TunIO: An AI-powered Framework for Optimizing HPC I/O

Neeraj Rajesh, Keith Bateman*, Jean Luca Bez†, Suren Byna‡,Anthony Kougkas, Xian-He Sun*
* Illinois Institute of Technology, † Lawrence Berkeley National Lab, ‡ Ohio State University & Lawrence Berkeley National Lab
*{nrajesh, kbateman}@hawk.iit.edu, †jlbez@lbl.gov, ‡byna.1@osu.edu, *{akougkas, sun}@iit.edu

***Abstract*—I/O operations are a known performance bottleneck of HPC applications. To achieve good performance, users often employ an iterative multistage tuning process to find an optimal I/O stack configuration. However, an I/O stack contains multiple layers, such as high-level I/O libraries, I/O middleware, and parallel file systems, and each layer has many parameters. These parameters and layers are entangled and influenced by each other. The tuning process is time-consuming and complex. In this work, we present TunIO, an AI-powered I/O tuning framework that implements several techniques to balance the tuning cost and performance gain, including tuning the high-impact parameters first. Furthermore, TunIO analyzes the application source code to extract its I/O kernel while retaining all statements necessary to perform I/O. It utilizes a smart selection of high-impact configuration parameters of the given tuning objective. Finally, it uses a novel *Reinforcement Learning* (RL)-driven early stopping mechanism to balance the cost and performance gain. Experimental results show that TunIO leads to a reduction of up to $\approx 73\%$ in tuning time while achieving the same performance gain when compared to H5Tuner. It achieves a significant performance gain/cost of 208.4 MBps/min (I/O bandwidth for each minute spent in tuning) over existing approaches under our testing.**

***Index Terms*—AI-powered I/O tuning, storage stack tuning, autotuning, source code transformations, and I/O performance optimization**

I. Introduction

Modern applications, often organized in coupled workflows, have grown complex in terms of software and hardware dependencies. Extracting maximum performance from the computing environment they run on is a complicated and costly activity [1]. Application developers often utilize an optimization process called tuning to determine the appropriate configurations for the application and its dependencies before they are run in production. Tuning is an iterative process that aims to determine which configuration to apply to an application or library to achieve a given objective. It has several essential steps, as follows: generate a set of initial configurations, evaluate the impact of the set of configurable parameters in terms of a given objective (e.g., maximize performance or minimize storage footprint), and reiterate the process until the optimal set of parameter values has been reached. The objective of tuning varies [2], [3], [4], but, most commonly, it is attempting to maximize application performance on the target environment in order to minimize the number of core hours spent by the user. Tuning is typically considered useful when the tuning benefit, or time gained running the application in production, supersedes the cost of tuning, or time spent tuning the application.

Scientific applications demonstrate many configurations and dependencies which modify their behavior and performance [5], [6], [7]. The application complexity is compounded by the intricacies of the HPC I/O stack, which is typically organized in a multi-layered design. High-level I/O libraries (e.g., HDF5 [8], PNetCDF [9], and ADIOS [10]) capture the data representation; middleware libraries (e.g., MPI-IO [11], Hermes [12], and UnifyFS [13]) map application I/O characteristics to the underlying I/O system; and the storage layer (e.g., Lustre [14], and BeeGFS [15]) interfaces directly with I/O hardware. Over time, there has been an explosion of I/O-related libraries that abstract the complexities of the underlying storage infrastructure. Each I/O library adds significantly to the configuration complexity of the I/O stack [16]. I/O is the major performance bottleneck of most modern scientific applications [17] and therefore represents the largest area for potential performance gain.

Existing tuners (e.g., Vizier [18], H5Tuner [19], [20], DEAP [21]) perform a search in a potentially large parameter configuration space to identify optimal parameter values, which results in a large tuning cost that may negate the benefits of tuning. For example, some applications may have low aggregated executions across their production run, which would not get adequately reimbursed unless the time spent tuning is small enough. Thus, it is imperative to balance the tuning cost and benefit gained. Three main challenges are associated with balancing the tuning budget while optimizing the tuning objective: (a) There is a need to prioritize tuning for I/O performance, which is the largest area of potential benefit for an application. As each run of the application is expensive, it is better to pick a single area of behavior to focus on when evaluating the application objective. (b) There is a need to prioritize the largest benefit parameters of the I/O stack. Since the configuration space experiences an exponential increase caused by the utilization of multiple I/O libraries, it is vital to minimize it by intelligently isolating high-impact parameters for the tuning objective. (c) There is a need for intelligent decision-making to determine whether to continue tuning so that unnecessary tuning is averted when the return on investment starts diminishing.

To address the above challenges in tuning, this paper introduces TunIO, an AI-powered optimization framework for tuning the HPC I/O stack. It consists of a set of optimizations that can be attached to any I/O tuning pipeline. TunIO builds upon the key insight that the best way to balance benefit and cost is to perform the most impactful tuning first, so that objective gains can be maximized when stopping tuning early to avoid diminishing tuning returns. One way to apply this logic is to reduce the part of the HPC stack being tuned to the category which provides the largest benefit. Therefore, TunIO prioritizes tuning I/O first and foremost. It has three components, each contributing to decreasing the tuning cost as follows:

- *Application I/O Discovery:* This component simplifies the application code for the purpose of objective evaluation so that only the critical sections with respect to I/O are run, resulting in a faster evaluation of the impact of tuning on the objective.
- *Smart Configuration Generation:* This component isolates high-impact parameters for the tuning objective using reinforcement learning, resulting in a smaller configuration space.

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00050

- *Early Stopping:* This component determines when to stop tuning using Reinforcement Learning, resulting in a balance between utilizing the tuning budget and optimizing the tuning objective.

In this paper, we present and evaluate the TunIO techniques integrated into H5Tuner. We discuss existing tuning techniques and motivation for TunIO's advancements in § II. We showcase TunIO's design, the reasoning for that design, and some limitations and implications in § III. We evaluate TunIO's design in § IV. We summarize related work and describe their distinctions from TunIO in § V. Finally, we present conclusions and future work in § VI.

II. Background

Tuning is an iterative, multistep process of finding optimal configuration values of a system by (1) generating multiple configurations, (2) evaluating, and (3) analyzing the impact of those configurations with respect to a well-defined objective, usually optimizing a measurable metric, such as maximizing performance, minimizing energy consumption, and/or minimizing latency. The process repeats until an adequate configuration is achieved or until the tuning budget has run out. The *tuning budget* is defined as the cost of tuning the application, expressed in units of resource allocation (e.g., CPU, storage) or time spent tuning (e.g., seconds, iterations).

Tuning is performed by either system admins deploying supercomputers or by users running applications on those supercomputers, depending on the system being optimized. Lower-level systems such as *Parallel File System*s (PFSs) or I/O forwarders can be configured by system admins, but user-level libraries and applications have to be configured by users. Tuning is time-consuming and complex to perform across many iterations. While system admins have weeks or months to optimize when deploying a system, the amount of time the users have to spend tuning is typically relatively small and will cost them allocation hours. In addition, users develop their applications in cycles, which leads to a variety of constraints. During the debug phase or when scaling up, the user needs to ensure the correctness of the application. During the final tuning for the target system, the user wants to optimize performance without exhausting their core hours. Users also have varying areas of expertise, and therefore some may lack knowledge of the exact parameters of the HPC stack which need to be configured.

A. I/O Stack Tuning

Historically, CPU performance has improved at a much faster rate than I/O performance [22]. Considering that I/O is the major bottleneck in most data-intensive applications, it follows that it represents an area of significant potential gains in performance [23]. The current HPC I/O stack is fairly complex and designed in a multi-layered manner, including the file system (e.g., XFS, F2FS, EXT2/3/4), low-level libraries (e.g., STDIO, POSIX I/O [24]), and high-level I/O libraries (e.g., Hermes [12], MPI-IO [25], HDF5 [8]) [26]. Tuning could optimize any of these layers, given the appropriate permissions. There is a myriad of configuration parameters across the entirety of the stack. For example, in HDF5 [27] one could configure the cache size or the alignment of I/O requests to the disk representation, and in Lustre [7] one could configure the stripe size or stripe count. These are highly dependent on the workload characteristics and are expected to result in subpar performance if misconfigured. Due to a lack of expertise, users tend to deploy I/O services with the default configuration parameter set [27]. However, these defaults are designed to support generic workloads and fail to take advantage of customization opportunities to improve performance. To make things worse, each of the layers in the I/O stack must be configured in coordination with the layers above or below [16]. This is necessary to avoid performance degradation, resource misutilization, and unintended behavior from the storage stack. For example, ideally, HDF5 chunks would correspond to Lustre stripes so that one chunk is written on one Lustre OSD, or HDF5 and MPI-IO would both either enable or disable collective I/O appropriately.

