Continuously Running Genetic Algorithm for Real-Time Networking Device Optimization

Amit Mandelbaum*
Nvidia Networking
Jerusalem, Israel
amitma@nvidia.com

Doron Haritan
Nvidia Networking
Jerusalem, Israel
doronh@nvidia.com

Natali Shechtman
Nvidia Networking
Yokneam, Israel
natalis@nvidia.com

ABSTRACT

Networking devices deployed in ultra-scale data centers must run perfectly and in real-time. The networking device performance is tuned using the device configuration registers. The optimal configuration is derived from the network topology and traffic patterns. As a result, it is not possible to specify a single configuration that fits all scenarios, and manual tuning is required in order to optimize the devices' performance. Such tuning slows down data center deployments and consumes massive resources. Moreover, as traffic patterns change, the original tuning becomes obsolete and causes degraded performance. This necessitates expensive re-tuning, which in some cases is infeasible.

In this work, we present ZTT: a continuously running Genetic Algorithm that can be used for online, automatic tuning of the networking device parameters. ZTT is adaptive, fast to respond and have low computational costs required for running on a networking device. We test ZTT in a diversity of real-world traffic scenarios and show that it is able to obtain a significant performance boost over static configurations suggested by experts. We also demonstrate that ZTT is able to outperform alternative search algorithms like Simulated Annealing and Recursive Random Search even when those are adapted to better match the task at hand.

CCS CONCEPTS

• **Networks** → *Network performance analysis*; **Data center networks**; **Network resources allocation**;

KEYWORDS

Genetic Algorithms, Networking, Adaptive Parameters Tuning

ACM Reference Format:
Amit Mandelbaum, Doron Haritan, and Natali Shechtman. 2021. Continuously Running Genetic Algorithm for Real-Time Networking Device Optimization. In *2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449639.3459272

*The first two authors have equal contribution

GECCO '21, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8350-9/21/07…$15.00
https://doi.org/10.1145/3449639.3459272

1 INTRODUCTION

In modern data centers, communication between servers is crucial to the operation, and as a consequence, the entire data center performance is highly determined by the performance of its networking devices (NDs)[24]. A ND has dozens of parameters that have to be configured. The different configurations have a significant impact on the performance of the ND as it is reflected in metrics like application throughput, latency, packet loss, and even power consumption.

Generally, the optimal configuration depends on the network topology and traffic patterns. A configuration that produces high performance on one scenario can significantly degrade performance on another and even result in service downtime. As different customers have different network topologies and traffic patterns, providing a single configuration that fits all customers is an impossible task, and manual tuning is required in order to optimize devices' performance. The manual tuning delay deployments on the data centers, affect customer experience and consume large amounts of R&D resources. More importantly, even within a single data center the traffic patterns change rapidly. This means that any *static* configuration is sub-optimal *by definition*. Moreover, if traffic patterns change significantly at the customer, the original tuning becomes obsolete and can cause degradation in performance[11], requiring expensive re-tuning. In some cases, the re-tuning process is infeasible.

In this work we present Zero Touch Tuning (ZTT): a modified Genetic Algorithm (GA)[18] that can be used for real-time, automatic optimization of the networking device parameters. The algorithm is designed to enable continuous operation, rapid reaction, and a low computational cost, which are all characteristics required for running on a ND, as discussed in Section 2.2. The contribution of this work is twofold: Firstly, we demonstrate how simple modifications to a basic GA allow it to run continuously (i.e., without the stopping criteria) and to respond to changes in the environment. Secondly, to the best of our knowledge, we are the first to introduce an algorithm that runs directly on a ND and optimizes its configurations continuously and in real-time. The benefits of using ZTT are as follows: **1. Performance improvement:** as ZTT dynamically adapts the ND configurations to changing traffic patterns it provides better performance compared to a static configuration. **2. Time and money savings:** ZTT can save significant amounts of man-hours spent on troubleshooting performance issues caused by obsolete static configurations and significantly improve customer experience.

To properly test out our algorithm, we build a physical arrangement of networking devices as described in Section 5. We test our algorithm on a diverse set of real-world traffic scenarios and show

in Section 6 that it significantly improves the performance over a static configuration found by domain experts. We also show that our method is able to outperform other successful optimization methods like Recursive Random Search (RRS)[47] and Simulated Annealing (SA)[25], even when those are adapted to better match the task at hand.

2 BACKGROUND

2.1 Networking Device Optimization

In general, NDs are network interface controllers (NICs), host channel adapters, network-enabled graphic processing units (GPUs), network switches or routers, which typically use InfiniBand™[33] or Ethernet as their communication protocol. In the context of our work, such NDs will contain some packet processing circuitry that is connected to the network and is configured to process packets for communicating over the network[14].

The packet processing circuitry contains a plurality of *configuration registers* that can be tuned on the fly while the device is processing packets. The circuitry also contains performance registers, typically in the form of counters, that measure the device's performance. These *performance metrics* may include transmit and receive bandwidth, latency, packet loss, memory usage, and power consumption.

