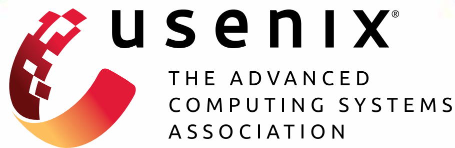


Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming

Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao,
University of California, San Diego

https://www.usenix.org/conference/atc21/presentation/huang-hanxian

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

**Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.**

Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming

Hanxian Huang Zixuan Wang Juno Kim Steven Swanson Jishen Zhao
University of California, San Diego

Abstract

Nonvolatile random-access memories (NVRAMs) are envisioned as a new tier of memory in future server systems. They enable a promising persistent memory (PM) technique, with comparable performance of DRAM and the persistence property of storage. However, programming PM imposes non-trivial labor effort on writing code to adopt new PM-aware libraries and APIs. In addition, non-expert PM code can be error-prone. In order to ease the burden of PM programmers, we propose *Ayudante*[1], a deep reinforcement learning (RL)-based PM programming assistant framework consisting of two key components: a deep RL-based PM code generator and a code refining pipeline. Given a piece of C, C++, or Java source code developed for conventional volatile memory systems, our code generator automatically generates the corresponding PM code and checks its data persistence. The code refining pipeline parses the generated code to provide a report for further program testing and performance optimization. Our evaluation on an Intel server equipped with Optane DC PM demonstrates that both microbenchmark programs and a key-value store application generated by Ayudante pass PMDK checkers. Performance evaluation on the microbenchmarks shows that the generated code achieves comparable speedup and memory access performance as PMDK code examples.

1 Introduction

Enabled by nonvolatile random-access memory (NVRAM) technologies, such as Intel Optane DIMM [33], persistent memory (PM) offers storage-like data persistence through a fast load/store memory interface [5, 63]. PM is envisioned to be a new tier of memory in future servers with promising benefits, such as fast persistent data access and large capacity.

However, the PM technique also introduces substantial challenges on programming. First, currently the burden of PM programming is placed on system and application programmers: they need to implement new PM programs or rewrite legacy code using PM programming libraries with a variety of programming interfaces and APIs [10,16,17,22,27,31,73]. Despite the promising development of libraries, implementing PM programs requires non-trivial labor efforts on adopting new libraries, debugging, and performance tuning. Second, the storage-inherited crash consistency property demands rigorous data consistency of the stand-alone memory system [44, 54] to recover data across system crashes; fully exploiting NVRAM's raw memory performance [15, 34, 76, 77, 80] requires programs to minimize the performance overhead of crash consistency mechanisms. As such, programmers need to understand the durability specifications and ordering guarantees of each PM library in order to provide sufficient persistence guarantee, while avoiding the performance overhead of adding unnecessary persistence mechanisms. Third, because various PM libraries provide different programming semantics, programmers need to manually transform the semantics, when switching from one PM library to another. As a result, PM programming is a tedious and time-consuming task even for the expert PM programmers, while imposing a painful learning period for non-experts. Moreover, the challenges significantly prolong the software development process and hold back the wide adoption of the PM technique.

In order to address these challenges, we propose Ayudante, a deep reinforcement learning (RL)-based PM programming framework to assist PM programming by transforming volatile memory-based code into corresponding PM code with minimal programmer interference. Ayudante consists of two key components as illustrated in Figure 1. First, Ayudante employs a deep RL-based code generator to automatically translate volatile memory-based C, C++, or Java code into corresponding PM code, by inserting PM library functions and instructions. Second, we design a code refining pipeline to parse the generated code to provide a report for programmers to further test, debug, and tune the performance of the code after the inference of our RL model. Ayudante intends to save the time and energy of programmers on implementing PM code from scratch. As such, Ayudante allows programmers to focus on leveraging or implementing volatile

[1] Ayudante source code: [1]

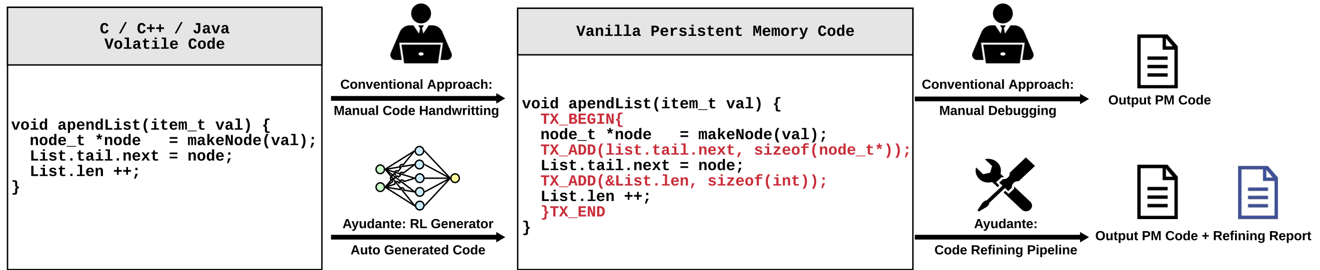


Figure 1: Ayudante framework overview. The framework takes conventional C, C++, or Java source code for volatile memory as an input and generates the corresponding PM code. Ayudante leverages a RL-based method to automatically translate the volatile version of source code to a nonvolatile version by inserting proper PM library annotations. In addition, Ayudante generates a code refining report to help reduce bugs and improve run-time performance.

memory-based code on traditional programming semantics that they are familiar with.

Due to the similarity to neural machine translation (NMT) problems in natural language processing (NLP), program translation is recently demonstrated to have promising results by adapting NMT techniques, such as sequence-to-sequence models [35, 79], word embedding [9, 12, 56], and tree-to-tree LSTM encoder-decoder on abstract syntax tree (AST) [11, 14]. However, the existing machine learning (ML)-based program translation methods focused on simple programs and data structures; the models fall short of handling sophisticated program syntax, data structures, and consistency reasoning, which yield large and complex search spaces [11, 14]. To this challenge, we integrate our RL model with Monte-Carlo tree search and carefully design our neural network architectures to improve generation efficiency. Furthermore, we adopt transfer learning to train the model for Java code generation based on the model trained for C and C++ languages to reduce training time. In summary, this paper makes the following contributions:

- We propose *Ayudante*, the first deep RL-based PM programming assistant framework, which automatically transforms volatile memory code to PM code. Our RL model mimics the behavior of expert PM programmers navigating through the input source code to add proper PM functions and instructions. We augment the RL model with Monte-Carlo tree search strategy to achieve efficient generation.
- We leverage a novel transfer learning model to transfer the PM programming semantics from the existing libraries of one programming language to another. In particular, this paper shows an example of transferring the knowledge of PM programming semantics from C/C++ to Java, saving training time for Java-based PM code generator. This approach sheds light on adapting PM code generation in various languages at low extra effort.
- We evaluate Ayudante with microbenchmarks incorporating various data structures and a key-value store application. Our results show that all the generated PM code passes PMDK checkers, with comparable performance on an Intel Optane DC PM server as code handwritten by experts.
- Ayudante assists novice PM programmers by reducing their time and energy spent on learning new PM libraries, automating the modifications on legacy code, and facilitating bug detection and performance tuning.

2 Background and Motivation

We motivate our Ayudante framework by PM programming challenges and opportunities in deep RL.

