Information Sciences Letters
An International Journal

http://dx.doi.org/10.18576/isl/120601

Optimizing Program Efficiency with Loop Unroll Factor Prediction

E. Alwan [*] *and R. Al Baity*

Department of Computer Science, Collage of Science for Women, University of Babylon, Iraq

Received: 22 Feb. 2023, Revised: 26 Feb. 2023, Accepted: 17 Mar. 2023.
Published online: 1 Jun. 2023.

Abstract: Loop unrolling is a well-established code transformation technique that can improve the performance of a program at runtime. The key benefit of unrolling a loop is that it often requires fewer instruction executions than the original loop. However, determining the optimal number of loop unrolling is a critical concern. This paper presents a novel method for predicting the optimal unroll factor for a given program. Specifically, a dataset is constructed that includes the execution times of several programs with varying loop unroll factors. The programs are sourced from different benchmarks, such as Ploybench, Shooutout, and other programs. Similarity measures between the unseen program and the existing programs are computed, and the three most similar programs are identified. The unroll factor that led to the greatest reduction in execution time for the most similar programs is selected as the candidate for the unseen program. Experimental results demonstrate that the proposed method can enhance the performance of training programs for unroll factors of 2, 4, 6, and 8 by approximately 13%, 18%, 19%, and 21%, respectively. For the unseen programs, the speedup rate is approximately 37.7% for five programs.

Keywords: loop unroll, compiler optimization, execution time.

1 Introduction

The execution time of programs is significantly allocated to a small proportion of their code, which is primarily located within loop constructs. It has been observed that programs expend approximately 90% of their execution time in 10% of the code. Therefore, directing efforts towards enhancing the frequently executed portions of the code can have a substantial impact on the overall program execution time. [1,2] As a result, code optimization techniques that accelerate loop execution are essential. [3,4]

One technique for improving program execution time is loop unrolling. This technique involves replicating the loop body multiple times while adjusting the loop termination code. By decreasing the overhead of loop termination, loop unrolling can improve code execution time. This is achieved by reducing the number of branch instructions needed at the end of the loop body. [5,2]

Loop unrolling is essential for certain optimizations, particularly those aimed at improving the memory system. Enabling loops unroll, generates numerous static memory instructions that can be rescheduled to take advantage of memory locality. In practice, loop unrolling enhances performance in nearly every scenario in which it is applied. However, if used improperly, loop unrolling can negatively affect other critical optimizations and decrease overall speed. Additionally, selecting the appropriate unrolling factor is crucial. An optimal unrolling factor reduces execution time while enhancing overall performance. [6] Although loop unrolling has numerous advantages, there are also several potential disadvantages that should be considered:

• The most well-known disadvantage of unrolling is that it can reduce the performance of the instruction cache.

• Additional scheduling freedom can lead to a rise in variable live ranges, resulting in extra register pressure. [7],8,9]

Based on the aforementioned, this study aims to create a model that can predict the optimal unrolling factor. The remaining sections of this paper are organized as follows: Section 2 provides an overview of related research on loop unrolling. Section 3 outlines the proposed approach for loop unrolling. Section 4 presents the results obtained from benchmark programs. Finally, Section 5 summarizes our conclusions and provides closing remarks.

[*]Corresponding author e-mail: esraa.hasi@uobabylon.edu.iq

© 2023 NSP
Natural Sciences Publishing Cor.

2 Literature Review:

This section discusses related work that is relevant to this research. We emphasize relevant work in this area because our study focuses on applying learning techniques to compilation.

Meisam Booshehri et al [10] emphasized on the loop unrolling approach and its impacts on power consumption, energy usage, and program speed by obtaining ILP (Instruction-level parallelism) (Instruction-level parallelism). Concentrating on superscalar processors, they studied J.C. Hang and T. Leng's idea of generalized loop unrolling and then presented a novel way to traverse a linked list to acquire a better outcome of loop unrolling in that circumstance. They ran their experiments using a Pentium 4 CPU (as an instance of super scalar architecture). In addition, the findings of some other experiments carried out on a supercomputer (the Alliat FX/2800 System) with superscalar node processors. These investigations demonstrated that loop unrolling had a minor detectable influence on energy and power consumption. However, it could be an efficient approach to speed up the program. Mark Stephenson and Saman Amarasinghe [7] demonstrated how machine learning methodologies may help compiler designers design complex systems. They focused on loop unrolling, a well-known approach for detecting instruction-level parallelism. They explained how to utilize the Open Research Compiler as a testbed to determine the effectiveness of loop unrolling using supervised learning methods. Over 2,500 loops from 72 benchmarks were utilized to train two separate learning algorithms to estimate unroll factors (the length of time a loop should be unrolled) for each new loop. The method accurately predicts the unroll factor for 65% of the loops in our sample, resulting in a 5% improvement in the overall performance of the SPEC 2000 benchmark suite.