Formally, the optimization problem can be formulated as follows (assume minimization): given a real-valued objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ find a global minimum:

$$x^* = \arg\min_{x \in S} f(x) \tag{1}$$

Such that x is the configuration registers, S is the search space, which is the range of valid values of the different configuration registers and f is a function of one or more of the performance metrics. In our case, the objective function $f(x)$ is analytically unknown and the function evaluation can only be performed via a physical setup or an equivalent simulator. Thus, this problem is treated as a "black-box" optimization problem[47].

In a manual workflow, the optimization process is done by combining trial and error with domain expertise. A performance engineer chooses a set of benchmark tests where f is measured and sets some value for the device registers x. While the tests are running, an external software samples the performance registers, usually at a low frequency, and collects performance statistics. The engineer then analyzes these statistics and repeats the process to determine which configuration has the best performance on average across all tests. The requirements, tests and definitions of performance can differ considerably between customers, and even for one customer this process can be quite costly. Furthermore, if any component changes, whether it is x, $f(x)$ or even S, the engineer often needs to start the optimization process from the very beginning.

A real-time optimization algorithm is also using the above circuitry but is running directly on the device itself as shown in Figure 1. This allows the algorithm to read the performance counters and write the configuration registers fast enough to react in real-time to changing traffic patterns[23]. As the algorithm adapts the ND to any traffic pattern, by definition, it is able to achieve better performance than any static configuration. Additionally, the algorithm does not require any offline training, so any changes to x, $f(x)$ or S will only require at the worst case, running the benchmark tests again just to ensure the algorithm is not failing. Finally, since the algorithm runs directly on the device, optimization occurs as the test is running. Thus, even in the case of searching for a static configuration, the process is orders of magnitude faster than the manual flow, where adjustments to x only occur at the end of a complete run of the benchmark tests.

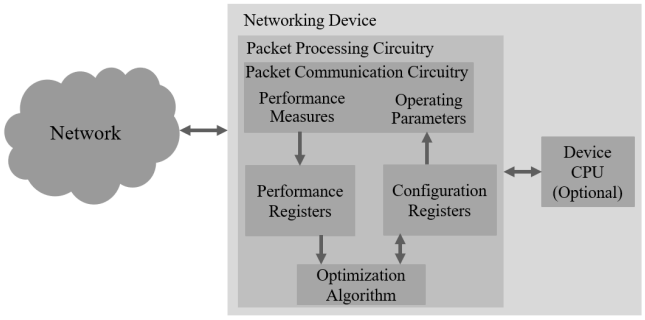

Figure 1: A diagram of a typical ND. The device will usually contain a packet communication circuitry located within a packet processing circuitry that is controlled by a firmware that runs on the ND compute. In some devices, an additional external CPU is available for tasks that require higher computation and can tolerate a longer latency. In order to meet requirements, the optimization algorithm is embedded directly in the device firmware and runs within the packet processing circuitry.

2.2 Algorithm Requirements

In order to design or select an efficient optimization algorithm, we must carefully look on the features of the given problem and the requirements it imposes on the algorithm. In the case of running an optimization algorithm on a ND the following requirements need to be met:

High sample efficiency: network traffic patterns normally vary significantly with time and the optimization algorithm should quickly find a better configuration before significant changes in the network occur.

Dynamism: as network traffic patterns change, the performance of a set of configurations might change. This means that the optimal configuration need to be searched again. Additionally, the possible values of the objective function f could change. For example, if f is a function of the transmit bandwidth and the maximum transmit bandwidth is now lower, something that often occurs, then the best objective function value may now be different. Any algorithm that runs on a networking device needs to be capable of handling such scenarios.

Low computational cost: as mentioned previously, in order for the algorithm to react quickly to changing traffic patterns, it runs directly on the ND's CPU. This imposes significant

constraints on the amount of computation that the algorithm can consume[35]. The CPU of the ND will also usually use a simpler set of instruction (like RISC-V[45] for example) compared to standard processors, further restricting the computation complexity of the algorithm.

Another consideration lies in the ability to model the dynamics of the network. Since the behavior of the real network changes frequently, modeling it becomes impractical. Furthermore, the traffic patterns of customers are rarely available due to privacy concerns. This requires the algorithm to run directly on the customers' network and be able to optimize it on real-time without being trained on an equivalent system first. It also means that methods that assume prior knowledge about the system[29] or try to model it[39] cannot be used here.

3 RELATED WORK

3.1 Machine Learning in Data Centers Networking

The field of machine learning for networking has seen a significant rise in recent years (see surveys [44], [5] and [12]). The works in this field employ the full range of machine learning techniques, including supervised learning, unsupervised learning, and reinforcement learning (RL). In the area of supervised learning we can find, for example, works on traffic prediction [22, 39, 51], load prediction[2, 3] and classification of network traffic[49, 50] or even specific packets[26, 35]. Unsupervised learning is used, for example, in the fields of cyber security[6, 8, 41] or for detecting anomalous performance behaviours[1, 19]. Finally, examples of usage of reinforcement learning techniques can be found in the fields of congestion control[21] adaptive routing[27, 28, 38] and cyber security[7, 13, 31, 36].

While most of these works cover a variety of network architectures, some are particularly relevant to data center networks. These include works on loads prediction[34], resources allocation[40] and congestion control[11, 21, 46]. From the mentioned above, the works on congestion control are similar to ours the most in the sense that they adjust parameters (e.g. sending rates) in order to maximize performance (e.g. total bandwidth). However, unlike our work, these works require some observations in addition to the cost function and the possible actions and most of them require training models on synthetic or simulated data sets[20, 48] while ours does not. Our approach is also different from other works mentioned here as it does not make any assumptions about the network topology or traffic patterns.