2.1 PM Programming Common Practice

PM systems introduce a new set of programming semantics that diverges from the conventional storage systems programming. Instead of extensively relying on slow system calls to access the persistent data, programmers now directly communicate with the byte-addressable nonvolatile main memory (NVMM) using load and store instructions. As PM combines the traits of both memory and storage, PM system requires crash consistency without hurting the memory-like access performance. One common practice of PM programming is to first use a PM-aware filesystem [20, 78] to manage a large memory region in NVMM. An application can then use a direct access (DAX) `mmap()` system call provided by the filesystem to map a nonvolatile memory region into its address space. From there, the application can directly access the NVMM. This programming model is portable and achieves high performance by reducing costly system calls that are on the critical paths [34, 80].

The PM programming model avoids directly using filesystem system calls for data accesses, making it difficult to use the conventional storage system's crash consistency and failure recovery mechanisms that extensively use system calls. As a result, PM programs need to maintain crash consistency and develop recovery mechanisms by themselves, rather than simply relying on the underlying filesystems. It is the programmers' responsibility to provide the crash consistency along with a proper recovery mechanism. Because PM programs rely on load and store instructions to access PM, a single mis-ordered store instruction or a missing cacheline

flush and write-back may make it impossible to recover to a consistent state after a system crash. As such, PM programs need to adopt (i) cacheline flush or write back instructions to ensure that data arrives at NVMM [20] and (ii) proper memory fence instructions to maintain the correct memory ordering [20]. To simplify the crash consistency guarantee in programming, many PM libraries [17, 22, 74] support fine-grained logging of the data structure updates, checkpointing [78], shadow paging [32], or checksum [40, 62] as the most common approaches to enforce failure atomicity to guarantee the recovery of data.

2.2 Challenges of PM Programming

Despite the promising performance and portability, the PM programming model imposes substantial challenges to software programmers on ensuring crash consistency and failure recovery, debugging, and performance optimization.

Labor Efforts of Adopting PM Libraries. Each PM library, such as PMDK [17], Atlas [10], go-pmem [22], and Corundum [31], typically defines its own failure model and provides a set of new programming semantics to hide the instruction level details. Although the high-level programming semantics ease the programming burden, it is still laborsome and error-prone to use those libraries as shown in Figure 3. In order for programmers to correctly use those libraries, they must learn the APIs and integrate them into the legacy code after fully understanding their own programming models.

We investigate code modification efforts of various PM programs as shown in Figure 2. It requires the change of 1547 lines of code (LOC) in order to transfer a volatile version of Memcached [28] into PM version [48] using PMDK library [17], which is 14% of the Memcached source code. Figure 2 also shows that other widely used data structures supported by PMDK require at least 15% LOC changes. It is invasive to perform such intensive modifications to a code base that is already complex and highly optimized for volatile memory operations.

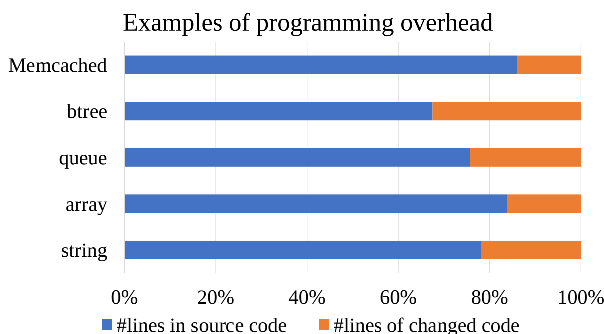


Figure 2: Proportion of lines of changed code to adopt a PM library, with PMDK microbenchmarks and a Memcached application.

Error-prone Non-expert Code and Debugging. Despite the high-level programming semantics provided by PM libraries, writing a PM code can be error prone as shown by previous studies [44, 45, 54]. It is typically programmer's responsibility to test and debug PM programs. Many crash consistency related bugs in PM code are hard to identify, as they may not interfere with program execution until the recovery stage after a system crash. Figure 3 shows such an example, where a programmer forgets to add a snapshot API offered by a PM library. The example executes normally but the program recovers to an undefined, non-consistent state undetected. Such bugs are not intuitive, and therefore extremely difficult to debug. Recent development of PM debugging tools leads to promising results [44, 45, 54]. However, the tools still rely on programmer's experience and knowledge on PM to annotate or navigate through the programs find bugs, making it hard to use by non-expert programmers.

```
1  int Queue::enqueue(...) {
2    ...
3    TX_BEGIN(pop) {
4      TX_ADD_DIRECT(&queue->back);
5      queue->back += 1;
6      TOID(struct entry) entry =
         TX_ALLOC(struct entry, sizeof(struct entry) + len);
7      D_RW(entry)->len = len;
8      memcpy(D_RW(entry)->data, data, len);
9      // the following snapshot code is missing:
10     // TX_ADD_DIRECT(&queue->entries[pos]);
11     queue->entries[pos] = entry;
12   } TX_ONABORT {
13     ret = -1;
14   } TX_END
15   ...
16 }
```

Figure 3: A buggy example of enqueue implementation using PMDK. The programmer forgets to add a snapshot function, and therefore violates the crash consistency requirement.

Performance Tuning. PM programming requires the cacheline flushes and fences to ensure that data updates arrive at NVMM in a consistent manner [13, 85]. These instructions introduce performance degradation by defeating the processor's hardware performance optimization mechanisms, including caching and out-of-order execution. Although it is generally acceptable to sacrifice certain levels of run-time performance to maintain data persistence, unnecessary cacheline flushes and fences will significantly hurt system performance, rendering undesirable deployment performance of PM in production environments. Therefore, it is essential to avoid using unnecessary cacheline flushes and fences. Furthermore, various PM libraries relax the crash consistency guarantee at different degrees for performance optimization. It is the programmers' responsibility to cherry-pick the optimal library for their application which in turn requires a vast amount of experience and prior knowledge.

2.3 Deep RL for PM Programming

We identify deep RL as a promising technique to perform automatic PM code generation, due to its powerful learning capability in tasks with a limited amount of available training

data [21, 29]. RL [69, 70] is a category of ML techniques that tackles the decision making problems. Deep RL [21, 29, 50, 51] augments deep neural networks with RL to enable automatic feature engineering and end-to-end learning [21]. We observed that deep RL is a better solution than RL for problems with higher dimensional complexity as it does not rely on domain knowledge. Due to these promising features, deep RL is widely used in games [51, 53, 68], robotics [36, 38, 60], and natural language processing [65, 75].

No end-to-end frameworks currently exist to automatically translate a piece of volatile memory code into PM code and perform crash consistency checking. State-of-the-art PM libraries [17, 22, 27, 31, 42, 47] and debugging tools [44, 45, 54] require programmers to annotate the code. Jaaru [24] does not require programmer's annotation but it relies on the program crash to detect bugs. It is also impractical to enumerate all the possible changes of each line of the code, while passing the checks of compilers and debugging tools. To address these issues, it is critical to automate PM code transformation, while achieving a high PM checker passing rate at a low transformation cost.

Fortunately, translating a volatile memory code into PM code can be formulated as a decision problem for sequential code editing, which is considered as solvable by deep RL. Moreover, augmented with Monte-Carlo tree search [8, 18] – a type of look-ahead search for decision making – deep RL is able to search the decision results more efficiently and effectively. Previous automatic code generation and translation studies focus on translation between different programming languages [41] and addressing debugging syntax errors [26]. To our knowledge, this paper is the first to formulate a ML-based PM program translation problem.

3 Ayudante Design

To embrace the opportunities and address the challenges described above, we propose Ayudante, a deep RL-based programming assistant framework as illustrated in Figure 1.