B. Configuration Space Optimizations

Tuning the I/O stack is ultimately a search problem, and therefore is bottlenecked by the number of parameters in the configuration space [28]. The search algorithms employed in user-level tuning have usually been AI techniques such as genetic algorithms [29], random search [30], hill climbing algorithms [31], and, more recently, reinforcement learning [32], [20].

The tuning search space has an expected space complexity of $O(m^n)$ [33], where m is the number of possible inputs to a parameter and n is the number of possible parameters. When performing a search on this space, the number of parameters n represents the most significant bottleneck to performance, even in optimized search algorithms [28].

Diverse HPC storage devices have the potential for high performance, but this has caused the software stack to become more complex with new libraries. This can be seen in Figure 1, which demonstrates the user-level parameter permutations of several HPC I/O libraries and storage systems. These are calculated utilizing a lower bound of two values for discrete parameters and five for continuous parameters. The baseline number of parameters was determined using existing information about HDF5 [5], PNetCDF [34], MPI [6], ADIOS [35], OpenSHMEMX [36], and Hermes [12]. These parameter counts represent lower bounds, as these libraries may contain additional configurable parameters. It can be seen from this table that the number of permutations of a full-stack tune would be very large. For example, a stack that includes HDF5 and MPI would have 3.81×10^{23} parameter value permutations. Multilayer tuning can cause an exponential increase in the search complexity in the worst case when all parameters are related. Even in the best case where each parameter is independent, there is still a linear increase in search complexity. Configuration space minimization is a practical and necessary technique for improving the average and best case performances of tuning by eliminating certain parameters from the search before exploring the parameter space [27]. However, configuration space minimization has the limitation that it may leave performance on the table and restrict the flexibility of the configuration, causing difficulty in adapting to changing conditions or requirements. Configuration space minimization has been accomplished via Bayesian Optimization [37], which determines the most valuable parameters from a static analysis and then groups the related parameters and tunes them together. Bez et al. [38] apply contextual bandits to determine which subsets of parameters need to be tuned to optimize the performance of the singular I/O forwarding layer.

C. Early Stopping

There can be instances when the tuning pipeline gets stuck in different local optima, which they are unable to escape within a reasonable amount of time [39], thus not justifying the benefit gained. In this case, quitting tuning and trying again would be best. Figure 2 demonstrates the I/O bandwidth achieved by HACC [40], FLASH [41], and VPIC [42] I/O kernels at different iterations when tuned using H5Tuner. It can be seen that application performance in tuning follows a logarithmic curve, where performance improvements

Layer	Name	Continuous Parameters	Discrete Parameters	Continuous Permutations	Discrete Permutations	User-Level Permutations
High-Level I/O	HDF5	17	10	7.63E+11	1024	7.81E+14
	ADIOS	57	21	6.94E+39	2097152	1.46E+46
	PNetCDF	5	1	3.13E+03	2	6.25E+03
Middleware	MPI	12	1	2.44E+08	2	4.88E+08
	OpenSHMEMX	11	1	4.88E+07	2	9.77E+07
	Hermes	20	0	9.54E+13	1	9.54E+13

Fig. 1: Explosion of user-level search space, discretizing continuous parameters to 5 values for simplicity.

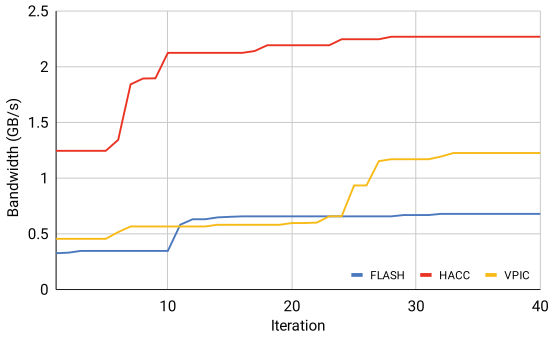


Fig. 2: Bandwidth (MB/s) across tuning iterations of HACC, FLASH, and VPIC I/O kernels at four node scale.

diminish over time. This observation suggests the need for early stopping [43], [18], a technique which involves prematurely ending a search after not observing a significant change in the results across a certain number of application runs.

III. TunIO

As an AI-powered I/O optimization framework, TunIO aims to address the challenges associated with the expensive costs of tuning HPC I/O stacks. Its main objective is to balance the cost and benefit of tuning an I/O stack. TunIO's set of optimizations is intended to be portable and used across any I/O tuning pipeline. TunIO conceptualizes optimizations to shorten the route to a fully tuned application by using Reinforcement Learning techniques to intelligently select a subset of parameters to be tuned, as well as to identify when to terminate the tuning pipeline without perceptible loss of optimization potential. TunIO is inspired by modern software stack compositions, where applications rely on a wide variety of I/O-related libraries running on top of modern hardware designs. TunIO targets I/O performance first, to mitigate known bottlenecks by capitalizing on improvements from a per-application tuned I/O stack. To achieve these goals, TunIO is designed and implemented with the following design principles in mind:

1) **I/O-focused Objective Evaluation**: Reduce the prohibitive expense of running the application via low-cost automatic isolation of the I/O kernel of the application.
2) **Impact-First Tuning**: Parameters that have a high impact should be tuned first. This allows for high-performant configurations to be discovered sooner in the tuning pipeline, effectively minimizing the configuration space.
3) **Constraint-based Tuning Cost and Benefit Balancing**: Searching should be tailored to the constraints of the user and their application and balance the impact of tuning with the cost of tuning. This will prevent the over-tuning of the application beyond the point where it stops being impactful.

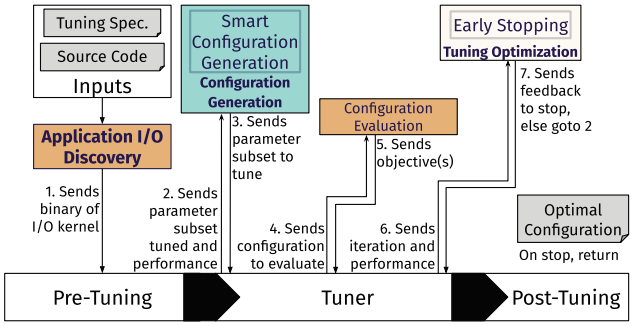


Fig. 3: Visualization of the tuning pipeline with TunIO.

A. High-Level Architecture

The design of TunIO can be seen in Figure 3. TunIO takes as inputs the tuning specification (including all user constraints) and source code. As a preprocessing step, the source code is passed through the Application I/O Discovery component (step 1), and an I/O kernel binary is generated. This kernel encapsulates all the I/O operations of the application and is passed to the tuner for evaluating the impact of tuning in a reduced manner. TunIO first uses the Smart Configuration Generation (§ III-C) component to decide on a set of configurations to evaluate (steps 2 and 3). This accomplishes the objective of Impact-First Tuning by intelligently selecting subsets of the total configuration space (demonstrated in the evaluation section - § IV-B). It passes the selected subset of parameters to be tuned to the Configuration Evaluation (steps 4 and 5), which evaluates the objective using the I/O kernel generated by the Application I/O Discovery (§ III-B) component. This accomplishes I/O-focused Objective Evaluation because the utilized application is reduced to only the calls essential to I/O. The Configuration Evaluation passes a list of measured objectives to Tuning Optimization (§ III-D) (step 6), which sends feedback (step 7) about whether to stop and return the optimal configuration found or continue tuning. This accomplishes Constraint-based Tuning Cost and Benefit Balancing by stopping the tuning pipeline when the cost outweighs the benefit (demonstrated in evaluation - § IV-C).

