD/MWRITE requests that opt to use P2P PCIe communication, the driver invokes the NVMe-P2P module to map the device memory of the peripheral that Morpheus-SSD wants to exchange data with to PCIe BARs.

On the GPU side, NVMe-P2P follows the conventional PCIe P2P mechanism. NVMe-P2P leverages AMD's DirectGMA [10] and NVIDIA's GPUDirect [11] technologies to program the GPU device memory to the PCIe BAR. After successfully programming PCIe BARs, NVMe-P2P responds to the extended NVMe driver. The NVMe driver then generates MREAD/MWRITE commands; these commands resemble conventional MREAD/MWRITE requests except that they use GPU device memory instead of main memory as the DMA target. Our implementation extends project Donard [2] to generate these commands with GPU memory addresses as their DMA targets. Upon receiving these commands, the Morpheus-SSD can directly pull data from or push data to the device memory address through the PCIe switch.

As NVMe SSDs cannot make their own device memory available to other peripherals, NVMe SSDs do not allow other PCIe devices to access them directly. NVMe-P2P still relies on the host system software stack to issue MREAD/MWRITE commands and uses the SSD to actively fetch or modify data on other devices. Therefore, NVMe-P2P does not create any new file system integrity issues for the Morpheus model.

V. THE MORPHEUS PROGRAMMING MODEL

To compose an application using the Morpheus model, we provide a programming framework including language extensions, libraries, and the compiler, for programmers to create a program using C/C++ programming languages. This section will briefly introduce the Morpheus programming model and show how the compiler, the runtime system, and the driver interact with Morpheus-SSD.

A. Composing a Morpheus application

In the Morpheus programming model, the programmer can define a StorageApp using a function in a high-level programming language like C or C++. The host program invokes the StorageApp as calling a function in the source code and shares data with the Morpheus-compliant storage device using the virtual memory abstraction.

```
StorageApp int inputApplet
(ms_stream ssd_input_stream, void *edge_array)
{
    Edge ssd_edge_array[4096];
    int i = 0;
    while(ms_scanf(ssd_input_stream, "%d %d",
                  &ssd_edge_array[i%4096].first,
                  &ssd_edge_array[i%4096].second)==2)
    {
        i++;
        if(i % 4096 == 0)
        {
            ms_memcpy(edge_array, ssd_edge_array,
                                  sizeof(Edge)*4096);
            edge_array += sizeof(Edge)*4096;
        }
    }
    ms_memcpy(edge_array, ssd_edge_array,
                         sizeof(Edge)*(i%4096));
    return i;
}
```

(a)

```
void test_distributed_page_rank(char* graphfilename,
int num_ofVertex, int num_ofEdges, int iterations)
{
    FILE *fin;
    ms_stream ssd_input_stream;
    void **arg_list;
    fin = fopen(graphfilename, "r");
    ssd_input_stream = ms_stream_create(fin);
    Edge *edge_array = (Edge *)malloc(
                              sizeof(Edge)*num_ofEdges);
    inputApplet(ssd_input_stream, edge_array);
    ms_stream_destroy(ssd_input_stream);
    // The rest of code ...
}
```

(b)

Figure 7. An example of StorageApp

1) Creating an StorageApp: To define a StorageApp, the programmer attaches a keyword – `StorageApp` – in front of a C/C++ function prototype. The StorageApp can declare and use local variables residing in the RAM space of storage processors. The StorageApp also allows passing arguments from the host application. The StorageApp can locate and access the file content using a special data structure: `ms_stream`. The StorageApp can also process or move data to or from the storage device using functions in the Morpheus device library.

Figure 7(a) shows an example of the StorageApp that we created for the PageRank application. The StorageApp `inputApplet` scans the input data from the `ssd_input_stream`, converts the input data into integers, stores the results in the `ssd_edge_array` data structure using the Morpheus library function `ms_scanf`, and copies the results to the host memory using `ms_memcpy` upon reaching the end of the `ssd_edge_array`.

