Rethinking Programming Frameworks for In-Storage Processing

Yu-Chia Liu
University of California, Riverside

Kuan-Chieh Hsu
University of California, Riverside

Hung-Wei Tseng
University of California, Riverside

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247919

***Abstract*—**
In-storage processing (ISP) is the most commercialized implementation of the near-data processing (NDP) model that executes tasks near their storage locations. However, programming ISP devices is complicated as it requires programmers to work closely with the underlying hardware, and even highly-optimized code can easily lead to suboptimal performance.

This paper introduces ActivePy. ActivePy makes the programmer completely agnostic to ISP hardware. ActivePy automatically, transpartently, and dynamically generates high-performance code to balance system-wide trade-offs and maximize the benefits of ISP. Our real system implementation shows that ActivePy can use ISP as efficiently as conventional C-based frameworks.

I. Introduction

The increasing popularity of in-storage processing (ISP) or computational storage solutions comes from two sources, the mismatching of hardware evolution and the rapid growth of application demands. With hardware accelerators offering orders-of-magnitude speedup in compute kernels, the relatively slow improvement in the interconnect bandwidth and non-volatile memory technologies have made data supply an emerging overhead. On the other hand, the datasets of applications have grown rapidly, increasing the demand for data storage.

ISP addresses both problems by leveraging or extending existing controllers near data storage. With more intelligent controllers pre-processing storage data or offloading other host computing resources' workloads, ISP reduces the amount of data volume going through the system interconnect or allows the system to use host computing resources more efficiently.

Unfortunately, programming existing ISP frameworks is challenging for the following reasons. First, identifying the most advantageous use of the ISP model or computational storage device (CSD) is not straightforward and relies heavily on programmers' knowledge of hardware and applications [2]. Second, composing a function on a CSD depends heavily on firmware programming [9], [15], [26], customized libraries [9], [13], [21], [25], specialized programming models [8], [20], low-level commands attached to storage objects [1], and hardware-description languages [5], [6], [11], [12], [27], and finally, even though programmers have optimized the application, the resulting program still lead to suboptimal performance if computational resources or data sources change as these frameworks have almost zero capability in dynamically adjusting/migrating workloads [2], [13].

This paper proposes a principled design of ISP programming frameworks to overcome the programming challenges. We argue that an ideal ISP programming framework should fulfill the following characteristics. First, the framework should decide on the most valuable CSD functions to release the burden on programmers. Second, the ISP device programming interface should be an integral part of the host programming interface and not require the programmer to compose the device function explicitly. In this way, the programming framework will have the flexibility to transparently decide the CSD function for the application and release the burden on programmers. Finally, the framework should allow flexibility in adjusting host/CSD workloads for system dynamics.

We designed and implemented *ActivePy* in response to the design principles. The programmer interacts with ActivePy using a high-level, interpreted, general-purpose programming language (e.g., Python in the current prototype), and the programmer is entirely agnostic to the presence of any CSD. ActivePy enhances the runtime interpreter to transparently, automatically, and dynamically compose functions for CSDs to perform through sampling and prediction. ActivePy compiles the resulting host application and the composed CSD functions into machine code to avoid the overhead of continuous runtime interpretation for better performance. ActivePy periodically monitors the program's execution and dynamically adjusts its assignments if necessary to provide performance guarantees.

In summary, this paper makes the following contributions:
(1) It is the first paper that explores the use of an interpreted language for programming CSDs in heterogeneous computing platforms to demonstrate the deficiency of conventional compiled programming languages on the same platform.
(2) It proposes a set of mechanisms to analyze and automatically generate programs on ISP platforms, making ActivePy the first ISP programming platform that does not rely on any programmer's annotation, pragma, or hint.
(3) It identifies and optimizes the performance bottlenecks of an interpreted-language runtime in heterogeneous computing.
(4) It evaluates ActivePy by building the complete system.

Our real system evaluation shows that ActivePy can allow Python programs without any ISP-related hints to achieve almost the same performance as equivalent, fully-optimized ISP programs written in C. The average speedup of the end-to-end latency from these applications is 1.33×, compared with equivalent baseline implementations written in C. ActivePy also makes the program less fragile to system dynamics than conventional programming frameworks.