In its reference implementation, TunIO targets HDF5 applications due to HDF5's ubiquitous use in different domains [44], as a vehicle to demonstrate the effectiveness of the proposed tuning techniques. It builds off of the existing H5Tuner library [45], [46], [27], using its mechanisms to override the configuration parameters of HDF5 applications via an XML file. The tuner code was implemented in Python 3.6+ [47] as with all other components mentioned below. The tuning framework is built using *Distributed Evolutionary Algorithms in Python* (DEAP) [21], a generic framework for evolutionary algorithms used to drive tuning forward. It is used to generate the configuration, use the results of the configuration evaluation to select the next generation's parents, and also provides a variety of other optimizations for generic evolutionary algorithms. The tuning pipeline employs elitism [48], a genetic algorithm technique used to ensure the best solution found so far is always carried through and not lost. Elitism, however, has the drawbacks of over-specializing the population and limiting the search space exploration, resulting in the genetic algorithm getting stuck at a local optimum. To account for these drawbacks, TunIO

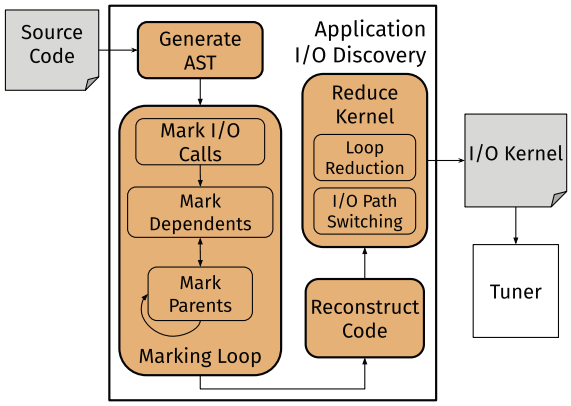


Fig. 4: Application I/O Discovery Component Architecture.

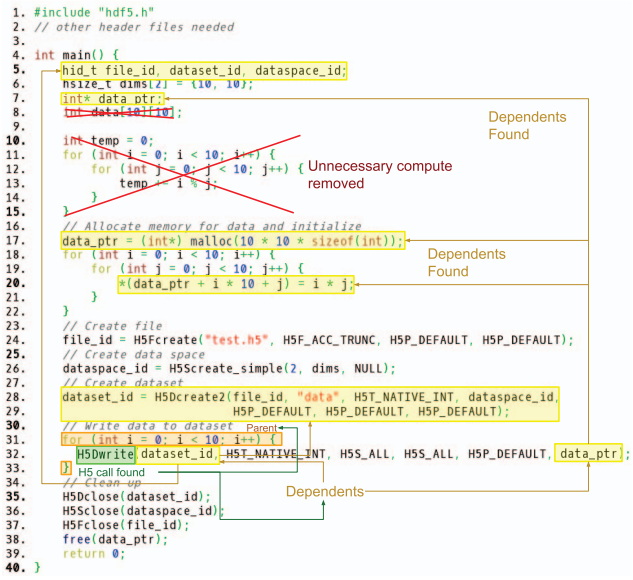


Fig. 5: Example of I/O Discovery; shows the partial consequences (one parent, immediate dependents, and eventual removals) of marking an H5 call.

employs tournament selection [49], a technique where three individuals are chosen randomly from the population of an iteration/generation, and the best two are carried forward as parents for the next generation.

B. I/O-focused Objective Evaluation

To collect objectives efficiently, TunIO passes the application through an Application I/O Discovery component before tuning begins. The application has to be passed through this component only once, but every evaluation of the objective will benefit from the improved runtime. There is no chance of an increase in the application runtime, as the number of instructions run will be the same or less so that, in the worst case, the application runtime will remain the same. In addition, if the I/O kernel of the application causes an error, TunIO will revert to using the full application. However, this should not happen under ordinary circumstances. To parse the source code of the application, the Application I/O Discovery component uses the Clang Python library [50] (particularly the *cindex* and *enumerations* modules).

The Application I/O Discovery component follows the process shown in Figure 4. It takes in the application's source code and outputs an I/O kernel, which the tuning pipeline can then use as the representative of its I/O behavior. To reduce source code to a kernel, the Application I/O Discovery component generates an Abstract Syntax Tree (AST), finds and marks I/O calls and related code in a marking loop, reconstructs the kernel from kept lines, and reduces the kernel by transforming those lines, ultimately outputting an I/O kernel to the tuner. In the prototype implementation of TunIO, the I/O calls are HDF5 calls. TunIO marks source code elements to keep on a per-line basis. An alternative would be to mark statements if the compiler-generated AST can distinguish them. Clang provides a large amount of nuance which makes it difficult to isolate statements [50], and therefore TunIO uses lines as a substitute. To get as close to one-statement-per-line as possible, TunIO uses a custom clang-format preprocessing step which avoids line breaking with a 200-character column limit while placing curly braces on distinct lines and splitting multi-statement lines.

To reduce the AST while maintaining consistency in application logic, the marking loop traverses the AST, finds I/O calls, and marks them to keep. TunIO continues by marking related source code elements as dependents of the kept lines. The dependents for a function call are its arguments and the left-hand side of its assignment (e.g., `dep = foo(dep, dep, ...)` where `dep` marks the dependents). The dependents for a conditional are the boolean statement (e.g., `if (dep) {body}`). The dependents for a loop are its initialization, update, and condition statements (e.g., `for (dep; dep; dep) {body}`). The dependents for an assignment are its left-hand side (e.g., `dep = rhs`). Whenever a variable is marked, a backward traversal must be applied to mark all assignments associated with that variable (e.g., `var = dep`). After marking dependents, TunIO uses the AST to find and mark the contextual parent of each dependent (e.g., the contextual parent of a `for` loop body statement is the loop header). Those parents will themselves have dependents and parents that need to be marked to maintain application logic, so the marking loop will continue until it reaches the source code's top-level.

Once the kept lines have been discovered for all I/O calls, the code is reconstructed with those lines and reduced using a technique such as loop reduction or I/O path switching. These exist to further improve the runtime of the application at the cost of the accuracy of its I/O kernel, and they are optional to apply (a null reduction step could be used instead). *Loop Reduction* incorporates a percentage decrease in iterations of loops containing I/O so that some locality can be maintained while significantly reducing the quantity of I/O operations performed to storage. The scalable metrics for that I/O are then multiplied by the loop reductions to achieve a prediction for the original loop. This drastically improves tuning time, but will trade some information about hardware locality and caching. Another technique that TunIO employs is *I/O Path Switching*, which prepends every path written or read with a path to memory (e.g., `/dev/shm` or *tmpfs*) so that calls are not actually performed to slow disks. This can be good for improving tuning time, but it loses some accuracy due to not tuning for the target storage device. Both these techniques are user-configurable and can capture the user tuning constraints (e.g., debugging or production job).

We show a partial example of the marking process in Figure 5. It can be seen that lines 8 and 10-15 contain compute and variable declarations not necessary to I/O, which will not be kept. Line 32 contains an `H5Dwrite` call and will be kept, along with its parent

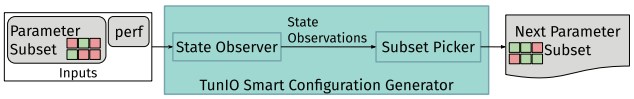


Fig. 6: Overview of TunIO Smart Configuration Generation.

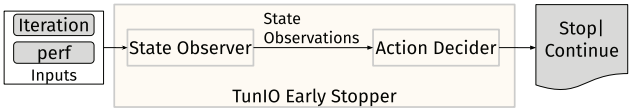


Fig. 7: Overview of TunIO Early Stopping.

for loop. That `H5Dwrite` call requires both `dataset_id` and `data_ptr` as dependents to function correctly, so lines 5 and 28-29 which assign to `dataset_id` and lines 7, 17, and 20 which assign to `data_ptr` will also be kept as a direct result of keeping line 32. There are other lines that contain H5 calls, and the process will continue after the results of this figure until it is finished inspecting them. In addition, the parent and dependent markings will recursively trigger further parent and dependent markings.

C. Impact-First Tuning

In application tuning, not all available parameters would impact the application's performance. Tuning all the parameters would explode the configuration space exponentially, making it harder to find an optimal configuration. The Smart Configuration Generation involves only tuning a subset of the parameters at each iteration to ensure that the search space utilized to find a tuned configuration is minimized and application evaluations are reduced. It then ranks the selected parameters in order of descending impact on the tuning objective to maximize benefit at earlier iterations.

