AUTOPAGER: Auto-tuning Memory-Mapped I/O Parameters in Userspace

Karim Youssef*§, Niteya Shah*§, Maya Gokhale†, Roger Pearce†, and Wu-chun Feng*
*Department of Computer Science, Virginia Tech
{karimy,niteya,wfeng}@vt.edu
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
{gokhale2,rpearce}@llnl.gov

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926409

***Abstract*—The exponential growth in dataset sizes has shifted the bottleneck of high-performance data analytics from the compute subsystem to the memory and storage subsystems. This bottleneck has led to the proliferation of non-volatile memory (NVM). To bridge the performance gap between the Linux I/O subsystem and NVM, userspace memory-mapped I/O enables application-specific I/O optimizations. Specifically, *UMap*, an open-source userspace memory-mapping tool, exposes tunable paging parameters to application users, such as page size and degree of paging concurrency. Tuning these parameters is computationally intractable due to the vast search space and the cost of evaluating each parameter combination. To address this challenge, we present AUTOPAGER, a tool for auto-tuning userspace paging parameters. Our evaluation, using five data-intensive applications with *UMap*, shows that AUTOPAGER automatically achieves comparable performance to exhaustive tuning with 10× less tuning overhead. and 16.3× and 1.52× speedup over *UMap* with default parameters and *UMap* with page-size only tuning, respectively.**

***Index Terms*—autotuning, virtual memory, big data, paging, memory-mapped I/O, memory, storage**

I. INTRODUCTION

The massive growth in dataset sizes across multiple domains necessitates the use of high-performance computing (HPC) to compute on such datasets at scale [1]. However, this growth has shifted the bottleneck from the compute subsystem toward the memory and storage subsystems, thus motivating the need for fast storage technologies, such as non-volatile memory (NVM) devices, in supercomputing design.

Fast storage technology can significantly improve the I/O performance of data-intensive applications, but at the expense of programming complexity to handle a complex HPC storage hierarchy [2], [3]. To address this challenge, data processing tools leverage memory-mapped I/O [4]. Memory-mapped I/O, i.e., *mmap()*, provides a unified interface to different storage types and enables applications to handle paging transparently for out-of-core data processing. However, memory-mapped I/O incurs significant performance limitations due to multiple bottlenecks in the Linux I/O subsystem [2], [4], [5]. Moreover, it lacks the flexibility to apply application-specific optimizations because paging parameters are system-wide [2].

To address these limitations, past research has studied handling paging management in userspace [2], [3], [5], [6]. For instance, *UMap*, a userspace memory-mapping library, provides an interface similar to *mmap()* while exposing tunable paging parameters [2]. These parameters, such as page size, degree of concurrency in paging management, and high/low eviction watermarks, are usually tuned for generality and then hard-coded into the OS kernel. *UMap* enables application-specific tuning of these parameters, which can lead to significant performance improvement [2]. Moreover, multiple applications can run on the same system with different per-application configurations.

However, identifying the optimal *UMap* paging parameters requires systems expertise and a deep understanding of an application's characteristics. Such requirements hinder the usability of *UMap*, particularly by domain scientists with limited systems' expertise. Unfortunately, an exhaustive search of this multidimensional *UMap* parameter space is computationally intractable due to the sheer size of the search space, along with the necessary time to evaluate each parameter combination.

Thus, we present AUTOPAGER, a tool for auto-tuning *UMap* paging parameters. In this paper, we leverage Bayesian optimization [7] for auto-tuning. Bayesian optimization is a widely-used algorithm for tuning complex, non-linear systems, where the cost of evaluating the objective function is significant [8]. We selected Bayesian optimization due to the complexity of the *UMap* parameter space and the cost of evaluating each parameter combination for applications with out-of-core processing. AUTOPAGER completed in less than 1% of the time that an exhaustive grid search took on a small dataset. In turn, the parameter-optimized *UMap* achieves up to a 16.3× speedup over *UMap* with its default parameters; in addition, it achieves up to a 1.52× speedup over page-size-only tuning.

We summarize our contributions below:

- AUTOPAGER, a framework for auto-tuning userspace virtual memory parameters.
- Applying AUTOPAGER to *UMap*, an open-source userspace paging tool, and leveraging Bayesian Optimization for parameters tuning.
- Balancing the trade-off between I/O performance and programming productivity (and portability) for designing scalable data processing systems.