Liu and Guo [8] employ a machine learning model to enhance the compiler's loop unrolling optimization capabilities. To begin, weighting and unbalanced dataset processing are applied to the basic random forest model. The training set is then constructed in order to train the model. According to the results of the experiment, the model can deliver the optimal or sub-optimal unrolling factor within 81% of the time after training. It is also put through numerous SPEC2006 test sets. The built-in loop unrolling model in Open64 can only increase program performance by 5% on average, however the technique suggested in this research for predicting loop unrolling components using weighted decision forest can enhance program performance by 12% on average. In [1], a loop unrolling approach based on enhanced random decision forest was developed in order to increase the accuracy of the compiler's loop unrolling factor. First, they enhanced the standard random choice forest by introducing weight values. Second, to address the issue of unbalanced data sets, a BSC approach based on the SMOTE algorithm was proposed. Almost 1000 loops were chosen from various benchmarks, and the features retrieved from these loops provide the training set for the loop unrolling factor prediction model. The model has an unrolling factor prediction accuracy of 81%, whereas the present Open64 compiler only has a 36% forecast accuracy.

3 Methodologies

As shown in Figure 1, the proposed model comprises four stages.

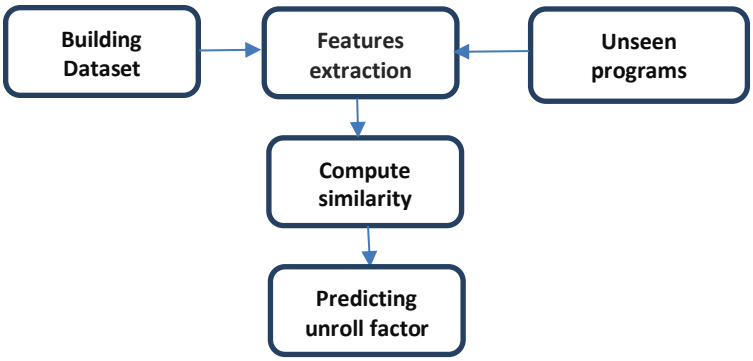

Fig. 1: Illustrates the proposed methods which consist of four stages

- **Building Dataset**

For this stage, a total of forty programs from diverse benchmarks are collected. The execution time for each program is measured using several loop factors. Four loop factors are chosen, as previous experiments have demonstrated that they have the most significant impact on program performance (2, 4, 6, 8). Each program is executed multiple times (more than five) with each one of these factors, and the average execution time is calculated.

© 2023 NSP
Natural Sciences Publishing Cor.

- **Features extraction using LLVM**

The features of C or C++ programs are extracted from their LLVM Intermediate Representations using the LLVM analysis pass -*instcount*, which tallies the different types of instructions present in the function. Table (1) displays the LLVM features that are gathered for each program. These features provide insight into the program's static behavior. A total of 39 features are distributed across various programs.

- **Computing similarity**

This stage consists of two steps as shwon below.

- Extract features for unseen program.
- Compute the similarity between the unseen program and the set of programs in the dataset. There are several scales for computing similarity. Cosine scale is used to compute the similarity as shown in equation 1:

$$\text{Sim(p,pi)}=\frac{\sum_{w=1}^{m} pw*piw}{\sqrt{\sum_{w=1}^{m} pw^2}*\sqrt{\sum_{w=1}^{m} piw^2}} \quad (1)$$