The Smart Configuration Generation component, as seen in Figure 6 is implemented as an RL agent. The agent gets the parameter subset and the best $perf$ achieved during that iteration, and returns the subset of parameters to use in the next tuning iteration. TunIO defines the I/O performance objective as $perf \equiv (1-\alpha)*BW_r+\alpha*BW_w$, where α denotes the ratio of data written over total data transferred, and BW denotes the bandwidth of either read r or write w operations in MB/s. The agent uses a State Observer, created using a *Neural Network* (NN)-based context bandit [51]. The observer uses the inputs provided to the RL agent to produce a state observation which represents a relationship between the application and the tuning environment; specifically, the state demonstrates how the performance of the application varies with inputs within the tuning environment as observed by the agent. The state observation is fed to the Subset Picker, created using an NN-based Q-Learning function, which determines the appropriate subset to use in the next tuning iteration based on the state observation. The NNs are created in Keras [52] and the RL agents using OpenAI Gym [53]. The reward function is dependent on the $norm_{perf}(perf) - norm_{param}(num\_parameters_{subset})$, where $norm()$ is a normalizing function, $num\_parameters_{subset}$ is the number of parameters used in the subset. The $perf$ is normalized using the constant $\frac{1}{BW_{single} \times num\_nodes}$ and $num\_parameters_{subset}$ is normalized by $\frac{1}{num\_parameters}$, the total number of parameters. It also utilizes a 5-iteration delay on the reward function to avoid bias introduced by short-term gains. To understand parameter impact efficiently, the agent is first trained offline to get a baseline model that improves the subsequent online training of the agent by minimizing the warm-up time for the model. The offline training process is accomplished by first doing a simple parameter sweep on some representative I/O kernels, including VPIC, FLASH, and HACC. During the parameter sweep, their appropriate $perf$ values are collected as performance metrics. The parameters swept are HDF5, MPI-IO and Lustre parameters, and include sieve_buf_size, chunk_cache, alignment, meta_block_size, col_meta_ops, mdc_conf, coll_metadata_write, striping_factor, striping_unit, cb_nodes, and cb_buffer_size. These parameters were chosen because they have demonstrated the best changes in performance for these libraries [5], [6], [7]. After performing a sweep on each I/O kernel, a Principal Component Analysis (PCA) analysis is performed on the parameters with respect to $perf$ to train the model to isolate the most impactful parameters. The trained model is ultimately utilized in the configuration generation phase of the tuning pipeline, where it continues to learn from the applications it is exposed to.

D. Balancing Tuning Cost and Benefit

The Early Stopping component is designed to determine whether to stop or continue the tuning pipeline. This component ensures that the tuning pipeline stops when there is no measurable improvement in performance over recent tuning iterations. This aims to balance the performance gained with the time spent tuning. This usually stops the tuning pipeline when the balanced configuration has been determined, without wasting tuning cycles. Nonetheless, in rare cases, it may get stuck in a local optimum, due to the overspecialization of the population, where it cannot determine whether further tuning will bring sufficient benefit.

The Early Stopping component, implemented with similar RL techniques as the Smart Configuration Generation component, gets the iteration and the performance from the tuner as inputs and returns whether the tuner should stop or continue. Its design is shown in Figure 7. The observed state is sent to the Action Decider, which determines the appropriate action. The inputs are $perf$ gained in the respective iteration and the number of iterations. Given this input, the agent decides if the tuning pipeline should stop or continue. To train the agent offline, tuning is emulated using generated log curves, as tuning performance follows a log curve (as was demonstrated in Figure 2) where performance is gained initially and attenuates as it approaches a tuned configuration. The log curves generated for training include noise in the form of randomized shifts down the curve to account for tuning cases where the wrong parameter is chosen briefly before adjusting. The agent utilizes a reward function with a 5-iteration delay. Each simulated application has a log curve with different characteristics such as initial value, growth rate, etc. The offline training runs until the average reward of the agent begins to stagnate in most simulated applications, indicated in this case by 5% or less increase across five iterations. Once trained, the agent is integrated into the tuning pipeline, where it learns from trends of the applications it is exposed to.

E. Implementation Details

API: TunIO separates its components and provides an interface so that they can be used by other tuning pipelines. The interfaces provided by the TunIO library are presented in Table I. It can be seen that the Early Stopping component takes the parameters of the current tuning iteration and the best $perf$ attained in that iteration as inputs, and outputs a suggestion to stop or continue. The Application I/O Discovery component takes the source code and options and outputs

a modified I/O kernel. Options may include manually indicated keep regions and flags for source code modifiers such as I/O path switching. Finally, the Smart Configuration Generation component takes the performance and the parameter subset used and outputs a subset of the parameter set to test.

Function	Input	Output
stop	current_iteration, best_perf,	stop/continue
discover_io	source_code, options	I/O kernel
subset_picker	perf, current_parameter_set	next_parameter_set

TABLE I: API for TunIO

Use Case: TunIO can be accessed as a user-level library and provides a CLI tool for the Application I/O Discovery component. This tool converts the source code to its equivalent I/O kernel, which the user can compile using their preferred method and use as a substitute for the application during the configuration evaluation phase. To demonstrate how these components are used, we create an example use case with DEAP and the H5Tuner library. DEAP provides a framework that allows the user to simply specify the termination condition, population generation, crossover, mutation, and fitness functions, as well as select optimizations for the genetic algorithm. This can represent a tuning pipeline, with the population generation mapping to configuration generation, the crossover and mutation mapping to search space optimization, and the fitness function mapping to configuration evaluation. In TunIO's case, DEAP calls the Smart Configuration Generation component within its population generation, calls Python `subprocess()` to spawn an I/O kernel job with the appropriate configurations set and monitor bandwidth (using monitoring hooks such as Darshan [54]) within its fitness function, and calls the Early Stopping component to determine its termination condition. To generate the I/O kernel, this use case calls the Application I/O Discovery component before using the DEAP framework.

F. Limitations and Considerations

While TunIO achieves its objective of balancing the tuning cost with the benefit gained, there are some considerations associated with its design. First, source code modification techniques that reduce the runtime will less accurately represent the I/O of the full application. It is possible that performance will be left on the table when tuning the kernel instead of the real application. Second, when only tuning the most impactful parameters, the configuration space is highly reduced, but performance may not be maximized for all workloads. This does imply that Impact-First tuning will likely never obtain the global optimal configuration set due to some parameters not being tuned. However, it will trade missed performance potential for better resource utilization while still acquiring near-optimal application I/O performance. Third, when early stopping happens based on statistical patterns in tuning performance and user constraints, there is a possibility that a better configuration would be discovered in a future iteration. However, if the application is stuck in a local optimum that is hard to escape from, it could take a long time before this improvement is seen. TunIO may choose to exit early even when suboptimal results have been obtained. The rationale behind this design choice is that tuning follows a logarithmic curve in performance (as in Figure 2), and minor improvements in performance do not justify the prolonged convergence of the tuning pipeline.

IV. Evaluation

Methodology: To assess the efficacy of our components, we conduct evaluations using the HDF5 library. This allows us to determine the viability of each component and measure their success in achieving their defined objectives. For our evaluations, we tune a subset of 12 parameters across HDF5, MPI, and Lustre, which gives a search space of over 2.18 billion permutations. Finally, a comprehensive evaluation is conducted to assess the impact of all components on the tuning pipeline.
Testing environment: All tests were performed on the Haswell nodes on Cori supercomputer [55]. Cori has a total of 2,388 Haswell nodes, each with a 16-core 2.3GHz Intel Xeon processor and 128 GB of 2133 MHz DDR4 memory. Where storage was required, tests used the scratch Lustre filesystem available on Cori [56], which has 30 PB of disk space and an aggregate I/O bandwidth of approximately 700 GB/sec. Each test uses 4 nodes and 128 processes for individual components. The end-to-end test was performed on Cori with 500 nodes and 1600 processes. To mitigate the volatility of the Cori platform, each application run is performed 3 times and bandwidths are averaged. The time cost of running the application is not accumulated across runs, because different systems have different volatility, and the extra runs can be seen as a necessary expense for a given platform.
Metrics: To quantify the cost-benefit balance of tuning, we need a metric that is proportional to the improvement in application performance gained from tuning and inversely proportional to the time spent tuning. We call this metric *Return on Tuning Investment* (RoTI), and it has its basis in financial return on investment metrics [57]. RoTI is represented by the formula

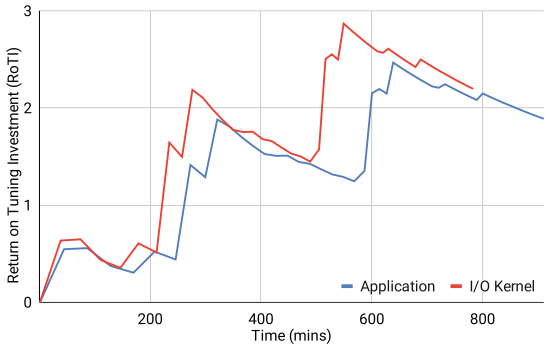
$$RoTI(t) \equiv \frac{perf_{achieved}(t) - perf_{achieved}(0)}{t}$$

where $perf_{achieved}(t)$ represents the maximum achieved $perf$ in MB/S at a certain time t in the tuning pipeline, $perf_{achieved}(0)$ represents the initial $perf$ in MB/s of the default configuration before tuning. This metric is an ascending value, with higher values providing higher returns in performance for the same tuning overhead (i.e., time spent tuning in minutes). Since these returns are bandwidths, an RoTI of 40 MB/s per minute spent tuning would represent an increase in bandwidth of 40 MB/s for each minute of tuning overhead. From this description, it can be seen that RoTI is fundamentally an application of return on investment to I/O tuning. We also use iterations as a duration metric, representing the current generation of the genetic algorithm.