3.1 Ayudante Framework Overview

Ayudante consists of two key components: a deep RL-based code generator and a code refining pipeline. Our code generator takes conventional source code developed for volatile memory systems as the input and generates a vanilla PM code through a RL model. We design our RL model to mimic programmer's behavior on inserting PMDK library annotations into the volatile version of code. In order to reduce the training time and effort, we first train our model for C/C++ code by RL, and then employ transfer learning to adapt our model to Java programs. We show that our model is generalizable to various PM programs in our test set on the open source Leetcode solution programs (Section 5). Our code refining pipeline integrates multiple PM checking and debugging tools to generate a report of syntax bugs (if any) and suggestions on run-time performance optimization. The report allows programmers to further improve and test the code.

Ayudante offers the following promising automated characteristics in assisting PM programming:

- **Efficient PM code generation** through a deep RL model augmented with Monte-Carlo tree search, which efficiently searches the correct code edits with significantly smaller search space.
- **Reduced bugs** through a deep RL model pre-trained to avoid bugs detected by checkers in the training environment.
- **Code refining reports and improved performance** through a code refining pipeline for programmers to further inspect the possible improvements to the generated programs if necessary.

3.2 Deep RL-based Code Generator

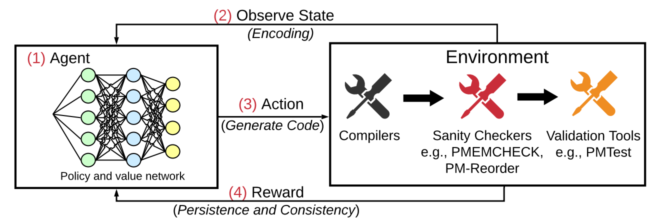


Figure 4: Ayudante's deep RL model consists of an agent and an environment. During model training, the agent repeatedly generates and sends actions to the environment based on the rewards and states it receives.

We design a deep RL network that generates the PM code. The trained network mimics the programmer's behavior; it navigates through the input source programs and adds proper code in the corresponding locations. Figure 4 shows Ayudante's deep RL model, which consists of (a) an agent with a *policy and value network* and (b) an *environment*. The policy and value network responds to a state (the encoded input source code) and outputs an action (which navigates through source code and inserts PM library functions). The environment applies the action to the last code state to generate a new code state, and tests it by a set of PM code checkers to return a reward to the agent. Details of integrating various PM checkers in the environment are discussed in Section 4.4.

The model is trained on a volatile version of PMDK example code [17] (by removing PM annotations). During training, the policy and value network is updated for a better action policy according to the reward function. After training offline, the RL generator performs online inference to generate the best actions and output a PM code according to the pre-trained policy and value network. We test on the data structures from open-source Leetcode solution code [6, 61, 83]. In the following, we describe the details of our RL and transfer learning models. Section 4 will discuss detailed implementation of training and inference of our models and the datasets used for training and testing.

3.2.1 Agent, State, Action, and Reward

As shown in Figure 4, our RL model consists of four major components: agent, state, action, and reward.

Agent. An agent is an executor of actions on a state. In our model, a state is a representation of source code, while an action is navigating through source code or inserting one PM library API function to source code. The agent repeatedly obtains a new state from the environment, executes actions on the state, and re-compiles the modified code if necessary, until the model outputs a complete PM code that passes the checkers in the environment.

State. A state is a pair of *string* and *position*. A string is a source code in plain text, while a position represents the current position in the code to take an action at. To preprocess the input and make the source code recognizable by the RL model, we perform an *encoding* operation, which transforms the source code into a sequence of *tokens* to feed into the RL model. The encoding using an Encoder-Decoder long short-term memory (LSTM) autoencoder [30] to fit sequence data. Once the autoencoder fits, the encoder in the model is used to encode or compress sequence data as a feature vector input to the RL model. Each encoded token represents either the program text or a program location for an API insertion. The string is initialized in a tokenized form of the input source code. The program location token is initialized to the first token in the program. We encode the program text, augmented with the program location using an LSTM network [30] and feed the state into the policy and value network. When taking an action, the modification is executed on the original program by the environment before compilation and testing.

Action. Our RL model has two types of actions: (1) navigation, which jumps to a new program location and (2) edit, which modifies the string (i.e. modifies the source code). The two actions are non-overlapping: navigation action does not update the code, while edit action only adds a PM library API function to the current program location without changing the location. We define two navigation actions, move-down and move-out. Move-down sets the program location to the next token. Move-out moves the program location to the current curly braces. A combination of these two navigation actions allows the model to enumerate all possible insertion locations of PM library APIs.

The edit action utilizes the possible library API annotations, which we manually select from PMDK [17]. The edit action either annotates one line of code or wraps a block of source code into a pair of annotations (*e.g.,* wrapping with `TX_BEGIN` and `TX_END`). We do not consider deletion of previous API annotations, because it is identical to do nothing in the deleted locations. The invalid edit action that causes syntax errors or bugs will be rejected by the environment and punished by the reward function.

Reward. In a typical RL model, a reward is used to measure the consequences of the actions and feedback to the agent to generate better actions. In our training process, the agent performs a set of navigation and edit actions to the program, and receives either the maximum reward if the generated code passed all the PM checkers in the environment, or a small reward formulated as follows, which consists of three penalties – step, modification, and code error reported by checkers, respectively:

$$r_t = \phi_1 \cdot lnS + \phi_2 \cdot lnM + \sum_{i=1}^{n} \rho_i \cdot E_i \tag{1}$$

The step penalty is defined by a penalty factor ϕ_1 and a step number S. A step refers to an iteration of taking a new action and feedback the corresponding reward. In each step, the agent is penalized with a very small step penalty (with a small ϕ_1) to motivate the agent to take fewer steps.

The modification penalty is defined by a factor ϕ_2 and a modification number M. ϕ_2 penalizes unnecessary edits and encourages the RL generator to complete the PM code with fewer modifications.

The code error penalty is defined as a summation of penalties given by multiple testing tools. For tool i, ρ_i represents the impact factor of the tool (i.e. how important the tool is), while E_i is the number of errors reported by the tool. The code error penalty penalizes those actions that introduce bugs and encourages the RL model to learn to avoid the bugs detectable by the checkers in the environment. The ρ_i can be tuned to give more attention to testing tool i, so as to reduce the corresponding bugs detectable by this tool in the ultimately generated code. In our model, the number of errors E_i is a few magnitudes lower than the number of steps S. Therefore, we use lnS in the reward instead of S to balance the penalties. The same reason applies to lnM.

3.2.2 An Example of RL-based Code Generation

Figure 5 shows an example PM code generated by our RL model. To generate such code in inference, our trained model takes an input of a pure C code (in black color in Figure 5) that is never seen during training. Our model encodes the code into tokens, then performs actions on the tokens as marked by arrows in Figure 5. At each step t, the agent of the model (i) retrieves a state $s_t \in S$ from the environment and (ii) selects an action a_t from all candidate actions with the maximum conditional probability value generated by the policy and value network. In Figure 5, the agent first chooses navigation actions for step ❶ and ❷, then an edit action for step ❸. At this point, as the agent changes the state (the source code), the environment executes the edit action to generate a new state s_{t+1} and a corresponding scalar reward r_t. Similarly, the agent performs edit actions at steps ❼ and ❿, and therefore generates a complete PM code. The generated code is further verified by the environment using a pipeline of multiple PM testing tools shown in Figure 4. In this example, the generated code passes all the tests. The code refining pipeline

(Section 3.3) will automatically generate code refining suggestions if it identifies bugs or performance degradation that requires programmer's attention.

```
1  int64_t Queue::pop(){
2    int64_t ret = 0;
3    auto pool = pmem::obj::pool_by_vptr(this);
4    obj::transaction::run(pool, [this, &ret] {
5      if (head == nullptr)
6        throw std::runtime_error("Empty queue");
7      ret = head->value;
8      auto n = head->next;
9      obj::delete_persistent<Node>(head);
10     head = n;
11     if (head == nullptr) tail = nullptr;
12   });
13   return ret;
14 }
```

Navigation Action ← Edit Action

Figure 5: An example of a sequence of actions taken by the trained agent to generate PM code based on its volatile memory version.