The limitations of the embedded core architecture in the Morpheus-SSD create some restrictions in our current programming framework. First, the programmer can only use the functions from the Morpheus library. The current implementation provides basic I/O parsing primitives including `ms_scanf` or `ms_printf`, which are similar to `scanf` and `printf` in the standard C library, to assist object serialization/deserialization in a StorageApp. The current library also contains primitives for obtaining file information from the `ms_stream` in a StorageApp. This limitation keeps the programmer from having to deal with low-level operations inside a storage device and maintains code portability when the underlying device changes. Second, the StorageApp cannot directly access the data in the host memory. The StorageApp can only receive data or send data to the host memory using the `ms_memcpy` in the Morpheus library. Third, due to the capacity of D-SRAM on the embedded core in Morpheus-SSD, the current implementation restricts the maximum working set size of a single StorageApp. If the data set size of the StorageApp exceeds the available embedded core D-SRAM capacity, the StorageApp needs to transfer part of the results to the destination and reuse the memory buffer in the StorageApp.

2) Invoking a StorageApp in an application: Invoking a StorageApp in the programming model resembles a function call, except that the application needs to prepare a `ms_stream` structure for file access and rely on the runtime system to handle the execution of StorageApp.

To feed a StorageApp with a file, the programming model requires the host application to create a `ms_stream` and pass this stream as an argument of the StorageApp. The Morpheus host-side library defines a `ms_stream_create` function that accepts a file descriptor as an argument and returns a `ms_stream` structure. The `ms_stream_create` function interacts with the underlying file system to get permission to access a file and information about the logical block addresses in file layouts. By using a `ms_stream` data structure, the Morpheus model leaves the file permission checks in the host operating system and avoids performing these complex operations on the SSD.

Figure 7(b) shows the host application that uses the StorageApp in Figure 7(a). To invoke this StorageApp in the `test_distributed_page_rank` function, we create an `ssd_input_stream` using the `ms_stream_create` function. Then, the host program prepares the argument for the StorageApp. With these modifications, the programmer can eliminate the CPU code that scans the input file and parses the data into `edge_array` since the `inputApplet` can offload this part to the SSD.

B. Code generation

A Morpheus application contains at least two kinds of binary executable: one is the application running on the host computer and the other is the StorageApp inside the storage device. The Morpheus programming model relies on the compiler to produce these two types of machine binaries and insert code that interacts with the runtime system. The runtime system translates application requests using extended Morpheus-SSD NVMe commands to allow communication between these two types of binaries.

The compiler acts as a regular C/C++ compiler of the host computer, except under the following two conditions: if the compiler reaches a call site of a StorageApp or if the compiler reaches the StorageApp code.

Unlike conventional function calls, a call to the Morpheus StorageApp requires the host application to initialize/deinitialize the StorageApp in the storage device and feed the StorageApp with data. Therefore, instead of calling the StorageApp function directly, the compiler inserts code for the runtime system. The runtime system employs the Morpheus-SSD driver to issue the MINIT command and install the StorageApp to the storage device. To distinguish the requesting thread in the host computer, the Morpheus-SSD runtime also generates a unique instance ID for each thread calling a StorageApp.

As a StorageApp may accept a pointer as the argument to store the application objects in the host main memory, the compiler analyzes the pointers passing to a StorageApp. If a pointer is used by a function in a device-side library that moves data between the storage device and a system memory address (e.g. `ms_memcpy`) , the compiler inserts runtime system calls that interact with the device driver to make these memory addresses available for the Morpheus-SSD to access through DMA. If the address points to a location in the GPU device memory, the runtime system and the driver can potentially use NVMe-P2P for more efficient data exchange.

Most Morpheus StorageApps consume a Morpheus stream (`ms_stream`) argument since most systems abstract data in storage devices as files. If the StorageApp consumes a stream, the compiled host program calls the runtime system to issue MREAD or MWRITE commands through the extended NVMe driver to transfer the file content into the StorageApp running on Morpheus-SSD. Because the NVMe standard limits the data length of each I/O request to 65536 blocks, the runtime system may break the request into multiple MREAD or MWRITE commands if the file contains more the 65536 blocks.

Finally, the compiler attaches code to the host program to send the MDEINIT command to complete the execution of a StorageApp and free up the resource in the Morpheus-SSD. When the Morpheus-SSD responds to the MDEINIT command, the host application can receive the return value from the complete StorageApp. After receiving the response from Morpheus-SSD, the compiler inserts code that allows the host application to work together with the Morpheus-SSD driver to make the content in the DMA addresses (and only these addresses) available for the host application before resuming the execution of the host application.