The rest of this paper is organized as follows. First, §II provides background about userspace paging and parameter

§Authors have equally contributed to this work.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

auto-tuning, followed by the design AUTOPAGER. Next, §IV evaluates AUTOPAGER using microbenchmarks and real-world applications. Finally, §V, and §VI discuss related work, and future directions, respectively, and §VII concludes.

II. BACKGROUND

In this section, we provide background on userspace memory-mapped I/O, including *UMap* and its parameter space. We then discuss different parameters auto-tuning approaches, highlighting Bayesian optimization as an approach that fits the *UMap* auto-tuning problem.

A. Userspace Memory-Mapped I/O

Memory-mapped I/O allocates virtual memory space and applies demand paging to transfer data between physical memory and persistent storage. Memory-mapped I/O provides a unified interface to different storage types [9]. Additionally, it has multiple performance benefits compared to read/write system calls for fast storage devices [4]. However, memory-mapped I/O, i.e., *mmap()*, suffers from lack of scalability for multithreaded applications [4], and lacks the flexibility for application-specific optimizations [2]. These limitations motivated the design of userspace memory-mapped I/O libraries [3], [6], [9]. Userspace memory-mapped I/O transfers page fault signals to the userspace, enabling application-specific optimizations while avoiding kernel-space bottlenecks. Specifically, *UMap* [9] exposes a wide set of paging parameters for application users to be tuned at runtime. Table I describes the *UMap* parameters and their values. This approach enables application-specific performance tuning. However, it challenges domain scientists who have minimal systems background to select the optimal parameters. To address this issue, we present and evaluate AUTOPAGER, a tool for auto-tuning *UMap* paging parameters. We designed AUTOPAGER to support different optimization algorithms. The complexity of the search space and the non-linearity of the objective functions motivate leveraging Bayesian optimization to identify near-optimal parameters.

B. Optimization Strategies

Optimization is the process of identifying the minimum or maximum of a given objective function $F(x)$. Traditional optimization techniques exploit the features of the objective functions, such as convexity and gradient, to quickly find the optimum. However, some complex objective functions exist that are non-trivial or computationally expensive to find the gradient and that no information other than the output exists. These types of functions are often labeled black-box functions, and the techniques that optimize them are called black-box derivative-free optimization techniques [10]. We discuss common optimization algorithms below:

1) Exhaustive Grid Search: Grid search is an optimization technique that performs exhaustive search over all the possible parameter combinations the objective function can take. In practice, if the objective function is expensive, certain parameter combinations may be pruned based on domain knowledge.

2) Coordinate Descent: Coordinate descent is an iterative method that selects a parameter to optimize for, fixes every other parameter, and then finds the minimum execution time for that parameter. It continues this until all the parameters are minimized. This method performs well when the parameters are independent, but may fail to find the optimal point if the parameters affect each other. Additionally, the order in which parameters are chosen is very important and can significantly affect the result. [11]

3) Bayesian Optimization: Bayesian optimization is one such technique that relies on the optimization of a surrogate model $I(x)$ that mimics the objective function $F(x)$. The surrogate model is continuously trained by sampling points using an acquisition function [12].

Algorithm 1: Bayesian Optimization

```
Initialize surrogate model I(x)
i = 1
while i ⩽ N do
    x_i = argopt_x I(x|(x_{1:i−1}, y_{1:i−1}))
    y_i = F(x_i)
end
```

The most commonly used surrogate model for Bayesian optimization is the Gaussian process [13]. The acquisition function is a heuristic function that determines the sampling points for the surrogate model by identifying the optimum point of the surrogate model. It balances exploitation, i.e., sampling points with a high probability of reward, and exploration, i.e., sampling points in regions of high uncertainty. Some commonly used acquisition functions are *probability of improvement*, *expected improvement*, and *lower confidence bound*. Finally, a Bayesian optimizer could be warm-started. Warm-starting is the process of transferring the learned parameters from one model to another model with similar computational patterns for faster convergence [14]. Bayesian optimization suits the complexity, non-linearity, and objective function cost of the *UMap* parameter space.

