Parametric Optimization of Storage Systems

E. Zadok, A. Arora, Z. Cao, A. Chaganti, A. Chaudhary, and S. Mandal
Stony Brook University

Abstract

Most storage systems come with large set of parameters to directly or indirectly control a specific set of metrics that may include performance, energy, etc. Often, storage systems are deployed with default configurations, rendering them sub-optimal. Finding optimal configurations is difficult due to the numerous combinations of parameters and parameter sensitivity to workloads and deployed environments. Previous research on parameter optimization was either limited to narrow problems or not widely applicable to storage stack parameter optimization in general. Based on promising early results, we propose using meta-heuristic techniques such as genetic algorithms to efficiently find near-optimal configurations for storage systems.

1 Introduction

Modern storage systems are often characterized by multiple connected hardware units with different software versions running user application workloads. There are many user controlled parameters at every layer that affect performance, energy, and more. Often, these systems are deployed with default settings that directly come from the vendor mostly due to two reasons: (1) administrators cannot be expected to know every parameter's effect across multiple layers and (2) vendors' default configurations are trusted to be "good enough." We showed previously that even a tiny set of parameter changes can lead to multi-fold difference in terms of performance and energy [12]. As Moore's law has slowed down, it is more important to squeeze every bit of performance out of deployed systems.

Finding optimal configurations is challenging. The first problem is the sheer size of the parameter space to try. For instance, we considered a storage system with NFS, and included only 18 useful tunable parameters (NFS version, r/wsize, a/sync, no_/wdelay, etc.). Based on their boolean, integer, or discrete values, there are around 10^{18} configurations to try. Such numbers show that it is impossible to exhaustively search these spaces even for a small set of parameters. Simple random sampling techniques do not scale in these large spaces. We need efficient and practical techniques to search the space.

The second problem is the optimization point's sensitivity to the environment and workload. Some parameters are important to optimize; others have insignificant impact and are a waste to try; and some are highly correlated with others and such dependencies must be discovered. The set of parameters that affect performance depends heavily on the combination of hardware, software, *and* workloads. Even a small change to one part of the environment can deviate the system away from its optimal performance. This problem results in a complex, very large, multi-dimensional search space with many suboptimal local maxima.

Related Work. Existing research in this field of optimization focuses on specific problems or uses methods that do not scale with the parameter space. Wang et al. investigated the application of a *machine learning* (ML) based methods for black-box storage device modeling and prediction [13]. The quality of ML models depends heavily on the amount and quality of training data which is either not available or difficult to generate for large spaces. Others including our group used *Control Theory* (CT) [9] to model energy consumption and performance of computing systems and found several problems. Alas, CT is ineffective for non-linear systems with non-numeric parameters. Like ML, CT (and variants we investigated) also requires training data and often handle only a small number of parameters.

It is impractical to manually explore large parameter spaces for optimizing storage systems. We propose to use *Meta-heuristic* (MH) stochastic search optimization techniques [11] to address this problem. Though the application of MH is not completely new in systems, their usage has been confined to building cost-efficient design and management solutions for storage systems [2, 4, 7]. To our best knowledge, MH is not used in parametric optimization of storage systems for IO performance. Based on promising early results, we believe that MH techniques have the potential to solve a wide range of storage system parameter optimization problems efficiently.

2 Background

Searching Large and Complex Spaces. Our search space has three unique and challenging properties: **(1)** It is very large: exploring even a major fraction exhaustively is impossible. A human expert cannot know where to search because people have specialized as experts (e.g., database vs. networking experts). **(2)** The space is very sparse. A vast majority of parameter combinations leads to bad configurations with poor results. Good search techniques must avoid these vast regions of "dead" search space: pure random search is a poor strategy. **(3)** It contains many local maxima that sometimes cluster together. Traditional search techniques, such as Hill Climbing, try to quickly walk up to the top of a

maxima—but often get stuck at local maxima, missing out better maxima elsewhere.

Meta-heuristic (MH) [11] search and optimization algorithms are a class of algorithms including Evolutionary Algorithms (EAs) [5, 6], Simulated Annealing (SA) [8], Particle Swarm Optimization, Tabu Search, Random Restart Hill Climbing, Memetic Algorithms, and variants thereof. MHs largely vary in two key properties: *exploration* vs. *exploitation*. **(1) Exploration** is how much the technique searches the space randomly. This often includes a combination of pure and guided random search in a given neighborhood—sometimes called a Monte Carlo process. **(2) Exploitation** is how much the technique leverages its neighborhood. When in a certain configuration, we try to find nearby better ones.

MH techniques have a small number of configurable parameters to guide the search technique more effectively. For instance, GAs have population sizes, crossover methods, and mutation rates to set. No single technique with specific properties can suit all environments and workloads. Still, we believe it is more effective to tune the fewer GA parameters than the numerous parameters of storage systems.

Genetic Algorithms. We explored a subclass of MHs called Evolutionary Algorithms (EAs), specifically Genetic Algorithms (GAs). Carefully used, GAs (and other MHs) have the ability to search large spaces very rapidly and zoom in on the right set of parameters and their values with the best fitness (e.g., performance) in a given environment. Theory developed by Holland [6] suggests that large search spaces can be reduced exponentially by finding dependent parameters and their values that go well together, and considering them as one.

GAs have several traits taken directly from nature and define nine key terms: **(1) Gene** represents a single configuration parameter, such as the number of threads a Mail server uses. A specific value of a gene is an *allele*. Genes can be of any size: a single bit can represent a Boolean choice; 8 bits can represent a numeric value from 0–256; or 2 bits can represent a choice between four file systems. **(2) Chromosome (or DNA)** is the schema that defines a configuration. A configuration is defined as a precise series of genes that code for various *traits* (e.g., hair color or amount of RAM). **(3) Population** is a set of unique configurations (chromosome instances) that are being evaluated. A population can grow or shrink over time. **(4) Environment** includes the underlying storage hardware, software and configuration, and the workloads being exercised. Environments change over time: software is upgraded, hardware wears out, and workloads change too. **(5) Fitness**: each configuration in a population is evaluated for its fitness in a given environment. We can define fitness as I/O throughput, CPU speed, energy used, and even complex cost functions [10] such as reliability, consistency, security, weighted combinations of these, etc. **(6) Generation**: the fitness of each population member is evaluated in the given environment. This is a single generation. **(7) Selection**: once the fitness of a population is evaluated, a selection process begins to determine the *next* generation. Key to GAs is their ability to *reinforce* better configurations: they live on and multiply in the next generation; and those that did poorly are discarded. Reinforcement is said to enable GAs to search vast spaces exponentially [6]. **(8) Crossover**: in Earth biology, the male and female of the species combine their respective genes in one or more offspring. This is called *crossover* and governs the GA's exploitation aspect. Crossover ensures that we continue to exploit configurations similar to the successful ones we already selected. **(9) Mutation**: in Earth biology, genes mutate regularly. Mutation modifies a gene randomly. Without mutation, a population is doomed to be stuck in exploring a narrow neighborhood and never stumble upon a much better neighborhood to explore. In nature, mutations often follow the power law: most are small but a few can be large [1]. Large mutations are important to allow a population to explore well outside its immediate neighborhood.

GA operation. A genetic algorithm process begins by selecting an initial population, which can be seeded with already known reasonable default configurations. An environment is setup: storage hardware, software, and the workload to evaluate. The fitness (e.g., performance, energy) of each population member is then evaluated. After the conclusion of this first generation, selection begins: some members are kept, some discarded, better ones are reinforced and crossed over with others, etc. The goal of GA is to find better configurations for the given environment. The longer you run the GA, the better configurations it would find. However, the speed at which it finds better configurations varies a lot. We know from our experiments that GAs can find near-optimal configurations faster than purely random search or exhaustive searches of the space, but there is no clear indication from the experiments as to how long it can take. In very large spaces, it is not guaranteed that a GA—or any search algorithm—can find a global optimum. But as long as we keep finding better and better configurations than default ones, then the systems' performance continues to improve.

3 Case Study: Optimize Storage Systems

Hardware setup. We first describe experiments on a 7-year-old server to show that even aging hardware can be useful after applying GA: a Dell PowerEdge SC1425 server with two Intel Xeon single-core 2.8GHz CPUs,

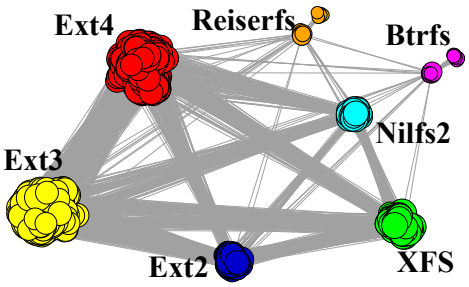

Figure 1: Configuration connectivity. Using a single mutation, an Ext4 configuration can easily reach an Ext3 one; but file systems such as Reiserfs and Btrfs have fewer reachable paths.

2GB RAM, and two 73GB Seagate ST373207LW SCSI drives.

Software. We used *Filebench* [3], to emulate I/O activity. Filebench is versatile and comes with multiple pre-configured *personalities*: we used its mailserver, fileserver, database, and webserver workloads. For our experiments, we formatted and mounted a storage device with specific file system, format, and mount options and then ran Filebench on it.

File systems used. We use a fitness value that is directly related to I/O performance, so we naturally considered the file system as the key parameter [12]. Based various factors such as the on-disk structures (BTree, LFS, FFS), user base, active development and support in all major Linux distributions, we chose the following seven file systems and their key parameters: **Ext2, Ext3, Ext4, Xfs, Btrfs, Reiserfs** and **Nilfs2**.

Applying GA. We used two chromosome versions in our experiments; see Table 1. The *Storage v1* chromosome defines these genes (or parameters): file system, block size, inode size, blocks per group, and options (mount, journal, etc.). Most file systems have the aforementioned common set of parameters, so we chose this design to generically represent any file system's configuration. Some genes take on powers-of-2 values within permitted ranges: block size, inode size, and blocks per group. The mount option is a Boolean to turn on/off flags such as `atime`. The journal gene supports three journaling modes. *Storage v2* adds the I/O scheduler (`deadline, noop, cfq` and disk-type (SAS, SATA, SSD) genes.

Every file system can have certain parameters that are unique to it. Currently, we do not create a new gene field for every such unique parameter; instead we defined a generic *special* options gene which holds such unique parameters for all file systems. For example, to define a chromosome for ReiserFS's unique `notail` option, we use the special options gene for it; this lets us define Btrfs's `compress, nodatacow, nodatasow` options. Certain combinations of genes could produce a "bad chromosome" or an *invalid* configuration. For example, journaling options make no sense for Ext2 because it does not have a journal. To handle this, we added a value *none* to the existing range of some genes. Any gene with *none* value is considered void. Since we cannot benchmark invalid configuration combinations, we give them a zero fitness value, which ensures they are purged in an upcoming generation.

Table 1 shows that total number of valid configurations in *Storage v1* is 2,074: 184 Ext2, 736 Ext3, 736 Ext4, 216 Xfs, 162 Nilfs2, 24 Btrfs, and 16 Reiserfs configurations. So if we run a GA over a completely random population, with a high probability, valid Ext3 and Ext4 configurations will dominate the file systems explored; few valid configurations like Btrfs and Reiserfs would be explored. Most of the Btrfs and Reiserfs configurations formed from a single gene (parameter) mutation from Ext3 or Ext4 often produce an invalid configuration. Only few transformations from other valid configurations can lead into valid Btrfs and Reiserfs configurations. We call this a *configuration connectivity problem*. Figure 1 illustrates all 2,074 valid configurations in our *Storage v1* as an undirected graph. Each node represents a single file system configuration. Each edge represents whether two configurations can be reachable from each other with just a single mutation. We can see that Ext3 and Ext4 dominate the space and are densely connected. Because uniform random selection can thus be lopsided, we initialized the population such that the default configuration of every file system is present. This way hard-to-reach configurations get a chance to participate and be reinforced over generations if they perform better. For all experiments, we used a population size of 80, the Roulette wheel method to select chromosomes for crossover, and a single point crossover with a probability 0.9. We mutated chromosomes uniformly at random with a rate of 0.02. These values are often used in the literature.

Results. We experimented with several GA parameters, two chromosomes (see Table 1), four workloads, and three classes of machines. For brevity, we report only a subset of representative results here. Table 2 summarizes the results of Filebench's mailserver workload using GA and the *Storage v1* chromosome of Table 1. We ran GA five times, each for 200 generations. We also exhaustively ran all 2,074 *Storage v1* configurations on all four workloads, to identify the *actual* global best configuration. We ran each of the 4 workloads multiple times to ensure statistical stability; this exhaustive set of tests took nearly 50 clock *days* at the rate of approximately 4 days per single run of each workload. With just the addition of two parameters in *Storage v2*, it is evident that run time for a single workload goes up by