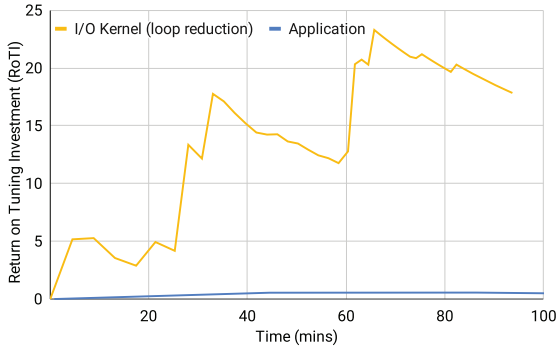
A. Analysis of I/O-focused Objective Evaluation

To demonstrate how the Application I/O Discovery component achieves I/O focused objective evaluation (objective 1), we ran the tuning pipeline on two versions of MACSio [58]: one which was reduced to its I/O kernel by the Application I/O Discovery component and one which was not. Since MACSio is a workload generator, we based the ratio of compute-to-I/O on observed values from running VPIC programs with the Dipole configuration. We show the RoTI for these two tuning curves in Figure 8(a).

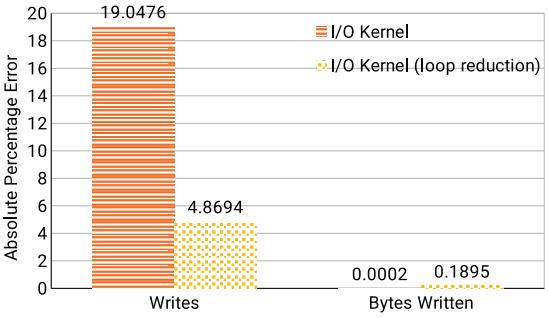
These results show an improved RoTI for the I/O kernel. The reason is that running only the parts of the application critical to I/O takes less time than running the full application at each step, which causes a smaller investment to achieve the same performance gain. As a result, the peak RoTI is 2.87 compared to the 2.47 peak RoTI of the regular application, which represents an additional improvement in application I/O bandwidth of 0.4 MB/s for each minute of tuning overhead when using I/O discovery compared to the regular application. The overall time to reach peak RoTI is reduced from 639 minutes to 549,



(a) Return on Tuning Investment with and without Application I/O Discovery.



(b) Return on Tuning Investment with and without loop reduction.



(c) Percentage Similarity of MACSio VPIC with and without loop reduction.

Fig. 8: I/O Discovery Evaluations

which represents a 14% decrease in tuning time. This improvement occurs because the application I/O Discovery component will remove unnecessary computing and networking statements, leaving only statements necessary to I/O. The exact improvement will vary depending on the application, and other techniques such as I/O path switching can be utilized to obtain an improvement in that case.

In Figure 8(b), we add the technique of loop reduction to our I/O Discovery process and observe the RoTI gained. The loop reduction applied was to perform 1% of the iterations. Loop reduction is a powerful technique, and in this particular case, it increases peak RoTI to 23.30, which is a very large boost over the 2.47 peak RoTI of the original application (over 9×). This represents an additional I/O bandwidth gain of 20.83 MB/s for each minute of tuning overhead with loop reduction compared to the original application. This is

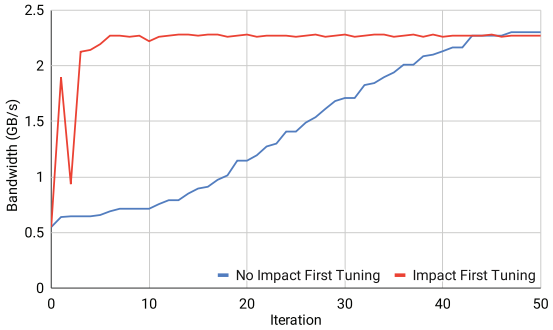


Fig. 9: Bandwidth achieved by the Smart Configuration Generation component over iterations of tuning FLASH.

because there is a significant reduction in I/O performed while maintaining enough calls to show the benefits of data locality. While this may introduce some inaccuracy, we found that the reported bandwidths, in this case, were 97.10% accurate. It is important to note that this technique will not always be applicable. Whenever the loop iterations are too small to reduce (less than one iteration on reduction), loop reduction will not be able to do anything.

In Figure 8(c), we show the absolute percentage error of a few metrics to contrast the generated I/O kernels with and without loop reduction to the original application. The percentage similarity is a simple division of the metric for the kernel by the metric for the original application. For the reduced kernel, we multiplied the metric by 100 to show the quantity of I/O that would be assumed by the kernel before doing the comparison. We find that the number of bytes written for the kernel and reduced kernel both have a very low absolute percentage error of less than 1% (0.0002% for kernel and 0.19% for reduced kernel). For the number of write operations, there is greater inaccuracy. The kernel has an error of 19.05%, which is due to the removal of some trivial writes, which may include logging operations or print statements. The reduced kernel has a lower error of 4.87%, which probably means that more I/O is being performed in early loop iterations than later ones so that the implied number of operations is greater than the true number that would have been performed and therefore makes up for a few of the absent logging operations when compared to the original application.

B. Analysis of Impact First Tuning

To evaluate the Smart Configuration Generation component, which achieves our criteria for impact first tuning (objective 2), we attach the Smart Configuration Generation component to the tuning pipeline for the FLASH I/O kernel, and we measure the overall performance gained over tuning iterations with respect to the time spent tuning. We compare this performance with the performance achieved in the tuning pipeline without the Smart Configuration Generation component. The result of this evaluation can be seen in Figure 9. We observe that Impact-First Tuning reaches a bandwidth of 2.3 GB/s at tuning iteration 6, while No Impact-First Tuning reaches this bandwidth at iteration 43. This represents an improvement of 86.05% in the number of tuning iterations. It is also worth noting that in the second iteration, Impact-First Tuning chooses a suboptimal subset (lower performance than the previous iteration) causing performance to drop between iterations, but based on that feedback it is able to

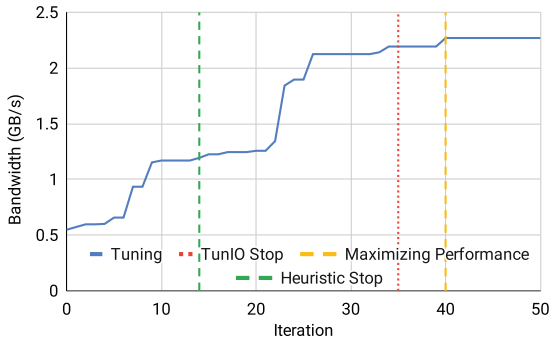
determine more optimal subsets to achieve optimal performance faster. The final configuration determined in tuning changes seven parameters from their default values, with the remaining five not having a significant impact on the tuning process.

