2022 IEEE International Conference on
Systems, Man, and Cybernetics (SMC)

Profile-Guided Optimization for Function Reordering: A Reinforcement Learning Approach

Weibin Chen[1], Yeh-Ching Chung[2]

2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC) | 978-1-6654-5258-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SMC53654.2022.9945280

***Abstract*—Profile-guided optimization (PGO) remains one of the most popular optimization strategies in code generation optimization. Function reordering is an essential step for profile-guided optimization. The state-of-the-art function reordering method performs on a unidirectional function call graph where nodes and edges define functions and caller-callee pairs. Each edge is labeled by its call frequency. However, we demonstrate that a bidirectional function call graph can represent the memory call better. We use a reinforcement learning algorithm SARSA to choose the appropriate order of functions by maximizing the total numbers of the function call through a bidirectional function call graph. In this paper, we use a self-developed tool to reordering functions. We first illustrate how our RL-based algorithm generates a new function order. Then we evaluate three algorithms on various applications, including Redis, Protobuf, and SPEC CPU benchmark. Our experiment results indicate that the new algorithm outperforms the other two algorithms in various applications, improving the resulting performance of practical applications. Especially on Redis, the performance is improved by 4.2% on SARSA, which is better than C^3 (3.4%) and ph (2.8%).**

***Index Terms*—Profile-guided Optimization, Function Reordering, Function Call Graph, Reinforcement Learning**

I. Introduction

Function reordering is an effective way of improving the performance of modern applications with lots of functions. There are many tools that are designed for improving code localities like AutoFDO [1], HFSort [2], and BOLT [3]. These tools improve the performance of different software by about 3% to 15%. In this paper, we build a tool to optimize applications in function level. Our research mainly focuses on improving function reordering with a reinforcement learning (RL) algorithm to improve the performance of different applications and information systems. Reducing I-cache miss and I-TLB miss are two main optimization targets for function reordering. A proper function order can speed up virtual-to-physical address translation. The latest function reordering algorithm focuses on putting caller before callee, greatly reducing I-TLB miss. Our algorithm also considers a sub-optimal strategy, placing the caller after the callee. This approach also decreases the call distance between caller and callee. Considering a large-scale application consisting of many functions, we cannot place caller before callee for all functions. Our new function reordering algorithm considers both strategies to generate a better function order. With the improvement of computer performance in recent years, using machine learning to optimize large-scale applications has been a research hotspot. Developers usually use machine learning algorithms to predict application characteristics in compilation stage, such as datacenter scheduling [4], resource allocation [5], and code-generation [6] etc. There are three steps to do optimization. First, we collect application runtime data. Then, we build a model for existing data based on the data characteristics. Finally, we use the existing data to train the model. Machine learning is divided into three categories, including supervised learning, unsupervised learning, and RL. RL is different from supervised learning or unsupervised learning. It is inspired by behaviorist psychology and uses the notion of rewards or penalties so that a software agent interacts with an environment and maximizes his cumulative reward. The reason why we choose RL [7] in this paper is that it does not need to collect a large amount of training data when finding the best order in the function call graph. After collecting runtime data from software, it will update the value function based on a policy to find the best strategy.

We use reinforcement learning with the best of our knowledge to design a function reordering algorithm that improves the performance of Redis, Protobuf, and SPEC2006. In this work, we have two main steps. Firstly, we use reinforcement learning to optimize function order. This method is achieved by (i) producing a function call graph based on the runtime profile data from applications, (ii) using state–action–reward–state–action (SARSA) [8], a reinforcement learning model that, given a function call graph, builds an improved ordering of functions optimizing the performance of the software. Then, we evaluate the performance impact of our algorithm and some most popular algorithms on a set of applications.

The contributions of the paper are as follows.

- We evaluate the impact of a widely used function reordering approach called Pettis and Hansen's (PH) and a state-of-the-art function reordering approach Call-Chain Clusters algorithm (C^3) on Redis, Protobuf, and SPEC2006.
- We analyze a potential opportunity to improve function reordering with a top-down algorithm. Then we propose a new reinforcement learning algorithm to improve function reordering.
- We extensively use several metrics to evaluate the new algorithm on various applications, including throughput, time, operation speed, I-cache miss, and I-TLB miss. The experiment results show that the new algorithm outperforms PH algorithm and C^3 algorithm in most of the situations.

The paper is organized as follows. We start by introducing the background of function reordering problem in Section II. After that, we present an overview of our system architecture to improve code layout (Section III) and three different kinds of dynamic call graphs for function reordering

[1]School of Science and Engineering, The Chinese University of Hong Kong (Shenzhen), Shenzhen, China
[2]School of Data Science, The Chinese University of Hong Kong (Shenzhen), Shenzhen, China

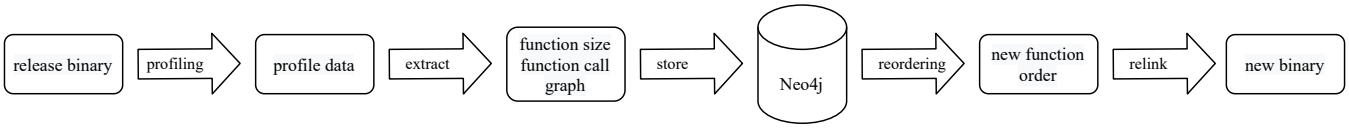


Fig. 1. Diagram showing a processes of profile-guided function reordering for a binary.