III. DESIGN OF AUTOPAGER

The AUTOPAGER tool consists of an objective function definition and an optimization workflow. The objective function we seek to optimize consists of an application's performance metric (e.g., execution time) as a function of the *UMap* paging parameters. The optimization workflow of AUTOPAGER consists of iteratively selecting and evaluating different *UMap* parameter combinations, then evaluating an optimization function (e.g., surrogate model). The objective function evaluation consists of running the target application as an external process and collecting the performance metric of interest. We define our parameter space as a Cartesian product of all possible *UMap* parameter combinations. We implement the Bayesian Optimization using the *Ax* library [15] by Facebook, using its default parameters. We discuss the details of the objective functions, workflow implementation, and validation methodology in the rest of this section.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

TABLE I: *UMap* userspace paging parameters that are tunable at application runtime

Parameter	Possible Values	Description
Page Size	Multiples of system's page size (4 KB)	Unit of transfer between DRAM and persistent storage
Page Fillers	1 to number of physical cores	Number of threads fetching pages from storage to DRAM
Page Evictors	1 to number of physical cores	Number of threads evicting and writing back dirty pages
Buffer Size	1 to (90% of available DRAM) (in pages)	Maximum memory buffer size in pages
Eviction High-Water Threshold (HWT)	1% to 100% of pages in buffer	Percentage of dirty pages at which eviction is triggered
Eviction Low-Water Threshold (LWT)	1% to HWT - 1 (where HWT $>$ LWT)	Percentage of dirty pages at which eviction is stopped

A. Objective Functions

Applications are designed with different use cases and constraints, so identifying performance metrics can become difficult. Metrics such as execution time, throughput, and peak memory utilization all play an important role in determining the effectiveness of a target application. We focused on execution time and throughout as the core objective functions in this work. We model the metric of interest as a function of the *UMap* parameters. For instance, let the objective function denote the application's execution time, and let U denote the *UMap* parameter space described in table I. The objective function could be represented as $T_{app} = f(x)$ where $x \in U$. In this case, we consider all other factors affecting the execution time, such as input size and algorithmic complexity, to be invariant for each optimization iteration.

B. Optimization Workflow

Figure 1 generally describes the AUTOPAGER iterative workflow to optimize the *UMap* parameters using a given optimization strategy. We use a Cartesian product of the parameter space for grid search. We uniformly sample each parameter dimension to perform the grid search in a tractable time. For Bayesian optimization, we sample 10 random points to initialize the model and then iteratively train the model for 20 more iterations.

The objective function depends on the application. In this work, we optimized either wall clock execution time or throughput. The Bayesian optimization software runs as a Python process that sets the selected *UMap* parameters as environment variables, then invokes the *UMap*-enabled application as an external process. To ensure we measure our metrics accurately, we disable page cache and swap on machines with super-user access. We run a script to overwrite the page cache on machines without root privileges. For small dataset experiments, we limit the Buffer Size parameter to push the *UMap* experiments out-of-core while not using swap.

C. Validation

Because an exhaustive grid search on large datasets is unfeasible, we take a smaller segment of the problem by reducing the dataset size and running an exhaustive search and the Bayesian optimizer on it. We compare the optimal point found by the exhaustive search to that found by the Bayesian optimizer. This technique allows us to compare the points found by the various optimization strategies. We also compare the optimal point found by the optimizers for the smaller dataset segment with that found for the large dataset.

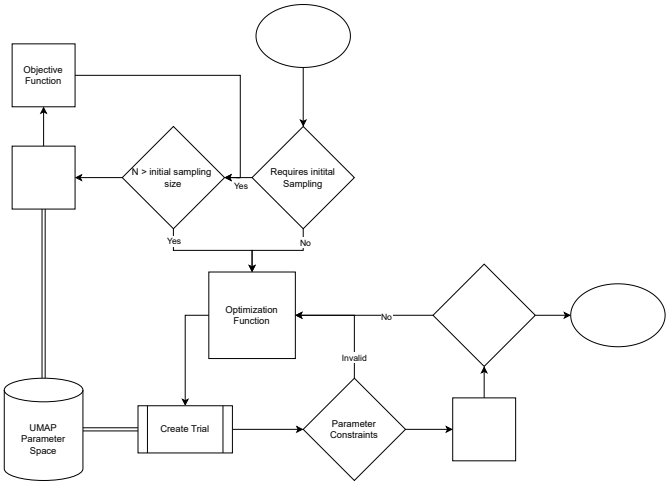


Fig. 1: Architecture of AUTOPAGER. Computing the objective function refers to running the application with *UMap* and collecting the performance metric of interest (e.g., total time).