As the Morpheus model executes StorageApps using the embedded cores, the compiler optimizes and assembles the StorageApp code using the instruction set of the embedded core architecture. Because each MREAD and MWRITE command can carry a limited amount of raw data, the library functions consuming the Morpheus stream, including the

Application Name	Benchmark Suite	Parallel model	Input Size
PageRank	BigDataBench	MPI	3.6 GB
CC	BigDataBench	MPI	62 MB
Breadth-First Search (BFS)	Rodinia	CUDA	2.53 GB
Gaussian Elimination (Gaussian)	Rodinia	CUDA	1.56 GB
Hybrid Sort	Rodinia	CUDA	3.14 GB
Kmeans	Rodinia	CUDA	1.30 GB
LU Decomposition (LUD)	Rodinia	CUDA	2.42 GB
k-Nearest Neighbors (NN)	Rodinia	CUDA	1.64 GB
Sparse Matrix Multiplication (JASPA)	JASPA	N/A	11 MB

Table I
THE APPLICATIONS AND THE INPUT DATA SIZES THAT WE USED IN THIS PAPER.

`ms_scanf` and `ms_printf` functions, respond to the host computer with a completion message after finishing data access operations of each command. The Morpheus-SSD can execute the following commands to keep receiving chunks of the raw data or finish executing the StorageApp. When the StorageApp calls the `ms_memcpy` function, the library code will send a message to the Morpheus-SSD to employ the DMA engine and move StorageApp data between the embedded core memory and the host main memory.

VI. EXPERIMENTAL METHODOLOGY

We build a Morpheus-SSD that supports the Morpheus model and attach the Morpheus-SSD to a high-end heterogeneous computing server with a GPU. We evaluate the performance of the resulting system with several benchmark applications that require object deserialization from text-based file formats. This section describes our prototype system setup and the benchmark selection.

A. Experimental platform

The experimental platform uses a quad-core Intel Xeon processor based on Ivy Bridge EP architecture. The processor runs at a maximum clock rate of 2.5 GHz and dynamically adjusts the frequency between 1.2 GHz and 2.5 GHz. The system also includes an NVIDIA K20 GPU that contains 2496 CUDA cores and 5GB GDDR5 memory on board. This computer hosts a Linux system with kernel version 3.13. We extend the NVMe driver in this system to support the Morpheus-SSD NVMe commands. The system contains a PCIe 3.0 I/O hub that connects the processor and other peripherals including the GPU and the SSD in the system. We measure the total system power using a Watts Up meter. The idle power of the experimental platform is 105 watts.

We build a Morpheus-SSD using a commercial SSD with a Microsemi's controller that has multiple general-purpose embedded processor cores [25]. The PCIe interface allows the Morpheus-SSD to communicate through PCIe 3.0 interconnect using up to 4 lanes. We also equip the Morpheus-SSD with 2GB DDR3 DRAM to store the StorageApp data and FTL mappings. The Morpheus-SSD provides 512GB of data storage using flash memory chips.

To support the Morpheus model, this SSD runs our modified firmware programs. This firmware is also compatible with standard NVMe; since we did not modify the code that handles regular NVMe commands, the firmware achieves the same performance as an NVMe SSD with the same hardware configuration. Because the Tensilica LX cores that we are using do not contain FPUs, the current library implementation for Morpheus-SSD relies on software emulation to handle floating point operations. However, as the cost of manufacturing embedded processors drops and the increasing demand of in-SSD processing, we expect that the next generation of SSD processors will provide native support for floating point operations.

B. Benchmarks

To evaluate the performance of the Morpheus-SSD, we select 10 benchmark applications from the BigDataBench [26], JASPA [27], and Rodinia [9] benchmark suites.

We select these 10 applications using the following standard: (1) The application accepts text-based file formats as input data. (2) The application provides (or the benchmark contains) tools to generate large and meaningful inputs that mainly consist of integers. We set this criteria as the embedded processor does not have an FPU. For benchmark applications with mostly floating point numbers, we expect that the conventional model would perform better than our current implementation. (3) The application contains optimized computation kernels representing the common case application behavior in current high-performance computers. (4) The application provides source code in C/C++ programming language that is compatible with our current framework.

Table I lists some important characteristics of the benchmark applications. We use input data up to 3.6 GB in size. These applications may apply MPI or CUDA [28] to parallelize the computation kernels. In this paper, we consider the *baseline* as running the unmodified version of these applications on the server machine described in Section VI-A with a standard NVMe SSD using the same hardware configuration as the one described but without the support of the Morpheus model. Section II of this paper demonstrates the behaviors of our baseline. In summary, our benchmark applications spend 64% of execution time in object deserialization.

For each application, we compose StorageApps to replace the object deserialization code in the baseline. These StorageApps create exactly the same data structures that the computational aspects of these applications consume. Since we do not change the data structures and do not offload the computation kernel to the Morpheus-SSD, the modified version does not affect the computation time or the parallel model in the computation kernel.