Table 1: program static features

Add instructions	FAdd instructions	GetElementPtr instructions	Ret instructions	SRem instructions	ZExt instructions
Alloca instructions	FCmp instructions	ICmp instructions	SDiv instructions	Shl instructions	basic blocks
And instructions	FDiv instructions	Load instructions	Sub instructions	Store instructions	memory instructions
AShr instructions	FMul instructions	Mul instructions	Switch instructions	Trunc instructions	non-external functions
BitCast instructions	FPExt instructions	or instructions	SExt instructions	URem instructions	
Br instructions	FPToSI instructions	PHI instructions	Select instructions	Unreachable instructions	
Call instructions	FSub instructions	PtrToInt instructions	SIToFP instructions	Xor instructions	

Where p represents the main program (unseen program) and pi represents the other programs (training programs).[2,11,12]

- **Predicting loop unroll factor**

- To begin, we select the three most similar programs to the unseen one and place them in the similarity set.
- Next, we rearrange these programs based on their potential benefits from the loop unroll factor. To do this, we examine the dataset of these programs to identify the loop factor that offers the greatest performance improvement for all three programs. This loop factor, which results in a high score, is presented as a potential candidate for the unseen program. Essentially, we choose the loop unroll factor that can reduce the execution time for the majority of similar programs. In summary, the objective is to select the optimal loop unrolling factor for a given set of programs to minimize their execution time. To achieve this goal, we analyze the dataset and determine the highest factor that can be applied to most of the programs. For instance, if a factor of 2 is effective for a majority of the programs, we evaluate the performance of a factor of 4. If this factor provides an improvement in execution time for most of the programs, we then examine a factor of 6. However, if the factor of 4 remains the most efficient option for a significant number of the programs, we consider it as a suitable candidate for unrolling loops in any future unseen program. On the other hand, if the factor of 4 is not appropriate for the majority of the programs, we revert to the initial factor of 2 for further analysis.

4 Performance Evaluation

This section aims to investigate whether a well-prepared dataset for loop unrolling factors can lead to faster program execution. To achieve this objective, we employ the acquired dataset from various benchmarks, including Polybench, Shootout, Stanford, and others to predict an appropriate unroll factor for each loop and compile the benchmark program accordingly. Notably, the dataset employed in this study comprises more than fifty programs, of which forty-one yielded satisfactory results with varying loop unroll factors, excluding those from the benchmark program used for evaluating the results. This enabled us to assess the effectiveness of the learned dataset on previously unseen loops.

© 2023 NSP
Natural Sciences Publishing Cor.

4.1. Training set programs

In this study, we compiled the programs using four different loop unroll factors, namely 2, 4, 6, and 8. To compute the entire runtimes, we employed the UNIX time command and performed five trials to obtain an average. Figure (2) illustrates the training set programs, or dataset, and displays the effects of these factors on program execution time compared to the normal case, i.e., no loop unrolling. The programs achieved speedups of 13%, 18%, 19%, and 21% for loop unroll factors of 2, 4, 6, and 8, respectively. It is noteworthy that all programs experienced a reduction in execution time with loop unroll factors of 2 and 4. However, few programs derived benefits from loop unroll factors of 6 and 8, respectively.