II. Background and Motivation

With commercialized products [6], [7], [22] and working groups on standards [23], the ISP model that offloads computation to computational storage drives (CSDs) starts to gain ground in data-intensive computing platforms. This section describes the architecture of modern CSDs and the challenges of exploiting performance using existing system frameworks.

A. Computational storage devices (CSDs)

Figure 1 depicts the high-level architecture of a CSD in a modern heterogeneous computer. In modern heterogeneous computers, any peripheral device attached to the host computer must communicate with another device through a host

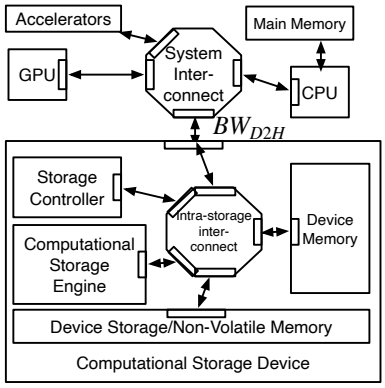

Fig. 1. The architecture of a computational storage device in a heterogeneous computer.

system interconnect (e.g., PCIe), except for the communication between CPU and main memory. Therefore, the system interconnect can become a bottleneck when applications must move large data volumes around different system components. For example, in modern computers using the PCIe 3.0 standard, the storage device can only share 4 GB/sec data bandwidth.

In addition to the storage controller, the host interconnect interface, the device memory (volatile), and the device storage (non-volatile) that a storage device typically contains, CSDs have a computational storage engine (CSE). The CSE communicates with the device memory/storage with relatively richer bandwidth of the exclusive intra-device interconnect, up to 16 GB/sec [18]. In commercialized products, the implementation of CSE can (1) leverage existing storage controllers [7], (2) add general-purpose processor cores, or (3) add an FPGA [6]. The processing power of the CSE allows the ISP model to improve program efficiency by (1) reducing the volume of data going through the relatively slow, narrow system interconnect, and (2) permitting tasks running on processors near data locations to receive data with richer bandwidth than the available external bandwidth going to the host [7]. Equation 1 quantifies the net profit (S) of performing a task (code region) in a program using a CSD [4], [24], instead of using the processors/accelerators from the host computer:

$$S = (\frac{DS_{raw}}{BW_{D2H}} + CT_{host}) - (CT_{device} + \frac{DS_{processed}}{BW_{D2H}}) > 0 \quad (1)$$

In this equation, DS_{raw} represents the raw input data size associated with executing the code on the host, CT_{host} represents the latency of executing the code using the host processor with all input data present in main memory, CT_{device} represents the latency of executing the equivalent code region on the CSD, $DS_{processed}$ represents the size of the intermediate data the device code produces, and BW_{D2H} represents the bandwidth between the device and the host. An ISP-assisted program is considered efficient if the program can make $S > 0$ in Equation 1.

B. Challenges

However, making $S > 0$ in Equation 1 is especially challenging in existing ISP platforms for the following reasons.

1) Limited CSE performance: Unlike GPU or ML/AI accelerators that deliver orders of magnitude speedup over the host CPU, the CSE in a modern CSD has limited processing power, typically slower than the host CPU. Previous studies [13], [25] have shown that the computation on the CSE is slower than the host CPU, and a majority of performance gain comes from reduced data volume. Therefore, a programmer must comprehensively understand application behavior and hardware characteristics to allocate application tasks to the most appropriate computational resources.

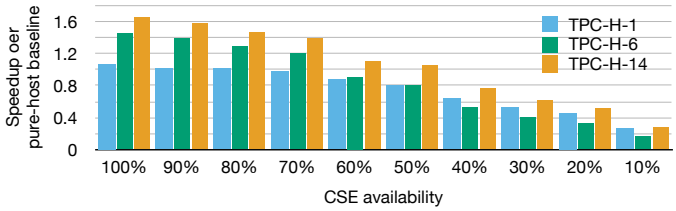

Fig. 2. The performance of the baseline task assignment compared with the "oracle" (optimized for the case where CSE are free) under different available hardware resources.