3.2 Online Optimization of Data Processing Systems

Big data processing systems have a large number of configuration parameters that control many aspects of the system. Like NDs, these systems require constant adaptation of their parameters to reach optimum performance, especially when dealing with streaming data. Recent survey [17] reviews the many methods used for this kind of tuning. With relation to our work, [42] used Bayesian optimization with a genetic algorithm to optimize the number of worker processes, while [43] uses a combination of a Random Forest and Recursive Random Search (RRS)[47] to automatically tune the data processing parameters. [16] also uses RRS for creating an elastic scaling data stream processing prototype, and finally [4] presents a framework for both black-box and gray-box optimization of parameters, though it requires some preliminary knowledge of the system which does not exist in our case.

4 PROPOSED METHOD

4.1 Basic Genetic Algorithm

The following section describes our proposed method in detail. However, since ZTT is a modified version of a GA, it is necessary to first provide a short introduction on basic GAs. GAs are a heuristic solution-search or optimization technique, originally motivated by the Darwinian principle of evolution through (genetic) selection and survival of the fittest. During the last few decades, GAs have been applied successfully in many areas[9, 32, 37] and are still today a popular technique for solving black-box optimization problems[15]. GAs are especially useful for cases like ours in which computation resources are limited, because GA arithmetic elements are, at least in their simplest form, very simple and fast to compute.

A basic GA search consists of four phases: initial population generation, selection, crossover and mutation. In the first step, a population of n "individuals" is created where each individual consists of real or integer coordinates, respectively corresponding to variables of the objective function at hand. In our case, an individual consists of a set of valid values for the configuration registers we are tuning. Using a fitness score, which depends on the objective function, and a selection strategy like elitism[10] or tournament selection[30], a subset k of the population is chosen to survive into the next generation. The crossover phase takes this subset of individuals and creates a new population from them by combining traits of pairs of individuals in a predefined manner such as averaging or swapping. To ensure that the genetic algorithm's evolution does not trap itself at a local minimum, the mutation phase randomly alters traits of certain members of the population, in accordance with predefined logic that can take into account the fitness score as well. The selection, crossover, and mutation steps are repeated in a cycle that is defined as a generation, until the stopping criteria is met, such as the number of generations passed or the diversity of the population going below a certain threshold.

4.2 Our Method

This section, describes ZTT in detail and explains the necessary modifications to the GA that are done to make ZTT continuously running and to increase its speed:

Removal of the stopping criteria: The first thing we notice is that the basic GA does not run continuously since it has a stopping criteria. One option to fix it is to use some mechanism that automatically restarts the algorithm[16]. Given the high frequency of changes in traffic and the fact that the changes are often gradual, this mechanism will not work here, therefore we remove the stopping criteria altogether.

Continuous selection phase: The regular genetic algorithm selection takes place after the crossover and mutation phases. This means that if the traffic pattern changes during a generation, the

selection is based on an objective function evaluation that is no longer valid. This can increase convergence time and lead to longer periods of degraded performance. Since our goal is to find as fast as possible a *better* solution rather than the *best* solution for the current scenario, we change the selection process. Rather than wait for the end of the generation to update the current population, we do it during the generation itself. This means that each time a new individual is created during the crossover or mutation phase, its fitness score is compared to the score of the best performing individual. If the score is better, we replace the lowest scoring individual with new one so that the population size always equals to k. This way the mutation and crossover phases are updated with the relevant population as they go, making the selection phase continuous.

Refresh of best individual's score: As explained, we insert new individuals based on their fitness score compared to the one of the best. However, the traffic pattern might have changed since we last measured the best individual's score. If the new minimum possible score of the objective function has increased due to traffic changes (as explained in Section 2.2), the new individuals might get a higher (worse) score than the current best while their *actual* fitness could be better. To solve this problem, we update the fitness score of the best individual before the crossover phase and again before the mutation phase to ensure the comparison is based on the current traffic pattern. As we show in Section 6, this further improves the results of our algorithm.

The full pseudo-code of ZTT is shown in Algorithm 1. If we look again at the requirements detailed in Section 2.2 we can see that ZTT meets them all. *High sample efficiency* is achieved by using a continuous selection mechanism which quickly updates the population to keep pace with changing traffic patterns. *Dynamism* is obtained by removing the stopping criteria in the algorithm, allowing it to run continuously, as well as the best score refresh mechanism, which ensures comparisons are done according to an updated objective function. Lastly, *low computational cost* is achieved by using a very simple GA that uses the most simple selection (elitism) and crossover (swapping) mechanisms. Additionally, the population size we use in practice is relatively small (see Section 6) which further reduces the computational cost of ZTT.