This approach allows us to understand the scaling of the performance and predict if the optimization on a smaller segment carries forward to its larger variant, which could save significant optimization time. Per-application validation details are explained in §IV.

IV. EVALUATION

We evaluated AUTOPAGER using five data-intensive applications: two micro-benchmarks, and three real-world applications. These applications represent diverse memory and storage access patterns. We evaluated each application using either their inbuilt benchmarking suite or a standard benchmark tool for that application type.

To compare AUTOPAGER's Bayesian optimization with an exhaustive search, we evaluated each of the five applications using a small dataset. We sampled each parameter space dimension to create a restricted search space to run exhaustive tuning in a tractable time. For instance, we chose page sizes between 4 KB and 16 MB with increment powers of two, fillers and evictors between 1 and maximum number of cores in increment powers of two, and eviction thresholds in 10% increment. It is worth noting that for different applications, and based on our initial investigation, we further eliminated parts of the sampled search space that we found to degrade performance significantly. For instance, running the LMAT genome analysis tool with a page size larger than 64 KB

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

caused significant performance degradation. We describe the per-application search space constraints under each application's description later in this section.

A. Applications

*1) **BFS**:* Breadth-first search (BFS) represents a core component in graph analytics. We used an existing memory-mapped BFS implementation shipped with *UMap*'s open-source release. We created a synthetic bidirectional graph using an R-MAT [16] graph generator and measured the time it takes to perform BFS on the entire graph starting from the vertex with ID 0.

*2) **QuickSort**:* Sorting is an essential building block in database systems and many data analytics workloads. In this work, we used *umapsort*, a sorting benchmark shipped with *UMap*'s open-source release, to evaluate AUTOPAGER. The *umapsort* benchmark creates a large array of long integers in reverse order and sorts it in ascending order using `__gnu_parallel::sort`. The objective function is the time taken to finish the sort.

*3) **LMDB**:* The Lightning Memory-Mapped Database (LMDB) [17] is a widely used key-value data store by many data analytics and machine learning frameworks, such as PyTorch. We used *dbbench*, a benchmarking tool provided by LevelDB [18], to evaluate and tune LMDB+UMap. The *dbbench* tool consists of reading and writing benchmarks with three different access patterns; (a) random, (b) sequential, and (c) reverse. Preliminary results indicated that *UMap* significantly improved LMDB throughput for the read workloads compared to baseline *mmap()*. On the other hand, *mmap()* yielded a higher throughput for write workloads. Hence, in this work, we focused on auto-tuning the read workloads for LMDB+UMap.

*4) **BLAST**:* The Basic Local Alignment Search Tool (BLAST) [19], [20] is the most widely used biological sequence comparison tool to date. BLAST implements a highly optimized memory management layer based on memory-mapped I/O to read the sequence database. However, recent studies, including [21], have shown that paging significantly degrades BLAST's performance when the database does not fit in memory. While distributing the sequence database across multiple nodes [21], [22] circumvents paging, it introduces high network overhead for processing significantly large output. Alternatively, we explore leveraging *UMap* with optimized parameters to mitigate paging overhead. We evaluate AUTOPAGER using SparkBLAST [23], a distributed BLAST implementation using Spark that partitions the query sequence and replicates the database to each compute node.

*5) **LMAT**:* The Livermore Metagenomic Analysis Toolkit (LMAT) [24] is an open-source tool for scalable metagenomic taxonomy classification. LMAT uses memory-mapped files to load large genome databases and enable out-of-core processing. We used an open-source version of LMAT that uses *UMap*. We used a real-world query consisting of nearly six million sequences in FASTA format. The query is searched against a 480 GB k-mer database. We used a random sample of the query file for the small dataset experiment that consisted of ten thousand sequences. For the large dataset experiment, we used the entire query file.