Configuration Type	#Useful Params.	#Unique Useful Configs.	Example Useful Params.
Storage v1	7	2,074	file system (e.g., Ext4), block size (e.g., 4KB), inode size (e.g., 256B), block groups size (e.g., 128MB), mount options (e.g., `noatime`, `journal=ordered`)
Storage v2	9	18,666	Storage v1 + I/O scheduler type (e.g., deadline, CFQ) + Disk type (e.g., SSD, SATA)

Table 1: No. of useful parameters and configurations that can be explored for performance optimization

#Run	Throughput ops/s	File System	Block Size	Blocks per group	Inode Size	Mount Options	Journal Options
Run 1	1,639	Ext4	2,048	4	128	noatime	journal=writeback
Run 2	3,669	Nilfs2	2,048	256	-	atime	order=relaxed
Run 3	1,903	Ext2	4,096	8	512	noatime	-
Run 4	3,677	Nilfs2	2,048	256	-	atime	order=relaxed
Run 5	3,677	Nilfs2	2,048	256	-	noatime	order=relaxed
Default	1,420	Ext4	4,096	32	256	atime	journal=ordered
Best	3,669	Nilfs2	2,048	256	-	atime	order=relaxed

Table 2: Results of GA experiments for Mailserver workload on Storage v1 chromosome

12×; thus exhaustive explorations of the space become quickly impractical.

Table 2 shows that in 3 of 5 runs, Nilfs2 produced a better configuration than the default Ext4 one: 2.6× better throughput. In fact, GA actually found the global best throughput configuration twice (modulo small standard-deviation fluctuations). In run 1, GA picked Ext4 but with different options that was still 15% better than Ext4's default options. Surprisingly, in run 3, GA picked the old Ext2 file system and a throughput 16% better than in run 1. These results demonstrate GA's ability to find significantly better—and sometimes surprising—configurations than default ones. For the three runs that selected Nilfs2, the mount option `noatime` was selected only once, but that did not impact overall throughput. We can conclude that `noatime` is unimportant when running the mailserver workload on Nilfs2.

Figure 2 details the results summarized in Table 2. The top figure shows the average throughput of the population at the end of each generation, over 200 generations, for all five runs. The fluctuations seen are a natural outcome of GA's random exploration of the search space. The middle figure is similar but instead records only the best maximum throughput found up to the given generation. The stair-step pattern represents plateaus in our search terrain with higher fitness as newer configurations are found. We can see that once GA found a better configuration, it is reinforced and carried over to future generations. The bottom figure is the same as the middle one, but instead shows the maximum throughput by clock time instead of generation number. It shows that the runs that happen to find better configurations early on (Nilfs2) where able to finish 200 generations 2–3× faster than the runs that got "stuck" on poorer performing Ext2 and Ext4 configurations. We can also infer the run time efficiency of GA from this graph. GA over *Storage v1* runs mostly around 10–15 hours whereas an exhaustive search takes 4 days. Figure 2 demonstrates the need to tune GA's parameters to search more promising areas of space and [re]seed the population according to the search space shape (see Figure 1). In most runs, 50 generations would have been enough; but for run 3, we needed 150 generations, suggesting the need for a good stopping criteria (convergence).

When we ran a different workload—fileserver—it converged on a different configuration than Nilfs2: an Ext4 configuration. This shows that the workload is a key part of the environment and significantly impacts the search space and optimal configurations. The optimal configuration is also sensitive to the changes in hardware. When we ran the mailserver workload experiments on a newer and faster machine with the *Storage v2* chromosome, Nilfs2 was no longer the best configuration for the mailserver workload: Ext2 was. Ext2 beat the default Ext4 configuration (on SSD), by about 7.4%. Another surprise was that the best configuration used the `noop` I/O scheduler rather than the default `deadline` one. This proves that default configurations are often sub-optimal and even common wisdom (i.e., best accepted practices) can be wrong. Expanding the chromosome to include I/O schedulers and disk types improved overall throughput by 13.4%. This shows the need to explore as many parameters as possible in large spaces.