(Section IV). Next, in Section V, we present two previous heuristic algorithms. Section VI describes our methodology for improving function reordering. In Section VII, we first discuss the parameters of our RL-based algorithm and then show the experiment results of latest techniques and our techniques for different applications. Finally, a discussion of related work is presented in Section VIII, and Section IX concludes the paper.

II. Problem Background and Definition

The goal of the most widely used approaches is to find a new function order so that a function will most likely be placed adjacent to the following called function. In other words, the caller locates right next to the callee. This idea aims to reduce the probability of page fault interrupts and TLB miss. The existing algorithms are bottom-up algorithms like merge sort. These algorithms maintain a set of nodes of functions. Each function forms a node, corresponding to the function call graph paths. Then two different nodes are merged based on the bottom-up algorithm optimization strategy. The algorithm ends until all nodes are merged entirely.

Unlike traditional approaches, we try a top-down algorithm to optimize the function reordering problem. Top-down algorithm keeps looking for the next node according to strategy until all nodes are found. We can formulate this problem as follow.

Definition 1 (The Function Call Maximization Problem). *Assume G = (V,A) denote a function call graph, where the vertex set V = 0,1,2...,n consists of n functions. The edge set A is the calls between functions with the call frequency as weight, where Fig. 2(b) shows an example. Given this function call graph G for the target application, we generate a decomposition of G into a chains C which maximizes the sum of function call frequencies in C.*

This types of problems are called **TRAVELING SALESMAN PROBLEM (TSP)** [9], which is a famous NP-hard combinatorial optimization problem. TSP-based algorithms [10] are very common in the code optimization community because these kinds of methods are simple and effective. This paper uses reinforcement learning to solve this formulated TSP problem.

III. System Overview

In this section, we give an overview of the whole system that helps to improve function reordering. Fig. 1 presents a diagram describing each part of the system. This system is similar to hfsort [11]. The main difference is that we use Neo4j [12] to store function call graphs. Since our target applications contain hundreds of functions, the function call graphs are enormous. Neo4j is a high-performance graph database, which can improve the access performance of the system. Besides, we modify the function call graph based on the function sizes as an input of the optimization algorithm, which we present in Section IV.

The first part of the system is to collect the profile data and store some helpful information in the database. To accomplish this work, we built a sample-based tool called Girasol. Girasol invokes the perf tool to collect profile data at specified intervals when we execute the original application. After the application completes, all profile data is stored in a perf file. A function call graph is extracted from profile data, as described in Section IV. Notice that our tool only takes the hot functions to the function call graph. Precisely, we define hot functions as callers or callees in the profile data, and we only consider these functions during function reordering. Finally, Girasol saves the function call graph to the Neo4j database.

In the second step, our tool takes function call graph from the Neo4j database. This function call graph is the input to the reordering algorithms. Since each reordering algorithm uses a different function call graph as input, our tool changes a function call graph for each optimization function after retrieving it from the database. The function call graphs and the reordering algorithms are described in detail in Section IV and Section V. Then our tool use a reorder algorithm to generate a new function order with a function call graph. To reorder functions in a specific layout, we use lld *–symbol-ordering-file* from LLVM 12.0 [13]. We generate a linker script containing all object files of the compiled application. Then our tool can link all symbols in the object files together by using lld after a new function order is generated by the reordering algorithm. Through this process, we end up with an optimized binary.

IV. Function call graph

Pettis and Hansen [14] first propose an undirected weighted call graph for a program. Fig. 2(a) shows an example of a graph. This call graph $G = (V, A)$ contains a set of nodes V. Each node corresponds to a function f in the binary. A set of arcs A represents the fact that function f calls function g ($f \rightarrow g$) or function g calls function f ($g \rightarrow f$). So the weights of the arc are the sum of the calls between f and g.

To better represent the function call, Ottoni and Maher [11] build a direct weighted call graph , which is presented in Fig. 2(b). Weight w ($f \xrightarrow{w} g$) represents the frequency of function f calls g at runtime. For example, the weight $W_{f \rightarrow g} = 60$ means that there were 60 call entries for f calling g in the *last branch records* (LBR) [15]. They choose a direct weighted call graph because the algorithm can place the caller before callee so that the call distance is shorter.

We propose a bidirectional graph for our TSP-based algorithm as input. Compared with bidirectional graphs, a reverse arc gives another choice to generate function order. We build a new function call graph based on the function size and

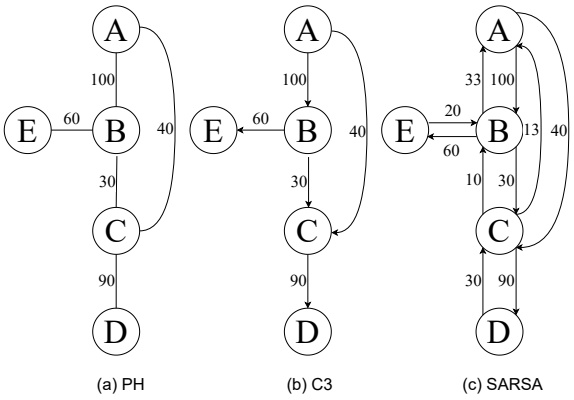


Fig. 2. Example of dynamic function call graphs for three different algorithms(Function sizes are the same).

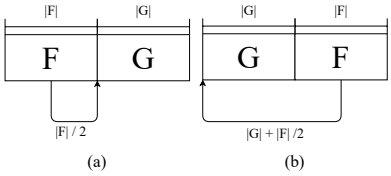


Fig. 3. Examples of call distance for functions F and G.

indirect function calls. Fig. 3 [2] shows a simple example with two functions, F and G, where function F calls G. The compiler usually puts the function entry at the lower address. Instructions are fetched from a higher address when the function is processed. In function F, $|F|/2$ is the average distance in the address space of each instruction from the entry of F. So Ottoni and Maher assume that the distance to be jumped in the address space when F call G is $|F|/2$ in the code layout of Fig. 3(a). For the same reason, Fig. 3(b) shows a different code layout. The call distance is $|G|+|F|/2$ when F call G. This phenomenon reveals that compared with the layout in Fig. 3(a), Fig. 3(b) needs a longer call distance when F calls G. This phenomenon means positioning G in front of F produces a penalty. This penalty is related to the function size of F and G, so we can use an equation to compute the weight of the reverse arc. Based on this characteristic, we proposed a new bidirectional function call graph. We calculate the reverse arcs as follow:

$$W_{G\to F} = W_{F\to G} \times \frac{\frac{|F|}{2}}{\frac{|F|}{2} + |G|} \tag{1}$$

where $W_{G\to F}$ means the weight of arc $G \to F$. With the increase of the size of G, the weight of arc $G \to F$ decreases. That is because, as the size of G becomes larger, it increases the distance to be jumped in the address space, and thus the penalty increases. Fig. 2(c) shows an example of a new bidirectional graph where each function in the graph has the same size. We do not update the weight of both arcs n the case of $f \to g$ and $g \to f$ in the sampled data.

V. Traditional function reordering Algorithm

This section presents two traditional function reordering algorithms. Section V-A introduces a bottom-up algorithm called Pettis and Hansen (PH) algorithm. While Section V-B describes Call-Chain cluster (C^3) algorithm, which is an extended version of PH.

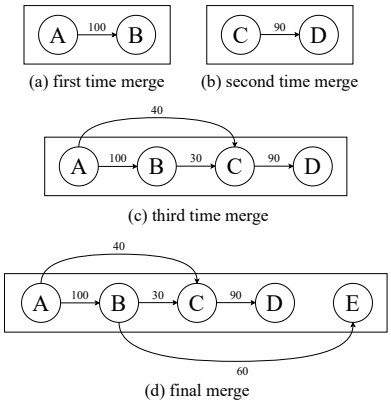


Fig. 4. Example of C^3 algorithm processing the call graph from Fig. 3(b).

A. Pettis-Hansen (PH) Algorithm

In 1990, Pettis and Hansen [14] proposed three approaches to improve the code layout, including reordering functions and basic blocks. These techniques are still widely used by many tools today. In this part, We briefly describe how Pettis and Hansen reordering functions to achieve the goal of improving code layout. We call this approach as PH algorithm.

As described in Section IV, the input of the PH algorithm is an undirected function call graph, which is shown in Fig. 2(a). PH sorts all arcs from highest to lowest. Each time, PH merges the two nodes with the highest arcs. Merge operation means two nodes are merged, and their weights are added up. Also, PH algorithm has a mechanism to check to increase weight. There is a list of the original nodes in order during the algorithm. We assume that node a and node b contain the first and last nodes separately. When merging node a and node b, PH evaluates the reversing of either a or b for increasing the weight of the new adjacent nodes. The algorithm stops until one node is left in the graph.

B. Call-Chain Cluster (C^3) Algorithm

Ottoni and Maher [11] proposed a new bottom-up algorithm called Call-Chain Cluster (C^3). One of the main differences between the C^3 and PH is that C^3 considers the caller/callee relationship. C^3 choose a directed call graph as input, which we discuss in Section IV.

The C^3 algorithm merges clusters as follows. C^3 use cluster instead of node. A cluster is a container used to store functions with a page size limit. When both clusters are larger than the page size, they stop merging. Before each function is placed in a cluster, C^3 sort each function in the call graph, in descending order of hotness. In this paper, we select the sum of the weights of clusters' incoming arcs to judge the hotness of clusters. Then, C^3 starts to append the function to the end of a cluster of its most common caller. C^3 stops merging clusters when the size of any two clusters that can be merged is greater than the page size. Finally, C^3 sorts the final cluster based on its "density." "density" is computed by equation 2. This equation calculates the sum of

the running times of all functions divided by the total size of all functions.

$$density(c) = \frac{time(c)}{size(c)} \tag{2}$$

We illustrate the operation of the C^3 algorithm in the example from Fig. 4. In the beginning, we assume that the amount of time spent in each function equals the sum of weights of its incoming arcs in the call graph. In the first step, C^3 B merged with the cluster containing function A in Fig. 4(a) because B has the greatest incoming arc. Next, function D is chosen. It is appended to function C in Fig. 4(b). Then function C is processed. Its cluster $(C; D)$ is appended to cluster $(A; B)$ in Fig. 4(c). Finally, function E has appended the $(A; B; C; D)$, resulting in a final cluster $(A; B; C; D; E)$ in Fig. 4(d).

