IEICE TRANS. INF. & SYST., VOL.E103–D, NO.9 SEPTEMBER 2020

LETTER

A Highly Reliable Compilation Optimization Passes Sequence Generation Framework

Jiang WU[†], **Jianjun XU**[†], **Xiankai MENG**[†a)], ***Nonmembers***, ***and*** **Yan LEI**[†], ***Member***

SUMMARY We propose a new framework named ROICF based on reinforcement learning orienting reliable compilation optimization sequence generation. On the foundation of the LLVM standard compilation optimization passes, we can obtain specific effective phase ordering for different programs to improve program reliability.

key words: *compilation optimization sequence, reinforcement learning, reliability, LLVM*

1. Introduction

At present, the research on compilation optimization sequence generation mainly focuses on program performance, power consumption, code size, etc. For example, in [1] the authors optimize the ordering for HLS programs and achieve an improvement in circuit performance and in [2] the authors achieve a better speedup by optimizing the search space. However, there is a small amount of research on program reliability [3], [4]. Compared with program performance-oriented research, in the ROPO (Reliability Oriented Phase Ordering) problem, the factors affecting the evaluation index are more complicated, and the acquisition of the evaluation index is more time-consuming. In order to find the approximate solution of the ROPO problem, it is necessary to explore in the compilation optimization space. The most commonly used exploration technology is iterative compilation technology. The iterative compilation optimization technology starts from an initial state, and through continuous trial, evaluation, and screening, finds the approximate solution of the optimal compilation optimization sequence. Iterative compilation optimization technology can be used alone [5] or combined with machine learning technology [6], [7], it can also be combined with some search strategies and heuristics. Based on the LLVM compilation framework, we design and implement an iterative compilation optimization framework named ROICF (Reliability Oriented Iterative Compilation Framework) for program reliability, then a reinforcement learning-based approach ROPOACER was designed as a targeted resolution of ROPO problem, which interacts with ROICF through multiple asynchronously running agents, and continuously adjust their own behavior mode according to the reward given by the environment, so as to learn to obtain high-reward value policy. For a certain program, we do not know which compilation optimization sequence is the most effective for it. We need an autonomous agent that will execute the compilation optimization pass (or sequence) as an action to interact with the environment, and through the interaction, the reward will return to an agent as a quantifiable scalar feedback signal, to evaluate the quality of an action. In this article, we continuously improve our behavior based on the reliability gain on the foundation of total variable alive time.

Manuscript received January 12, 2020.
Manuscript revised March 18, 2020.
Manuscript publicized June 22, 2020.

[†]The authors are with National University of Defense Technology, China.
a) E-mail: mengxiankai0364@163.com (Corresponding author)

DOI: 10.1587/transinf.2020EDL8006

2. ROICF Approach

Based on the LLVM compilation framework, we have designed and implemented the ROICF for program reliability. This framework can be combined with reinforcement learning technology to solve the phase ordering problem. The agent obtains information from the ROICF framework to form a strategy-exploring agent. Benchmarks are the set of target programs that are generated compilation optimization passes.

ROICF consists of seven components: Program random selector, Intermediate code generator, Compilation optimizer, Feature extractor, Counter of sequence-length, Reliability analysis tool, and History manager. The functions of each component are as follows. **Program random selector**: responsible for selecting a target program from benchmarks as a round of exploration, each target program can contain multiple source code files and input data required when the program is running; **Intermediate code generator**: responsible for converting and merging target program code into an LLVM intermediate representation (IR); **Compilation optimizer**: responsible for compiling and optimizing the current IR file and generating a new IR file; **Feature extractor**: responsible for extracting the features of the current IR file and forming a feature vector; **Counter of sequence-length**: responsible for recording the number of compilation optimization passes (or subsequences) that the current IR file passes, that is, the length of the compilation optimization sequence currently that has been explored; **Reliability analysis tool**: reliability analysis of current IR files, using a combination of dynamic analysis and static analysis; **History manager**: responsible for managing IR files and IR features generated under the compilation optimization passes that have been explored. The history includes data such as features, reliability gains, etc. So that they can be quickly provided to

agents when the same compilation optimization passes are explored again. Before the compilation optimizer, feature extractor and reliability analysis tools perform their respective analyses through the historical accessor.