4 Conclusion and Future Work

Storage stacks are getting more complex with added controls to optimize. Optimizing such systems is becoming exponentially more challenging. Users still rely on unsuitable or manual methods using limited, narrow domain expertise that lacks a global system view. Worse, optimizing one system often does not carry over when the environment (hardware, software, or workloads) inevitably changes. We believe that Meta-heuristic search techniques (GA, SA, etc.) are key to optimizing storage systems. Our preliminary experiments have shown their potential to find near-optimal configurations regardless

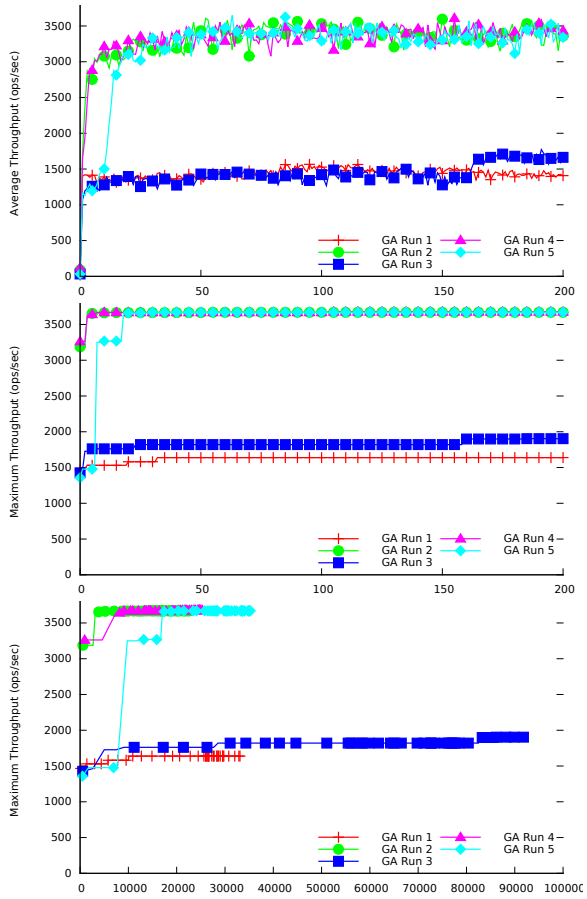

Figure 2: Throughput GA results for Mailserver workload: average by generation (top), max by generation (middle), and max by time (bottom).

of changing environments—often finding surprisingly unexpected configurations that defy existing practices.

GA has a few tunable parameters that speed up exploring the search space. We plan to study the effects of different selection and crossover methods, mutation rates, and adaptive techniques. We plan to combine some MH techniques together (e.g., GA and SA), especially in enabling the automatic removal of less important parameters, thus reducing the search space. Currently our fitness function is defined as a numeric throughput. We are exploring more complex cost functions involving energy efficiency, reliability, and even economic models. In more practical scenarios, workloads change with time. Such scenarios demand continual GA evaluation depending on the *speed* of change. We plan to develop automated stopping/resuming criteria for GA based on the observed differences in fitness values for known environments. Moreover, it may not be feasible to change some system parameters frequently. Mount parameters like `noatime` can be set during run time, but file systems cannot be formatted/changed without downtime. In the future, we will explore and add the costs of such changes to the fitness function.

Acknowledgements. We thank the anonymous reviewers and our shepherd for their helpful comments. This work was made possible in part thanks to NSF awards CNS-1302246, CNS-1305360, CNS-1522834, and IIS-1251137.

References

[1] E.D. Beinhocker. *The Origin of Wealth: Evolution, Complexity, and the Radical Remaking of Economics*. Harvard Business School Press, 2006.

[2] E. Dicke, A. Byde, P. Layzell, and D. Cliff. Using a genetic algorithm to design and improve storage area network architectures. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 1066–1077. Springer, 2004.

[3] Filebench. `http://filebench.sf.net`.

[4] S. Gaonkar, K. Keeton, A. Merchant, and W. H Sanders. Designing dependable storage solutions for shared application environments. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):366–380, 2010.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[6] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[7] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: restoring data after disasters. *ACM SIGOPS Operating Systems Review*, 40(4):235–248, 2006.

[8] S. Kirkpatrick, C D. Gelatt, M. P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[9] Z. Li, R. Grosu, K. Muppalla, S. A. Smolka, S. D. Stoller, and E. Zadok. Model discovery for energy-aware computing systems: An experimental evaluation. In *Proceedings of the 1st Workshop on Energy Consumption and Reliability of Storage Systems (ERSS'11)*, Orlando, FL, July 2011.

[10] Z. Li, A. Mukker, and E. Zadok. On the Importance of Evaluating Storage Systems' $Costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[11] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013.

[12] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[13] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *IEEE MASCOTS 04*, pages 588–595, 2004.