Algorithm 1 ZTT

$x_{default}$ = the set of default static configurations;
$init\_population$ = initialize $n-1$ random individuals from search space S;
$init\_population = init\_population + x_{default}$;
$score_i = f(x_i)$ for x_i,$i = 1...n$ in $init\_population$;
$population$ = take x_i, i=1...k with lowest $score_i$ values;
$x_{opt} = argmin_{1 \le i \le k}(score_i)$, $f_{opt} = f(x_{opt})$;
while True **do**
  for $i = 0$ **to** $k-1$ **do**
    Select two random individuals x_m and x_l from the current population;
    $x_i = crossover(x_m, x_l)$;
    $score_i = f(x_i)$;
    if $score_i < f_{opt}$ **then**
      Update population with x_i; {see text for update process}
    end if
  end for
  $x_{opt} = argmin_{1 \le i \le k}(score_i)$;
  $f_{opt} = f(x_{opt})$; {best score refresh}
  for $i = 0$ **to** $k-1$ **do**
    Select random individual x_m from the current population;
    $x_i = mutate(x_m)$;
    $score_i = f(x_i)$;
    if $score_i < f_{opt}$ **then**
      Update population with x_i;
    end if
  end for
  $x_{opt} = argmin_{1 \le i \le k}(score_i)$;
  $f_{opt} = f(x_{opt})$; {best score refresh}
end while

5 EXPERIMENTAL SETUP

The following section describes the physical setup used in our experiments, the tests that we ran, the configuration registers we tuned and the objective function we used. First, in order to test our algorithm in an environment close as possible to an actual data center networking cluster, we built a physical setup as shown in Figure 2. The setup is composed of three main components: two physical NICs, two servers with virtual machines emulating two additional NICs, and a switch connecting all the NICs together. Each of the NICs (both virtual and physical) have 4 hosts that can transmit and receive data at a maximum rate of 25GB/s (100GB/s in total). The switch enables each host on each NIC to transmit or receive data from and to any other host on any other NIC. The switch also allows several hosts to be connected to the same host in parallel. Since in data centers it is common to find different types of NICs connected to one another, we run ZTT only on the two physical NICs and use the virtual ones to simulate NICs that cannot run ZTT, to better simulate a real environment. In terms of specific devices, for the two physical NICs we use Nvidia® Mellanox® ConnectX®-6 Dx network adapters and for the switch we use Nvidia® Mellanox® Spectrum®-3 Open Ethernet switch, though the specific type of switch should not affect the results since the ZTT only affects the NICs. The specific setting of the virtual NICs should not affect the results either as long as they can send and receive data at a rate of 25GB/s per host.

In order to test ZTT we created 20 tests with varying characteristics that reflect the diversity of real-world traffic patterns. The tests differ in several aspects including the number of hosts sending or receiving data (participating hosts), the rate of the data that is sent or received, and the number of hosts connected in parallel to each host. We also mixed between bi-directional tests where all participating hosts transmit and receive data simultaneously and uni-directional tests where they only receive data. Another variable which changes across tests is the *overloading* of one or more of the participating hosts: In **regular** tests, *all* participating hosts in the NIC receive or transmit data at a rate that is below or equal to their full line-rate (25GB/s). In **aggressor/victim** tests, one or more of the participating hosts are *overloaded*, meaning they are receiving data at a rate that is higher than their full line-rate (aggressors) while the rest of the hosts do not (victims). On these tests, we also

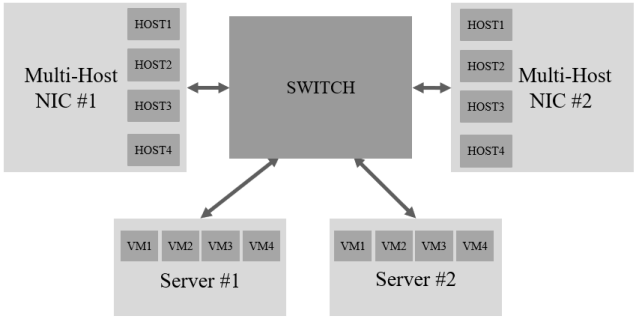

Figure 2: The experimental setup we used is composed of two multi-host NICs running ZTT and 2 servers simulating NICs that do not run ZTT. The 4 NICs are connected through a switch such that each host on any of the NICs can transmit or receive data to and from each host on any other NIC.

need to ensure that the overloaded hosts do not cause starvation of the regular hosts. In all tests, performance is measured on a single NIC and thus they are designed accordingly. The full list of tests can be found in the supplementary material.

To optimize the NIC, we used 12 *configuration registers* which affect different aspects of how it processes data. These configurations include the maximum number of packets allowed to be "in-the-air" (en-route to another ND), the credits of each host before giving back-pressure to the physical port, and parameters related to the Tx burst. In total, our search space S contains about $2.5 * 10^{50}$ different parameter configurations. The *objective function* is a combination of the transmitted bandwidth, the received bandwidth and the number of packet dropped per second, where *bandwidth* should be *maximized* and *packets drop rate* should be *minimized*. Increased bandwidth and reduced packets drop rate means that the applications on the server are receiving data at a higher rate resulting in a better overall performance. Therefore, this objective function reflects the real-world needs. We note here that generating a new individual requires setting values for the configuration registers and then sampling the performance over a pre-defined period of time called *sampling period*. As traffic is never completely stable, the lower the sampling period, the noisier the performance measures become, so we had to choose a sampling period that balances how fast the algorithm runs with how noisy its fitness function is.