TABLE II: Hardware and systems specifications

CPU (cores)	Memory	Storage	Kernel
Intel Xeon Platinum 9242 (96)	384 GB	3.2 TB NVMe	3.10
AMD EPYC 7702 (128)	256 GB	480 GB SSD	3.10
AMD EPYC 7401 (48)	256 GB	1.6 TB SSD	5.10.28

B. Computing Platforms

We conducted our experiments on three different computing platforms. We used compute nodes of a production cluster with Linux kernel 3.10 for read-only benchmarks, and specific clusters with Linux kernel 5.10.28 for read/write benchmarks. Table II describes the computing platforms used in running our experiments.

C. Experiment Setup

- **Default *UMap* Parameters:** We note the performance of each application using the default *UMap* parameters.
- **Exhaustive:** We run the applications on a sample of the complete parameter space and note the best point found.
- **Page Size Only:** We measure the performance of the applications on a line-search along the page size parameter and note the best point found.
- **AUTOPAGER:** We optimize the parameter configuration of the applications with AUTOPAGER. We note the best performing parameter configuration as our result.

We evaluated all the aforementioned setups for the small datasets. For the large datasets, we did not evaluate exhaustive tuning since it was computationally intractable. We compared the net performance gain from the parameter configuration, as well as the time taken to identify that configuration, for both the small data and large data. The best parameters found by AUTOPAGER for all applications are shown in table III.

D. Results

*1) **BFS**:* We generated R-MAT graphs of scales 26 (34 GB) and 31 (519 GB) with a connectivity of 16. We measured the time taken to find element 0 and recorded that as the objective function. We ran the scale 26 small dataset with a limited *UMap* Buffer Size of 16 GB to force out-of-core execution. For the small dataset, we get an 8.26× speedup using exhaustive grid search compared to *UMap* with default parameters. Using AUTOPAGER, we get a 7.69× speedup compared to default parameters. For the scale 31 large dataset, the best parameters found by AUTOPAGER achieved 6.84× speedup compared to default *UMap* parameters, while page size tuning achieved 6.29× speedup. Hence, AUTOPAGER achieved 1.08× speedup compared to coordinate descent tuning on the page size parameter.

*2) **Quicksort**:* Using the *umapsort* benchmark, we optimized the *UMap* parameters for sorting a small array of 16 GB and a large array of 512 GB. For the small array, we limited the *UMap* buffer size to 8 GB to force out-of-core execution.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

For exhaustive tuning, we started the page size range from 128 KB, since previous *UMap* performance studies showed that smaller page sizes significantly degraded performance. The best parameters found by AUTOPAGER achieved comparable performance to those found by exhaustive tuning (3.9% slower). On the other hand, these parameters achieved 19.29× speedup compared to the default *UMap* parameters, and 1.23× speedup compared to page-size only tuning. For the large dataset, the best-found parameters by AUTOPAGER achieved 16.34× speedup compared to the default *UMap* parameters, and 1.06× speedup compared to page-size only tuning.

*3) **LMDB**:* The optimization goal for LMDB is to maximize the throughput. We created a 49 GB database to run exhaustive tuning and limited the *UMap* buffer size to 32GBs. For the large experiment, we created a 384 GB database. As shown in figure 2a, for the small dataset, the best parameters identified by AUTOPAGER achieved 6.49×, 6.6×, and 3.9× higher throughput compared to the default *UMap* parameters for *readseq*, *readrandom*, and *readreverse* respectively. These parameters also yielded 1.1X, 1.338×, and 1.332× better throughput compared to exhaustive tuning for *readseq*, *readrandom*, and *readreverse*. For the large datasets, the best parameters identified by AUTOPAGER achieved 6.25× and 6.37× better throughput compared to the default *UMap* parameters for *readseq* and *readreverse* respectively, as shown in figure 2b. For the *readrandom* benchmark, it was shown that the best parameters found by the Bayesian optimizer achieved a slowdown of 1.42× compared to the default *UMap* parameters. It also yielded a similar slowdown compared to page-size-only tuning, since the default page size achieved the best performance. Conversely, for *readseq* and *readreverse*, the speedups of Bayesian optimization compared to page size tuning were 1.2× and 1.24×, respectively.