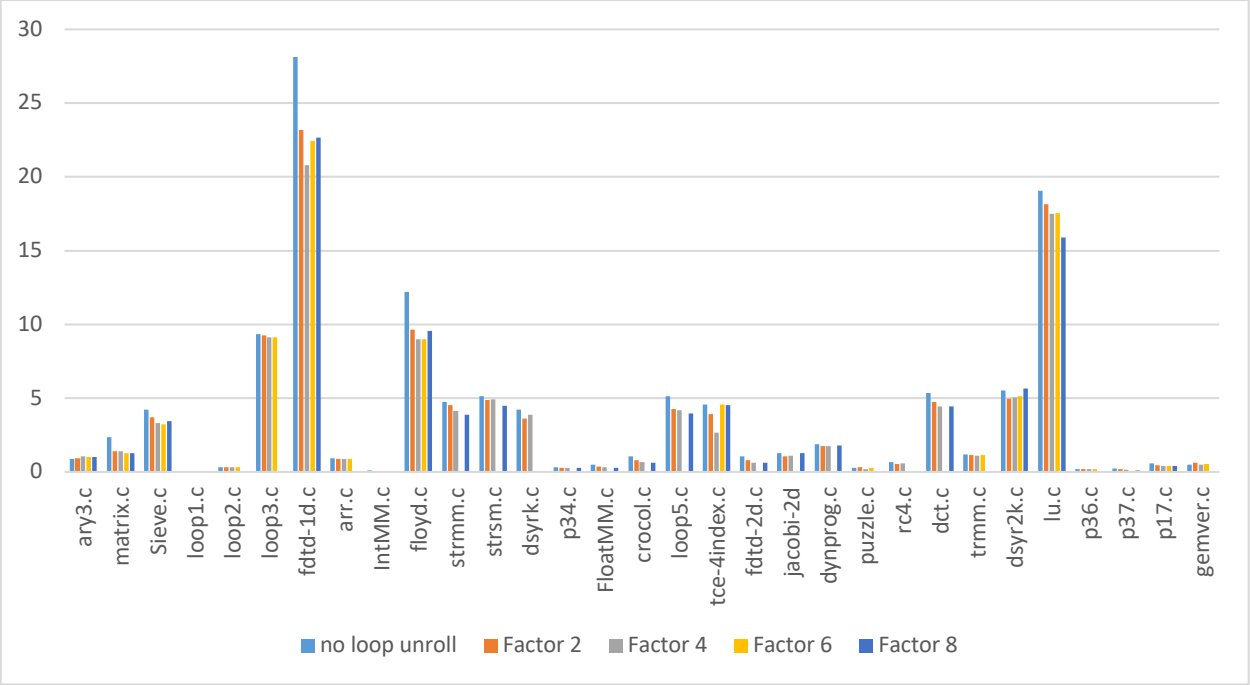

Fig. 2: training set programs

4.2 Results of unseen programs

In these experiments, five unseen programs are used to validate the proposed method. For each unseen program we computer its similarity with the data set and extract the three of most similar programs and candidate high loop unroll factor that most of the similar programs get benefit from it (speed up their execution times).

Symmat.c

We compute the similarity between this program and the programs in the dataset and we extract the most similar three programs which are loop3.c, strmm.c, dct.c. Then we scan the dataset and we get two similar programs benefit from the highest factor 8. Figure (3) illustrates the effect of the loop unroll factors on the program execution time where factor 8 is the best.

© 2023 NSP
Natural Sciences Publishing Cor.

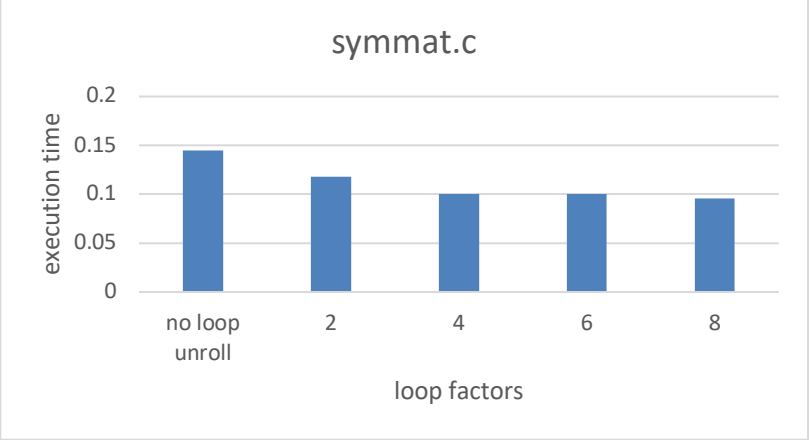

Fig. 3: illustrates the predicted factor for symmat.c program

syrk.c

The second unseen program is syrk.c. We find the most similar three programs are ary3.c, floatMM.c, trim.c and the predicted factor is 4 as illustrated in the figure below.

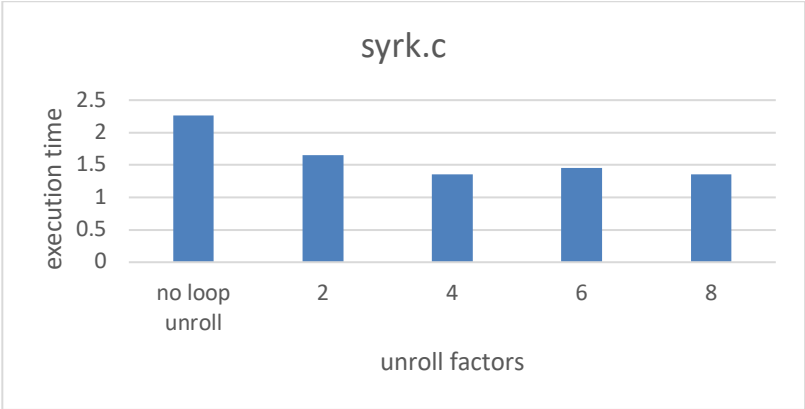

Fig. 4: illustrates the predicted factor for syrk.c program

convariance.c

The third unseen program is convariance.c. We find the most similar three programs are strmm.c.c , p34.c ,dct.c and the predicted factor is 8 as illustrated in figure (5)