When receiving an action command *action-x*, ROICF first determines whether the received motion *action-x* is a STOP command. If it is a STOP command, a reliability analysis tool is used to perform a reliability analysis on the current IR file to obtain its reliability gain, return to the agent as a reward; if it is not a STOP command, then *action-x* must correspond to a compilation optimization pass or a compilation optimization sub-sequence, and ROICF will perform the following operations: (1) Compilation optimizer performs compile optimization on the current IR file according to the compile optimization pass (or subsequence) corresponding to *action-x*, generates a new IR file instead of the original IR file, and stores the original IR file in the historical database; (2) Extracting the features of the current IR file by feature extractor and sending it to the agent in a vector form; (3) Counter of sequence-length add 1; (4) Determine whether sequence-length exceeds the preset maximum compilation optimization sequence length M. If it does not exceed, return reward = 0; otherwise, analyze the reliability gain of the current IR file with a reliability analysis tool and return it to the agent as the value of the reward.

3. Reinforcement Learning Model

Through a series of interactions with the ROICF (sending action commands to the ROICF, observing the IR feature vectors returned by the ROICF, obtaining Reward, etc.), the agent learns the ability to solve the problem of reliability-oriented compilation optimization sequence generation in the exploration. The process from when the agent issues a Reset command to when the agent issues a STOP command or the number of compilation actions sequence-length exceeds M is called a round of exploration. Because the Reset command is only executed at the beginning of each round of the exploration process, each round of the exploration process can only be performed for the same target program. At the end of each round of exploration process, the agent will explore a compilation optimization sequence and a reward value corresponding to it. During a round of exploration, each interaction between the agent and the ROICF is called an iteration step. As shown in Fig. 1, in the iteration step t, the agent collects actions from the action set $\mathrm{A} = \{\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{n-1}, \mathrm{STOP}\}$ according to the current IR feature X_t, and send it to ROICF for execution. The action a_t will affect the IR feature X_{t+1} in the next iteration step of ROICF and the reward value that the agent will get at the end of this round of exploration as shown in Fig. 1. The process can be regarded as the task T that the agent interacts in the environment E (ROICF). The goal of task T is to enable the agent to obtain the largest cumulative reward value in the interaction. Task T can be formally described as a Markov decision process through a five-tuple $\langle \mathrm{S}, \mathrm{A}, \rho, f, \gamma \rangle$:

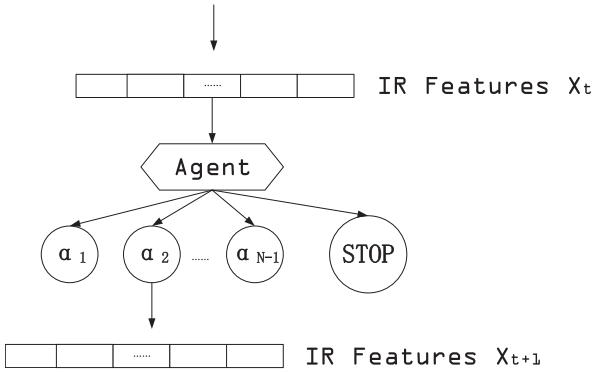

Fig. 1 Agent exploration process.