*4) **BLAST**:* To compare with exhaustive tuning, we used a small dataset that consists of a randomly sampled subset of the query file *Geobacter Metallireducens* against the *large environmental sequencing projects database (env_nr)* on a single node. For the large dataset, we searched the entire query file against the *non-redundant protein database (nr)* using sparkBLAST to partition the query and run it on 16 nodes of an HPC cluster. We measured the total query execution time as our objective function. For the small dataset, exhaustive tuning, page size tuning, and AUTOPAGER achieved an 7.59X, 8.15X, 8.2X speedup respectively against the default parameters. For the large dataset, page-size tuning and AUTOPAGER achieved an 1.95X and 2.36X speedup respectively against the default parameters.

*5) **LMAT**:* To compare with exhaustive tuning, we used a subset of 10K sequences sampled from the query file. We optimized the *UMap* parameters for searching the query subset against the full database using exhaustive tuning, page size tuning, default *UMap* parameters, and AUTOPAGER. Figure 2a shows the performance comparison of exhaustive tuning, page size tuning, and AUTOPAGER to the default *UMap* parameters. For the small dataset, AUTOPAGER yielded 1.35×, 1.06×, and 1.35× speedups compared to default parameters, exhaustive tuning, and page size tuning. While the random I/O access pattern of LMAT favors smaller page-sizes, tuning only the page size yielded suboptimal performance. For the large dataset results, we searched the entire query file against the full database. Figure 2b shows the performance comparison of different tuning techniques to the default parameters. For the large dataset, AUTOPAGER yielded a 2.32× speedup compared to default *UMap* parameters, and 1.29× speedup compared to page-size tuning.

E. Tuning Overhead

The best parameters found by AUTOPAGER achieved comparable performance to those found by exhaustive tuning. However, AUTOPAGER found these parameters significantly faster. Specifically, AUTOPAGER identified the best parameters in a time that is between 10× and 156× less than the time exhaustive tuning took. It is also worth noting that the AUTOPAGER tuning overhead is amortized since the best-found parameters could be re-used. For instance, different queries with the same BLAST database could re-use the best-found AUTOPAGER parameters.

V. RELATED WORK

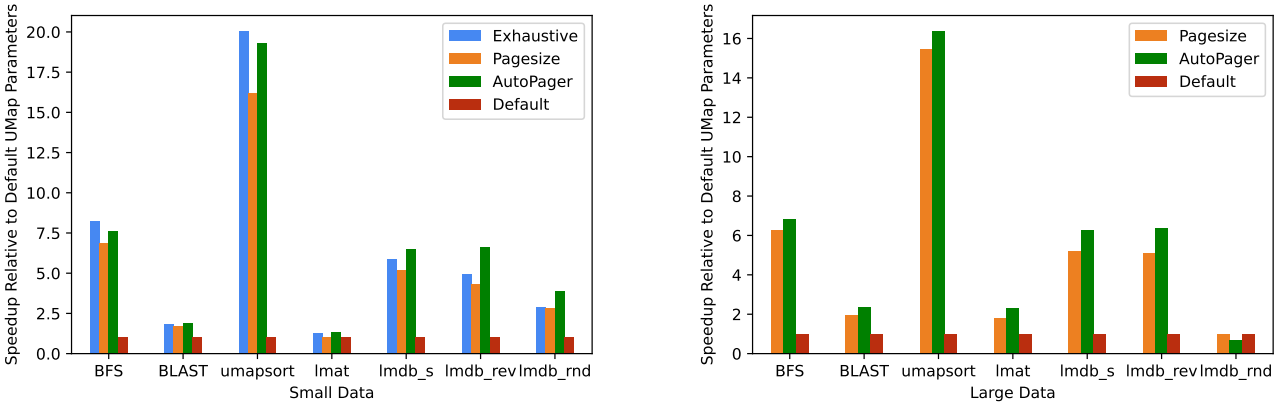
We discuss related work in the directions of userspace memory-mapped I/O, tuning memory management parameters, and, more generally, tuning different aspects of high-performance computing systems parameters.