C. Tuning Cost-Benefit Analysis

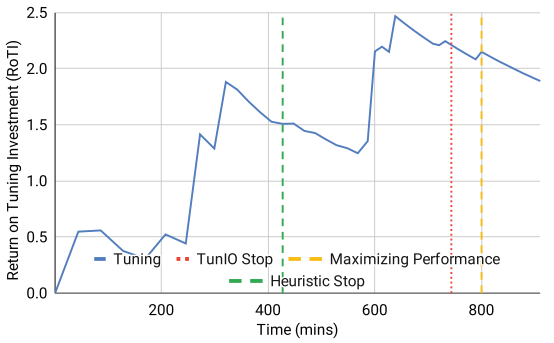
To evaluate the effectiveness of the Early Stopping component to balance the tuning cost and benefit (objective 3), we attach the component to the tuning pipeline for HACC and measure the performance gained. Additionally, we keep tuning to determine if there was any further increase in performance and compare it against heuristic early stoppers [18]. Heuristic-based early stoppers decide to stop the tuning pipeline when there is no improvement in the performance over a specified number of iterations. We set those thresholds at 5% improvement and 5 iterations, since 5% is a common measure of statistical significance [59].

In Figure 10(a), we observe that TunIO's early stopper terminates tuning at the 35^{th} of 50 generations of the tuning pipeline, achieving 2.2 GB/s bandwidth ($\approx 4\times$ improvement from the non-tuned application bandwidth of 0.55 GB/s). If the tuning was allowed to continue for the remainder of the 15 generations, the performance would have only gained a negligible 0.08 GB/s (i.e., $0.14\times$ gain) over the bandwidth chosen by TunIO's early stopper. In addition, TunIO's Early Stopping component intelligently avoids getting caught in the plateau around the 10^{th} to 20^{th} iterations. In contrast, the traditional heuristic-based early stopper is undoubtedly affected at iteration 14 when it decided to stop, achieving only 1.2 GB/s bandwidth. This results in a mere $2\times$ performance improvement over the non-tuned application. This is $\approx 83\%$ less than TunIO's outcome (i.e., TunIO boosted performance by $4\times$); this loss in potential is significant, considering the time spent tuning the application across 14 generations would be expected to precipitate more performance than this.

In Figure 10(b), we plot the RoTI of tuning HACC. The perfect RoTI for this application would be 2.31, achieved by stopping at iteration 35. We observe that TunIO's early stopping mechanism has an RoTI of 2.09, which is 90.5% of the best return. Maximizing Performance represents an alternate stopping method where tuning stops immediately on getting the optimal performance. Models which utilize Maximizing Performance stopping would typically take a few iterations to determine that the true optimal was reached, but we assume a perfect model for this evaluation. The Maximizing Performance stopping method gets 1.99 RoTI or 86.1% of the best return; TunIO stopping achieves an additional 0.1 MB/s improvement in I/O bandwidth for each minute of tuning overhead compared to this method. The heuristic model of stopping achieves 1.37 RoTI or 59.3% of the best return; TunIO stopping achieves an additional 0.72 MB/s of I/O bandwidth for each minute of tuning overhead compared to this method. If the user had a maximized tuning budget of 50 iterations, that would give an RoTI of 1.8 or 77.9% of the best return. In addition to giving a better RoTI, TunIO stops at 744 minutes as opposed to the 800 minutes of Maximizing Performance stopping, which represents a 7.61% time improvement. The findings indicate that the use of TunIO stopping results in better returns compared to not using early stopping or methods that stop based on maximizing performance or heuristic criteria. Additionally, TunIO reaches near-optimal returns faster than all other methods.



(a) Tuning while using Early Stopper.



(b) Return on Tuning Investment.

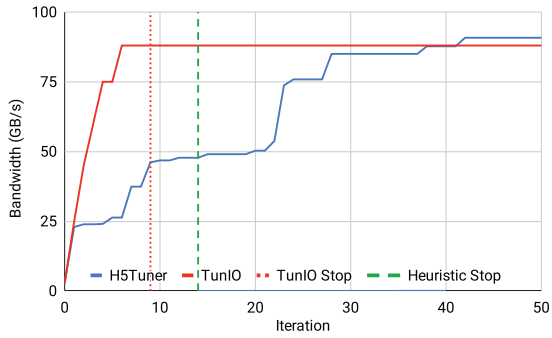
Fig. 10: Visualization of Tuning HACC with Early Stopping.

D. TunIO Pipeline Analysis

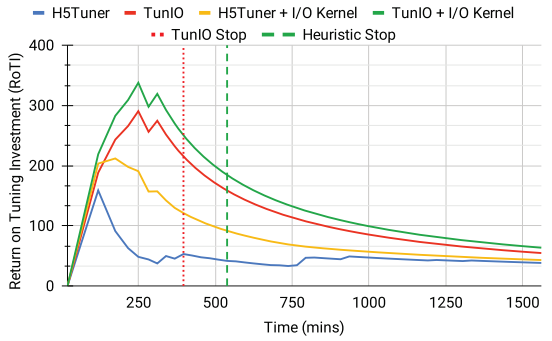
We need to quantify the overall benefit of using all the TunIO components in their respective phases of the tuning pipeline.

We measure the performance gained and the number of iterations needed to get to a tuned configuration. We run several tuning jobs for BD-CATS [60], [61], H5Tuner with No Stop, H5Tuner with Heuristic Stop, TunIO, H5Tuner + I/O Kernel with No Stop, H5Tuner + I/O Kernel with Heuristic Stop, and TunIO + I/O Kernel. The Heuristic Stop refers to the same 5% heuristic explained in section IV-C.

Figure 11(a) shows the bandwidth of tuning the job, with stop markers indicating where TunIO finds that RoTI diminishes or where the Heuristic decides to stop. TunIO observes no improvement from iterations 4 to 5, and this could be due to a suboptimal subset being tuned, resulting in a reduction in performance for that iteration. However, due to elitism, which carries over the best configuration from the previous generation, the performance stays the same over the 4^{th} and 5^{th} iterations. By the 6^{th} TunIO iteration, the application reaches its peak bandwidth at 88 GB/s. The RL-based Early Stopping component stops the tuning pipeline at the 9^{th} iteration due to a lack of further improvement of the bandwidth of the application. The reduction in the search space by the parameter selection agent helps the tuning pipeline converge faster. We compare this with an application being tuned in H5Tuner, which does not have TunIO's optimizations (Application I/O Discovery, Smart Configuration Generation, and RL-based Early Stopping). We observe that it is harder for the tuning pipeline to find a good configuration due to the noise of parameters that do not impact the performance of the application within a large



(a) Bandwidth of BD-CATS with TunIO.



(b) RoTI of BD-CATS with H5Tuner.

Fig. 11: End-to-end evaluation of BD-CATS.

search space. This, unfortunately, prolongs the tuning time and ends with the application using a large allocated tuning budget of 1750 minutes. TunIO, by contrast, only uses a tuning budget of $\approx$ 468 minutes, an improvement of $\approx$73%. Note that H5Tuner without stop is able to leverage additional parameter configurations to achieve a better max bandwidth of 90.8 GB/s, but this 3% I/O bandwidth improvement is only achieved after significant time and tuning effort. Alternatively, we can compare to H5Tuner with Heuristic Stop, which uses a tuning budget of $\approx$ 538 minutes to achieve a bandwidth of 47.7 GB/s. TunIO achieves 1.84$\times$ I/O bandwidth in 87% of the time.

Figure 11(b) plots an RoTI curve for the tuning pipeline. We observe that, compared to H5Tuner with Heuristic Stop, TunIO provides a higher RoTI of 215 compared to the H5Tuner with Heuristic Stop return of 41.6. This represents a gain of 173.4 MB/s of I/O bandwidth in application performance for each minute of tuning overhead. We further observe that using the I/O kernel instead of the original application improves return significantly, with TunIO achieving an RoTI of 250, which represents an additional performance improvement of 35 MB/s for each minute of tuning overhead compared to H5Tuner with Heuristic Stop. When using the I/O kernel instead of the original application in H5Tuner with Heuristic Stop, there is an RoTI of 91.6, representing an additional performance improvement of 50 MB/s for each minute of tuning overhead.

We observe that TunIO provides the right balance of performance and investment of tuning time and resources, and consistently provides a higher Return on Tuning Investment for the user.