1. Define a state space S and use the current IR feature vector X_t to represent the environment state s_t, that is, for any $s_t \in \mathrm{S}$, there is $s_t = X_t$;
2. The action space A is the set of actions that the agent can perform. In task T, $\mathrm{A} = \{\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{n-1}, \mathrm{STOP}\}$, where each $\alpha_i \in \mathrm{A}$ is a compilation optimization pass (or subsequence), STOP is the end action command;
3. ρ: $\mathrm{S} \times \mathrm{A} \rightarrow \mathrm{R}$ is the reward function. $r_t = \rho(s_t, a_t)$ represents the immediate reward value obtained by the agent performing the action a_t in the state s_t, in task T, when a_t is the STOP command or sequence-length > M, the reward value is the current reliability gain of IR, otherwise the immediate reward value r_t is 0;
4. f: $\mathrm{S} \times \mathrm{A} \times \mathrm{S} \rightarrow [0, 1]$ is the state transition probability distribution function, $f(s_t, a_t, s_{t+1})$ represents the probability of the agent transitioning to the state s_{t+1} after performing the action a_t in the state s_t in task T, the distribution of state transitions is implicit in the compilation optimization sequence and IR characteristics;
5. γ is a predefined loss discount factor, and $\gamma \in [0, 1]$, which reflects the degree of attenuation of the current action's impact on future returns. $\gamma = 0$ means that the current action can only affect the current reward value, and $\gamma = 1$ means the impact of the current action on future returns is determined. Generally, γ takes a value of 0.9 or 0.99, which means that the current action will affect future returns, but the influence gradually weakens as the iterative step progresses.

Among many reinforcement learning methods, ACER [8] is widely used for its high efficiency, simplicity, and stability. ACER algorithm is based on the reinforcement learning algorithm Actor-Critic (AC) [9]. Actor-Critic is a Temporal Difference (TD) algorithm that combines the advantages of two types of reinforcement learning methods, value-based and policy-based. The Actor maintains a policy function $\pi(a_t|\boldsymbol{s}_t; \theta)$, and perform action selection based on the current state $\boldsymbol{s}_t$. Critic maintains a value function $\mathrm{V}(s_t; \theta_V)$, which is responsible for estimating the value of state $\boldsymbol{s}_t$. Based on the traditional Actor-Critic algorithm, ACER utilizes the parallel capabilities of multi-core CPUs to run multiple agents asynchronously to learn separately.

Each thread has an agent running in the copy of the environment. Each step generates a gradient of parameters. These gradients of multiple threads are added up and the shared parameters are updated together after a certain number of steps. Then introduces the advantage function, experience replay and off-policy training, which greatly improves the learning efficiency and sample utilization efficiency. We describe the ROPO problem as a Markov decision process that the agent explores under the working environment ROICF. Based on this description, we build a model orienting ROPO problem named ROPOACER. In order to alleviate the problem of perceived confusion, ROPOACER uses LSTM [10], and historical observation, action and feedback sequences are included in the calculation range and participate in action selection together.

4. Program Reliability Measure

During the program execution, a variable can be read or written. The variable is alive from the first write (followed by a read) to the last read (before the next write or the ending of the program), otherwise, it is dead. When a variable is dead, its content is irrelevant to the correct program execution since it will be either rewritten or never used again. In basic block, the positions between adjacent sentences are called points, and the positions before the first sentence and after the last sentence are also called points [11]. The path from point P_1 to point P_n is such a sequence of points P_1, P_2, ..., P_n. For a statement S, we define the point immediately before S as the entry point of the statement and the point immediately after S as the exit point of the statement. Based on this we define several sets:
in[S]: Set of alive variables at the entry point of statement S;
out[S]: Set of alive variables at the exit point of statement S;
def[S]: Set of variables defined in statement S and not referenced in S before the definition;
use[S]: Set of variables referenced in the statement S and not defined in S before the reference;
according to the above definition of the set, we can get two data flow equations, S_N is all successors of S:

$$\text{in[S]} = \text{use[S]} \cup (\text{out[S]} - \text{def[S]}) \quad (1)$$

$$\text{out[S]} = \cup\, \text{in[S}_\text{N}\text{]} \quad (2)$$

In this way, we can get the set of alive variables at the exit of each statement in the intermediate code.