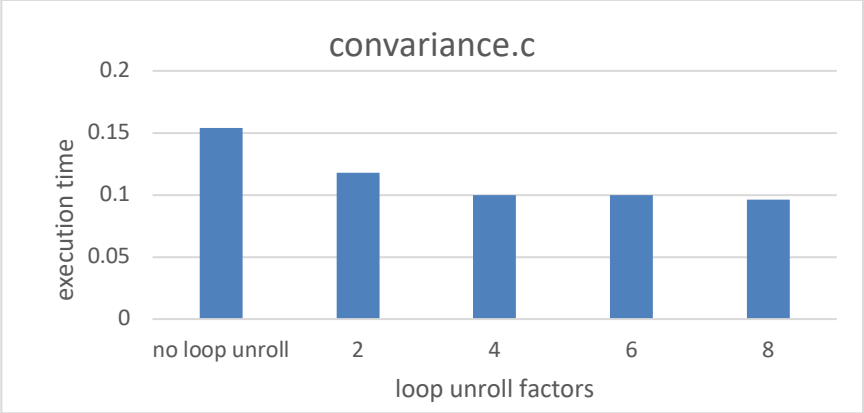

Fig. 5: illustrates the predicted factor for convariance.c program

Jacobi-1d-imper.c

The fourth unseen program is Jacobi-1d-imper.c. We find the most similar three programs are floy.c , fdtd.c ,dct.c and

© 2023 NSP
Natural Sciences Publishing Cor.

the predicted factor is 4 as illustrated in figure (6)

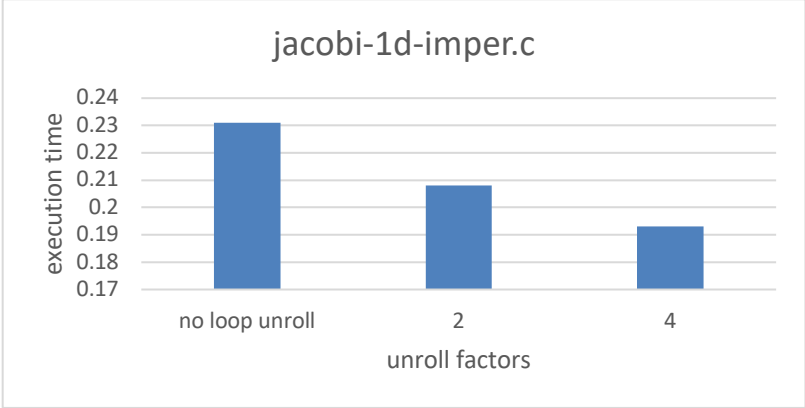

Fig. 6: illustrates the predicted factor for jacobi-1d-imper.c program

Doitgen.c

The fifth unseen program is Doitgen.c. We find most similar three programs are dct.c , Jacob-2d-imper.c , strmm.c and the predicted factor is 8 as illustrated in figure(7)