VI. Reinforcement learning algorithm

Pettis and Hansen formulate [14] the basic block reordering problem as a TSP-based problem. They search a basic block order, starting from a block to search the next block based on the arc's weight, finally generating a path. The order of this path is the order of blocks. The function call graph is quite similar to the basic block graph. So we can use a similar approach to find an effective solution. However, unlike the basic block graph, the nodes in the function call graph have more arcs than the basic block graph because one caller can have more than two callees. In addition, the structure of a basic block graph is more straightforward than a function call graph. Therefore, if we want to use the same method to identify an excellent solution, we must consider finding the best adjacent function for the subsequent placement.

Reinforcement learning aims to maximize long-term rewards in a dynamic environment. Fig. 5(a) describes the operation steps of reinforcement learning. First, The exploratory agent of RL explores via the trial and error method in a complex environment. Then we generate a policy to select the best action for the current state. The value function measures the quality of action in a state by accumulating rewards for taking this action. RL accumulates rewards or penalties to value function by the last train and error actions. Fig. 5(b) illustrates a detailed version of our RL algorithm. We set functions as states. Action is a function to find the following function connected to it. Each time, our algorithm chooses the next function(t+1) based on the ϵ-greedy policy. Based on the function call graph, the policy will be updated by the reward(t+2). Then the functions(t+2) will become the current state function(t+2).

This section describes a new efficient heuristic. Algorithm 1 shows the operation process of SARSA. We illustrate this algorithm in three parts. In Section VI-A, we present the basic idea of Q-table and how this algorithm updates Q-table. Then, detailed information about how our algorithm chooses the following function from Q-table is shown in Section VI-B. Finally, Section VI-C describes a comparison between C^3 and SARSA.

A. Q-table Update Rule

We use a widely used RL algorithm SARSA to find the new function order. Unlike PH and C^3, we search from one function to find the next adjacency function. Usually, we use

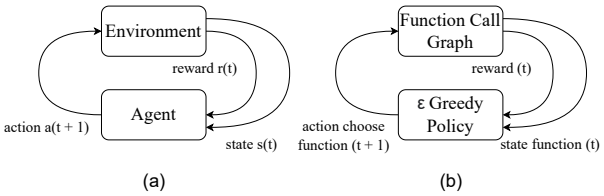


Fig. 5. (a) Operation steps of basic reinforcement learning; (b) RL-based method for searching new function order

main function as the first function. Besides, we construct a bidirectional graph as an input, which we present in Section IV. An example is shown in Fig. 2(c).

SARSA uses Q-table instead of a value function to store the accumulated awards. Q-table is a matrix table storing the accumulated rewards of each exploration. For each element in Q-table, we call it Q-value, which represents a value for every single state and action pair. The equation below shows how SARSA updates Q-table:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{3}$$

Here, r means that the expected rewards collected when state s takes action a, followed by a transition to next state s'. α is a learning rate parameter that helps to converge the Q-table in a non-deterministic environment. When the agent chooses a larger learning rate, the new rewards from the last action will influence the agent more than its previous accumulated rewards. On the contrary, the algorithm will learn very little from new experiences. Therefore, a suitable α can help agents learn faster in the new environment. γ is a discount rate parameter for future rewards, greater than or equal to 0 ($\gamma \geq 0$). This parameter affects the rewards from the next state. A higher discount rate indicates that the future gain prospected by the agent is more important than the current gain. It is good for fast convergence. However, based on our experience, a higher discount rate might influence the long-term benefit from the prospect gain. Thus, we should choose a reasonable discount rate to balance the short-term rewards and long-term benefits. SARSA can always find the optimal scheduling policy in the limit with 100% probability where each table has a limited number of visits [16].

B. ϵ-greedy Policy

Usually, the highest number of Q-value available from the Q-table means the best index for the next function. However, at the beginning of this algorithm, Q-table is initialized as 0. So maximum Q-value can not be considered the next selected function in the Q-table. In this situation, we need a random selection to start exploration. In addition, random selection can prevent the search process from getting stuck at the local maximum.

Our algorithm uses ϵ-greedy policy to choose the next state. Instead of always selecting the highest Q-value, ϵ allows SARSA to choose the next function randomly. This strategy enables SARSA to explore the new environment. Specifically, the largest number in Q-table is selected with probability ϵ, and the next function is selected randomly with probability 1 - ϵ. In this paper, we set a self-adaptive ϵ as i/N, where i is current exploring times and N is total training times. With this self-adaptive ϵ, SARSA does not trust the

Q-table at the beginning because i/N is a tiny number. As the training increases, SARSA will gradually start trusting the Q-table. SARSA chooses the following function based on Q-table without random action when ϵ = 1.

C. Quantitative Comparison

Our goal is to use SARSA to detect a better function ordering that considers a set of characteristics in runtime. The complete proposed algorithm is given in Algorithm 1. The environment is the dynamic call graph. Each time, SARSA starts from a function. Then it finds the following function, according to the value in Q-table. Q-table is used to select the best action for the current state. SARSA chooses the next function using ϵ-greedy policy for each action. Initially, all Q-values in Q-table are zero. Then SARSA randomly selects different paths to update Q-table. As the number of iterations increases, SARSA starts to trust Q-table and look for the best action from Q-table. To better update Q-table, we usually repeat it thousands of iterations.

To compare with C^3, we generate a function order by SARSA on N = 60. We use a graph of the structure of Fig. 2 (c) as input to SARSA. Then, we obtain a function order (A;B;E;C;D) with the sum of call frequency 250. We now compare the final layouts obtained by SARSA (A;B;E;C;D) and C^3 (A;B;C;D;E). We make use of the approach from Ottoni and Maher [11] to evaluate the amount of distance jump in these two layouts. We assume that all five functions have the same size $|g|$. Callers start from a $\frac{|g|}{2}$ of themselves. According to the arc weights from Fig. 2 (c) and two layouts extracted by two algorithms, we calculate the total distance jumped through the calls in each case:

$$\begin{aligned}&\text{cost(SARSA)} = 100 * 0.5 * |g| + 40 * 2.5 * |g| \\ &+30 * 1.5 * |g| + 90 * 0.5 * |g| + 60 * 0.5 * |g| \\ &\text{cost(SARSA)} = (50 + 100 + 45 + 45 + 30) * |g| \\ &\text{cost(SARSA)} = 270 * |g|\end{aligned}$$

$$\begin{aligned}&\text{cost}\left(C^3\right) = 100 * 0.5 * |g| + 40 * 1.5 * |g| + 30 * 0.5 * |g| \\ &+90 * 0.5 * |g| + 60 * 25 * |g| \\ &\text{cost}\left(C^3\right) = (50 + 60 + 15 + 45 + 150) * |g| \\ &\text{cost}\left(C^3\right) = 320 * |g|\end{aligned}$$

The results show that SARSA has a 19% reduction in the total call distance in this situation, compared with C^3. In the next section, we further investigate the impact of these algorithms on Redis, Protobuf, and SPEC2006 with a series of experiments.

VII. Evaluation

This section first shows the analysis of SARSA and then evaluates three algorithms on different scenarios of different applications. The evaluation was conducted on a Linux-based server powered by 2.4 GHz Intel Xeon E5-2640 v4 microprocessors with ten cores. The total number of Random Access Memory is 192 GB. For all experiments in this section, we take five times the test and eliminate the largest and smallest results. Then we calculate the mean of the rest three results.

Algorithm 1 An reinforcement algorithm for function reordering

Input: bidirectional dynamic call graph G
Output: ordering of functions
Algorithm parameter: learning rate $\alpha > 0$ and discount factor $\gamma > 0$
State $\mathcal{S}$ and **action** $\mathcal{A}(s)$ are set of functions in the graph.
Reward $r \in W(s \rightarrow a)$
Initialize Q(s,a) table, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ with 0
s' is the next action of s, a' is the next action of a
Repeat (N times, for each episode):
  Initialize $\mathcal{S}, \mathcal{A}(s)$
  Repeat (for each step of episode):
    Take function a, observe r, s'
    Choose a' from s' using ϵ-greedy policy
    if (rand() $< \epsilon$) **then** //ϵ = *current time i / N*
      select random function
    else
      select function with maximum Q-value
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'; a \leftarrow a'$
  until no possible function left
until finish N loop

A. Analysis of SARSA

In this part, we show an evaluation of SARSA. We take some experiments to answer two questions: (a) How do we choose the parameters of SARSA? (b) How long does SARSA generate a new order compared with the existing heuristic?

Considering the first question, we mainly focus on two parameters that describe the behavior of SARSA, which are discount factor γ and exploration parameter ϵ. This part only discusses the ϵ parameters because we use a self-adaptive γ strategy, which we explain in Section VI-B. We calculate the total function calls on Redis for different values of the discount factor γ where the learning rate α = 0.1. Remember that γ measures how essential future rewards are compared to the current reward. However, we found less than 2% error between these sum of function calls, which has little effect on compiled program performance. We set γ to 0.1 for the rest of the experiments.

TABLE I shows the runtime of SARSA, PH, and C^3 on Redis and GCC, which contain 230 and 700 hot functions, respectively. Because PH and C^3 are bottom-up algorithms, they only cost less than 1 second to find a new function order. SARSA needs dozens of seconds to generate a new function order on these two applications. However, large-scale applications need to run for a long time with heavy workloads in the data center. 1% of performance improvement can reduce the processing time of much significant software. It is still worth developers taking a longer time to search for a better function order with SARSA.

B. Redis and Protobuf

For Redis, the performance evaluation indexes are throughput, I-TLB miss, and I-cache miss. Throughput is the key performance measure. It means the number of operations per second a server can deliver. Higher throughput implies a

TABLE I
OPERATION TIME FOR THREE DIFFERENT APPROACH TO GENERATE A NEW FUNCTION ORDER ON REDIS AND GCC

Algorithm	PH	C3	SARSA
Time (Redis)	0.33	0.33	36.72
Time (GCC)	0.42	0.42	75.49

TABLE II
YCSB BENCHMARK AND PERFORMANCE CHARACTERISTICS OF REDIS. (MPKI AND OPS/SEC MEANS MISSES PER 1000 INSTRUCTIONS AND OPERATION PER SECONDS RESPECTIVELY)

Insertion Rate	Read Rate	Record	Throughput (ops/sec)	I-TLB MPKI	I-cache MPKI
20%	80%	200000	33009	0.74	0.39
40%	60%	200000	25909	0.68	0.51
60%	40%	200000	21707	0.63	0.57
80%	20%	200000	19101	0.64	0.60
100%	0%	200000	16984	0.64	0.57

better performance of Redis. For Protobuf, we measure the performance with operation speed, I-TLB miss, and I-cache miss.

1) Characteristics of Applications: The first system is the Redis (4.0), a highly optimized application storing data for various applications. Because of its high performance read and write, it is often used to store frequently accessed data. Redis contains about 200MB of program text after compiling by standard Redis makefile with GCC [17] -O2 optimization level. Google developed Protobuf to serialize structured data. Users can use the Protobuf compiler to automatically generate different kinds of language source code with only one definition for data format. It has four steps for serialization including new, $newarena$, $reuse$ and $serialize$ [18]. There are two reasons why we chose Redis and Protobuf as our studied applications. One reason is that they account for a large portion of the computing cycles spent caching data for varieties of modern applications. The other is that they have more than 200 hot functions in profile data which is a suitable application to show the effect of different function replacement strategies.

Frequent cache line conflicts cause a high cache miss rate. This phenomenon mainly happens in Redis, which experiences many computing cycles to find the key and value. TABLE II shows the benchmark information, throughput, and cache performance of Redis. Yahoo Cloud Serving Benchmark (YCSB) is a performance measurement tool for performance tests. It [19] is Yahoo's open-source framework for testing the performance of modern databases. TABLE III demonstrates a CPP benchmark provided by Google, including the speed of four steps, I-TLB and I-cache.

In this experiment, YCSB has 200000 records that have been inserted or written. To make the data stable, we modified YCSB to use a fixed random seed. The sum of insertion rate and read rate is 100% for each test. The insertion rates range from 20% to 100%, while the read rate ranges from 0% to 80%. It is interesting to note that MKPI and throughput change with the insertion rate and read rate. The program has various behaviors for different benchmark parameters, resulting in different performances. That is why we use different parameters for Redis experiments. In these experiments, we use the different function orders computed by each algorithm for each evaluation of the insertion rate.

TABLE III
GOOGLE MESSAGE BENCHMARK AND PERFORMANCE CHARACTERISTICS OF PROTOBUF. (MPKI AND MB/SEC MEANS MISSES PER 1000 INSTRUCTIONS AND OPERATION SPEED PER SECONDS RESPECTIVELY)

Method	new (MB/s)	reuse (MB/s)	newarena (MB/s)	serialize (MB/s)	I-TLB MPKI	I-cache MPKI
baseline	216.34	630.49	478.18	908.22	0.0336	7.5700
PH	216.90	636.29	493.83	915.31	0.0316	7.5588
C3	217.673	640.21	495.18	916.92	0.0268	7.5592
SARSA	**218.09**	639.56	491.22	**919.98**	0.0273	7.5689

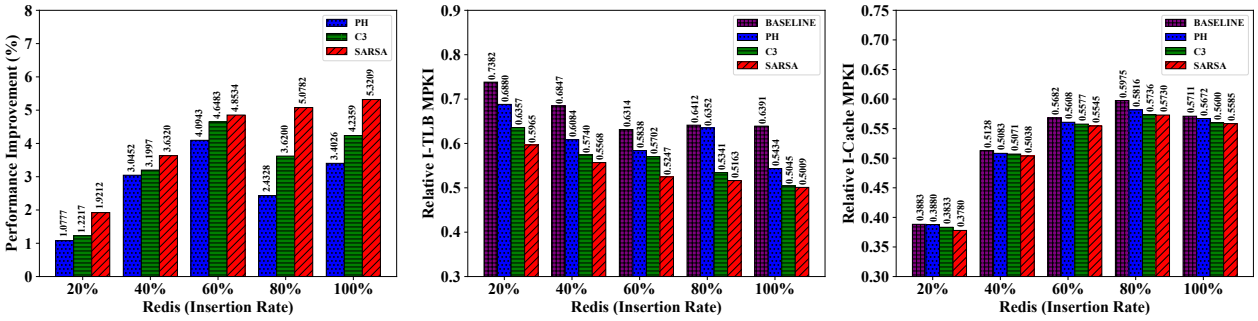
C. Experiment Results

We evaluate performance results using three different approaches to order the functions. The function order of the baseline is the default order. Then we use PH, C^3 and SARSA to obtain the new order. We use the Linux perf tool to obtain all I-TLB and I-cache performance information from the hardware performance counters.

1) Performance Results: For the parameters of SARSA, we select a small learning rate and discount factor to converge the Q-table better. Because small parameters can better converge, we set the parameters of SARSA as $\alpha = 0.1$, $\gamma = 0.1$. To ensure that Q-table has converged, we set iteration time N = 1000 and then calculate the sum of call frequency on N and 1.5*N. If the error of two sums of call frequency is less than 5%, we believe that the Q-table will converge after the N iteration. If the error is greater than 5%, we will increase N by 50% and calculate the sum of call frequency for new N and new 1.5*N again. We continue to cycle this process until we find an N value to make the Q-table converge. Since SARSA cannot guarantee to find the highest sum of call frequency at the last time, we record the function sequence of the last 50 times. Then we use the sequence with the highest sum of function calls as a result. TABLE III presents a performance comparison of three function reordering algorithms on the Protobuf. We observe that SARSA performs better in Parse_new step and serialize step with an average of 1.1% improvement, while C^3 outperforms other algorithms in Parse_reuse and Parse_newarena. By running the YCSB test, we calculate the throughput (operation per second) of three different approaches. Fig. 6(a) indicates that three algorithms improve the performance of Redis. The results show that SARSA performs better than C^3 and PH with an average throughput improvement of 4.2%, compared to 2.8% for PH and 3.4% for C^3. The most significant performance improvement was the 100% insertion rate and 0% reading rate, where SARSA improved performance by 5.3%. It is interesting to note that SARSA and C^3 have similar performance improvements for 60% insertion rates. Overall, SARSA outperforms C^3 and PH. C^3 have better performance improvement than PH in all scenarios.

2) I-TLB Performance Comparison: We use the Linux perf tool to record cache performance during steady-state execution many times to understand the performance improvements better. This section compares the effect of the different scenarios on the I-TLB miss, and the next part compares I-cache misses.

Fig. 6(b) shows the performance of I-TLB miss rates for the various scenarios on Redis. PH reduces I-TLB miss by



(a) Throughput performance comparison (b) I-TLB performance comparison (c) I-cache performance comparison

Fig. 6. Performance comparison on YCSB benchmark for Redis with five different insertion rate

9% on average, which is a minimal improvement. Compared with PH, C^3 and SARSA have better improvement, which reduces the metric by 15.5% and 19.8% on average over the baseline. As shown in Fig. 6(b), SARSA has the highest I-TLB miss reduction by 100% insertion rate and 0% reading rate. This phenomenon corresponds to the previous section that SARSA has the best performance improvement by 100% insertion rate, which means that I-TLB miss reduction is the main effect of function reordering. TABLE III presents the I-TLB miss in four operation steps of Protobuf. SARSA's I-TLB miss is slightly large than C^3 with an approximate 18.9% improvement over the baseline.

3) I-cache performance Comparison: Fig. 6(c) shows the comparisons of I-cache performance for the different scenarios on Redis. SARSA constantly improves I-cache miss, providing an average reduction of 2.6% in this metric. However, PH and C^3 sometimes have fewer improvements for the I-cache miss. Their results on a 1.1% average reduction and a 2.0% average reduction in this metric. Fig. 6(c) also indicates that Redis has the lowest I-cache miss reduction by 20% insertion rate and 80% reading rate, which means more insert operation is more likely to cause a higher I-cache miss rate. TABLE III shows that the I-cache miss rate of baseline is relatively the same as the three optimized I-cache miss rates in Protobuf.

D. SPEC CPU 2006

This section evaluates function reordering on the SPEC CPU 2006 benchmark. We utilized five relatively large C/C++ programs compiled using Clang with -O2 optimization level and ran experiments on the same hardware as the previous part. GCC, hmmer, and gobmk are from the integer benchmark. milc and lbm are from the floating-point benchmark. In this experiment, we analyze the performance of the binaries optimized by our tool with different ordering algorithms, using the original as the baseline. We use a separate SPEC train model to collect profile data. In this experiment, we mainly evaluate the execution time of each application.

Although we chose five relatively large SPEC programs, these programs are still much smaller than large-scale software used in modern data centers. Our optimization approaches are unlikely to reduce too many I-cache miss and I-TLB miss. Table IV shows the results of our experiments with the binaries that have several functions between 10 to 1000 based on the collected profile data. We do not find an outstanding advantage in using our optimization strategies for all applications. SARSA algorithm has statistically significant improvement compared with the other two algorithms for three binaries: gcc (2.90%),hmmer (1.42%) and milc (1.16%). For the most extensive program, gobmk, the C^3 achieves 2.37% speedup outperforming the ph algorithm by 1.8%. It is interesting to see that C^3 and SARSA have the same effect on lbm. Since lbm only has dozens of functions, C^3 and SARSA likely produce the same function order.

TABLE IV
PERFORMANCE COMPARISON ON SPEC CPU 2006 (MEASURED IN TIME)

Applications	C3	PH	SARSA
403.gcc	2.43%	1.86%	**2.90%**
445.gobmk	2.37%	0.56%	1.53%
456.hmmer	1.31%	0.59%	**1.42%**
433.milc	0.88%	0.88%	**1.16%**
470.lbm	1.53%	0.80%	1.53%
mean	1.70%	0.91%	1.70%

VIII. RELATED WORK

In this section, we first present some of the previous related work on profile-guided optimizations. Then we discuss previous works about using RL in program optimization.

Pettis and Hansen [14] first proposed code reordering at the function level. This algorithm is the PH algorithm described in Section V-A. This algorithm is implemented in many compilers and binary optimization tools. In 2017, Ottoni and Maher [11] presented an improvement version, which is the C^3 algorithm described in Section V-B. They choose to use a directed call graph instead of an unidirected call graph. The common denominator of these two algorithms is greedily merging functions and is designed to reduce I-TLB and I-cache miss primarily. SARSA is another algorithm proposed by this paper to improve the code layout introduced in Section VI. Note that, unlike these two algorithms, SARSA starts to search function order from top to bottom.