2) Ad hoc, difficult-to-use and error-prone programming frameworks: Programming of CSDs is further complicated by the programming models. Existing platforms rely on (1) firmware programming in C [9], [15], [26], (2) limited function through customized libraries [9], [13], [21], [25], (3) specialized programming models [8], [20], (4) low-level commands attached to storage objects [1], and (5) hardware-description languages (e.g., Verilog) [5], [6], [11], [12], [27]. As a result, the development of CSD functions are separated from the application and requires significant changes in the original application. The interface between different languages potentially incurs additional memory and data exchange overhead. The absence of hardware virtualization in these platforms makes CSD programming error-prone and non-portable.

3) Lack of flexibility during execution: Most importantly, even though the programmer carefully and exhaustively profiled and composed the application, there is still no guarantee of performance gain from using CSDs. The parameters in Equation 1 can change due to (1) resource contention coming from other applications, (2) resource contention coming from the storage management workloads (e.g., garbage collection), and (3) the change of input datasets itself. With the programmer-directed design of CSD functions in modern ISP platforms, there is no flexibility for existing ISP platforms to be adaptive to these system dynamics. Take Figure 2 as an example, we leverage three TPC-H workloads used for evaluating a representative CSD, Summarizer [13]. We performed the same code optimization/distribution, all written in C, on our baseline CSD described in Section IV-A. We change the available CSE time for programs running on the CSD to emulate the changes of computing resources. The code optimization from Summarizer is based on the case when the CSE is 100% available to the application. The x-axis in Figure 2 represents the portion of CSE available for the program, and the y-axis represents the speedup of the end-to-end latency on the optimized workload under different CSE availabilities. When the CSE fully dedicated its resources to tasks from the CSD-assisted program (i.e., 100% in Figure 2), these workloads are 1.25× faster than their baseline (i.e., the same workload but not using CSDs at all). However, the same optimizations to these workloads suffer from performance loss when the CSD has less than 60% computation time available.

III. ActivePy

Figure 3 illustrates the workflow of ActivePy. ActivePy takes a typical interpreted language (i.e., Python in the current

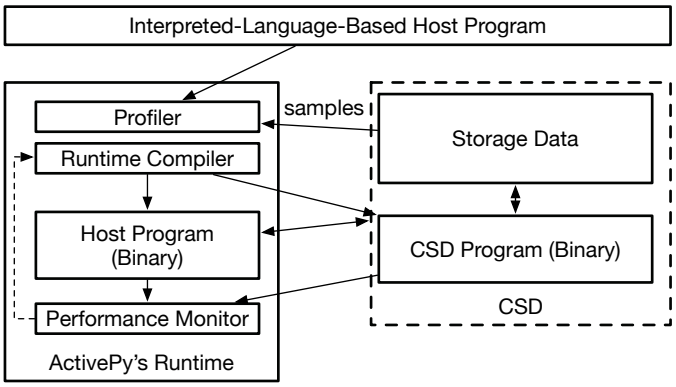

Fig. 3. The architecture of a computational storage device in a heterogeneous computer.

Algorithm 1: CSD code assignment

Input: P, a collection of lines (L_0, L_1, . . ., L_n) in the original program
Result: P_{host}, a collection of lines in the host program, and P_{csd}, a collection of lines in the CSD program

$T_{csd} = T_{host}$;
foreach *:* $L_i \in P$ **do**
  if $i == 0$ *or* $L_{i-1} \in P_{csd}$ **then**
    $T_{L_i \in P_{csd}} = T_{csd} - CT_{i,host} + CT_{i,device} - \frac{Din_i}{BW_{D2H}} + \frac{Dout_i}{BW_{D2H}}$;
  else
    $T_{L_i \in P_{csd}} = T_{csd} - CT_{i,host} + CT_{i,device} + \frac{Din_i}{BW_{D2H}} + \frac{Dout_i}{BW_{D2H}}$;
  end
  if $T_{L_i \in P_{csd}} < T_{csd} \leq T_{host}$ **then**
    $P_{csd} = P_{csd} \cup L_i$;
    $P_{host} = P_{host} - L_i$;
    $T_{csd} = T_{L_i \in P_{csd}}$;
  end
end

implementation) based program that does not contain any programmer's annotation or hint about the ISP model as the input. The host runtime system works with the CSD to form sample inputs from the referenced files of the running program to perform a sampling phase and collect statistics. The host runtime system then estimates the latency of each line of code when executing the code on the host processor and the CSD to develop an initial task-assignment plan. In ActivePy, we define a *task* as a program's dynamic instance of a code region.