Userspace paging was proposed to overcome the performance limitations incurred by system-level *mmap()* [2], [4]. The most recent userspace memory-mapped I/O tools include Userland CO-PAGER [5], uMMAP-IO [3], and *UMap* [2]. The advantages of *UMap* include higher flexibility for application-specific tuning by exposing a larger number of paging parameters. This flexibility motivated the design of AUTOPAGER. Moreover, *UMap* uses *userfaultfd* [25] to capture page faults, which incurs less overhead than the *sigaction*-based fault handling [26] that was used by the other solutions.

Tuning paging parameters have been recently explored by Park et al., who devised a performance model that estimates different workload performance as a function of the page size [27]. Their work yielded a 38% performance improvement when using their model to select the optimal page size for a virtual machine on a virtualized system. As shown in our evaluation, the page size is not the only parameter that affects application performance. Simultaneously modeling other paging parameters using a white-box model is not feasible. This motivated the use of Bayesian optimization in our case. Moreover, tuning paging parameters in userspace enables multiple applications running on the same system to be tuned differently without affecting the global system environment.

Auto-tuning memory management parameters was shown to be successful when applied to memory configurations of distributed data processing frameworks. In [28], a hybrid model was devised that consisted of a Bayesian optimizer guided by an analytical model. The parameter space consisted of memory configurations of the Java Virtual Machine (JVM)

978-1-6654-9786-2/22/$31.00 ©2022 IEEE



(a) (b)

Fig. 2: Speedups relative to the default *UMap* parameters for (a) small datasets and (b) large datasets.

TABLE III: Best parameters found by the Bayesian optimizer for each application. Buffer size was fixed for small data.

	Page Size (KB)		Page Fillers		Page Evictors		Evict HW Threshold		Evict LW Threshold		Buffer Size %	
	Small	**Large**	**Small**	**Large**	**Small**	**Large**	**Small**	**Large**	**Small**	**Large**	**Small**	**Large**
BFS	128	64	66	62	63	57	100	100	99	40	47	90
BLAST	32	128	1	53	85	128	100	77	59	34	75	90
QuickSort	16384	16384	10	36	1	1	98	78	51	64	50	87
LMAT	8	4	16	48	64	3	77	74	11	15	50	80
LMDB ReadSeq	16384	16384	76	1	76	79	91	51	22	41	65	86
LMDB ReadRandom	16384	128	27	47	28	3	87	73	66	15	65	84
LMDB ReadReverse	16384	16384	1	1	47	34	67	27	60	1	65	84

parameters. In our work, we optimize memory-mapped I/O parameters for more generic workloads.

Auto-tuning systems parameters is of significant importance due to the increased complexity of modern systems. For instance, HPC systems expose multiple tunable parameters at the hardware, compilers, system software, and application layers. The work in [29] studies auto-tuning of different HPC application parameters using Bayesian optimization. At the hardware level, accelerators are examples of systems with complex parameters, where application developers need to perform prohibitive tuning to leverage their performance. There have been multiple research achievements to auto-tune GPU parameters, including [30], [31]. To the best of our knowledge, auto-tuning virtual memory paging parameters is under-explored in this field, which motivated the design of AUTOPAGER for tuning the *UMap* userspace paging parameters.

VI. FUTURE DIRECTIONS

Our findings regarding auto-tuning the *UMap* paging parameters open multiple future directions, including other dimensions for the optimization space, guided auto-tuning, and optimizing paging for multi-tiered storage systems. UMap supports a fine-tuned eviction policy that allows selecting pages to be fetched or evicted. Combining predictive pre-fetching with adaptive parameters auto-tuning could be explored. Also, AUTOPAGER can be guided with domain knowledge and heuristics that can yield efficient convergence. One approach is by pruning parameter combinations that the expert knows will not perform well. Also, a workload characterization could be performed, where per-workload-category heuristics could be applied to guide AUTOPAGER. Moreover, a white-box model could be devised for each application category that guides AUTOPAGER parameter selection, e.g., guided Bayesian optimization [28]. Finally, large-scale data analytics require efficient utilization of deep memory and storage hierarchies [32]. Tuning *UMap* parameters for multi-tiered storage systems is a promising future direction.