The design of the Morpheus model and the Morpheus-SSD also support object serialization. However, as our

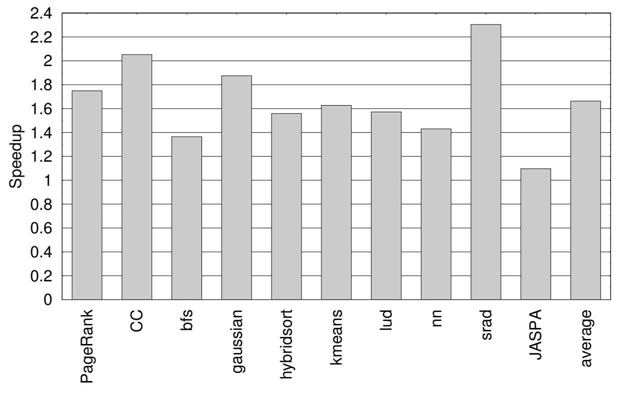

Figure 8. The speedup of object deserialization using Morpheus-SSD

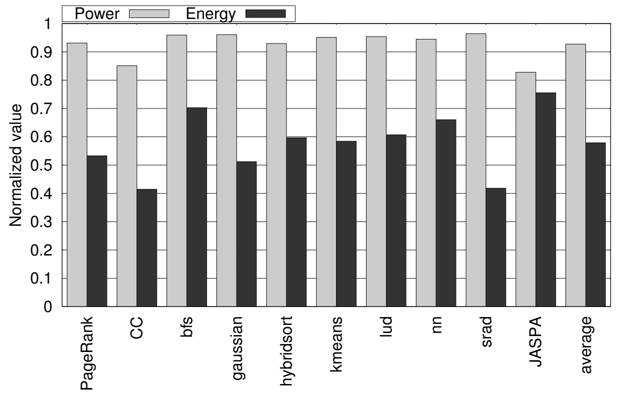

Figure 9. The normalized power and energy consumption during object deserialization

benchmark applications spend a relatively small amount of time or almost no time in serializing objects to the storage device, we did not evaluate the effect of using the Morpheus model for optimizing object serialization.

VII. Results

This section presents the performance of using the Morpheus model for object deserialization and discusses the impact of this model on application performance in a heterogeneous computing platform with a Morpheus-SSD.

A. Object deserialization performance

Figure 8 shows the speedup gained in object deserialization by using Morpheus-SSD. With less powerful embedded cores, Morpheus-SSD still achieves as much as 2.3× speedup and an average of 66% performance gain compared to the baseline.

For JASPA, we see only a 10% performance gain in object deserialization. This is because 33% of the strings in the input data represent floating point numbers, and the lack of FPUs increases the object deserialization overhead inside Morpheus-SSD.

Instead of using high-performance but more power-hungry host CPUs for object deserialization, Morpheus-SSD allows applications to use more energy-efficient embedded cores

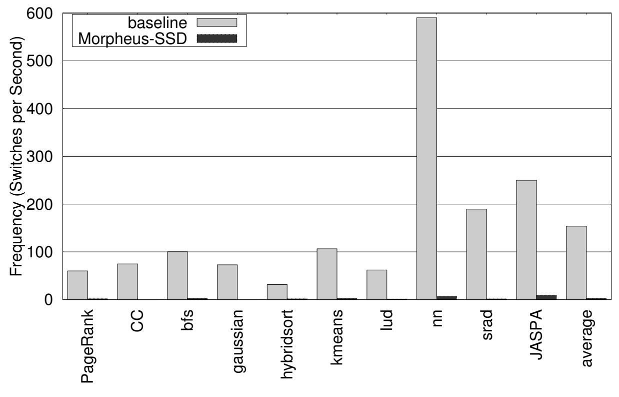

Figure 10. The context switch frequencies (number of context switches per second) during object deserialization

for the same purpose, thus saving power and reducing energy consumption. Figure 9 lists the normalized power and energy consumption of using Morpheus-SSD for object deserialization, compared to the baseline. Because it heavily relies on the host processor, the baseline increases the average power required from the idle system by 10.4 W. With Morpheus-SSD, the system uses the embedded cores to perform object deserialization, demanding only 1.8 W of total system power. Compared to the baseline system, Morpheus-SSD can reduce the power consumption of the total system for all applications by up to 17%, with an average of 7%. The effect of Morpheus-SSD on energy saving is more significant. Morpheus-SSD can reduce energy consumption by 42%, as Morpheus-SSD reduces both the amount of power required and execution time.

As StorageApps sends application objects instead of raw data strings to host applications, this model can potentially reduce both the traffic going outside Morpheus-SSD and between the CPU and the main memory. Compared to the conventional model, using the Morpheus model reduces the bandwidth demand of these applications by 22% on PCIe interconnect and the traffic on the CPU-memory bus by 58%.

The Morpheus model can also reduce context switches from system calls and long latency operations. Figure 10 lists the context switch frequencies of these benchmark applications. Across all applications, Morpheus-SSD can lower context switch frequencies by an average of 98%, and it can reduce the total number of context switches by an average of 97%.

B. Morpheus-SSD and NVMe-P2P

The Morpheus model enables GPU applications (e.g. applications from the Rodinia benchmark suite) to benefit from more efficient P2P data communication than the PCIe interconnect can provide. In this work, we implement NVMe-P2P to provide this support. Since NVMe-P2P does not affect the object serialization performance, but only reduces the data movement overhead in applications, we compare the end-to-end latencies for applications.

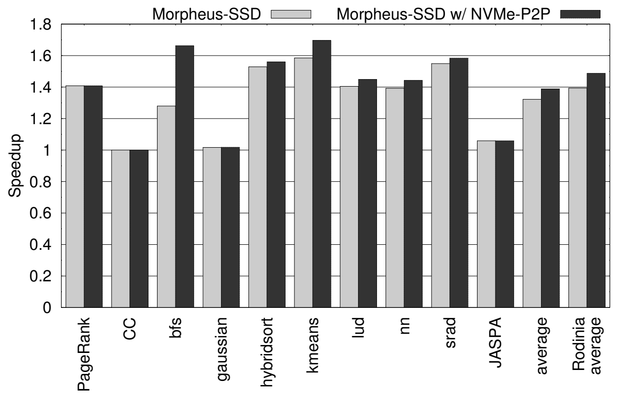

Figure 11. The overall application speedup using Morpheus-SSD and Morpheus-SSD w/ NVMe-P2P

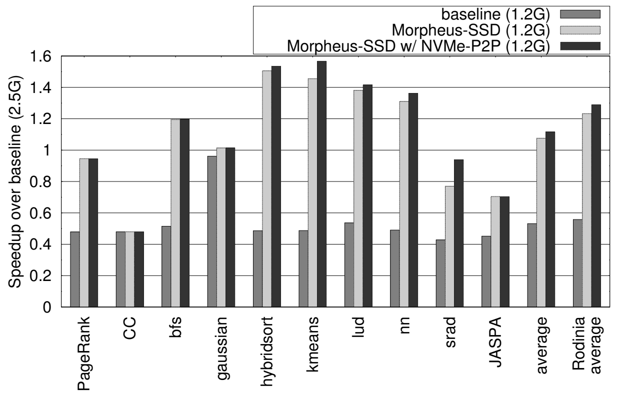

Figure 12. The overall application speedup using Morpheus-SSD and Morpheus-SSD w/P2P on a lower-clocked CPU

Figure 11 shows the overall application speedup gained by using Morpheus-SSD and NVMe-P2P. For GPU applications from the Rodinia benchmark suite, Morpheus-SSD can achieve an average speedup of 1.39×. With NVMe-P2P that can bypass the CPU and the main memory overhead, we can further achieve an average speedup of 1.49× — representing a 10% performance gain from P2P PCIe communication. If we include those CPU applications that cannot benefit from NVMe-P2P, we can see an average speedup of 1.39× when applying both Morpheus-SSD and NVMe-P2P. By using Morpheus-SSD alone, we can achieve only a 1.32× speedup.

C. Sensitivity to CPU performance

The Morpheus model does not rely on the CPU for the time-consuming object deserialization, making applications using this model potentially less sensitive to the performance of underlying CPUs. To investigate the impact of CPU performance on the Morpheus model, we clock the CPU to 1.2 GHz, 52% slower than the standard 2.5 GHz.

Figure 12 compares the end-to-end latencies of running applications using this under-clocked computing platform. With Morpheus-SSD, these applications still run 2% faster than the baseline at 2.5 GHz. If we enable NVMe-P2P, these applications gain 6% over the 2.5 GHz baseline. Compared with running baseline using 1.2 GHz processor, Morpheus-SSD speeds up applications by 2.10×. With NVMe-P2P, Morpheus-SSD further achieves 2.19× speedup.

By maintaining the same level of performance for applications, Morpheus-SSD makes servers with less powerful processors an attractive option as we can lower the power, energy, and machine costs. Baseline implementations that rely heavily on the CPU for object deserialization and moving data among heterogeneous computing units suffer a 42% degradation in performance with the slower clock speed. As a result, the energy-efficiency of the slower baseline system cannot compete with a high-end server or servers using the Morpheus-SSD.

VIII. Related work

The Morpheus model has its roots in early works that promote adding processors to disks [29][30][31][32][33][34]. However, due to the limitations of disk access latencies and processor technologies in the last century, these works did not provide enough performance gain to justify the increased cost.

With the advancement of storage technologies, recent works have re-examined this concept and shown promising results [35][36][37][38][39][40][41][42][43][44][45]. These works have mostly focused on trying to offload compute kernels that make it difficult for current SSD processors to deliver compelling performance, including data analytics, SQL queries, operating system operations, graph traversal, image processing, and MapReduce operations. Therefore, the Morpheus model applies in-storage processing to a completely different domain of applications that can maximize the potential of using this type of models.

The implementation of Morpheus-SSD uses general-purpose embedded processors and enables use of high-level programming languages to make it easy to customize the StorageApp. This approach is similar to IDisks, SmartSSD and Willow [34][37][40]. However, unlike SmartSSD (which uses the less efficient SATA interface) or Willow (which uses PCM as the storage medium), Morpheus-SSD uses the more flexible and efficient NVMe interface and adopts flash memory as the storage medium.

Although we currently implement the Morpheus model on an SSD, the Morpheus model can apply to any kind of device with input data and computing resources, including computational memories [46][47][48][49][50], NVRAM [51], ioMemory [52], or programmable network interface cards [53][54][55]. Section II demonstrates that object deserialization is inefficient even with DRAM as the data storage and that implementing the Morpheus model in these computational memories would improve performance.

The Morpheus model is complementary to existing programming language optimizations for object serialization/deserialization [56][57][58][59][60]. Programmers can implement these techniques using the processing power that our model exposes. As the Internet becomes the main medium for interchanging files, several works have also tried to improve the efficiencies of processing data interchange formats including XML and JSON [61][62]. To further improve the performance of exchanging objects between computers, several projects propose remote method invocation or adding runtime code [63][64][65]. The Morpheus model can support this optimizations and leverage NVMe-P2P to further reduce overhead.

Morpheus is fully compatible with existing file formats and requires only minor changes to the applications. ProtocolBuffers and Thrift propose new binary-based schema for data interchange; these change existing file formats and require the programmer to change both the application generating and the one receiving the data [66][67]. The Morpheus model can also provide more flexibility to create arbitrary types of objects for various applications in object-based storage [68][69][70].

NVMe-P2P implements P2P PCIe communications between Morpheus-SSDs and GPUs using the similar approach as in project Donard [2], NVMMU [21] and Gullfoss systems [71]. GPUDrive [22] also provides similar functionality, but it uses a customized PCIe switch to provide access to SATA SSDs. However, the set of GPU applications we examine in this paper cannot make use of these strategies without using the Morpheus model and Morpheus-SSD.

Before the emergence of the NVMMU and Gullfoss systems, existing works leveraging AMD's DirectGMA or NVIDIA's GPUDirect focus P2P communication between two GPUs or between the GPU and an Infiniband device [10][11] to improve inter-node communication within GPU clusters [12][13][14] or intra-node communication between GPUs or other devices [15][16][17][18][19][20].

The Morpheus model makes applications less sensitive to CPU performance in heterogeneous computing platforms. This model makes server systems with less powerful processors, including FAWN [72], Gordon [73] and Blade [74], appealing options for data centers.

IX. Conclusion

This paper presents the Morpheus model and the Morpheus-compliant SSD, which provide a framework for moving computation to the SSD. While this model is applicable to many domains, we evaluate this framework by using it to target a common, but under-represented bottleneck in computer architecture — object deserialization. In a conventional high-performance server using high-speed storage devices, we observed that a set of applications spent 64% of execution time in object deserialization.

The Morpheus model allows the programmer to offload inefficient object deserialization code to storage devices, where the source data reside. This model applies energy-efficient processors that are already present in emerging storage devices to generate application objects in storage devices. Deserializing application objects inside storage devices avoids host system overhead, reduces bandwidth, improves power consumption, frees up host processor resources, and enables peer-to-peer communications between the storage device and heterogeneous computing units.