6 RESULTS AND DISCUSSION

In the following section we evaluate ZTT in two different scenarios. Firstly, while ZTT is designed to adapt the ND to *changing* traffic patterns we want to ensure it is still able to provide good performance when traffic is *static*. For this reason, in the first scenario we run each test independently and for a long period of time. In the second scenario, we run all tests in succession thus reflecting real-world traffic patterns. We compare ZTT to the default static configuration, and to two successful optimization methods, SA[25] and RRS[47], and show that ZTT is able to get better results in terms of increased bandwidth and reduced packets drop rate. We discuss performance when running Aggressor/Victim tests and show that ZTT is able to improve overall performance while not degrading the victim received bandwidth. We also discuss convergence times and show that ZTT converges faster than SA and RRS. Finally, we discuss performance stability and show that ZTT provides more stable performance across all measured metrics. To increase statistical significance, we run each test four times and average the results.

Table 1: Performance of Random Configurations

Configuration	Rx BW	Tx BW	Drops
Random 1	-0.44%	-0.58%	-4.51%
Random 2	-0.34%	-0.28%	-3.91%
Random 3	-0.38%	-0.64%	-5.09%
Random 4	-0.43%	-1.10%	-5.95%
Random 5	-0.43%	-2.77%	-5.59%

Table 1 legend. *Tx BW* and *Rx BW* denote transmit and receive bandwidth respectively. *Drops* denotes packets drop rate. Results are shown for the top performing 5 random static configurations out of the 20 we tested. Results represent the average percentage of degradation over the performance of the default configuration across all tests.

The parameter settings are as follows: For all 3 algorithms, we set the initial population size to 48 with the default static configuration being part of that population. The rest of the RRS parameters are set as in [47]. For SA we use a linear cooling schedule with $\alpha = 0.1$, $T_{init} = 1$ and $T_{final} = 0.025$. For ZTT we set k to 20. In all tests we use a *sampling period* of 40ms, which, for our experimental setup, provides a good balance between algorithm speed and performance stability.

Considering the fact that the range of bandwidth and packets drop rate depends on various characteristics of each test, the correct way to compare performance is to compare percentage of improvement over the baseline performance as averaged across all tests and not by comparing actual bandwidth and packets drop rate values. For this reason, results in this section are presented in this way, where our baseline performance is the one with the default static configuration suggested by experts. In terms of notation, *Tx BW* and *Rx BW* refer to transmit and receive bandwidth in GB/s respectively and *Drops* refer to the packets drop rate, or packet drops per second. The full numerical results per test can be found on the supplementary material.

In order to establish a baseline, we begin by comparing our default static configuration with a set of random static configurations. For this, we run the 20 tests 20 times, each time with a different set of random static configurations. In Table 1 we see the performance of the best 5 random static configurations, compared to that of the default one. The default static configuration clearly outperforms the random ones, which means we can safely take the default configuration's performance as our baseline.

6.1 Independent Tests

In this experiment we examine how ZTT performs on static traffic. To accomplish this, we run each test independently for a minute and measure the performance of each algorithm over the entire minute and just on the last twenty seconds. With a sampling period of 40ms both SA and RRS have enough time to converge to a good set of configurations. In Table 2 we can see the full results. We clearly see that all three algorithms outperform the default static

Table 2: Results On Independent Tests

Alg.	Full Minute			Last 20 Seconds		
	Rx BW	Tx BW	Drops	Rx BW	Tx BW	Drops
ZTT	**0.92±0.10%**	**4.95±0.23%**	**20.39±0.89%**	**0.96±0.16%**	**4.98±0.05%**	**22.17±1.58%**
SA	0.85±0.12%	4.48±0.48%	17.56±3.52%	0.91±0.15%	4.66±0.63%	19.41±3.28%
RRS	0.81±0.03%	3.69±0.38%	15.79±1.26%	0.81±0.04%	3.71±0.42%	17.18±1.86%

Table 2 legend. Notations are the same as Table 1. Results represent the average and standard deviation of the percentage of improvement over the performance of the default configuration across all tests.

Table 3: Results On Continuous Tests - 30 Seconds

Alg.	Without Refresh			With Refresh		
	Rx BW	Tx BW	Drops	Rx BW	Tx BW	Drops
ZTT	**0.83±0.33%**	**4.37±0.22%**	**20.48±2.76%**	**0.89±0.16%**	**4.48±0.82%**	**21.48±2.67%**
SA	0.64±0.28%	3.05±1.31%	16.93±6.57%	0.74±0.3%	4.10±0.62%	18.35±8.7%
RRS	0.64±0.16%	2.55±0.2%	12.56±4.09%	0.66±0.06%	3.99±0.22%	15.81±2.42%

Table 3 legend. Notations and results format are the same as Table 2

configuration, especially after the initial convergence time. This is not surprising since the default configuration aims to achieve a good performance in all tests on *average*, rather than on each test individually. More importantly, we see that although ZTT runs continuously, it performs just as well when the traffic is static and even outperforms other algorithms designed to find good static configurations.

6.2 Continuous Tests

In order to test our algorithm under changing traffic patterns we run all 20 tests in succession and measure performance across all tests. We evaluate two scenarios, one where each test runs for 30 seconds and another that runs each for 10 seconds. As discussed in Section 2.2, to be able to handle rapidly changing traffic patterns, any algorithm running on a ND needs to be able to adapt itself to the changing objective function. As ZTT deals with this using a *refresh* mechanism, we are adding a similar mechanism to both RRS and SA in order to make a more meaningful comparison. To do this, for both algorithms we do a *refresh* by updating the score of the best found individual $f(x_{opt})$ every k individuals, where k is the same as in ZTT.