VII. CONCLUSION

We present AUTOPAGER, a tuning methodology for optimizing paging parameters in the userspace. We apply AUTOPAGER to *UMap*, a userspace memory mapping tool. We show that tuning the paging parameters of *UMap* using AUTOPAGER leads up to a 16.34× performance improvement over the default *UMap* parameters. AUTOPAGER optimizes the trade-off between the programming productivity of *mmap()* and the performance gained by complex application-specific memory management designs. We also show that the parameter space for optimization is enormous, and it is often intractable to perform a grid search on it. We highlight the pertinence of Bayesian optimization in automatically tuning the parameters. We show that Bayesian optimization with AUTOPAGER can find parameters that achieve 1.52× performance improvement compared to page-size only tuning, which was shown to be the most effective paging parameter.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-839425). Funding from LLNL LDRD project 21-ERD-020 was used in this work. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

REFERENCES

[1] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[2] I. B. Peng, M. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce, "Enabling scalable and extensible memory-mapped datastores in userspace," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[3] S. Rivas-Gomez, A. Fanfarillo, S. Valat, C. Laferriere, P. Couvee, S. Narasimhamurthy, and S. Markidis, "ummap-io: User-level memory-mapped i/o for hpc," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 363–372.

[4] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 813–827.

[5] F. Li, D. G. Waddington, and F. Song, "Userland co-pager: boosting data-intensive applications with non-volatile memory, userspace paging," in *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*, 2019, pp. 78–83.

[6] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "Dimmap—a scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, vol. 18, no. 1, pp. 15–28, 2015.

[7] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[8] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast Bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 528–536.

[9] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "Umap: Enabling application-driven optimizations for page management," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 71–78.

[10] J. Larson, M. Menickelly, and S. M. Wild, "Derivative-free optimization methods," *Acta Numerica*, vol. 28, pp. 287–404, 2019.

[11] H.-J. M. Shi, S. Tu, Y. Xu, and W. Yin, "A primer on coordinate descent algorithms," 2016. [Online]. Available: https://arxiv.org/abs/1610.00040

[12] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems*, vol. 24, 2011.

[13] E. Schulz, M. Speekenbrink, and A. Krause, "A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions," *Journal of Mathematical Psychology*, vol. 85, pp. 1–16, 2018.

[14] V. Perrone, R. Jenatton, M. W. Seeger, and C. Archambeau, "Scalable hyperparameter transfer learning," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[15] E. Bakshy, L. Dworkin, B. Karrer, K. Kashin, B. Letham, A. Murthy, and S. Singh, "Ae: A domain-agnostic platform for adaptive experimentation," in *Conference on Neural Information Processing Systems*, 2018, pp. 1–8.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[17] G. Henry, "Howard chu on lightning memory-mapped database," *IEEE Software*, vol. 36, no. 06, pp. 83–87, 2019.

[18] L. Wang, G. Ding, Y. Zhao, D. Wu, and C. He, "Optimization of LevelDB by separating key and value," in *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2017, pp. 421–428.

[19] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[20] T. Madden, "The blast sequence analysis tool," *The NCBI Handbook*, 2003.

[21] K. Youssef and W.-c. Feng, "SparkLeBLAST: Scalable parallelization of blast sequence alignment using spark," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 539–548.

[22] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and I/O in massively parallel sequence search," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 529–543, 2010.

[23] M. R. de Castro, C. d. S. Tostes, A. M. Dávila, H. Senger, and F. A. da Silva, "Sparkblast: scalable blast processing using in-memory operations," *BMC bioinformatics*, vol. 18, no. 1, pp. 1–13, 2017.

[24] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen, "Scalable metagenomic taxonomy classification using a reference genome database," *Bioinformatics*, vol. 29, no. 18, pp. 2253–2260, 2013.

[25] T. kernel development community. Userfaultfd. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html

[26] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[27] Y. Park and H. Bahn, "Modeling and analysis of the page sizing problem for NVM storage in virtualized systems," *IEEE Access*, vol. 9, pp. 52 839–52 850, 2021.

[28] M. Kunjir, "Guided Bayesian optimization to autotune memory-based analytics," in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019, pp. 125–132.

[29] 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)*. IEEE, 2020, pp. 831–840.

[30] X. Cui and W.-c. Feng, "Iterative machine learning (iterml) for effective parameter pruning and tuning in accelerators," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 16–23.

[31] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.

[32] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager *et al.*, "Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1033–1042.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE