An Autonomic Resource Allocating SSD

Dongjoon Lee, Jongin Choe, Chanyoung Park, Kyungtae Kang, Mahmut Kandemir*, and Wonil Choi
Hanyang University, Ansan, Republic of Korea, *Pennsylvania State University, University Park, PA, USA
{dongjoonlee, jichoe, chanyoung, ktkang, wonilchoi}@hanyang.ac.kr, *mtk2@psu.edu

***Abstract*—When an SSD is used for executing multiple workloads, its internal resources should be allocated to prevent the competing workloads from interfering with each other. While channel-based allocation strategies turn out to be quite effective in offering performance isolation, questions like "what is the optimal allocation?" and "how can one efficiently search for the optimal allocation?" remain unaddressed. To this end, we explore the channel allocation problem in SSDs and employ a reinforcement learning-based approach to address the problem. Specifically, we present an autonomic channel allocating SSD, called `AutoAlloc`, which can seek near-optimal channel allocation in a self-learning fashion for a given set of co-running workloads. The salient features of `AutoAlloc` include the following: (i) the optimal allocation can change depending on the user-defined optimization metrics; (ii) the search process takes place in an online setting without any need of extra workload profiling or performance estimation; and, (iii) the search process is fully-automated without requiring any user intervention. We implement `AutoAlloc` in LightNVM (the Linux subsystem) as part of the FTL, which operates with an emulated Open-Channel SSD. Our extensive experiments using various user-defined optimization metrics and workload execution scenarios indicate that `AutoAlloc` can find a near-optimal allocation after examining only a very limited number of candidate allocations.**

I. Introduction

It is quite common for a single resource-rich contemporary solid-state drive (SSD) to accommodate and execute multiple workloads simultaneously. Such a shared SSD requires its internal resources to be partitioned and allocated to co-running workloads for preventing the competing workloads from interfering with one another. While there exist various ways of allocating the resources in an SSD, *channel-based* allocation strategies are desirable from a performance standpoint, as they can eliminate any potential interference between the workloads, and thus, offer performance isolation to each workload [8].

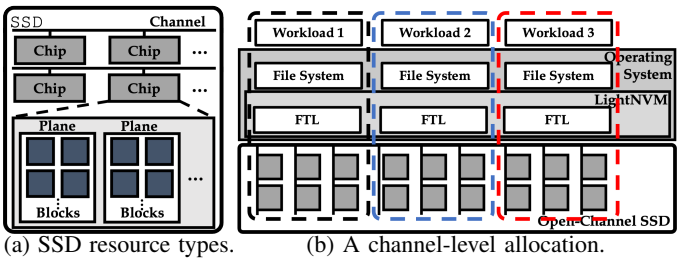
Unfortunately, existing channel-based allocation strategies *assume* a specific scenario in which the number of co-running workloads is quite small, and in turn, each workload can obtain as many channels as it wants. However, such situations may not occur in some environments (e.g., university, company) where many users in an organization share a fixed amount of resources. Instead, in such environments where the collective demand of workloads exceeds the available channel capacity, device operators can face a new critical problem: "what is the optimal channel allocation for a given set of co-running workloads?" In this paper, we refer to this problem as the "channel allocation problem" and present a novel online solution for it.

The channel allocation problem is quite challenging due to the following reasons. First, there is no accepted, unified definition of "optimality". In situations of channel scarcity, an increase in the channel allocation for a workload necessarily leads to a decrease in the channel allocation for another workload, in which case the former workload experiences improved performance whereas the latter workload suffers from degraded performance. Then, what is the desirable standard that can be employed to favor one allocation over the other? Second, the performance of each of the co-running workloads under varying number of allocated channels should be known in advance for making an accurate allocation decision. Actually, the benefits brought by allocating additional channels to workloads can vary significantly across the different workloads and within each workload, which brings us to the next question: can such a time-consuming, offline process of profiling the performance of each (potentially) co-runner workload be avoided? Finally, the number of possible allocations can be very large (i.e., the channel allocation problem can have a huge search space). Specifically, as the numbers of available channels and/or target (co-running) workloads increase, the number of possible allocations can increase significantly. Therefore, is there an efficient strategy that can be adopted to search over such a large space?

To address these problems in a systematic fashion, we present an autonomic resource allocating SSD, called `AutoAlloc`, which determines and applies the optimal channel allocation to a given set of co-running workloads. Inspired by its self-adapting capability, `AutoAlloc` introduces *Reinforcement Learning* (RL) in our context to solve the channel allocation problem. Specifically, given a set of co-running workloads, beginning with the even-allocation, `AutoAlloc` selects (and enforces) a candidate channel allocation, executes the workloads on the allocation (for a certain period in time), and obtains the performance results. This process is repeated until no better candidate allocation could be found. The key features of `AutoAlloc` in the process of exploring for an optimal channel allocation include the following: (i) it seeks the best allocation based on the optimality which is defined and given by an administrator; (ii) it begins the search right away without requiring any offline process of workload profiling; and, (iii) it explores the target search space efficiently and automatically without any need of administrator intervention.

We implement and evaluate `AutoAlloc` in a full-system emulation environment, where FEMU [13] imitates an Open-Channel SSD [1] and real applications are executed on it. Specifically, we add two modules – the *RL-agent* module and the *Dev-mgmt* module – to the Linux kernel (LightNVM [4] where the FTL functionalities are implemented). While the former executes the Q-learning algorithm [5] by evaluating the current allocation and selecting the next candidate allocation, the latter is responsible for enforcing the candidate allocation, executing the workloads, and measuring their performance.

979-8-3503-4859-0/DATE24/© 2024 EDAA



(a) SSD resource types. (b) A channel-level allocation.

Fig. 1: The resource allocation for multi-workload execution.

In scenarios where 4 co-runner workloads are executed on a 32-channel, 128GB SSD (and thus, there exist 969 different possible channel allocations), `AutoAlloc` finds a near-optimal allocation (i.e., one within 8% of the globally optimal allocation) after evaluating an average of 77 different allocations. Overall, this paper makes the following major **contributions**:

- We propose an autonomic channel allocating SSD, called `AutoAlloc`, which employs an RL-centric approach to explore channel allocations for a given set of co-running workloads. `AutoAlloc` can reach a near-optimal allocation automatically and gradually at runtime.
- We introduce various desirable definitions of the optimality and employ them in `AutoAlloc`. Our experiments reveal that, for a fixed set of co-running workloads, `AutoAlloc` navigates over different candidates and selects different near-optimal allocations under different definitions.
- We observe that the search space of the allocation problem can be extremely large, where `AutoAlloc` becomes a more effective alternative. In particular, increasing the allocation granularity can reduce the number of evaluated allocations significantly while still coming up with a near-optimal allocation.

II. Preliminaries and Related Work

A. SSD Resource Allocation under Multi-Workload Scenarios

As illustrated in Fig. 1a, an SSD includes four types of resources, namely, channels, (flash) chips, planes and blocks, and in principle, each of these can be an allocation unit. A large body of prior works [8], [12], [16] explored those units and investigated the impact of using them on the workload performance. Compared to other resource types, channel-level allocation can offer the best performance isolation for each of the co-running workloads (since a channel is the largest unit for accessing the storage media). Fig. 1b illustrates an example where three co-running workloads are executed on their own channel partitions with their own FTL and file system.

A recent work [14] explored the channel allocation problem using a machine learning (ML) based approach. However, (i) it considers only the latencies of the workloads for the optimal allocation; (ii) it generates a learned model in an offline setting and infers the optimal allocation for given set of workloads; and (iii) its validation is performed based on the old-fashioned simulation with only a few tiny-scale scenarios. In contrast, our proposed system can (i) employ any user-defined optimization metric (this work uses 4 alternate metrics); (ii) find the optimal allocation *automatically* and *gradually* in an online setting by actually executing the workloads in a full-system environment; and finally, (iii) test a wide range of different workload/SSD scenarios, which require a significant evaluation time.

B. Applying Learning-Based Approaches to SSD Problems

There have been several attempts to employ an ML-based approach in the SSD-specific problems. A group of works [17], [19] used ML to improve the device lifetime, while another body of works [9], [10] reduced tail latencies by improving the efficiency of the current GC operations by learning the SSD behavior in the past. There have also been ML-based approaches to manage the device-internal traffic efficiently, targeting DRAM caching [15] and SLC caching [20].

III. The Challenges and Our Approaches

Finding the optimal allocation is a challenging task. Here, we elaborate its challenges and our corresponding approaches.

- **Pareto Efficiency:** In the channel allocation problem, it is not possible to increase the allocation of a workload without decreasing the allocation of at least another workload (such a situation is called "Pareto efficiency"). Fig. 2a shows the experienced throughput of each workload under different channel allocations, when two workloads are executed on a 16-channel SSD. Obviously, if a workload experiences improved throughput due to the increased number of the channels, the other suffers from a decreased throughput coming from the correspondingly reduced number of the channels. The lack of a standard that can be used to favor one allocation over others is a fundamental research problem in this context.

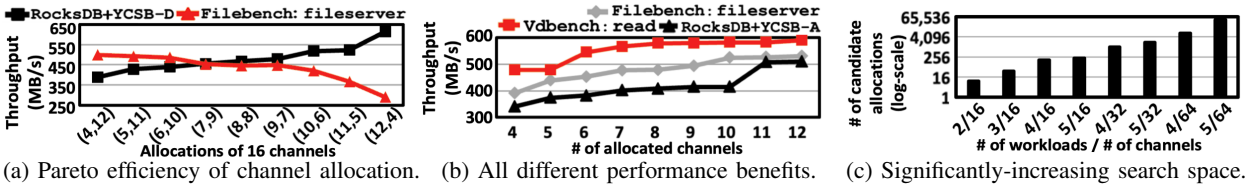
To this end, we introduce four desirable standards (metrics) to measure each allocation, and make `AutoAlloc` search for the optimal allocation based on the given metric.

(i) Aggregate Throughput: Maximizing the collective throughput value of all the co-running workloads can be a simple yet commonly-used optimization goal. The optimal allocation tries to maximize $\sum_{i=1}^{n} th(i)$, where n is the number of workloads and $th(i)$ is the throughput of workload i.

(ii) Weighted Speedup: The weighted speedup can be a promising choice, if a notion of fairness across the competing workloads is to be taken into account. In this case, the optimal allocation maximizes $\sum_{i=1}^{n}(th(i)/th_{even}(i))$, where $th_{even}(i)$ is the throughput value of workload i under an even-allocation.

(iii) Device Lifetime: If the administrator considers the budget rather than the user performance, she may want to reduce the consumption of the device lifetime. Here, the optimal allocation minimizes $\sum_{i=1}^{n} writes(i)$, where $writes(i)$ is the total number of page writes. In the current implementation, we include two dominant contributors: a) writes directly issued by the workload i, and b) writes generated from the garbage collection (GC) for the workload i, in $writes(i)$.

(iv) Throughput-Lifetime Product: For the administrator who may consider both the performance of the workloads and the device lifetime, we define a new metric, called *throughput-lifetime product*, which is calculated by *multiplying* the aggregate throughput by an inverse of the written data volume for a second. That is, the optimal allocation maximizes $\sum_{i=1}^{n} th(i) \times (T/\sum_{i=1}^{n} writes(i))$, where T is the execution time of the workloads (or the evaluation time of an allocation).



(a) Pareto efficiency of channel allocation. (b) All different performance benefits. (c) Significantly-increasing search space.

Fig. 2: The key factors which render our channel allocation problem quite challenging.

• **Different Performance Benefits:** The magnitude of the performance improvement that can be achieved as a result of allocating extra channels varies significantly within each workload as well as across different workloads. Fig. 2b shows the achieved throughput of three workloads under varying number of allocated channels (when each workload is executed individually). While an increase in the allocation generally leads to an increase in the throughput of a workload, the throughput improvement changes radically across the workloads. Even for each individual workload, the increase in its throughput does not exhibit any uniform patterns. Due to such irregular benefits of workloads under varying allocations, finding an optimal allocation requires either an offline profiling or a model-based performance estimation (which may not be accurate).

To address this, we develop an *online*, *real execution-based* approach, which does not require any profiling or model-based strategy. Specifically, inspired by the self-adapting nature of reinforcement learning (RL), `AutoAlloc` evaluates each allocation by executing the target workloads on it at runtime, and identifies an optimal allocation in a gradual fashion based on the cumulative evaluation results.

• **Huge Search Space:** The number of possible channel allocations increases significantly, as the number of the available channels in the target device and/or the competing workloads increases. Fig. 2c shows the number of possible channel allocations for a few problem scenarios. Even in a tiny-scale scenario where 4 workloads share 32 channels, there exist up to a thousand of different allocations, which may require an extensive experimental effort to evaluate candidate allocations.

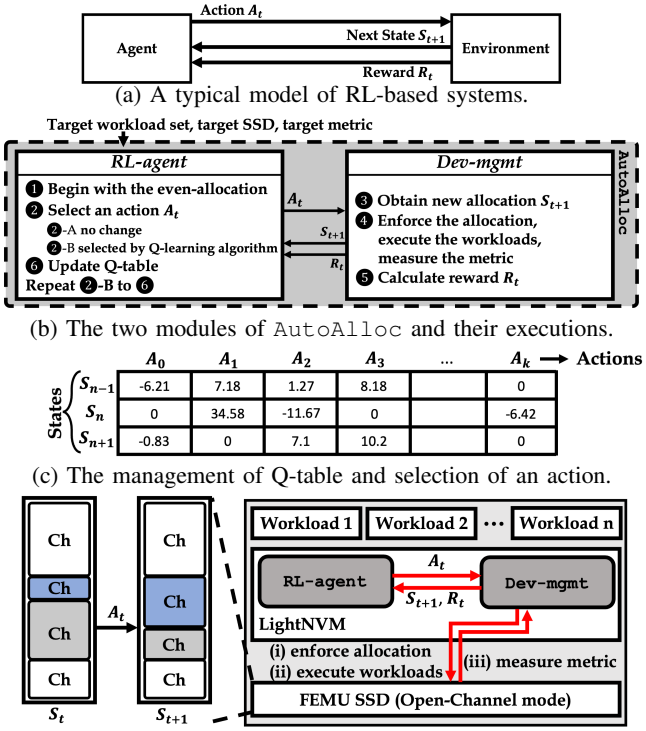
To this end, we devise an *autonomic* system, which searches for the optimal allocation by itself without any need of human intervention. Specifically, based on the self-operating nature of RL, `AutoAlloc` automatically selects/evaluates candidate allocations and stops its operation once it finds the best allocation.

IV. Proposed Autonomic System

A. Applying Reinforcement Learning to Our Problem

Fig. 3a depicts a typical model of RL-based systems, where the *agent* is a self-trained entity and the *environment* is the target problem space including all possible states. The agent takes an action (A_t) in the environment at each turn (t), and consequently, the environment produces the next state (S_{t+1}) and the reward (R_t) for the corresponding action. The agent repeats to take actions in the varying states, with the goal of maximizing the cumulative rewards. In our context, the possible channel allocations can be regarded as states. Then, the key components can be modeled as follows:

(1) Action (A_t): The change that is made in the current channel allocation (S_t). In the current version of `AutoAlloc`, an



(a) A typical model of RL-based systems.

(b) The two modules of `AutoAlloc` and their executions.

(c) The management of Q-table and selection of an action.

(d) The current implementation and channel allocations.

Fig. 3: Our proposed `AutoAlloc` system.

action can be a channel increment for a workload and a channel decrement for one of the other workloads. For example, in a 4-workload scenario, an action can be expressed as (0, +1, -1, 0), which means that the number of allocated channels for the second workload is increased by one while that for the third workload is decreased by one. Since the possible number of actions can increase significantly as the number of co-running workloads increases, we limit the number of possible actions by increasing and decreasing the channel count *only for any two neighboring workloads*. So, in a 4-workload scenario, there exist 9 possible actions: (0, 0, 0, 0), (+1, -1, 0, 0), (-1, +1, 0, 0), (0, +1, -1, 0), (0, -1, +1, 0), (0, 0, +1, -1), (0, 0, -1, +1), (+1, 0, 0, -1), and (-1, 0, 0, +1). Here, (0, 0, 0, 0) means that no change is made in the current state.[1]

(2) Next state (S_{t+1}): The channel allocation that stems from performing the action (A_t) from the current channel allocation (S_t). For instance, in a 4-workload, 32-channel scenario, if S_t = (8, 8, 8, 8) and A_t = (0, +1, -1, 0), then S_{t+1} = (8, 9, 7, 8).

(3) Reward (R_t): The change in the metric value as a result of

[1]Under this model, the total number of actions can be limited to $(2 \times n) + 1$, assuming a n-workload scenario. It should be noted that this limited set of actions can explore all possible allocations. For example, a potential action (+1, 0, -1, 0) can be achieved by performing two different actions, (+1, -1, 0, 0) and (0, +1, -1, 0), consecutively.

TABLE I: The configurations of the tested SSDs.

128GB	32 channels, 2 chips/ch, 2 planes/chip, 16KB page, channel-level allocation
	tested scenarios: S-1, S-2, S-3, S-4, S-5, S-6, S-7, S-8
256GB	64 channels, the same for the other configurations; tested scenarios: S-9, S-10

changing the allocation from S_t to S_{t+1} by A_t. R_t is obtained by $M_{t+1} - M_{avg}$, where M_{t+1} and M_{avg} indicate, respectively, the metric value in the new allocation (S_{t+1}) and the average of the metric values in the allocations evaluated so far.

Fig. 3b illustrates the overview of our proposed AutoAlloc system, where the *RL-agent* and *Dev-mgmt* modules act as an agent and an environment, respectively, and exchange the action, the reward, and the next state between the two.

(4) Agent: The RL-agent module which implements an RL algorithm. While there exist various sophisticated RL algorithms, we use the representative Q-learning [5], since it fits into our channel allocation problem well and generates the promising results. This module is responsible for (i) the management of the Q-table (a core data structure in the Q-learning) and (ii) the selection of the next candidate allocation (the action).

(5) Environment: The Dev-mgmt module which evaluates each of the candidate channel allocations. Specifically, this module (i) enforces the channel allocation based on the action from the RL-agent module, and (ii) executes the target workloads on it. Then, it (iii) measures the target metric value, generates the reward value, and provides it to the RL-agent module.

B. Detailed Execution Process

As described in Fig. 3b, the execution process of AutoAlloc is as follows: ❶ AutoAlloc begins with an "even-allocation" (the channels are evenly distributed over the workloads); so, the Dev-mgmt module sets the even-allocation as the initial state. ❷ the RL-agent module selects an action and passes it to the Dev-mgmt module; here, (❷-A) the first action is an action for making no change (i.e., (0, 0, 0, 0)) to evaluate the initial even-allocation, whereas (❷-B) all the following actions are selected by the Q-learning algorithm. ❸ the Dev-mgmt module obtains a new channel allocation based on the action, which is the next state (S_{t+1}). ❹ the Dev-mgmt module enforces the new allocation, executes the workloads on it, and measures its optimization metric. Here, we avoid repeated evaluation of the same allocations by keeping the metric values for the "already-evaluated" allocations in the Dev-mgmt module and supplying those values when we see the same allocations. ❺ the Dev-mgmt module calculates the reward (R_t), which is sent to the RL-agent module along with the next state (S_{t+1}). ❻ the RL-agent module updates the Q-table with the received reward. The process of ❷-B to ❻ is repeated until the state does not change for N_{stop} consecutive actions, and the current state becomes the final channel allocation.

The Management of the Q-table: The Q-table in the RL-agent module records the Q-values for each combination of a state and an action. Fig. 3c depicts a sample Q-table. A Q-value represents that the "cumulative value" of an action from a state, which is updated whenever the action is taken from that state. Note that the zero values indicate that the actions in the states have not been taken so far. The Q-value ($Q(S_t, A_t)$) is

TABLE II: The tested multi-workload scenarios.

Individual workloads

W1(Vdbench [3] BasicReadWrite (rdpct=80)) W2(Vdbench [3] WebServer) W3(RocksDB [7] + YCSB-A [6]) W4(RocksDB [7] + YCSB-D [6]) W5(RocksDB [7] + YCSB-F [6]) W6(Filebench [18] webproxy) W7(Sysbench [11] fileio) W8(Filebench [18] fileserver) W9(Filebench [18] varmail) W10(Vdbench [3] BasicReadWrite (rdpct=20)) W11(Fio [2] (rw=randrw, bs=16k))

Multi-workload scenarios

Scenarios	Label	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11
4-workload	S-1					•	•	•			•	
	S-2				•			•	•			•
	S-3	•	•				•	•				
	S-4			•				•	•	•		
	S-5		•	•					•	•		
	S-6				•			•		•	•	
	S-7	•			•	•					•	
	S-8	•				•	•		•			
5-workload	S-9	•	•		•			•	•			
6-workload	S-10			•		•	•		•	•	•	

calculated as a "weighted sum" of the old value and the learned value as follows:

$$Q(S_t, A_t) \leftarrow (1-\alpha) \cdot Q(S_t, A_t) + \alpha \cdot \{R_t + \gamma \cdot argmax_A Q(S_{t+1}, A)\}, \quad (1)$$

where α and γ are the learning rate and the discount factor, respectively. The learning rate (α) determines the weight between the old value (the former term) and the learned value (the latter term). For the learned value, the estimate of the optimal future value can be taken into account along with the reward (R_t obtained in ❺). Specifically, $argmax_A Q(S_{t+1}, A)$ indicates the maximum Q-value in the next state S_{t+1} (as a result of performing A_t in S_t). The discount factor (γ) determines how much we consider the future value for the learned value.

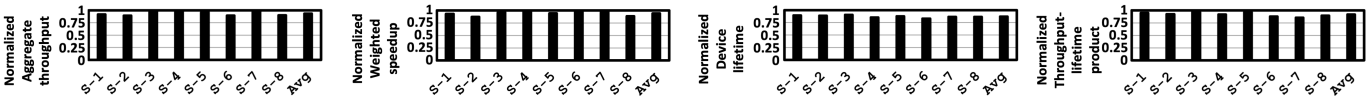
The Selection of an Action in a State: What action to be performed next is determined by the RL-agent module (❷-B) based on the ϵ-greedy policy of the Q-learning. Specifically, whenever an action is selected, a random number between 0 and 1 is picked. If the number is smaller than ϵ, an action is determined randomly among the possible actions. Otherwise, the action whose Q-value is the largest is picked from the Q-table. For example, in Fig. 3c, if the random number is larger than or equal to the ϵ value, then A_3 is selected in the current state S_{n-1} as it is the maximum Q-value in the state.

The Channel Allocation in SSD: In the current implementation, we use FEMU [13] for an emulated SSD, which offers a channel partitioning capability. As illustrated in Fig. 3d, the Dev-mgmt module, which is implemented in LightNVM (as part of FTL), is responsible for the management of the SSD (❹). Specifically, it (i) enforces an allocation, (ii) executes the workloads, and (iii) measures the optimization metric.

C. Increasing Allocation Granularity for Huge Search Space

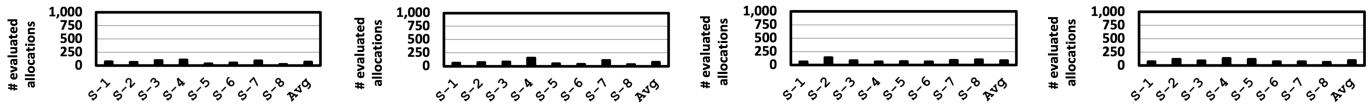
The search space of the channel allocation problem increases significantly as the number of available channels and co-running workloads increase. For example, the number of ways for allocating 64 channels to 7 workloads can increase to 1.6 million. Introducing an RL approach into such extreme-scale search space leads to a significant increase in the number of states and actions, which may not be feasible in an SSD or need a considerable amount of time to find an optimal allocation.

To this end, AutoAlloc can increase the allocation granularity for such large search spaces, thereby reaching a near-optimal allocation quickly. Specifically, the actions are modeled as increasing/decreasing 2, 3, or 4 channels at a time for



(a) Aggregate throughput (b) Weighted speedup (c) Device lifetime (d) Throughput-lifetime product

Fig. 4: The metric values for the found near-optimal allocations, which are normalized to those for the optimal allocations.



(a) Aggregate throughput (b) Weighted speedup (c) Device lifetime (d) Throughput-lifetime product

Fig. 5: The number of the evaluated allocations during the learning process (the total number of possible allocations is 969).

any two neighboring workloads. For example, in 4-workload scenarios, if the allocation granularity is set to 2 channels, the 9 actions are (0, 0, 0, 0), (+2, -2, 0, 0), (-2, +2, 0, 0), (0, +2, -2, 0), (0, -2, +2, 0), (0, 0, +2, -2), (0, 0, -2, +2), (+2, 0, 0, -2), and (-2, 0, 0, +2). Doing so enables us to pick and evaluate a representative allocation by skipping similar neighboring allocations. For example, employing a 2-channel allocation granularity in 4-workload, 32-channel scenarios evaluates (8, 10, 6, 8) without visiting (8, 9, 7, 8) or (8, 11, 5, 8). It is to be noted that, according to our analysis, there are little differences in the workloads performance among neighboring allocations.

D. Design Overheads

`AutoAlloc` requires to maintain two types of data structures in the host DRAM: (i) the Q-table in the RL-agent module and (ii) the collection of the metric values for the evaluated allocations in the Dev-mgmt module. The size of a Q-table is determined by the numbers of possible states and actions (as shown in Fig. 3c). The 4-workload, 32-channel problem including 969 states and 9 actions requires about 68KB for the Q-table (assuming that each Q-value needs 8 bytes), which is a quite marginal increase in memory consumption. Even in much larger search spaces such as the 7-workload, 64-channel problem, the number of the states can be limited to 12,376 if 2-channel allocation granularity is employed while the number of the actions is at most 15. Consequently, the size of the Q-table is 1.4MB, which is still affordable in the host system. Compared to the size of the Q-table, the total volume of the metric values for the evaluated allocations is much smaller; the maximum number of such values is the total number of the states. Assuming that each metric value requires 8 bytes, the 4-workload, 32-channel problem needs at most 7KB.

E. Discussion

Employing Other Allocation Units: While `AutoAlloc` targets the channel allocation problem, it can be applicable to the allocation problems based on other units such as planes or blocks, if the target SSD offers such opportunities. Note however that, the numbers of states and actions can increase significantly, as there can exist a large number of possible allocations based on such plane or block units.

Modeling the Actions in Different Ways: While we model the actions as increasing/decreasing the same number of channels only for any two neighboring workloads (Section IV-A) in the current version of `AutoAlloc`, there can exist various possible ways of defining the actions. For example, we tried to model actions as increasing/decreasing the same number of channels for all the combinations of any two workloads. However, such a choice led to an increase in the number of actions, and thus, required much more steps in the learning process (while ending up with comparable near-optimal allocations).

Impacts of the Key Parameters in Q-learning: There are three major parameters in the Q-learning algorithm, which are (i) learning rate (α in Equation 1), (ii) discount factor (γ in Equation 1), and (iii) the ϵ value of the ϵ-greedy policy. While (i) and (ii) affect the Q-values, (iii) is involved in the selection of actions. Even though one can vary these parameters when targeting different problem scenarios (e.g., workload counts, channel counts) and execution settings (e.g., target workloads, target devices), in the current version of `AutoAlloc`, we used fixed values for Equation 1 (i.e., α of 0.3, γ of 0.4), and gradually decreased the value of ϵ from 0.8 to 0.01 by 0.01 for each action, since our initial analysis has revealed that doing so consistently led to the best results across different scenarios.

V. Evaluation

A. Methodology

Testbed: We constructed a full-stack SSD system by using FEMU [13] with a channel partitioning capability. `AutoAlloc` is implemented in LightNVM [4], which enables it to manage channel allocations of the emulated SSD.

Device Configurations and Workloads: While `AutoAlloc` can be applied to any device configurations and multi-workload scenarios, in this study, we employed a 32-channel, 128GB SSD for various 4-workloads scenarios by considering evaluation time and available computing resources. To investigate larger search spaces, we also employed a 5-workload scenario and a 6-workload scenario, each of which is executed on a 64-channel, 256GB SSD. Various workload scenarios were constructed by selecting and combining candidates from 11 individual workloads with diverse I/O behaviors. The details of the device configurations and workloads can be found in Tables I and II.

B. Near-Optimal Allocations and Learning Steps

Fig. 4 shows the metric values for the near-optimal allocations that `AutoAlloc` finds, which are normalized to those for the "globally-optimal" allocations (determined by executing all the possible allocations), under different types of metrics. One can see that the metric values of the near-optimal allocations decrease quite marginally (by 8% on average), compared to those of the global-optimal allocations, (i) for the eight 4-workload scenarios and (ii) under the different metric types.

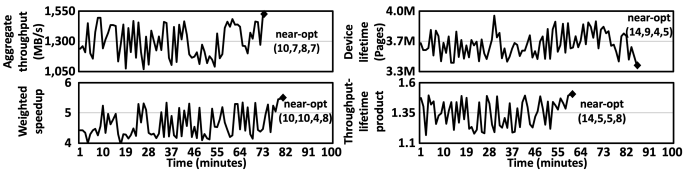
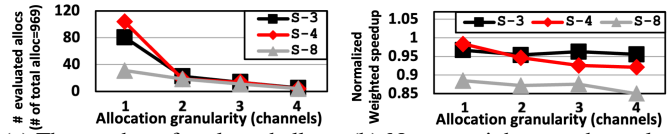


Fig. 6: The learning process of AutoAlloc for S-6.



(a) The number of evaluated allocs. (b) Norm. weight. speedup value

Fig. 7: The results of increasing allocation granularity.

Fig. 5 shows the number of evaluated allocations in the course of navigating over search space. In the 4-workload, 32-channel problem where there exist 969 possible allocations, AutoAlloc evaluates only an average of 77 allocations for the 8 execution scenarios and our 4 metric types. One can conclude that AutoAlloc can find a high-quality allocation after evaluating a very small number of allocations.

C. Allocations under Different Optimization Metrics

Since the optimal allocation can change depending on the target optimization metric, AutoAlloc navigates the search space in different ways based on the given metric types. Fig. 6 shows how AutoAlloc seeks the optimal allocation over time for a representative workload scenario (S-6) under our 4 metrics. One can see that AutoAlloc selects entirely-different candidate allocations and finally reaches different near-optimal allocations when targeting different types of metrics.

D. Impact of Increasing the Allocation Granularities

Fig. 7 plots the search results of AutoAlloc with increasing allocation granularity (from 1 channel to 4 channels) for the three scenarios (S-3, S-4, and S-8) under the weighted speedup metric. As shown in Fig. 7a, a larger granularity can reduce the learning process (i.e., the number of evaluated allocations) significantly, which is quite useful when the search space is very large. However, the quality of the selected allocation can decrease further from the optimal allocation, as illustrated in Fig. 7b. Note that, a larger granularity can find a better allocation than a smaller granularity by chance since (i) using different allocation granularities and (ii) the random nature of the Q-learning in the selection of next states can collectively lead to different series of evaluated allocations.

E. Extreme-Scale Search Space

Table III shows the results of two large-scale channel allocation problems, S-9 and S-10, on the 64-channel, 256GB SSD. For the former, the number of possible allocations is increased by 40×, compared to that in the 4-workload, 32-channel problem. However, AutoAlloc can find high-quality near-optimal allocations after exploring only 6.6% of the entire search space. For the latter where the search space is much larger, AutoAlloc with a 2-channel allocation granularity can reduce the total number of candidate allocations to only 11,628, and find near-optimal allocations after examining only an average of 429 different allocations.

TABLE III: The results in the extreme-scale problems.

Scenarios	Optimization Metrics	# of Evaluated Allocations		Optimization Metric Values	
		# of possible allocations	AutoAlloc	Globally optimal	AutoAlloc
S-9	Aggregate throughput	40,920	2,026	2,836.4	2,701.8
	Weighted speedup		3,107	5.6	5.2
	Device lifetime		2,252	4,332,288	4,879,040
	Throughput-lifetime product		3,488	0.0275	0.0262
S-10	Aggregate throughput	11,628	505	2,184.1	2,178.8
	Weighted speedup		397	8.827	8.5
	Device lifetime		375	5,500,224	5,829,248
	Throughput-lifetime product		439	0.0253	0.0245

VI. Conclusions

Allocating SSD channels to the co-running workloads is a critical but challenging task, due to a large search space that needs to be traversed and the lack of uniform definition of so called optimal allocation. We introduced a RL approach in such a context, and presented an autonomic channel-allocating SSD, called AutoAlloc. The navigation process of AutoAlloc is performed (i) at runtime, (ii) in a fully-automated fashion, and (iii) based on a given optimization metric. Our evaluations with various channel allocation scenarios demonstrate that AutoAlloc can find a near-optimal allocation after learning a quite small number of candidate allocations.

Acknowledgment

This work was supported by NSF grants 2211018, 2008398, 1822923, and 1908793. This work was also supported by MOTIE (1415181081), KSRC (20019402), and IITP (2022-0-00117, RS-2023-00221040). Dongjoon Lee and Jongin Choe are in the Department of Computer Science and Engineering (Major in Bio Artificial Intelligence) of Hanyang University ERICA. Wonil Choi is the corresponding author.

References

[1] "Open-channel solid state drives specification revision 2.0." http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf, 2018.
[2] J. Axboe, "Flexible i/o tester," https://github.com/axboe/fio, 2005.
[3] A. Berryman *et al.*, "Vdbench," https://www.oracle.com/downloads/server-storage/vdbench-downloads.html, 2010.
[4] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem," in *FAST*, 2017.
[5] J. Clifton and E. Laber, "Q-learning: Theory and applications," *Annu. Rev. Statist. Appl.*, 2020.
[6] B. F. Cooper *et al.*, "YCSB," https://ycsb.site, 2010.
[7] Facebook, "Rocksdb," http://rocksdb.org, 2011.
[8] J. Huang *et al.*, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds," in *FAST*, 2017.
[9] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in ssd," in *DAC*, 2018.
[10] ——, "Q-value prediction for reinforcement learning assisted garbage collection to reduce long tail latency in ssd," *TCAD*, 2019.
[11] A. Kopytov, "Sysbench," https://github.com/akopytov/sysbench, 2004.
[12] M. Kwon *et al.*, "Dc-store: Eliminating noisy neighbor containers using deterministic i/o performance and resource isolation," in *FAST*, 2020.
[13] H. Li *et al.*, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in *FAST*, 2018.
[14] R. Liu *et al.*, "Ssdkeeper: Self-adapting channel allocation to improve the performance of ssd devices," in *IPDPS*, 2020.
[15] W. Liu, J. Cui, T. Li, J. Liu, and L. T. Yang, "A space-efficient fair cache scheme based on machine learning for nvme ssds," *TPDS*, 2023.
[16] Z. Liu *et al.*, "Ocvm: Optimizing the isolation of virtual machines with open-channel ssds," in *ICA3PP*, 2020.
[17] Y. Pan *et al.*, "Lightwarner: Predicting failure of 3d nand flash memory using reinforcement learning," *IEEE Trans. Comput.*, 2023.
[18] V. Tarasov, E. Zadok, and S. Shepler, "Filebench," https://github.com/filebench/filebench, 2016.
[19] F. Xu *et al.*, "General feature selection for failure prediction in large-scale ssd deployment," in *DSN*, 2021.
[20] S. Yoo and D. Shin, "Reinforcement learning-based slc cache technique for enhancing ssd write performance," in *HotStorage*, 2020.