Next, ActivePy's host runtime system separates the CSD tasks and automatically rewrites parts of the program code to enable the interaction between the CSD program and the host program. Finally, ActivePy's host runtime system will generate binary running on both the host and the CSD and start executing the program with the raw input. When the ActivePy program runs, the host runtime system continues monitoring task performance on each computing unit. If it leads to performance degradation, ActivePy's runtime system can reassign the tasks between the host and the CSD, repartition and regenerate the code, and migrate tasks between the host processor and CSDs.

We describe the key elements of ActivePy in the followings.

A. Sampling Phase

Finding the sweet spot of slicing code from the original program to maximize the benefits of using available CSDs requires ActivePy to determine the parameters in Equation 1 for each line of code. ActivePy achieves this goal by running a sampling phase that collects necessary statistics to guide the values when evaluating Equation 1.

The design of ActivePy's phase leverages two general observations. (1) A small subset of input data can capture properties of the original input [14]. (2) the latency and resulting data size are relational to the input data size. Therefore, the sampling phase starts by heuristically selecting data from raw inputs to create sample inputs of different sizes. The current implementation generates inputs using four scaling factors (F): (1) tiny, $2^{-10}\times$, (2) small, $2^{-9}\times$, (3) medium, $2^{-8}\times$, and (4) large, $2^{-7}\times$.

After generating sample inputs, ActivePy will start running the program using these sample inputs. When running each line of code, ActivePy will record the execution time, the input data size, and the output data size. If the code contains access to stored data, ActivePy will separate the data access time from the code execution time. This is because the data access time is typically linear to the data size but not necessarily true of the computation time. The sample inputs do not guarantee the program to generate meaningful computation results for the workload. However, since the purpose of the sampling phase is simply collecting information to estimate the benefit of ISP, this method is satisfactory for ActivePy. In this work, we implemented the sampling phase using line profiler [19].

During sample runs, ActivePy generates a set of estimated performance metrics for each line of code (L_i), including (1) $CT_{i,host}$, the estimated computation time on the host, (2) $CT_{i,device}$, the estimated computation time on the CSD, (3) Din_i, the estimated volume of data inputs, and (4) $Dout_i$, the estimated volume of data outputs. Since our sampling mechanism grows F exponentially, ActivePy can extrapolate the execution time and change to the raw data size for each line once four sample runs are complete. ActivePy predicts the execution time and data-size changes by selecting the closest fit from one of five curves—$O\left(1\right)$, $O\left(n\right)$, $O\left(n\ log\ n\right)$, $O\left(n^2\right)$, and $O\left(n^3\right)$. ActivePy then estimate the expected execution time on the target CSD by multiplying the predicted computation time on the host with a constant factor (C). ActivePy calculates C by either (1) querying the CSD's performance counters (e.g., retired instructions per cycle), or (2) running a small sample program on both a CSD and the host computer if performance counters are not available.

The aforementioned execution time estimation mechanism in ActivePy aims at a "good enough" rather than a highly accurate one for the following reasons. First, ActivePy uses the estimation for initial task allocation and can gradually adjust and optimize the decision later. Second, as system dynamics can change anytime, highly accurate estimation is not very useful in most scenarios. Finally, the overhead of applying highly accurate estimation mechanisms may not be able to outweigh the gain from a "good enough" but lightweight one.

B. Identifying CSD code regions

Algorithm 1 summarizes ActivePy's algorithm in identifying the CSD functions. The design of the algorithm uses one line of Python code as the basic unit of forming code regions, but not finer-grained in the translated code level as: (1) each line of Python code is a single-entry-single-exit code region that facilitates code generation and optimization due to the nature of line-by-line interpretation of the language, and more importantly, (2) as CSDs must communicate with the rest of the system through bandwidth-constrained, relatively-long-latency interconnect, the BW_{D2H} factor in Equation 1 is usually small and makes the data movement overhead

significant. Therefore, ISP models cannot take advantage of fine-grained task allocations that arbitrarily distribute tasks among different components since the cost of sending data back and forth through the interconnect is expensive.

Algorithm 1 initializes the set of code (i.e., CSD code) to execute on the CSD, P_{csd}, as an empty set and uses the total execution time for all code on the host, T_{host}, as the projected execution time on the CSD, T_{csd}. Using the projected execution time and data-size changes, ActivePy (1) examines every line to determine whether adding the line into the P_{csd} can reduce execution time T_{csd}, and (2) records the assignment that yields the shortest execution time, until the algorithm went through every line of code.

C. Code generation

After ActivePy identifies P_{csd}, the computation and data accesses in the CSD function, ActivePy will generate the machine code on each hardware and automatically instrument the program code for the following purposes: (1) memory allocations, (2) CSD function invocations, and (3) elimination of redundant memory operations.

a) Memory allocations: ActivePy adopts a shared memory address space between the host program and the CSD program. ActivePy's memory allocation policy prefers to place data near their consumers. This design reduces the software overhead and redundant memory copies by leveraging existing architectural supports. For CSDs attached to the host computer using PCIe interconnect, the CSD can expose its memory available for ActivePy's CSD functions by declaring them in the PCIe BARs (base address registers). ActivePy's kernel memory module can work with the OS's virtual memory subsystem to map these CSD locations into any ActivePy program's virtual memory address. The host program can later directly access these locations using load/store instructions without the demand for additional memory buffers and calls to I/O library functions. If the CSD attaches to the host computer through storage protocols over the network (e.g., NVMeoF), the CSD can leverage the RDMA hardware infrastructure NVMe already uses to support the storage function to map the device's internal memory into the host program's virtual address space. Similarly, the host program can use load/store instructions to directly access the CSD's device memory without additional memory and library overhead.

b) CSD function calls: As modern CSDs use fast, non-volatile memory, ActivePy's approach to making CSD function calls mimics the mechanism NVMe uses to communicate with devices for shorter latency [10]. Like the concept of queue-pairs in NVMe [10], [16], [17] and several CSD prototype systems [5], [8], ActivePy maintains a function call queue mapped from each CSD's memory location visible to the host. The CSD's CSE fetches a request from the call queue whenever the CSE is free.

To enable feedback and migration when running a CSD function, ActivePy also patches the status update code that reports the current execution status, typically the execution rate, through the completion/response queue. ActivePy patches status update code only at the end of each line of code at the interpreted language level, typically once every tens of machine instructions. The status update code also checks if the host computer has any request that CSD needs to handle with high priority. The overhead of status update code takes very little overhead in code execution time.

c) Elimination of redundant memory operations: ActivePy also patches the code to avoid Python's overhead of calling libraries written in heterogeneous programming languages. ActivePy does so by placing memory objects exchanged through different function calls directly into mutable memory, potentially a memory location on a CSD. By placing values in mutable memory objects, passing inputs when calling wrapper functions is similar to call-by-reference; the caller and callee share the same memory locations. If ActivePy can determine the target type of memory objects, the relevant library functions in ActivePy can produce memory objects in the target data type (e.g., NumPy) directly to the destination memory locations, further avoiding conversion overhead and bypassing the memory buffer.

d) Code distribution: ActivePy compiles the CSD function into the CSD's machine binary and the rest as the host machine binary for performance. ActivePy leverages the code generation functions in Cython [3] to produce high-performance machine code. ActivePy invokes Cython functions after ActivePy starts running the program and makes the decisions of task and data allocation. ActivePy distributes the CSD function using the mapped device locations. As mentioned before, CSDs supporting ActivePy make their device memory available to the host computer. Therefore, ActivePy can directly copy or emit the generated CSD binary into the target device memory location without additional commands or protocols.

D. Runtime monitoring and migration

ActivePy monitors the performance of code executing on CSDs and potentially adjusts the workload distribution for situations in which (1) the target device needs to handle high-priority tasks or (2) the device's projected performance does not match its real performance. For the first case, the CSD will signal ActivePy through the command pages to notify ActivePy to take corresponding actions immediately. ActivePy detects the second case by checking the throughput of the CSD code. ActivePy will use the measured IPC to re-estimate the time required for the remaining tasks on CSD if any of the following cases occur: (1) the rate of instruction throughput (i.e., instructions per cycle [IPC]) from the CSD code segment is decreasing, or (2) the IPC significantly below the estimated instruction throughput (i.e., the total amount of estimated instructions divided by estimated execution time on CSD). If the re-estimated execution time is longer than the total cost of migrating the remaining task to the host computer, including the host computation time and the data movement overhead, ActivePy will initiate task migration to the host.

Once the system decides to migrate the CSD task, ActivePy will break the execution of CSD code at the end of the currently executing line of Python code. With the help of a single memory abstraction for all computing units, migrating tasks among different computing units only requires ActivePy to save the local variables and the data in the shared memory space. ActivePy will regenerate the machine code for the new target-computing unit and resume execution of the offloaded code segment at the breakpoint of the Python code. The target-computing unit can resume execution of the originally offloaded code segment by using the regenerated machine mode and data/variables in the shared memory abstraction.

IV. Experimental methodology

This section describes both the hardware platform and the applications that we established to evaluate ActivePy.

Name	Data Size	Name	Data Size
blackscholes	9.1 GB	KMeans	5.3 GB
LightGBM	7.1 GB	MatrixMul	6.0 GB
MixedGEMM	9.4 GB	Pagerank	7.7 GB
TPC-H-1	6.9 GB	TPC-H-6	6.9 GB
TPC-H-14	7.1 GB		

TABLE I
THE APPLICATIONS, THEIR INPUT DATA SIZES AND DESCRIPTION ON THEIR SINGLE-ENTRY-SINGLE-EXIT CODE REGIONS.

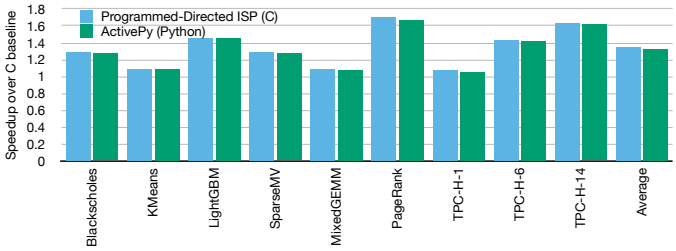

Fig. 4. The speedup of ActivePy compared to the speedup of a static, C-based ISP platform

A. Experimental platform

a) The Host Computer: The experimental platform we used has an octa-core AMD AMD Ryzen 7 3700X processor with a base clock rate of 3.6 GHz. We installed Ubuntu 16.04 (Linux kernel version 4.15), extended the language runtime, and added user-space drivers to support the ActivePy model. The motherboard contains a PCIe 3.0 I/O hub that connects the processor and other peripherals, including a Mellanox InfiniBand NIC that connects to the CSD. The machine also contains an NVIDIA RTX 2080 GPU.

b) The CSD: To evaluate the performance issues and our proposed design, we built a CSD that includes a system-on-chip (SoC) with 8 ARM Cortex-A72 processor cores and uses 2 TB flash memory arrays for data storage. The SoC and the device DRAM have access to the internal NAND flash-based arrays through an internal interconnect. We measured an effective peak bandwidth at 9GB/sec when accessing the internal NAND array. The CSD uses NVMe [17] to communicate with the host computer with up to 5GB/sec bandwidth, half of that to the internal device bandwidth. In addition, the CSD's hardware supports RDMA/InfiniBand, which can help map the CSD's internal memory locations to the host computer. The assembly of our CSD matches the specifications of commercialized products or prototypes [7], [13] and delivers similar performance gain as prior research prototypes [13].

B. Applications

We used nine Python workloads to evaluate the performance of ActivePy. Table I provides a summary of each application's input data size and code regions. We select these workloads as these applications have both Python and C implementations with optimized compute kernels to allow this work to compare the performance of optimized, programmer-directed versions and the Python version automatically optimized by ActivePy. These applications are also proved to beneficial through using CSDs with general-purpose processors [8], [9], [13], [25].

V. RESULTS

ActivePy's overall performance The evaluation result in Figure 4 shows that the automatic ActivePy without any programmer's hint achieves almost the same performance gain as optimal programmer-directed programming. Figure 4 shows ActivePy's performance for the total execution time of these applications when the CSD fully dedicated itself to the running application. We normalize the speedup to the baseline application (C-based, without ISP). ActivePy achieves almost the same performance gain as the programmer-directed ISP —1.34× vs. 1.33×—because ActivePy successfully identified *exactly* the same set of code regions for our CSD to perform as the optimal programmer-directed configuration. The execution time of the baseline workloads varies from 11 secs (TPC-H-6) to 73 sec (KMeans) on our machine. The small (1%) performance difference between these two configurations reflects the negligible overhead, typically 0.1 sec latency, of the sampling mechanisms and the code-generation phase.

The programmer-directed ISP is a collection of manually optimized C-based, ISP programs running on the same hardware platform using the system infrastructure with a CSD. In C-based implementations, programmers have to manually specify/compose ISP code regions. On the other hand, the proposed automatic ActivePy does not rely on *any* programmer's hint. To create an optimal programmer-directed code for each C application, we exhaustively tried to offload all reasonable combinations of single-entry-single-exit code regions in Table I to the CSD when the CSD entirely dedicated itself to the running program. We select the combination that delivers the shortest end-to-end latency as the optimal programmer-directed version for each application.

ActivePy's capability in identifying and composing CSD code ActivePy's mechanism usually makes very accurate predictions on data volume changes, the main factor affecting ISP program performance. As the data volume reduction after processing on our CSD is the main factor that leads to the performance gain, the accuracy in predicting volume changes compensates for the errors in estimating computation time. The only exception is the conversion to CSR format in PageRank and SparseMV – ActivePy can over-estimate the data volume that our CSD produced after generating CSR by up to 2.41×. The geometric mean of our error rate that discounts the outliers (e.g., CSR format) is only 9%. ActivePy does not predict accurately for CSR format because the sparsity is challenging to estimate with the limited number of samples we created in our algorithms. Our experiments on different input matrices show that ActivePy always over-estimates the data volume after generating CSR on our CSD, meaning that our algorithm under-estimates the potential of our CSD. Therefore, ActivePy at least makes no harm to performance if ActivePy conservatively schedules tasks on the host machine due to the under-estimated data reduction benefit from our CSD.

ActivePy's optimizations in its language runtime It is worth mentioning the performance of code optimizations of ActivePy in Section III-C0d. F Without any optimization, the baseline Python code without using CSDs is 41% slower than the C baseline due to the overhead of the original Python runtime. Using Python-to-C compiler (e.g., Cython) to generate code helps close the performance gap and shrink the slowdown to 20%. By eliminating unnecessary memory copies, the end-to-end latency of running the baseline Python program (ActivePy program without using ISP) makes almost no difference as the C baseline, excluding the 1% compilation overhead.

ActivePy with dynamic task migration Unlike traditional compiled-language-based platforms that cannot easily migrate tasks once assigned, ActivePy can reassign task locations and generate high-performance binaries on each computing unit with reasonably low overhead. To demonstrate ActivePy's task-migration capabilities, we created a version of ActivePy that cannot migrate tasks dynamically (ActivePy w/o migration) and compared its performance with the full-fledged

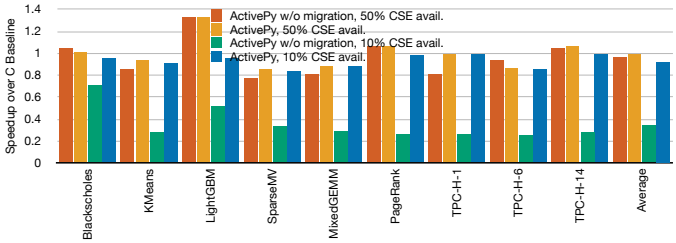

Fig. 5. The speedup of all workloads on ActivePy under 50% and 10% available CSP resources with and without task migration

ActivePy (ActivePy). We stressed the CSD processor by executing similar workloads right after each application's ISP tasks make 50% of their progress to simulate a situation where the CSD must load multiple tasks.

Figure 5 lists the cases when only 50% and 10% of the CSD computing resource is available to ISP workload. For both cases, full-fledged ActivePy outperforms ActivePy w/o migration. When the CSD has only 10% computing resources available for the assigned CSD tasks, ActivePy w/ task migration outperforms ActivePy w/o task migration by 2.82×. Relative to the no-CSD-assisted, baseline condition, ActivePy suffers 8% slowdown on average for the overhead of regenerating code on the host and accessing live data in CSD from the host computer. Without ActivePy's capability in migrating tasks dynamically, always using CSD can lead to an average of 67%, up to 88% performance loss when only 10% computing resource is available.