3.2.3 Policy and Value Network

The *policy and value network* (Figure 4) determines which action to take based on an action-value function. In Ayudante, we use deep Q-learning [50, 72], a representative and widely adopted policy and value network. Q-learning is model-free and generalized not to depend on any specific RL models. In Q-learning, the function Q calculates an expected reward given a set of states and actions, which includes the reward of future subsequent actions. In Ayudante, we use deep Q-learning to combine Q-learning with a deep neural network to form a better end-to-end generator. The function Q is defined as:

$$Q(s_t,a_t) = (1-\alpha)Q(s_t,a_t) + \alpha(r_t + \gamma \cdot \max_{\alpha} Q(s_{t+1},a)) \quad (2)$$

where t represents a time step that requires an action to be taken; s_t is a sequence of actions; a_t is the action; r_t is the reward. The decay rate is $0 \le \alpha \le 1$. The discount factor is $0 \le \gamma \le 1$. We apply such a Q function to our Deep Q Network, working as an iterative decision making algorithm outlined in Algorithm 1. The objective of deep Q-learning is to minimize $(Y - Q(\phi, a; \theta))^2$ based on sequences of actions and observations $s_t = x_1, a_1, x_2, a_2, ..., a_{t-1}, x_t$. Here, Y represents the expected reward, while $Q(\phi; a; \theta)$ is the reward calculated from the Q function, with trainable weight parameters θ. The Q function works on fixed-length representation of code modification histories collected by function ϕ. M is the maximum number of epochs and T is the number of iteration to modify the code used to simulate the descent process, which are user-defined parameters. This process will also generate the training datasets of Q-learning, stored in a replay memory $\mathcal{D}$, with a maximum capacity N. When $\mathcal{D}$ is inquired for a minibatch of transitions, it will return Y_j and $Q(\phi, a_j; \theta)$.

Algorithm 1 The policy and value network function.

1: Initialize replay memory $\mathcal{D}$ to capacity N and random initialize θ
2: **for** epoch from 1 to M **do**
3:   Initialize sequence $s_1 = \{x_1\}$ and $\phi_1 = \phi(s_1)$
4:   **for** t from 1 to T **do**
5:     With probability $< \varepsilon$ select a random action a_t
6:     Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
7:     Execute action a_t: navigate or insert an API
8:     Get reward r_t, and next state x_{t+1}
9:     Set $s_{t+1} = s_t, a_t, x_{t+1}$ and $\phi_{t+1} = \phi(s_{t+1})$
10:     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
11:     Sample a minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$
12:     Set $Y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$ for a non-terminal ϕ_{j+1} or $Y_j = r_j$ for a terminal ϕ_{j+1}
13:     Minimize Loss $(Y_j - Q(\phi_j, a_j; \theta))^2$
14:   **end for**
15: **end for**

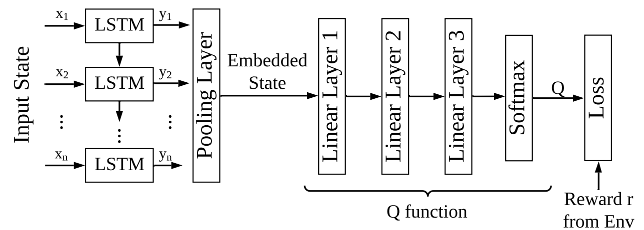


Figure 6: Neural architecture of the policy and value network.

Figure 6 shows the neural network architecture of our policy and value network. We first embed the input source code with LSTM and a pooling layer. Then, we pass the embedded state to three fully-connected layers. The Softmax function will output the Q value; the action will be selected either by maximizing the reward or with a small probability to play a random action. Finally, we calculate the loss function by the real reward r and the Q value, and update the trainable parameters θ in the Q function. Here we adopt two sets of parameters (θ and θ') in the same shape. One is for selecting an action and another one is for evaluating an action. They are updated alternatively.

3.2.4 Monte-Carlo Tree Search

Searching for correct edits is non-trivial due to two key challenges – exponentially large search space and unnecessary decision search. First, with the increase of lines of code in programs and the number of actions, the search space grows exponentially. This leads to exponential search time using uninformed search algorithms such as enumerative search [3, 4]. Second, a correct decision search requires both localize and make precise edits to generate a PM code as illustrated in

Figure 5. An invalid edit that introduces bugs leads to unnecessary decision search, if the model is unaware of such bugs and proceeds to search for new actions. To tackle the first challenge, we adopt an efficient search strategy Monte-Carlo Tree Search (MCTS) [67], which is highly efficient and effective in searching exponentially increased states. To address the second challenge, we combine the policy and value network (Section 3.2.3) with MCTS to guide selection towards potentially good states and efficiently evaluate the playout.

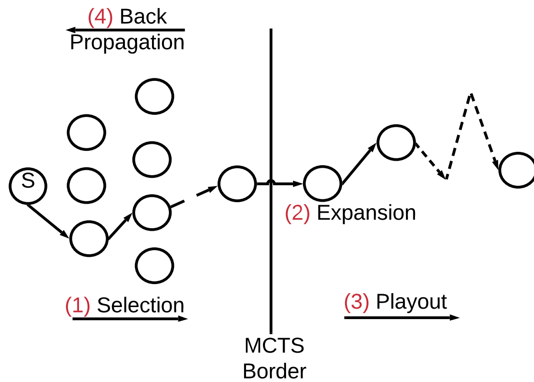


Figure 7: Overview of a simulation of Monte-Carlo tree search.

As shown in Figure 7, one round (or "simulation") of MCTS consists of four phases. (1) **Selection** starts from the root node and selects the successive child nodes until reaching a leaf node. Here, the root node represents the current state, while a leaf node is a potential child node with at least one unexplored child (i.e., not covered by previous search). We employ a widely-used Upper Confidence Trees (UCT)-based strategy [37] to select the node with the maximum value V

$$V = q_i + c\sqrt{\frac{lnN_i}{n_i}} \tag{3}$$

where q_i is the current action-value estimate after the i-th move; n_i is the number of simulations for the node considered after the i-th move; N_i is the total number of simulations after the i-th move run by the parent node of the one considered. c is the exploration parameter that is theoretically equal to $\sqrt{2}$; in practice, it is typically chosen empirically. The first term is high for a move that gets a high successful edits rate. The second term is high for a move with few simulations. Therefore, a large V value will lead the search to a better final solution. (2) **Expansion** unless reaches a goal state, create one or more valid child nodes. (3) **Playout** instead of evaluating the position after running a full simulation and sampling the moves until reaching a goal state in the vanilla MCTS, we approximate the value of the position by the deep Q-learning network. (4) **Back-propagation** utilizes the result from playout to update node information in reverse order.

3.2.5 Transfer Generating Knowledge of C/C++ to Java

To fully leverage the code translation knowledge learned from the RL generators for C and C++, we train a Java generator model by employing transfer learning [64]. Transfer learning is an effective technique to take features learned from one domain and leverage them in a new domain with certain similarity, to save training time and achieve higher inference accuracy on the new domain. The most commonly used transfer learning technique is fine-tuning [82], which freezes (sets as un-trainable) some layers of a pre-trained model and train the other trainable layers on a new dataset to turn the old features into predictions on the new dataset. However, the fine-tuning is a destructive process, because the new training will directly discard the previously learned function and lose the pre-trained knowledge on generating C/C++ code.

To address this issue, we employ a progressive network technique [64]. Progressive network also leverages the pre-trained models, whereas utilizing the output rather than the pre-trained parameters. Doing so overcomes the problem of discarding the prior knowledge in further training. Given a pre-trained RL generator G_j, with hidden activations $h^{(1)_j} \in R^{n_i}$ from layer i, where n_i is the number of units at layer i, the progressive network utilizes a single hidden layer multi-layer perception σ to adjust different scales of various hidden layer inputs and adapt the hidden input to the new model by

$$h_i^k = \sigma(W_i^{(k)} h_{(i-1)}^k + U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})) \tag{4}$$

where the $h_i^{(k)}$ is the hidden activations for layer i in task k, $W_i^{(k)}$ is the weight parameters for layer i in task k, $U_i^{(k:j)}$ are the lateral connections (a $n_i \times m_j$ matrix) from layer $i-1$ of task j to layer i of task k, $V_i^{(k:j)}$ is the projection matrix to be trained, and α is a learned scalar initialized to a small random value. The progressive network fully exploits the hidden knowledge represented by the hidden activation in multiple similar tasks in sequence.

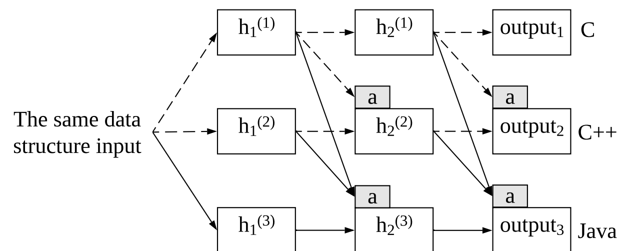


Figure 8: The workflow of the progressive network to transfer from generating C and C++ code to generating Java code, where a refers to the adaptor function shown by Equation 4.

By adopting fine-tuning and progressive networking, we take advantage of the generators for C and C++ which are well-pretrained on a sufficient training dataset and show the

potential of assisting writing PM code for other programming languages.

3.3 Code Refining Pipeline

Previous ML-based code generators pay much attention to improving the inference accuracy. Yet, even on well-investigated translation tasks (*e.g.,* CoffeeScript-JavaScript Task [14] and FOR/LAMBDA translation task [19]) on popular programming language (e.g., C/C++/JAVA/Python), state-of-the-art deep learning models only achieve up to 70% $\sim$ 90% inference accuracy [14, 19]. Due to the inherent fuzzy and approximate nature of deep learning, it is impossible to achieve 100% accuracy for all generated programs only by an ML model, let alone many complicated bugs that require programmer's domain knowledge to detect and fix.

To address the limitations of ML-based method, we design a code refining pipeline to search for bugs that may remain in the code generated by the RL model. As shown in Figure 4, we incorporate a set of code refining tools in three levels: (1) *compilers* check on the syntax and library API usage bugs; (2) *sanity checkers*, including PMEMCHECK and PM-Reorder [17], check on the consistency and reordering bugs; (3) *validation tools*, including PMTest [45], XFDetector [44], and AGAMOTTO [54], perform a deep bug search in terms of crash consistency and failure recovery. We organize these tools as a pipeline to ensure that high-level bugs (*e.g.,* syntax bugs) are identified as early as possible before starting the time-consuming deep bug search. The output of the code refining pipeline is a code refining suggestion report for programmers to further manually inspect the code.

Figure 9 shows an example output of our code refining pipeline. In this example, the vanilla generated code is called `node_construct` followed by a `pmemobj_persist`. Our code refining pipeline identifies that `node_construct` already persists the tree node data. Therefore, there is no need to persist it again. This optimization suggestion is reported to the programmer, who decides to remove the redundant persistence API call, leading to the improved code.

Vanilla generated code

```
int node_construct() {
// set up btree node
...
pmemobj_persist(pop, node,
                a->size);
// persist data
...
}
```

Call

```
void btree_insert() {
// set up btree stucture
...
POBJ_ALLOC(..., node_construct);
pmemobj_persist(pop, dst, args.size);
// duplicated persistence
...
}
```

Improved code

```
int node_construct() {
// set up btree node
...
pmemobj_persist(pop, node,
                a->size);
// persist data
...
}
```

```
void btree_insert() {
// set up btree stucture
...
POBJ_ALLOC(..., node_construct);
...
}
```

Figure 9: A piece of B-Tree code improved by the code refining pipeline.

4 Implementation

4.1 Training and Testing Dataset

We are the first to develop both training and testing datasets for ML-based PM code translation problems. We preprocess the datasets by including header files, creating the pool file (memory-mapped file) and initializing the root object for further read and write operations.

Training Set. We use the dataset from the PMDK library [17] with 18 C code examples, 14 C++ programs, and two Java programs. Because the PMDK example code is the nonvolatile version expert code, we obtain a volatile version of each program by manually discard the PMDK APIs in the code.

Testing Set. To show the generalization ability of our model, we test it with code never seen by the model and measure how well the model generates for such new code in the inference process. Our testing dataset consists of volatile code of 48 C programs, 42 C++ programs, and 13 Java programs obtained from the open-source Leetcode solution [6, 61, 83]. These programs perform similar functionality as PMDK example programs on various data structures widely used in persistent memory applications, including array, string, list, queue, btree, rbtree, hashmap, and combinations of multiple data structures.

4.2 Training Configuration

Deep RL Implementation. We implement the RL generator in PyTorch [58]. We identify the suitable hyper-parameters of the RL generators and update the weight parameters in the policy and value network in the training process. In particular, the LSTM encoder in our model has two recurrent layers, each with 128 cells. The *string* has on average 112 tokens, which are embedded into 48-dimensional vectors. The LSTM encoder for state embedding and the policy network are trained together in an end-to-end manner. We set the decay rate $\alpha = \frac{K}{K+T}$ to achieve the temporal decay, with $K = 10$ and T as the epoch number. We set the discount rate γ as 0.95. For the reward function, when using different checkers, the relative relationship between step, modification, and code error penalties can be different. In our model, we set step penalty factor ϕ_1 as -0.01 and modification penalty factor ϕ_2 as -0.005. For the impact factor of each tool in the code error penalty, we set $\rho_1 = -0.1$ and $\rho_2 = -0.06$ and $\rho_3 = -0.065$ for PMEMCHECK, PM-Reorder and PMTest respectively. We set the maximum reward as 1. We train the RL generator on two Nvidia GeForce GTX 2080 Ti GPUs with 11 GB Memory for 60 epochs.

Transfer Learning Implementation. The progressive network is also implemented in PyTorch [58], and trained on two NVIDIA GeForce GTX 2080 Ti GPUs with 11GB Memory for 30 epochs. We use a dynamic learning rate scheduler in PyTorch with an initialization of 0.001 for progressive networks.

4.3 PMDK APIs

PMDK [17] is a collection of libraries developed for PM programming. These libraries build on the DAX feature that allows applications to directly load and store to PM by memory-mapping files on a PM-aware file system. We employ the *libpmemobj* library, which provides a transactional object store, memory allocation, and transactions with atomicity, consistency, isolation, and durability for PM systems. We pick 21 library functions (such as `pmemobj_persist`, `pmemobj_create`, `pmemobj_open`) and 78 macros (such as `TOID`, `POBJ_ALLOC`, `D_RW`, `D_RO`, `POBJ_ROOT`). For C++ code, we employ *libpmemobj-cpp*, a C++ binding for *libpmemobj*, which is more approachable than the low-level C bindings. For Java code, we use persistent collections for Java based on the libpmemobj library.

4.4 Checkers

We use checkers in both the *environment* of our RL model and the code refining pipeline. As illustrated in Figure 4, the checkers are organized into three levels. First, we use compilers (such as gcc) to detect syntax errors. Second, we use sanity checkers, such as PMEMCHECK and PM-Reorder, to detect consistency and reordering bugs. Finally, we use high-level validation tools, such as PMTest [45], XFDetector [44], and AGAMOTTO [54], to further capture durability and ordering violations in PM operations, cross-failure bugs [44], and redundant cache line flushes and fences [54].

As demonstrated in Section 5.1, the more checkers used, the higher the PM checker passing rate and robustness. However, increasing the number and complexity of checkers also leads to a much longer training time. Therefore, we only use PMTest [45] in the *environment* in RL model training. The code refining pipeline adopts all three aforementioned high-level validation tools one after another to validate a program generated by our RL model. As these are independent of each other, the order of running the tools does not matter. In the following, we discuss the implementation details of integrating various checkers in Ayudante.

Compilers. We adopt `gcc` version 7.5.0. as the compiler in this paper.

PMEMCHECK. PMEMCHECK is a persistent memory analysis tool that employs the dynamic memory analysis tool Valgrind [55], to track all stores made to persistent memory and inform programmers of possible memory violations. Other than checking and reporting the non-persistent stores, PMEMCHECK also provides other options to look out for memory overwrites, redundant flushes, and provides transaction checking such as check stores that are made outside of transactions or regions that overlapped by different transactions. Here we mainly feedback the error number from the error summary reported by PMEMCHECK to the reward in the model.

PM-Reorder. PM-Reorder is another tool for persistence correctness checking. It will traverse the sequences of stores between flush-fence barriers made by the application, and replays these memory operations many times in different combinations, to simulate the various possible ways the stores to the NVDIMM could be ordered by the system. Given an exhaustive consistency checking function, this process will uncover potential application bugs that otherwise could have been encountered only under specific system failures. It provides various optional checking orders, such as *ReorderFull* to check all possible store permutations, *ReorderPartial* to check 3 random order sequences, *ReorderAccumulative* to check a growing subset of the original sequence. Here we mainly use *ReorderPartial* to achieve consistency checking while keeping training efficiency. PM-Reorder requires users to provide a user-defined consistency checker, which is a function that defines conditions necessary to fulfill the consistency assumptions in source code and returns a binary value (0 represents consistent and 1 represents inconsistent). With each value in a data structure, Ayudante provides a default consistency checker, which determines whether each value field is properly assigned compared to the number that we store in the value field [59]. For example, if the main function sets nodes 0, 1, and 2 of a list as 100, 200, and 300, Ayudante will provide a consistency checker that traverses these three nodes to evaluate whether their corresponding values are correctly assigned or not. Here we mainly leverage the inconsistency number reported by PM-Reorder.

PMTest. PMTest is a fast and flexible crash consistency detecting tool, which reports violations in durability and ordering of PM operations, such as whether a persistent object has been persisted, ordering between persistent updates, and unnecessary writebacks or duplicated logs. With C/C++ source code, Ayudante automatically generates annotations using the C/C++-compatible software interface offered by PMTest, including (i) wrapping the entire code with `PMTest_START` and `PMTest_END` functions and (ii) using `TX_CHECKER_START` and `TX_CHECKER_END` to define the boundary of each transaction as required by the PMTest high-level checkers. These annotations will be removed after testing the generated code. We use the high-level checkers to validate three items: (1) the completion of a transaction, (2) the updates of persistent objects in the transaction are recorded in the undo log before modification, and (3) the code is free of unnecessary writebacks or redundant logs that constitute performance bugs. An issue with (1) or (2) will be reported as a `FAIL`, while issues with (3) are identified as `WARNING`s. During the training process, we use the number of `FAIL`s from PMTest as feedback to the neural network. In the refining pipeline, we use the `WARNING` information to suggest code refining on removing the redundant writebacks and logs.

XFDetector. XFDetector detects cross-failure bugs by automatically injecting failures into a pre-failure execution; it also checks cross-failure races and semantic bugs in the post-failure continuation. With C/C++ source code, Ayudante generates the annotations by wrapping the

Table 1: Intel server system configuration.

CPU	Intel Cascade Lake engineering sample 24 Cores per socket, 2.2 GHz 2 sockets, HyperThreading off
L1 Cache L2 Cache L3 Cache TLB	32KB 8-way I$, 32KB 8-way D$, private 1MB, 16-way, private 33MB, 11-way, shared L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries
DRAM	DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket
NVRAM	Intel Optane DIMM, 256 GB, 2666 MHz 2 sockets, 6 channels per socket
Kernel	Linux 5.1
Software	GCC 7.1, PMDK 1.7

code with `RoIBegin(1, PRE_FAILURE|POST_FAILURE)` and `RoIEnd(1, PRE_FAILURE|POST_FAILURE)` functions offered by XFDetector's C/C++-compatible interface to define the region-of-interest (RoI); such annotations will be removed after code testing.

AGAMOTTO. AGAMOTTO detects persistency bugs using two universal persistency bug oracles based on the common patterns of PM misuse of C++ code: it identifies (i) modifications to PM cache lines that are not flushed or fenced; (ii) duplicated flushes of the same cache line or unnecessary fences. AGAMOTTO symbolically executes PM workloads to identify bugs without requiring annotations.

5 Evaluation

Experimental Setup. Table 1 shows configuration of the Intel server adopted in our experiments. We configure the Optane DIMMs in App Direct interleaved mode [33] and employ ext4 filesystem in DAX mode. We disable hyper threading and boost the CPUs to a fixed frequency to maintain a stable performance. We run each program 10 times and report the geometric mean.

5.1 PM Checker Passing Rate in Inference

We employ PM checker passing rate (CPR) defined in Equation 5 – the percentage of generated code that passes all the PM checkers – to measure the generalization ability of the trained model.

$$CPR = \frac{\#PassCheckers}{\#Generated} \tag{5}$$

In our experiments, we use PMEMCHECK as the checker to verify the PM CPR in inference. We train three versions of RL generators with various checker combinations in the environment, including (1) PMEMCHECK; (2) PMEMCHECK and PM-Reorder; (3) PMEMCHECK, PM-Reorder, and PMTest. separately and Table 2 shows the CPR in inference tested on microbenchmarks, a key-value store application, and Leetcode solution set. Our results show that all the generated code of RL generator with checker combinations (2) and (3) passes the checkers with the key-value store workload testing and microbenchmarks incorporating array, string, list, queue, btree, rbtree, and hashmap data structures. Our experiments also show that Ayudante can handle programs with multiple data structures. For example, the Leetcode solution for merging k-sorted lists [7] adopts both `list` and `queue` data structures; the generated code makes both data structures persistent and passes the PMEMCHECK checker. Moreover, we observe that the more checkers used to feedback the reward, the higher CPR is achieved in inference. This is because checkers will complement each other to penalize different types of bugs and encourage more precise edits on the source code to avoid the bugs. Note that none of the existing ML-based models can achieve 100% inference accuracy due to the inherent approximate nature of ML. However, our method improves the CPR in inference by effectively taking advantage of different checkers in the training process and the refining pipeline to further report the testing and show improvement suggestions.