We implement and evaluate Morpheus-SSD, an SSD that supports the Morpheus model, using a commercially available NVMe SSD. The evaluation shows that with current SSD technologies, offloading object deserialization to the SSD improves object deserialization performance by 1.66× and energy consumption by 42%, leading to overall application speedup by 1.32×. With NVMe-P2P removing the CPU and the main memory overhead for heterogeneous computing applications, Morpheus-SSD further achieves an average speedup of 1.39×. The Morpheus model is more effective in a lower-end server setup. Morpheus-SSD and NVMe-P2P can accelerate applications by 2.19×.

Acknowledgments

We would like to thank Stephen Bates of Microsemi for his support in developing the SSD firmware and his insights in implementing NVMe-P2P. This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA.

References

[1] Amber Huffman, "NVM Express Revision 1.1." http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf, 2012.

[2] Stephen Bates, "Project donard: Peer-to-peer communication with nvm express devices." http://blog.pmcs.com/project-donard-peer-to-peer-communication-with-nvm-express-devices-part-1/, 2014.

[3] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13, 2011.

[4] PCI-SIG, "PCI-Express Specification." https://www.pcisig.com/specifications/pciexpress/.

[5] The Serial ATA International Organization, "SATA-IO Releases SATA Revision 3.0 Specification." https://www.sata-io.org/sites/default/files/documents/SATA-Revision-3.0-Press-Release-FINAL-052609.pdf, 2009.

[6] , "NVM Express Explained." http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf, 2013.

[7] AMD, "Compute Cores White Paper." https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014.

[8] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 457–467, 2013.

[9] M. B. S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, pp. 44–54, Oct 2009.

[10] Advanced Micro Devices, Inc., "FirePro DirectGMA Technical Overview." http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firepro-directgma-sdk/, 2014.

[11] NVIDIA Corporation, "Developing a Linux Kernel Module Using RDMA for GPUDirect." http://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf, 2014.

[12] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[13] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Gpu peer-to-peer techniques applied to a cluster interconnect," in *IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pp. 806–815, 2013.

[14] J. Jenkins, J. Dinan, P. Balaji, N. Samatova, and R. Thakur, "Enabling fast, noncontiguous gpu data movement in hybrid mpi+gpu environments," in *2012 IEEE International Conference on Cluster Computing*, CLUSTER, pp. 468–476, Sept 2012.

[15] J. Stuart and J. Owens, "Multi-GPU MapReduce on GPU Clusters," in *2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS ' 11.

[16] S. Potluri, H. Wang, D. Bureddy, A. Singh, C. Rosales, and D. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, IPDPSW ' 12.

[17] L. Oden and H. Froning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing*, CLUSTER ' 13, 2013.

[18] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 2595–2605, Oct 2014.