We check all three algorithms with and without the *refresh* mechanism. Results for the 30-seconds scenario are presented in Table 3, where we see that in both cases ZTT achieves the best results. As expected, the added *refresh* mechanism improves results of all three algorithms. We also see that the performance difference between ZTT and the other algorithms is more significant here compared to the difference on the independent tests. This demonstrates that ZTT is indeed better at dealing with continuously changing traffic. Figure 3 shows an example of a single run of three consecutive 30-seconds tests. We can see that ZTT performance is better across the entire run and is also more stable. In Table 4 we can see the results

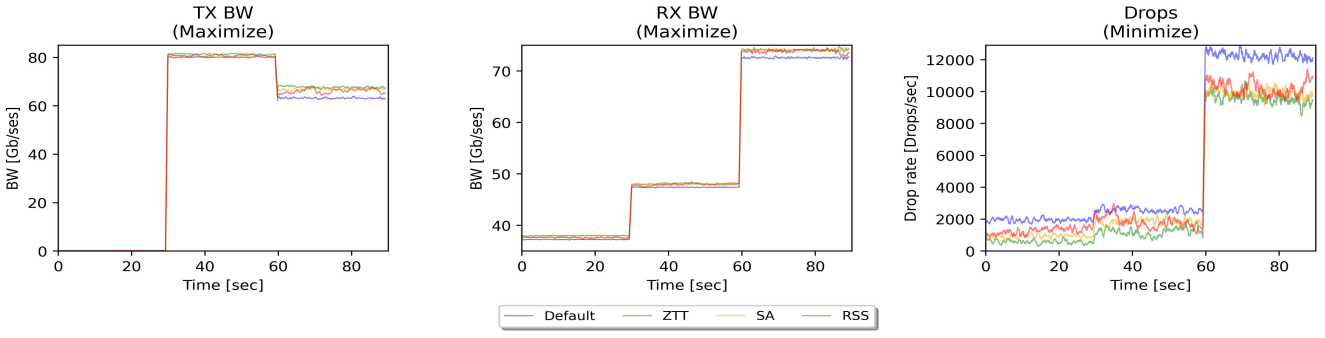

Figure 3: Example of a single run of 3 continuous tests. Performance is shown for the default static configuration and for the 3 optimization algorithms with the refresh mechanism active. *Rx/Tx BW* denotes receive and transmit bandwidth respectively. *Drops* denote packet drops per second. Each test is running for 30 seconds and we can see that traffic patterns change significantly between tests. The first test is of uni-directional traffic and thus the transmit bandwidth is 0.

Table 4: Results On Continuous Tests - 10 Seconds

Alg.	Without Refresh			With Refresh		
	Rx BW	Tx BW	Drops	Rx BW	Tx BW	Drops
ZTT	**0.44±0.22%**	**2.83±1.03%**	**19.86±6.12%**	**0.79±0.2%**	**3.60±1.23%**	**22.14±2.68%**
SA	0.12±0.38%	2.78±1.19%	16.66±5.21%	0.57±0.35%	3.30±0.84%	18.81±2.41%
RRS	0.28±0.19%	1.96±0.39%	14.29±3.12%	0.50±0.43%	2.60±1.01%	15.60±3.36%

Table 4 legend. Notations and results format are the same as Table 2

when running each test for 10 seconds. ZTT outperforms the other algorithms and the improvement in results when adding the refresh mechanism is more significant than the one on the 30-second tests, at least for ZTT and SA. This further emphasizes the importance of the refresh mechanism especially when traffic patterns change rapidly.

6.2.1 Aggressor/Victim Tests. As mentioned in Section 5, 14 out of the 20 tests we ran are defined as aggressor/victim tests. In these tests, one or more hosts (aggressor) are receiving data faster than their full line rate (25GB/s), while the other hosts do not (victims). Using the configuration registers, an optimization algorithm which tries to increase the received bandwidth of the aggressor host(s), can potentially cause a degradation of the received bandwidth of the victim host(s). The default static configuration do not cause this problem since these scenarios are taken into account. In Table 5 we can see the results of these tests. ZTT is able to significantly increase the aggressor bandwidth while not degrading the victim bandwidth when compared to the default static configuration. RRS and SA on the other hand, are also improving the aggressor bandwidth but this comes, at least partially, on the expanse of the victim bandwidth. We note here that since traffic is received in bursts a constant full utilization of the line rate is impossible. Thus the numbers in the table are already very close to the practical maximum.

6.2.2 Convergence Time. The ability of the optimization algorithm to respond rapidly to changing traffic patterns is one of the most important aspects of its design. To measure the algorithms' performance in this area, we examine the transitions between traffic patterns and count the number of individuals the algorithm need to check before its performance is steadily better than that of the default configuration. Results are presented in Table 6 and we can see that ZTT significantly outperforms both algorithms. We note here that on some transitions the algorithms' performance is better than that of the default configuration right out of the gate. In these cases the count of individuals is zero. Since this situation occurs more frequently in ZTT, the second line of the table shows the results when zero counts are excluded. We can clearly see that ZTT still outperforms the other two algorithms and we can conclude that its response time is indeed faster.