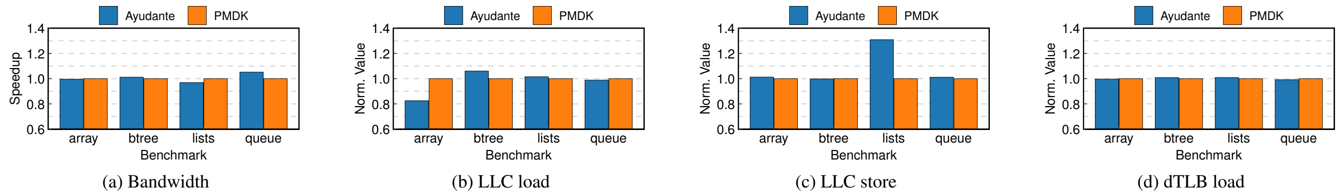
Table 2: The PM CPR in inference and the average percentage lines of code (LOC) changes, compared among using various checkers in the environment during training process of deep RL-based code generator.

Testing Set	Checkers in Environment	CPR	LOC
Microbenchmarks and KV store application	PMEMCHECK	87.5%	12.3%
	PMEMCHECK & PM-Reorder	**100%**	13.4%
	PMEMCHECK & PM-Reorder & PMTest	**100%**	13.8%
Leetcode solution	PMEMCHECK	60.2%	12.5%
	PMEMCHECK & PM-Reorder	62.1%	13.1%
	PMEMCHECK & PM-Reorder& PMTest	78.7%	13.4%

5.2 Execution Performance

We also evaluate the execution performance of the generated code under various input sizes by running the programs on an Intel Optane DIMM server configured as Table 1. We show the memory bandwidth performance in Figure 10a and Figure 10 on `lists`, `array`, `queue`, and `btree` microbenchmarks. As shown in Figure 10a, the generated code achieves a similar bandwidth compared with expert code provided by PMDK code examples. We also employ `perf` [25] to profile the last level cache load, store, and data TLB (dTLB) load events. As shown in Figure 10, the generated code has a cache and TLB performance comparable to PMDK. Although `lists` has a higher number of (30%) LLC store events due to unnecessary persistence API calls, it does not significantly affect the overall program execution performance.

We further evaluate the scalability of generated code by investigating the performance with different input sizes. Figure 11 shows that the bandwidth of the generated code is comparable to PMDK examples. Therefore, our code generator



(a) Bandwidth (b) LLC load (c) LLC store (d) dTLB load

Figure 10: Performance evaluation of code generated by Ayudante, on bandwidth, LLC load, LLC store, and dTLB load on Intel server with Optane DC PM. The bandwidth is calculated based on the modified data structure size. Numbers are normalized to corresponding PMDK microbenchmark performance.

is highly stable and reliable in generating high-performance PM code.

5.3 Bug Avoidance and Reduction

Ayudante employs two mechanisms to avoid and reduce both syntax and performance bugs.

Avoidance. During the training process, we utilize the checkers to penalize our model from buggy edits to avoid bugs in the generated code. We use PMEMCHECK to penalize non-persistent stores and misused transactions. We adopt PMReorder to penalize in-consistency stores. We employ PMTest to penalize non-persistent objects, wrong ordering, and redundant logs. As an example in practice, we observe 17 non-persistent errors in an intermediate code state during the training process; after training, all the bugs are eliminated after step-by-step searches and edits. This also demonstrates that the RL generator is able to help to debug.

Reduction. Beyond the sanity checks using PMEMCHECK and PMReorder, we also design a code refining pipeline to further perform deep bug search and code refining suggestions generation. An example is shown in Section 3.3 to demonstrate the results of the code refining pipeline.

5.4 Labor Effort Reduction

We evaluate the reduction of programming labor effort with average lines of code (LOC) changed as shown in Table 2. In our experiments, the percentage of LOC changes is typically $12\% \sim 15\%$, which significantly reduces the labor effort on developing PM applications. We test the LOC of three different versions of models by adopting different numbers of checkers. Intuitively, more checkers leads to more robust generated code, hence more APIs inserted to guarantee consistency; this results in more LOC changes. However, the difference of the total LOC is small among different versions of models, while our models bring significant CPR improvement as demonstrated in Table 2.

6 Discussion

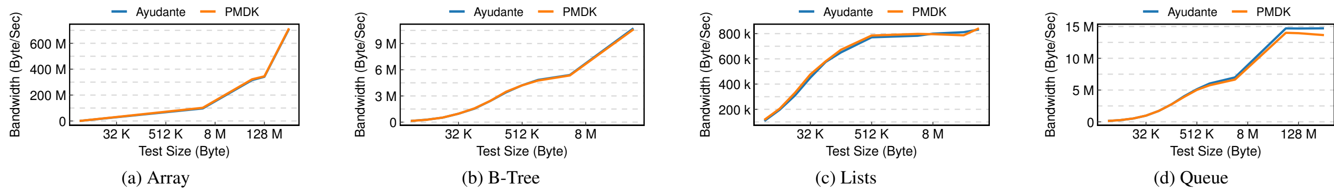
Limitations of ML-based Approaches. Due to limitations of ML as discussed in Section 3.3, it is impossible for an ML model to achieve 100% inference accuracy with all programs due to the inherent fuzzy nature of ML [11, 14, 19]. In fact, inference accuracy improvement remains a critical challenge in ML community [23,49]. To address the accuracy limitation, Ayudante's code refining pipeline effectively reduces user efforts on debugging and optimizing the generated programs.

Limitations of PM Checkers. Ayudante relies on PM checkers during model training to provide rewards and in the code refining pipeline to provide code optimization suggestions. Therefore, the capability of PM checkers is critical to the quality of code generated by Ayudante. So far, none of the existing PM checkers detects all PM bugs or programming issues. Ayudante addresses the issue by adopting a three-level checking process, which incorporates multiple PM checkers, to generate the rewards during RL model training (Section 4.4); different checkers complement each other during training. As demonstrated in Table 2, the more checkers used, the higher the PM checker passing rate and robustness. Furthermore, our code refining pipeline adopts multiple high-level validation tools, such as XFDetector [44], and AGAMOTTO [54], to further detect bugs that are not captured in training. Once more comprehensive PM checkers are developed by the community, we can retrain our model by replacing the current PM checkers to further improve PM code generation and refinement.

7 Related Work

To our knowledge, this is the first paper to design an ML-based automatic PM program assistant. This section discusses related works.

PM Programming Tools. Prior works focused on developing PM debugging and testing tools [44,45,54], programming libraries and APIs [10,17,22,31,74]. The tools are helpful for PM programming. However, these tools require users to manually write PM code from the scratch, which is challenging for non-expert programmers and tedious and time-consuming work for expert programmers as discussed in Section 2. Recent works also explored user-friendly libraries and high-level programming semantics for converting the data structures developed for volatile memory into PM programs [27,42,47]. Concurrent work TIPS [39] goes further and provides a systematic framework to convert DRAM-based indexes for the NVMM with plug-in APIs to plug-in a volatile index and facade APIs to access the plugged-in index. However, Ayudante focuses on automatically generate PM code by inserting library API functions in the source code. As such, Ayudante is orthogonal to these libraries and programming semantics;



(a) Array (b) B-Tree (c) Lists (d) Queue

Figure 11: Bandwidth comparison between Ayudante generated code and PMDK microbenchmarks on the Intel server.

the model can also be trained based on such libraries as a substitute or complementary of PMDK.