Mukherjee et al. [12] divided all the bits in the system and all points in the space formed by the execution cycle into ACE (Architecturally Correct Execution) bits and un-ACE bits. The concept of ACE bits formalizes this notion. Let us assume that a program runs for 10 billion cycles through a microprocessor chip. Out of these 10 billion cycles, let us assume that a particular bit in the chip is only required to be correct 1 billion of those cycles. In the other 9 billion cycles, it does not matter what the value of that bit is for the program to execute correctly. The bit is an ACE bit—required for ACE—for 1 billion cycles. For the rest of the cycles, the value of the bit is unnecessary for ACE and therefore termed un-ACE. We introduce the idea of the ACE bits into variables. Similarly, the longer the alive time of a variable, it belongs to the ACE variables in more instruction cycles, and it will be stored in registers in more instruction cycles and even more vulnerable to soft errors like SEU. The more ACE variables in a cycle, the more vulnerable the program is. We consider that each LLVM instruction is executed in one clock cycle, in the foregoing, we defined that there is an alive variable set out[S] at the exit of each statement S (equivalent to LLVM instruction here), and each alive variable in out[S] is alive during the period in which S is executed, so in the cycle where S is executed, the number of alive variables is the quantity of out[S]. For an LLVM instruction S, we define that during the complete execution of the program, the total number of alive variable cycles corresponding to this instruction is the product of the quantity of out[S] and the total number of S executions. Then for the entire program, during the complete execution of the program, the total number of variable alive cycles is the sum of the number of variable active cycles corresponding to each instruction. Our experiment will discuss the RG (reliability gain) brought by the compilation optimization sequence to the program on the basis of TNVAC (total number of variable alive cycles). We define the RG of a program after compilation optimization sequence C for a given target program P as:

$$RG_{P,C} = 1 - \frac{TNVAC_{P,C}}{TNVAC\_baseline_P} \quad (3)$$

where $TNVAC_{P,C}$ refers to the corresponding TNVAC under program P and compilation optimization sequence C, and $TNVAC\_baseline_P$ refers to TNVAC when program P is not optimized.

5. Experiment and Results

The ROPOACER model was fully implemented based on the ROICF framework, and the ROPOACER algorithm framework is implemented by the Coach [13]. We randomly selected 200 programs from the *Singlesource* (Contains test programs that are only a single source file in size) and *Multisource* (Contains subdirectories which entire programs with multiple source files) collections in the LLVM test suite, while expanding the number of programs in each category in Mibench [14] to 10. IR features include 123 pieces of information in 4 categories: instruction technical information, data-dependent information, memory dependent information, and alias information. We can directly use the information provided by LLVM to save the time overhead of extracting program features from raw code [15]. We apply the trained ROPOACER model to the test set, and generate a compilation optimization sequence for each program in the test set, and the reliability gain calculated from the previous definition is compared with the GA (Genetic Algorithm)-based [16] and the SA (Simulated Annealing Algorithm)-based [17] compilation optimization sequence generation.

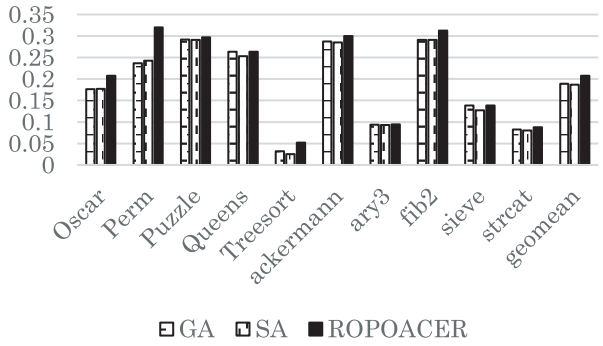

Fig. 2 Comparison of RG on the LLVM test suite.

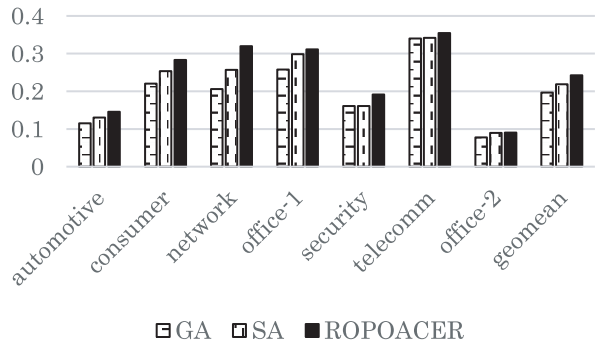

Fig. 3 Comparison of RG on Mibench.