6.2.3 Stability. The algorithm's stability is another important feature of its performance. An optimization algorithm might improve performance on average, but momentary performance would be highly stochastic. In the networking world, such performance behaviour can be detrimental to performance and lead to degradation in the long-term. To measure the algorithms' stability we measure the average standard deviation for each of our metrics across the test. To be able to average the standard deviation across all tests we divide it by the mean performance of each test. Results are presented in Table 7 and we can see that ZTT achieves the lowest standard deviation on all metrics, demonstrating the most stable performance.

Table 5: Results on Aggressor/Victim Tests

Host Type	Default	ZTT	SA	RRS
Victim	**96.8%**	**96.8%**	96.4%	96.5%
Aggressor	94.0%	**95.8%**	95.1%	95.4%

Table 5 legend. *Default* denote the default configuration. Results represent the Rx BW percentage out of the maximum possible value, which is 25Gb/s for the aggressor host(s) and 5, 10 or 20 GB/s for the victim host(s), depending on the test settings. In case we have more than one victim or aggressor hosts, results are averaged across hosts. Note that the Tx BW is not effected by the aggressors.

Table 6: Comparison of Convergence Times

Tests	ZTT	SA	RRS
All	**2.05**	4.20	5.30
No Zero Count	**2.56**	4.67	5.30

Table 6 legend. Results represent the average number of individuals the algorithms need to check before its performance is steadily better than that of the default configuration. *All* denote results when averaging across all transitions between tests. *No Zero Count* denote results after excluding transitions where the algorithm was better immediately after the transition.

Table 7: Comparison of Performance Stability

Alg.	Rx BW	Tx BW	Drops
ZTT	$\mathbf{1.22e^{-3}}$	$\mathbf{7.13e^{-3}}$	$\mathbf{1.19e^{-1}}$
SA	$1.89e^{-3}$	$9.25e^{-3}$	$1.22e^{-1}$
RRS	$4.81e^{-3}$	$16.15e^{-3}$	$1.57e^{-1}$

Table 7 legend. Results represent the normalized standard deviation averaged across all tests. Notation are the same as in Table 1.

7 CONCLUSIONS

In this paper we presented ZTT, an online parameters optimization algorithm that automatically tunes the configuration registers of a ND for maximum performance in real-time. We showed that several modification to a simple GA allows the algorithm to work fast, in a continuous manner and fulfill the requirements needed for an algorithm to run on a ND. We designed a physical experimental setup

that runs real world traffic patterns. For this setup we showed that ZTT is able to increase bandwidth by 5% and decrease the packets drop rate by 21% compared to expertly crafted static configurations. We also showed that some of the modifications made for ZTT are also applicable to other search algorithms like SA[25] and RRS[47] and improve their performance in our experiments. Nevertheless, ZTT managed to get the best results in all the tests we conducted. ZTT was also able to outperform the other algorithms when we inspected other metrics such as convergence time, stability and victim host starvation.

We note that while the experimental setup here just focused on optimizing the bandwidth and packets drop rate of NICs, the implementation behind ZTT can be used for numerous other applications. ZTT can be applied, for instance, to optimize other metrics like power consumption, latency, CPU usage, and much more. ZTT can also be used in other NDs like switches, routers and host-channel adapters. The simplicity of ZTT makes it easy to deploy it directly on the device firmware. This enables ZTT to be embedded in a wide variety of devices even when their computing power is very low. In light of the growing demand for high-performance communication networks within data centers and the increasing complexity of such systems, adaptive optimization algorithms are needed more than ever to efficiently operate NDs. It is our belief that ZTT and the various related principles discussed in this paper would lead to many applications in the years to come, applications that would drive increased performance, reduce human effort and improve the overall customer experience.

REFERENCES

[1] Paul Barford, Nick Duffield, Amos Ron, and Joel Sommers. 2009. Network performance anomaly detection and localization. In *IEEE INFOCOM 2009*. IEEE, 1377–1385.

[2] Dario Bega, Marco Gramaglia, Marco Fiore, Albert Banchs, and Xavier Costa-Perez. 2019. DeepCog: Cognitive network management in sliced 5G networks with deep learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 280–288.

[3] Paola Bermolen and Dario Rossi. 2009. Support vector regression for link load prediction. *Computer Networks* 53, 2 (2009), 191–201.

[4] Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*. 189–200.

[5] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. 2018. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications* 9, 1 (2018), 16.

[6] Anna L Buczak and Erhan Guven. 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials* 18, 2 (2015), 1153–1176.

[7] James Cannady. 2000. Next generation intrusion detection: Autonomous reinforcement learning of network attacks. In *Proceedings of the 23rd national information systems security conference*. 1–12.

[8] Pedro Casas, Johan Mazel, and Philippe Owezarski. 2012. Unsupervised network intrusion detection systems: Detecting the unknown without knowledge. *Computer Communications* 35, 7 (2012), 772–783.

[9] SS Chaudhry* and W Luo. 2005. Application of genetic algorithms in production and operations management: a review. *International Journal of Production Research* 43, 19 (2005), 4083–4101.