Conventional Code Translation. Code translation problem on popular and familiar programming language, *e.g.,* C/C++/Java/C#/Python/Go, has been well investigated [2, 43, 46, 57, 71]. Previous tasks focus on adding annotation and source-to-source code translation can be categorized as 3 main folds: (1) Adding specific annotations to achieve certain constraints. An example is parallelization annotation, annotating to a program with proper statements that are required by parallel computing libraries and make the source program can run on parallel hardware, *e.g.,* OpenCL [71], CUDA [57] and OpenMP [43]. (2) translate between different types of source code. Such as source-to-source compilers LLVM [46] to translate between C and C++ by first compiling source code to LLVM bitcode then decompiling the bitcode to the target language. (3) translate code between different versions of one programming language. For example, python's 2to3 tool [2] translates from python version 2 to version 3 by parsing the program to abstract syntax tree then translate it. Though transcompilers are preferred among the software developers for its definite transformation process, it suffers from tremendous developing labor efforts and is error-prone.

ML-based Program Generation. To model the edits to transform code into a target code, one needs to learn the conditional probability distribution of the target code version given the source code. A good probabilistic model will assign higher probabilities to plausible target versions and lower probabilities to less plausible ones. Neural Machine Translation models (NMT) are a promising approach to realize such code edit models and use distributed vector representations of words as the basic unit to compose representations for more complex language elements, such as sentences and paragraphs, *e.g.,* the sequence-to-sequence (Seq2Seq) models [35, 79] and word embedding [9, 12, 56]. However, code edits also contain structural changes, which requires the model is syntax-aware. To overcome the rigid syntax constraints in programming language, recent studies leverage tree-to-tree LSTM encoder-decoder on abstract syntax tree for program statements translation [11, 14]. However, these work are either rule-based that requires additional knowledge of the programming languages, such as grammar [11, 52, 66, 81, 84], or applying a model to implicitly learn the translation policies [14, 81] that require enough training dataset to achieve the high inference performance, which is challenging to apply to a new task without training dataset.

8 Conclusions

We propose Ayudante, a deep reinforcement learning based framework to assist persistent memory programming. Ayudante provides a deep RL-based PM code generator that mimics programmers behavior to add proper PM library APIs to the volatile memory-based code. Ayudante also provides a code refining pipeline that reports code improvement suggestions to the programmers, to help reduce bugs and improve run-time performance. Ayudante utilize a novel transfer learning to transfer PM programming semantics from C/C++ to other languages like Java. We evaluate Ayudante with microbenchmarks of various data structures which pass all code refining pipeline checkers and achieve comparable performance to expert-handwritten code. Ayudante significantly reduce the burden of PM programming, and shed light on machine auto programming for PM and other domains.

9 Acknowledgement

We thank our shepherd Dr. Changwoo Min and the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1829524, 1817077, 2011212 and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

[1] Ayudante source code. https://github.com/Hanxian97/Ayudante.

[2] Python 2to3 migration tool, 2020. https://github.com/erdc/python/blob/master/Doc/library/2to3.rst.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part*

I, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[5] A. Badam. How persistent memory will change software systems. *Computer*, 46(8):45–51, 2013.

[6] begeekmyfriend. Leetcode solution in C. https://github.com/begeekmyfriend/leetcode.

[7] begeekmyfriend. Leetcode solution in C++ and Python. https://github.com/begeekmyfriend/leetcode/blob/master/0023_merge_k_sorted_lists/merge_lists.c.

[8] Bruno Bouzy and B. Helmstetter. Monte-carlo go developments. *Advances in Computer Games*, 135, 10 2003.

[9] Nghi DQ Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36, 2018.

[10] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications*, OOPSLA '14, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural machine translation. *arXiv preprint arXiv:1810.00314*, 2018.

[12] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019.

[13] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[14] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.

[15] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091. ACM, 2020.

[16] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[17] Intel Corporation. Persistent memory programming. https://pmem.io.

[18] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, page 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *arXiv preprint arXiv:1807.01784*, 2018.

[20] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

[22] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872. USENIX Association, July 2020.

[23] Sina Ghiassian, Banafsheh Rafiee, Yat Long Lo, and Adam White. Improving performance in reinforcement learning by breaking generalization in neural networks. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 438–446. International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[24] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS 2021, page 415–428, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Brendan Gregg. Linux perf examples, 2019. http://www.brendangregg.com/perf.

[26] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 930–937, 2019.

[27] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Alan Harris. Distributed caching via memcached. In *Pro ASP. NET 4 CMS*, pages 165–196. Springer, 2010.

[29] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[31] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[32] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 468–482, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Intel. Intel® Optane™ DC Persistent Memory, 2019.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[35] Yonghae Kim and Hyesoon Kim. A case study: Exploiting neural machine translation to translate cuda to opencl. *arXiv preprint arXiv:1905.07653*, 2019.

[36] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[37] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[38] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[39] R. Madhava Krishnan, Wook-Hee Kim, Fu Xinwei, Sumit Kumar Monga, Hee Won Lee, Jang Minsung, Ajit Mathew, and Changwoo Min. Tips: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[40] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, February 2017. USENIX Association.

[41] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[42] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[44] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing

framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[46] LLVM. The llvm compiler infrastructure, 2020. https://llvm.org/.

[47] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Amirsaman Memaripour and Steven Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 413–422. IEEE, 2018.

[49] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[52] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[53] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*, 2015.

[54] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.

[55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[56] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758. IEEE, 2016.

[57] Cedric Nugteren and Henk Corporaal. Introducing 'bones' a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 1–10, 2012.

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[59] pmdk. pm-reorder consistency checker. https://pmem.io/2019/02/04/pmreorder-basics.html.

[60] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[61] pozy. Leetcode solution in C++ and Python. https://github.com/pezy/LeetCode.

[62] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, October 2005.

[63] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

[64] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[65] Akanksha Rai Sharma and Pranav Kaushik. Literature survey of statistical, deep and reinforcement learning in natural language processing. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 350–354. IEEE, 2017.

[66] Hui Shi, Yang Zhang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Deep symbolic superoptimization without human knowledge. In *International Conference on Learning Representations*, 2019.

[67] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[68] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[69] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[70] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[71] Krishnahari Thouti and SR Sathe. A methodology for translating c-programs to opencl. *International Journal of Computer Applications*, 82(3), 2013.

[72] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[73] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[74] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[75] William Yang Wang, Jiwei Li, and Xiaodong He. Deep reinforcement learning for nlp. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, pages 19–21, 2018.

[76] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.

[77] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of Intel's Optane DC persistent memory module and its impact on high-performance scientific applications. In *SC*, 2019.

[78] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[79] Shaofeng Xu and Yun Xiong. Automatic generation of pseudocode with attention seq2seq model. In *2018 25th Asia-Pacific Software Engineering Conference APSEC*, pages 711–712. IEEE, 2018.

[80] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[81] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[82] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3320–3328, 2014.

[83] yuanguangxin. Leetcode solution in Java. https://github.com/yuanguangxin/LeetCode.

[84] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*, pages 1737–1746, 2018.

[85] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2013.