Code reordering at the basic block level is also initiated by Pettis and Hansen [14]. They merge a chain of basic blocks that are frequently executed together. This technique is also implemented in many tools. An improvement has been proposed by Newell and Pupyrevs [10] lately. Similar

to our algorithm, their techniques are trying to solve a kind of **TRAVEL-SALESMAN** problem. The difference is that their input is a control flow graph rather than a function call graph.

Lagoudakis and Littman [20] started using reinforcement learning for program optimization 20 years ago. They use reinforcement learning to search the cutoff point to switch between quick sort and insertion sort. Nowadays, more researchers use reinforcement learning to improve the program's performance. Noureddine [21] used deep reinforcement learning to achieve the optimal task allocation through a series of actions in a dynamic environment. Ipedk [22] employed reinforcement learning to schedule RAM traffics. Porter [23] applied reinforcement learning to select software component configurations at runtime. The reason why we choose reinforcement learning is that it is very suitable for modeling problems, such as dynamic task scheduling [24], where a series of actions achieve the best solution.

IX. Conclusion

This paper introduced a new method for function reordering in a bidirectional function call graph. This algorithm is used on a tool we built to reorder functions based on the profile data. We evaluate the impact of PH, C^3 and SARSA over seven widely-used programs. Although SARSA needs longer time to find the new function order, it outperforms the other two traditional function reordering heuristics on Redis, Protobuf, and part of SPEC2006. For Redis, we selected a database performance testing framework called YCSB. Our experimental evaluation indicated that SARSA had the best results on Redis, increasing the performance by 4.2% on average. For Protobuf, SARSA outperforms the other two algorithms in Parse_new and serialize, 0.8% and 1.2%, respectively. For SPEC2006, SARSA has better performance in gcc, hmmer and milc.

Although the presented algorithms were studied in 7 programs, we are convinced that the benefits of our techniques are applicable in other applications. The optimization effect of SARSA is better than PH and C^3 that in most scenarios. These techniques can also improve the performance of other tools and applications, including some large-scale applications running on the data center, which can provide better software services for developers or users. In the future, we will try to combine other optimization strategies with our work to improve the overall performance of large-scale applications.

Acknowledgment

We would like to thank Yifan Zhu and Wenlong Sun for implementing the optimization tool described in Section III. We also appreciate the anonymous reviewers for their valuable feedback. This work was supported in part by the Alibaba AIR project for Large-scale Graph Pattern Query System and its Optimization Strategy.

References

[1] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 12–23.

[2] G.Ottoni, "hfsort: a tool to sort hot functions," Website:https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort.

[3] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[4] W. Tang, Y. Wang, H. Liu, T. Zhang, C. Li, and X. Liang, "Exploring hardware profile-guided green datacenter scheduling," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 11–20.

[5] D.-J. Oh, Y. Moon, E. Lee, T. J. Ham, Y. Park, J. W. Lee, and J. H. Ahn, "Maphea: a lightweight memory hierarchy-aware profile-guided heap allocation framework," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2021, pp. 24–36.

[6] R. Gupta, D. Benson, and J. Z. Fang, "Path profile guided partial dead code elimination using predication," in *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 1997, pp. 102–113.

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[8] T. L. Thorpe, "Vehicle traffic light control using sarsa," in *Online]. Available: citeseer. ist. psu. edu/thorpe97vehicle. html*. Citeseer, 1997.

[9] H. A. Abdulkarim and I. F. Alshammari, "Comparison of algorithms for solving traveling salesman problem," *International Journal of Engineering and Advanced Technology*, vol. 4, no. 6, pp. 76–79, 2015.

[10] A. Newell and S. Pupyrev, "Improved basic block reordering," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1784–1794, 2020.

[11] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 233–244.

[12] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 217–218.

[13] M. Panchenko, "Building a binary optimizer with llvm," in *European LLVM Developer's Meeting*, 2016.

[14] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 16–27.

[15] Intel Corporation, *Intel® 64 and ia-32 architectures software developer's manual*, 2011.

[16] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic programming*. Athena Scientific, 1996.

[17] G. Team, "Gnu compiler collection," Website:http://gcc.gnu.org.

[18] Google, "Protobuf," Website:https://github.com/protocolbuffers/protobuf/blob/master/docs/performance.md., 2012.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[20] M. G. Lagoudakis, M. L. Littman *et al.*, "Algorithm selection using reinforcement learning." in *ICML*. Citeseer, 2000, pp. 511–518.

[21] D. B. Noureddine, A. Gharbi, and S. B. Ahmed, "Multi-agent deep reinforcement learning for task allocation in dynamic environment." in *ICSOFT*, 2017, pp. 17–26.

[22] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 39–50, 2008.

[23] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "{REX}: A development platform and online learning approach for runtime emergent software systems," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 333–348.

[24] C. Shyalika, T. Silva, and A. Karunananda, "Reinforcement learning in dynamic task scheduling: A review," *SN Computer Science*, vol. 1, no. 6, pp. 1–17, 2020.