[10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[11] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. 2015. {PCC}: Re-architecting congestion control for consistent high performance. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 395–408.

[12] Zubair Md Fadlullah, Fengxiao Tang, Bomin Mao, Nei Kato, Osamu Akashi, Takeru Inoue, and Kimihiro Mizutani. 2017. State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2432–2455.

[13] Rajesh Ganesan, Sushil Jajodia, Ankit Shah, and Hasan Cam. 2016. Dynamic scheduling of cybersecurity analysts for minimizing risk using reinforcement learning. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–21.

[14] José Luis García-Dorado, Felipe Mata, Javier Ramos, Pedro M Santiago del Río, Victor Moreno, and Javier Aracil. 2013. High-performance network traffic processing systems using commodity hardware. In *Data traffic monitoring and analysis*. Springer, 3–27.

[15] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.

[16] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 276–287.

[17] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–37.

[18] John Henry Holland et al. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[19] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 1–35.

[20] Teerawat Issariyakul and Ekram Hossain. 2009. Introduction to network simulator 2 (NS2). In *Introduction to network simulator NS2*. Springer, 1–18.

[21] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*. 3050–3059.

[22] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. 2016. {CFA}: A practical prediction system for video qoe optimization. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 137–150.

[23] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. 2017. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 393–406.

[24] Christoforos Kachris and Ioannis Tomkos. 2012. A survey on optical interconnects for data centers. *IEEE Communications Surveys & Tutorials* 14, 4 (2012), 1021–1036.

[25] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[26] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*. 256–269.

[27] Shih-Chun Lin, Ian F Akyildiz, Pu Wang, and Min Luo. 2016. QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach. In *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 25–33.

[28] Michael Littman and Justin Boyan. 1993. A distributed reinforcement learning scheme for network routing. In *Proceedings of the international workshop on applications of neural networks to telecommunications*. Erlbaum Hillsdale, NJ, USA, 45–51.

[29] Daniel A Menascé, Daniel Barbará, and Ronald Dodge. 2001. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*. 224–234.

[30] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[31] Thanh Thi Nguyen and Vijay Janapa Reddi. 2019. Deep reinforcement learning for cyber security. *arXiv preprint arXiv:1906.05799* (2019).

[32] Mantas Paulinas and Andrius Ušinskas. 2007. A survey of genetic algorithms applications for image enhancement and segmentation. *Information Technology and control* 36, 3 (2007).

[33] Gregory F Pfister. 2001. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O* 42, 617-632 (2001), 102.

[34] John J Prevost, KranthiManoj Nagothu, Brian Kelley, and Mo Jamshidi. 2011. Prediction of cloud data center networks loads using stochastic and neural models. In *2011 6th International Conference on System of Systems Engineering*. IEEE, 276–281.

[35] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2020. A Computational Approach to Packet Classification. *arXiv preprint arXiv:2002.07584* (2020).

[36] Arturo Servin and Daniel Kudenko. 2005. Multi-agent reinforcement learning for intrusion detection. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*. Springer, 211–223.

[37] Mandavilli Srinivas and Lalit M Patnaik. 1994. Genetic algorithms: A survey. *computer* 27, 6 (1994), 17–26.

[38] Giorgio Stampa, Marta Arias, David Sánchez-Charles, Victor Muntés-Mulero, and Albert Cabellos. 2017. A deep-reinforcement learning approach for software-defined networking routing optimization. *arXiv preprint arXiv:1709.07080* (2017).

[39] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. 2016. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 272–285.

[40] Gerald Tesauro et al. 2005. Online resource allocation using decompositional reinforcement learning. In *AAAI*, Vol. 5. 886–891.

[41] Marina Thottan and Chuanyi Ji. 2003. Anomaly detection in IP networks. *IEEE Transactions on signal processing* 51, 8 (2003), 2191–2204.

[42] Michael Trotter, Guyue Liu, and Timothy Wood. 2017. Into the storm: Descrying optimal configurations using genetic algorithms and bayesian optimization. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 175–180.

[43] Guolu Wang, Jungang Xu, and Ben He. 2016. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 586–593.

[44] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. 2017. Machine learning for networking: Workflow, advances and opportunities. *IEEE Network* 32, 2 (2017), 92–99.

[45] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. 2014. *The risc-v instruction set manual. volume 1: User-level isa, version 2.0.* Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

[46] Keith Winstein and Hari Balakrishnan. 2013. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 123–134.

[47] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 196–205.

[48] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. 1998. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings. Twelfth Workshop on Parallel and Distributed Simulation PADS'98 (Cat. No. 98TB100233)*. IEEE, 154–161.

[49] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. 2014. Robust network traffic classification. *IEEE/ACM transactions on networking* 23, 4 (2014), 1257–1270.

[50] Jun Zhang, Yang Xiang, Yu Wang, Wanlei Zhou, Yong Xiang, and Yong Guan. 2012. Network traffic classification using correlation information. *IEEE Transactions on Parallel and Distributed systems* 24, 1 (2012), 104–117.

[51] Yan Zhu, Guanghua Zhang, and Jing Qiu. 2013. Network Traffic Prediction based on Particle Swarm BP Neural Network. *J. Networks* 8, 11 (2013), 2685–2691.