When the CSD has 50% computing resource, the case shows that ActivePy is still capable of balancing the trade-offs of migration or slower ISP tasks. ActivePy decides to migrate for Blackscholes, KMeans, SparseMV, MixedGEMM, TPC-H-1, and TPC-H-14. ActivePy's decisions outperform ActivePy w/o task migration in all cases except for Blackscholes.

VI. Conclusion

This paper presents ActivePy, a dynamic-language-based programming framework for ISP. ActivePy makes ISP more accessible to application designers by generating code that works on processors near data storage without programmer involvement. ActivePy can easily migrate tasks among different compute units to prevent performance degradation when workloads change. Through experiments conducted with the prototype ActivePy platform, this paper shows that intepreted languages with our proposed optimizations can be as competitive as compiled languages. ActivePy successfully identifies ISP use cases in a diverse range of applications, yielding a speedup of 1.33×. ActivePy's flexibility is underscored by its capacity to migrate ISP tasks among different units while minimizing the impact of inappropriate task assignments when workloads change.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. We would like to thank Fazil Osman and his team from Broadcom Inc. for their support in developing the Serpentry prototype. This work was sponsored by the two National Science Foundation (NSF) awards, CNS-1940048 and CNS-2007124. This work was also supported by new faculty start-up funds from North Carolina State University and University of California, Riverside.

References

[1] I. F. Adams, J. Keys, and M. P. Mesnier, "Respecting the block interface – computational storage using virtual objects," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[2] A. Barbalace and J. Do, "Computational storage: Where are we today?" in *CIDR*, 2021.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *IEEE Computing in Science & Engineering*, vol. 13, no. 2, 2011.

[4] 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*, 2012.

[5] 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*, ser. MICRO '43, 2010.

[6] 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*, ser. SIGMOD '13, 2013.

[7] J. Do, S. Sengupta, and S. Swanson, "Programmable Solid-State Storage in Future Cloud Datacenters," *Commun. ACM*, vol. 62, pp. 54–62, 2019.

[8] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," *SIGARCH Comput. Archit. News*, 2016.

[9] Y.-C. Hu, M. T. Lokhandwala, T. I, and H.-W. Tseng, "Dynamic Multi-Resolution Data Storage," in *52th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2019, 2019.

[10] D. B. M. Jisoo Yang and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, ser. FAST' 2012, 2012.

[11] 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*, ser. ISCA '15, 2015.

[12] 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.

[13] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, S. Swanson, H.-W. Tseng, and M. Annavaram, "Summarizer: Trading bandwidth with computing near storage," in *50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2017, 2017.

[14] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[15] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, , and M. Annavaram, "GraphSSD: Graph Semantics Aware SSD," in *46th International Symposium on Computer Architecture*, ser. ISCA 2019, 2019.

[16] NVM Express, Inc., "Nvm express explained," http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf, 2013.

[17] NVM Express, Inc., "NVM Express(R) Moves Into The Future," https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf, 2016.

[18] PMC-Sierra, "Flashtec NVMe Controllers," http://pmcs.com/products/storage/flashtec_nvme_controllers/, 2014.

[19] Robert Kern, "line_profiler and kernprof," https://github.com/rkern/line_profiler, 2017.

[20] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage computing system for emerging High-Performance drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[21] 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)*, 2014.

[22] SNIA, "What is computational storage?" 2020. [Online]. Available: https://www.snia.org/education/what-is-computational-storage

[23] SNIA Computational Storage Technical Working Group, "Computational Storage Architecture and Programming Model Version 0.8 Revision 1," https://www.snia.org/sites/default/files/technical-work/computational/draft/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09-DRAFT.pdf, 2021.

[24] 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*, ser. HotPower'12, 2012.

[25] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[26] Y. P. Yanqin Jin, Hung-Wei Tseng and S. Swanson, "Kaml: A flexible, high-performance key-value ssd," in *High Performance Computer Architecture (HPCA)*, 2017.

[27] J. Zhang and M. Jung, "Flashabacus: A self-governing flash-based accelerator for low-power systems," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. ACM, 2018, pp. 15:1–15:15.