Tuning an application will achieve a better runtime for that

Fig. 12: Demonstration of overall lifespan time of BD-CATS when incorporating various tuning methodologies for different numbers of executions in production.

application in production, but at the cost of the time spent tuning. When running an application only once, tuning will only achieve a time cost benefit for the extreme case of poorly configured programs, if at all. For production workloads, however, an application is expected to execute more times across multiple steps and iterations and achieve a benefit. This benefit is achieved when the overall time spent across the tuning process and the totality of the period of continued execution becomes less than it would be if tuning was not applied. In figure 12, we demonstrate the overall time of the application lifecycle across various numbers of executions applied to BD-CATS, comparing TunIO with H5Tuner and No-Tuning cases. We do this to demonstrate the time it takes to tune (shown as the y-intercept), the time at which tuning becomes viable over No-Tuning (shown as intersections), and the overall time cost benefits that TunIO has over H5Tuner across the lifespan of BD-CATS. TunIO takes 403 minutes to tune BD-CATS, while H5Tuner takes 1560 minutes to tune BD-CATS. TunIO has a viability point of 1394 executions, while H5Tuner has a viability point of 5274 executions. Note that this viability is dependent on the application being tuned and the tuning method, and cannot be generalized. In this case, TunIO achieves tuning viability in 73.6% fewer executions than H5Tuner. TunIO maintains a better overall time than H5Tuner until 3.99 million executions. This occurs because the benefit that H5Tuner achieves takes a very long time to manifest, and therefore TunIO retains its advantage for numerous executions.

V. Related Work

A. Tuning

Some works on optimizing the tuning pipeline involve changing how the search space is represented. Chen et al. [39], proposed a meta multi-objectivization model (MMO) that utilizes an auxiliary performance objective (e.g., throughput in addition to latency) to prevent the search from being trapped in a local optimum. This technique leads to a 42% gain in overcoming the trap while achieving the same performance targets. However, these improvements are domain-agnostic, as opposed to TunIO, which addresses the ever-increasing search spaces specific to I/O tuning.

The work by Bez et al. [38] has performed I/O stack tuning, as it uses contextual bandits to optimize the I/O forwarding layer. This technique leverages I/O access patterns to determine the performance impact of a parameter. While this is tuning of a layer of the I/O stack, it is aimed at tuning performed by the system admin rather than by

the users. TunIO is aimed at a higher level in the I/O stack, and, as a result, it can apply different techniques, such as I/O kernel reduction and collaborative tuning.

H5Tuner [45], [46] has proposed user-level tuning of the HDF5 I/O library using genetic algorithms. However, this work lacks TunIO's multilayer tuning, smart configuration generation, and early stopping mechanisms. While H5Tuner could utilize I/O kernels, it lacks TunIO's unique I/O Discovery, which enables the application configuration to be incorporated into the tuning process if desired.

B. I/O Discovery

The detection of critical regions of code is a process which is usually performed manually. It can be utilized alongside semi-empirical performance modeling in order to improve the tuning pipeline, as in the work by Hoefler et al. [62]. However, this work suffers from its generality. It requires manual determination of critical sections of code because generic application tuning is not allowed to make unsupervised assumptions about which regions are critical. Since TunIO is specific to tuning for I/O, it can automatically determine which regions of the code are critical to I/O via the application of I/O knowledge. This sort of determination of source code sections critical to I/O has been seen before in Vidya [63], which uses source code analysis to isolate I/O regions in order to determine I/O phases of the application and improve profiling. However, Vidya does not reassemble the I/O regions of the code to produce an I/O kernel as TunIO does, nor does it apply the technique to tuning. There are tools, such as in Behzad et al. [64] or Skel [65], which utilize trace files or ADIOS configurations to generate a similar I/O kernel, but, curiously, none directly utilize the source code. Admittedly, the Behzad work points to a limitation that source code may not always be available, but the benefit of using source code is that the I/O kernel generated will be robust to the different potential configurations of the application. This is necessary if the application is included in a tuning pipeline, as each application configuration would require its own trace obtained via running the application, exploding the required number of runs and therefore removing any value the I/O kernel had.

C. Smart Configuration Generation

Works to reduce the configuration space to aid in better configuration generation include Menon et al. [37]. However, their process reduces the parameter set, implying that it needs to run some configurations with the entire parameter space. Depending on the size of the parameter set, this would require a significant number of configurations to be evaluated prior to identifying the relationships between parameters with respect to the objective. This differs from the TunIO approach in that we start by determining a subset of parameters that allow the application to achieve performance improvements. This approach allows us to arrive at a tuned configuration faster. Additionally, when the component is exposed to new applications, it can learn from the new trends it sees and can provide improved results in the future. H5Tuner [20] uses modeling to reduce the search space to the 20 parameters that are the most impactful to I/O performance. However, this work has limitations with the applicability of the model across different applications, as well as with the same application at different scales. Additionally, the choice of a specific number of impactful parameters could lead to an erroneous number of parameters being tuned.

D. Early Stopping

Early stopping has been explored in works by Golovin et al.[18]. They propose two methods, one which creates a performance prediction curve using regression and uses it to estimate the final value, and the other checks for a decrease in the objective from a running average. The heuristic nature of these methods makes them lightweight, but this limits them to not being able to adapt to application trends and changes in trends that deviate from their initial model. Additionally, early stopping as a technique is also used in training *Artificial-Intelligence* (AI) models. It is used similarly to its application within the context of I/O tuning, but to maximize accuracy or minimize a relevant error metric. Ultimately, they follow a heuristic where they stop if the performance objective does not improve by a threshold over a period of time [66]. By contrast, TunIO's approach uses RL to achieve robustness to varied application patterns.

VI. Conclusions and Future Work

We have presented TunIO, an AI-powered I/O optimization framework, and its three primary components. The Application I/O Discovery component reduces an application to an I/O kernel, which aids in reducing the time spent evaluating the objective during the tuning pipeline. The Smart Configuration Generation component provides a subset of parameters to be tuned, which aids in reducing the overall tuning search space. Finally, the Early Stopping component determines whether to stop the tuning pipeline based on appropriate feedback, which aids in balancing the tuning budget and the application performance improvement. We observe that incorporating all the techniques of TunIO into the tuning pipeline can improve the time it takes to achieve a tune of comparable performance by $\approx 73\%$ over H5Tuner. TunIO efficiently tunes applications, with an expected additional performance improvement of 208.4 MB/s in I/O bandwidth achieved for each minute of tuning overhead for select applications.

During designing and developing TunIO, we discovered several key points in the tuning space that require further consideration. As future work, we will continue improving TunIO by addressing some of the limitations mentioned in Section III-F. Specifically, we will explore more use cases of source-code modification techniques. There are a wide variety of techniques that can be utilized to transform the generated I/O kernel in interesting ways, such as simulating loops, removing blind writes, simulating necessary compute, and more. While these techniques were dismissed for use in TunIO, they could still apply elsewhere. The generality of source code modification needs further work to properly explore its possibilities. We would like to explore adding an interactive session feature where a configuration can be refined over time across a series of runs. Finally, we want to improve our early stopping model by including the expected number of production runs as input, to allow TunIO to continue tuning if the user knows that they expect to run the application long enough for the extra tuning to be worthwhile.

VII. Acknowledgement

This research was partially supported by the National Science Foundation under grants OCI-1835764 and OAC-2104013, and by the Department of Energy under grant DE-SC0023263. This research used the resources of the National Energy Research Scientific Computing Center (NERSC). This research was supported in part by The Ohio State University under a subcontract (GR130493) that was funded by the director of the Office of Science, the Office of Advanced Scientific Computing Research (ASCR) of the U.S. Department of Energy (DOE) under contract number DE-AC02-05CH11231 (at LBNL), with program managers Hal Finkel and Margaret Lentz.

References

[1] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018.

[2] R. Schoonhoven, B. Veenboer, B. van Werkhoven, and K. J. Batenburg, "Going green: optimizing gpus for energy efficiency through model-steered auto-tuning," *arXiv preprint arXiv:2211.07260*, 2022.

[3] M. F. Ringenburg, A. Sampson, L. Ceze, and D. Grossman, "Profiling and autotuning for energy-aware approximate programming," in *Workshop on approximate computing across the system stack (WACAS)*, 2014.

[4] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.

[5] B. Xie, H. Tang, S. Byna, J. Hanley, Q. Koziol, T. Li, and S. Oral, "Battle of the defaults: Extracting performance characteristics of hdf5 under production load," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 51–60.

[6] S. Pellegrini, R. Prodan, and T. Fahringer, "Tuning mpi runtime parameter setting for high performance computing," in *2012 IEEE International Conference on Cluster Computing Workshops*. IEEE, 2012, pp. 213–221.