[19] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA Communication Via PCI Express," *Cluster Computing*, vol. 17, pp. 339–348, June 2014.
[20] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pp. 170–178, 2013.
[21] J. Zhang, D. Donofrio, J. Shalf, M. Kandemir, and M. Jung, "Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures," in *The 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2015, 2015.
[22] M. Shihab, K. Taht, and M. Jung, "Gpudrive: Reconsidering storage accesses for gpu acceleration," in *Workshop on Architectures and Systems for Big Data*, 2014.
[23] ARM Ltd, "Storage must be fast, reliable and low power. ARM technology powers the worlds' leading HDDs and SSDs." http://www.arm.com/markets/embedded/hdd-ssd.php, 2016.
[24] tensilica Inc., " Xtensa microprocessor." http://www.tensilica.com, 2014.
[25] PMC-Sierra, "Flashtec NVMe Controllers." http://pmcs.com/products/storage/flashtec_nvme_controllers/, 2014.
[26] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 488–499, Feb 2014.
[27] Y. F. Hu, R. J. Allan, and K. C. F. Maguire, "Comparing the performance of JAVA with Fortran and C for numerical computing." http://yifanhu.net/SOFTWARE/JASPA/index.html.
[28] NVIDIA Corporation, "CUDA C Programming Guide v6.0." http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2014.
[29] S. Schuster, H. Nguyen, E. Ozkarahan, and K. Smith, "Rap: An associative processor for databases and its applications," *Computers, IEEE Transactions on*, vol. C-28, pp. 446–458, June 1979.
[30] 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.*, vol. 1, pp. 53–65, Mar. 1976.
[31] 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, pp. 280–287, VLDB Endowment, 1978.
[32] J. Banerjee, D. K. Hsiao, and K. Kannan, "Parallel architectures for database systems," ch. DBC&Mdash;a Database Computer for Very Large Databases, pp. 134–149, Piscataway, NJ, USA: IEEE Press, 1989.
[33] 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, (New York, NY, USA), pp. 81–91, ACM, 1998.
[34] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, vol. 27, pp. 42–52, Sept. 1998.
[35] 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*, pp. 1–12, April 2012.
[36] 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, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2012.
[37] 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 SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1221–1230, ACM, 2013.
[38] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, pp. 963–974, July 2014.
[39] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, 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, (Washington, DC, USA), pp. 385–395, IEEE Computer Society, 2010.
[40] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 67–80, USENIX Association, Oct. 2014.
[41] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 1–13, ACM, 2015.
[42] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, pp. 68–74, June 2001.
[43] Y. Kang, Y.-S. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *Mass Storage Systems and Technologies (MSST)*, 2013.
[44] 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, (New York, NY, USA), pp. 267–280, ACM, 2012.
[45] I. S. Choi and Y.-S. Kee, "Energy efficient scale-in clusters with in-storage processing for big-data analytics," in *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, (New York, NY, USA), pp. 265–273, ACM, 2015.
[46] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent ram (iram): chips that remember and compute," in *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*, pp. 224–225, Feb 1997.
[47] J. Torrellas, "Flexram: Toward an advanced intelligent memory system: A retrospective paper," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 3–4, Sept 2012.
[48] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the diva processing-in-memory chip," in *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pp. 14–25, 2002.
[49] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart memories: a modular reconfigurable architecture," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 161–171, 2000.
[50] P. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, vol. 1, pp. 77–84, 1994.
[51] PMC-Sierra, Inc., "Flashtec nvram drives." http://pmcs.com/products/storage/flashtec_nvram_drives/, 2015.
[52] Fusion-io, "iomemory virtual storage layer (vsl)." http://www.fusionio.com/overviews/vsl-technical-overview, 2015.
[53] 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, pp. 7–12, 1998.
[54] 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*, pp. 67–74, 2002.
[55] 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*, pp. 96–107, 2005.
[56] F. Breg and C. D. Polychronopoulos, "Java virtual machine support for object serialization," in *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pp. 173–180, 2001.
[57] M. D. Bond and K. S. McKinley, "Tolerating memory leaks," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, (New York, NY, USA), pp. 109–126, ACM, 2008.
[58] M. Dias, M. M. Peck, S. Ducasse, and G. Arévalo, "Clustered serialization with fuel," in *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, (New York, NY, USA), pp. 1:1–1:13, ACM, 2011.
[59] J. Ross and P. Chandran, "Object serialization support for object oriented java processors," in *Information Technology, 2008. ITSim 2008. International Symposium on*, vol. 3, pp. 1–6, Aug 2008.
[60] L. Opyrchal and A. Prakash, "Efficient object serialization in java," in *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*, pp. 96–101, 1999.
[61] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi, "Xml screamer: An integrated approach to high performance xml parsing, validation and deserialization," in *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, (New York, NY, USA), pp. 93–102, ACM, 2006.
[62] "json-smart: A small and very fast json parser/generator for java." https://code.google.com/p/json-smart/, 2011.
[63] N. Abu-Ghazaleh and M. Lewis, "Differential deserialization for optimized soap performance," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 21–21, Nov 2005.
[64] S. Kamin, L. Clausen, and A. Jarvis, "Jumbo: run-time code generation for java and its applications," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 48–56, March 2003.
[65] M. Gligoric, D. Marinov, and S. Kamin, "Codese: Fast deserialization via code generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 298–308, ACM, 2011.
[66] K. Varda, "Protocol buffers: Google's data interchange format," tech. rep., 2008.
[67] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," tech. rep., 2007.
[68] ANSI standard INCITS 400-2004, "Object-Based Storage Device Commands ," 2004.
[69] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pp. 2:1–2:17, 2008.
[70] Y. Xie, D. Feng, Y. Li, and D. D. Long, "Oasis: An active storage framework for object storage platform," *Future Generation Computer Systems*, 2015.
[71] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources," Tech. Rep. CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego technical report, 2015.
[72] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 1–14, 2009.
[73] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pp. 217–228, 2009.
[74] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pp. 315–326, 2008.