In our experiments, the subsequence partitioning method proposed in [18] is followed. According to the function of the compilation optimization pass, the compilation optimization pass or sequence is divided into four categories: C1-code modification, C2-code motion, C3-code elimination, and C4-loop-related optimization. The number of passes (or subsequence) contained in the four categories is 35, 6, 20, and 14. The compilation optimization subsequence and the end command STOP constitute the action space $A = \{C1, C2, C3, C4, STOP\}$. On this basis, we do not want to have too many repeated subsequences in the compilation optimization sequence. In addition, each subsequence has a chance to appear at least once, so the maximum compilation optimization sequence length is set to 5. The results are shown in Figs. 2 and 3. The data set is divided according to the LLVM test suite and Mibench, and every data set is divided into a training set, a validation set, and a test set according to a ratio of $8 : 1 : 1$.

6. Conclusion

From the experimental results, in terms of reliability oriented compilation optimization sequence generation, our framework ROICF based on reinforcement learning has better results than SA and GA. On the LLVM test suite, the geomean RG improves by 0.0208 than SA and 0.018 than GA, on Mibench, the geomean RG improves by 0.0233 than SA and 0.0453 than GA. And our model has a good generalization ability. The model trained on small-scale programs is applied to large-scale programs with an accuracy rate of 95%.

The advantages of reinforcement learning in the sequential decision have been reflected. Compared with the previous work that leverages RL to tackle the phase-ordering problem to optimize for performance, our research combines reinforcement learning methods with iterative compilation, and it is well known that program reliability measurement costs a lot. We set the optimization goal as program reliability, which provides a powerful reference for software reinforcement research.

References

[1] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "AutoPhase: Compiler phase-ordering for HLS with deep reinforcement learning," IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), p.308, 2019.

[2] L.G.A. Martins, R. Nobre, J.M.P. Cardoso, A.C.B. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," ACM Trans. Architecture and Code Optimization (TACO), vol.13, no.1, Article No.8, 2016.

[3] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," 2016 12th European Dependable Computing Conference (EDCC), pp.1–12, IEEE, 2016.

[4] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: Embedded code generation aiming at reliability," Proc. seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp.237–246, IEEE, 2011.

[5] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," Workshop on Profile and Feedback-Directed Compilation, Paris, France, Oct. 1998.

[6] G. Fursin, Y. Kashnikov, A.W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C.K.I. Williams, and M. O'Boyle, "Milepost GCC: Machine learning enabled self-tuning compiler," International Journal of Parallel Programming, vol.39, no.3, pp.296–327, 2011.

[7] A.H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "COBAYN: Compiler autotuning framework using Bayesian networks," ACM Trans. Architecture and Code Optimization (TACO), vol.13, no.2, Aricle No.21, 2016.

[8] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," arXiv preprint, arXiv:1611.01224, 2016.

[9] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," Proc. 33rd International Conference on Machine Learning, pp.1928–1937, 2016.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Comput., vol.9, no.8, pp.1735–1780, 1997.

[11] F.E. Allen and J. Cocke, "A program data flow analysis procedure," Commun. ACM, vol.19, no.3, p.137, 1976.

[12] S. Mukherjee, Architecture Design for Soft Errors, Morgan Kaufmann, 2011.

[13] https://nervanasystems.github.io/coach/usage.html

[14] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proc. Fourth Annual IEEE International Workshop on Workload Characterization, WWC-4. pp.3–14, IEEE, 2001.

[15] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp.219–232, IEEE Computer Society, 2017.

[16] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Summary the Applications of GA-Genetic Algorithm for Dealing with Some Optimal Calculations in Economics, Addison Wesley, Reading, MA, 1989.

[17] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning," Operations Research, vol.37, no.6, pp.865–892, 1989.

[18] R. Coke, K. Pappas, A. Garimella, et al., "Optaw: Beating LLVM-O3 using source code features."