[7] M. Seiz, P. Offenhäuser, S. Andersson, J. Hötzer, H. Hierl, B. Nestler, and M. Resch, "Lustre i/o performance investigations on hazel hen: experiments and heuristics," *The Journal of Supercomputing*, pp. 1–29, 2021.

[8] T. H. Group, "The hdf5(r) library & file format - the hdf group," 2022. [Online]. Available: https://www.hdfgroup.org/solutions/hdf5

[9] E. C. Project, "Pnetcdf," 2023. [Online]. Available: https://parallel-netcdf.github.io/

[10] O. R. L. C. Facility, "Adios – oak ridge leadership computing facility," 2023. [Online]. Available: https://www.olcf.ornl.gov/center-projects/adios/

[11] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in mpi–io," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[12] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 219–230.

[13] L. L. N. S. LLC, "Unifyfs: A file system for burst buffers – unifyfs 1.0.1 documentation," 2023. [Online]. Available: https://unifyfs.readthedocs.io/en/dev/

[14] Oracle, "Lustre* software release 2.x operations manual," Oracle, Tech. Rep., 2017. [Online]. Available: https://doc.lustre.org/lustre{_}manual.pdf

[15] J. Heichler, "An introduction to beegfs," 2014.

[16] H. Tang, B. Xie, S. Byna, P. Carns, Q. Koziol, S. Kannan, J. Lofstead, and S. Oral, "SCTuner: An Autotuner Addressing Dynamic I/O Needs on Supercomputer I/O Subsystems," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, Nov. 2021, pp. 29–34.

[17] P. Harrington, W. Yoo, A. Sim, and K. Wu, "Diagnosing parallel i/o bottlenecks in hpc applications," in *International Conference for High Performance Computing Networking Storage and Analysis (SCI7) ACM Student Research Competition (SRC)*, 2017.

[18] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google Vizier: A Service for Black-Box Optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 1487–1495. [Online]. Available: https://doi.org/10.1145/3097983.3098043

[19] B. Behzad, J. Huchette, H. V. T. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat, "A framework for auto-tuning hdf5 applications," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–128. [Online]. Available: https://doi.org/10.1145/2462902.2462931

[20] B. Behzad, S. Byna, S. M. Wild, M. Snir *et al.*, "Dynamic model-driven parallel i/o performance tuning," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 184–193.

[21] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[22] A. F. Furtunato, K. Georgiou, K. Eder, and S. Xavier-de Souza, "When parallel speedups hit the memory wall," *IEEE Access*, vol. 8, pp. 79 225–79 238, 2020.

[23] V. Averbukh, A. Bersenev, M. Forghani, A. Igumnov, D. Manakov, A. Popel, S. Sharf, and P. Vasev, "Visualizing a supercomputer: A case of objects regrouping," in *CEUR Workshop Proceedings*, vol. 2485. CEUR-WS, 2019, pp. 74–76.

[24] D. Hildebrand, A. Nisar, and R. Haskin, "pnfs, posix, and mpi-io: a tale of three semantics," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, 2009, pp. 32–36.

[25] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong, "Overview of the mpi-io parallel i/o interface," in *Input/Output in Parallel and Distributed Computer Systems*. Springer, 1996, pp. 127–146.

[26] C. Wang, K. Mohror, and M. Snir, "File system semantics requirements of hpc applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 19–30.

[27] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O Performance of HPC Applications with Autotuning," *ACM Transactions on Parallel Computing*, vol. 5, no. 4, pp. 15:1–15:27, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3309205

[28] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel I/O complexity with auto-tuning," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–12, iSSN: 2167-4337.

[29] S. Sivanandam and S. Deepa, "Genetic algorithms," in *Introduction to genetic algorithms*. Springer, 2008, pp. 15–37.

[30] Z. B. Zabinsky *et al.*, "Random search algorithms," *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.

[31] R. Greiner, "Palo: A probabilistic hill-climbing algorithm," *Artificial Intelligence*, vol. 84, no. 1-2, pp. 177–208, 1996.

[32] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[33] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[34] P. Wauteleta and P. Kestenera, "Parallel io performance and scalability study on the prace curie supercomputer," *White paper, Prace*, vol. 115, p. 180, 2011.

[35] O. R. N. Laboratory, "Supported engines," 2020. [Online]. Available: https://adios2.readthedocs.io/en/latest/engines/engines.html#supported-engines

[36] Cray, "Tuning parameters – cray openshmemx 10.1.0 documentation," 2023. [Online]. Available: https://cray-openshmemx.readthedocs.io/en/latest/tuning_parameters.html

[37] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 831–840, iSSN: 1530-2075.

[38] J. L. Bez, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. Navaux, "Adaptive request scheduling for the i/o forwarding layer using reinforcement learning," *Future Generation Computer Systems*, vol. 112, pp. 1156–1169, 2020.

[39] T. Chen and M. Li, "Multi-objectivizing software configuration tuning (for a single performance concern)," *arXiv preprint arXiv:2106.01331*, 2021.

[40] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[41] G. Lukat and R. Banerjee, "A gpu accelerated barnes–hut tree code for flash4," *New Astronomy*, vol. 45, pp. 14–28, 2016.

[42] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[43] M. Li, M. Soltanolkotabi, and S. Oymak, "Gradient descent with early stopping is provably robust to label noise for overparameterized neural networks," in *International conference on artificial intelligence and statistics*. PMLR, 2020, pp. 4313–4324.

[44] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 2011, pp. 36–47.

[45] B. Behzad, J. Huchette, H. V. T. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat, "A framework for auto-tuning HDF5 applications," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, ser. HPDC '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 127–128. [Online]. Available: https://doi.org/10.1145/2462902.2462931

[46] B. Behzad, R. Aydt, S. Byna, and J. Huchette, "Auto-Tuning of Parallel I/O Parameters for HDF5 Applications," p. 3.

[47] P. S. Foundation, "Python release python 3.6.0 — python.org," 2023. [Online]. Available: https://www.python.org/downloads/release/python-360/

[48] C. W. Ahn and R. S. Ramakrishna, "Elitism-based compact genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 4, pp. 367–385, 2003.

[49] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[50] T. L. Team, "Clang indexing library bindings – libclang 14.0.6 documentation," 2022. [Online]. Available: https://libclang.readthedocs.io/en/latest/index.html

[51] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire, "Taming the monster: A fast and simple algorithm for contextual bandits," in *International Conference on Machine Learning*. PMLR, 2014, pp. 1638–1646.

[52] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.
[53] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.
[54] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 9–17.
[55] NERSC, "Cori - nersc documentation," 2022. [Online]. Available: https://docs.nersc.gov/systems/cori/
[56] ——, "Cori scratch - nersc documentation," 2023. [Online]. Available: https://docs.nersc.gov/filesystems/cori-scratch/
[57] H. Erdogmus, J. Favaro, and W. Strigel, "Return on investment," *Ieee Software*, vol. 21, no. 3, pp. 18–22, 2004.
[58] L. L. N. Laboratory, "Github - llnl/macsio: A multi-purpose, application-centric, scalable i/o proxy application," 2023. [Online]. Available: https://github.com/LLNL/MACSio
[59] R. M. Craparo, "Significance level," *Encyclopedia of measurement and statistics*, vol. 3, pp. 889–891, 2007.
[60] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, and P. Dubey, "Bd-cats: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
[61] t. L. B. N. L. The Regents of the University of California, "Bd-cats-io benchmark," 2017. [Online]. Available: https://github.com/glennklockwood/bdcats-io
[62] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2011, pp. 1–12.
[63] H. Devarajan, A. Kougkas, P. Challa, and X.-H. Sun, "Vidya: Performing code-block io characterization for data access optimization," in *2018 IEEE 25th International Conference on High Performance Computing, Data, and Analytics*. IEEE, 2018.
[64] B. Behzad, H.-V. Dang, F. Hariri, W. Zhang, and M. Snir, "Automatic generation of i/o kernels for hpc applications," in *2014 9th Parallel Data Storage Workshop*. IEEE, 2014, pp. 31–36.
[65] J. Logan, S. Klasky, J. Lofstead, H. Abbasi, S. Ethier, R. Grout, S.-H. Ku, Q. Liu, X. Ma, M. Parashar *et al.*, "Skel: generative software for producing skeletal i/o applications," in *2011 IEEE Seventh International Conference on e-Science Workshops*. IEEE, 2011, pp. 191–198.
[66] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approximation*, vol. 26, no. 2, pp. 289–315, 2007.