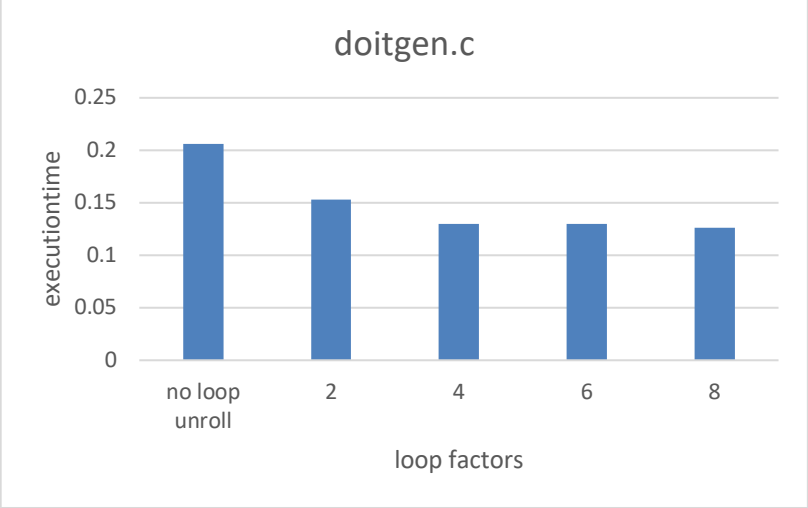

Fig. 7: illustrates the predicted factor for doitgen.c program

4.3 Discussion of the results

Our study has revealed that for various benchmark programs, approximately 40-50% of all loops could not be unrolled using the loop unroll LLVM opt command line due to one or more of the subsequent causes: (1) the initial loop induction variable value is not a high value, (2) the loop comprises conditional control. Additionally, we found that the highest loop unroll factor that can yield a speedup for very few programs was eight, and the speedup remains the same even with the highest factor, such as 16. However, for some programs, their performance was degraded.

A noteworthy observation is that a loop with a high unroll factor, such as eight, is already performing well with a small unroll factor, such as two, whereas the opposite is not necessarily true. Moreover, the findings indicate that no single unrolling factor significantly outperforms others.

5 Conclusions

In order to enhance program speed, the compiler applies various LLVM transformation passes with

loop unrolling. This study proposes a method to improve the loop unrolling optimization ability of the compiler. First, a dataset comprising the execution time of a set of programs with varying loop unroll factors is constructed. Then, the similarity with unseen programs is computed, and the similar programs are reordered based on their potential benefits from loop unroll factors. The highest loop unroll factor that can reduce the execution time for most of the similar programs

© 2023 NSP
Natural Sciences Publishing Cor.

is selected. The experimental results demonstrate that the proposed method can accelerate the training programs with varying factors by approximately 17%, while the speedup for unseen programs with different candidate factors is approximately 37.7%.

Conflict of interest

The authors declare that there is no conflict regarding the publication of this paper.

References

[1] S. Singh, R. Singh and S. Kumar, "Efficient Loop Unrolling Factor Prediction Algorithm using Machine Learning Models," in Proc.INCET, 1-8, (2022).

[2] M. Almohammed, A. Fanfakh, and E. Alwan, “Parallel genetic algorithm for optimizing compiler sequences ordering,” in Proc.CCIS, 128-138, (2020).

[3] G. Zacharopoulos, A. Barbon, G. Ansaloni and L. Pozzi, "Machine Learning Approach for Loop Unrolling Factor Prediction in High Level Synthesis," in Proc. HPCS, 91-97, (2018).

[4] P. R. Panda, N. Sharma, S. Kurra, K. A. Bhartia and N. K. Singh, "Exploration of Loop Unroll Factors in High Level Synthesis," in Proc. VLSID, 465-466, (2018).

[5] J. C. Huang and T. Leng, "Generalized loop-unrolling: a method for program speedup," in Proc. ASSET'99,244-248, (1999)

[6] L. Domagała, D. V. Amstel, F. Amstel, P. Sadayappan, “Register allocation and promotion through combined instruction scheduling and loop unrolling”. In Proc CC, 143-151, (2016).

[7] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in Proc. CGO,123-134, (2005).

[8] H. Liu and Z. Guo, “A loop unrolling method based on machine learning,” in Prdc. Vibroengineering PROCEDIA, 215–221, (2018).

[9] M., Hall, J., Chame, C., Chen, J., Shin, G., Rudy, M.M Khan. Loop Transformation Recipes for Code Generation and Auto-Tuning. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds) Languages and Compilers for Parallel Computing. LCPC 2009. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg vol 5898, (2010)

[10] M. Booshehri, A. Malekpour, and P. Luksch, “An improving method for loop unrolling”, International Journal of Computer Science and Information Security,vol. 11,No. 5,(2013).

[11] J. W Davidson and S. Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In International Conference on Compiler Construction. In: Gyimóthy, T. (eds) Compiler Construction. CC 1996. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg., vol 1060, 59–73. (1996).

[12] L. H. Alhasnawy, E. H. Alwan, and A. B. M. Fanfakh, “Using machine learning to predict the sequences of optimization passes,” in Proc. CCIS,139-156, (2020).

© 2023 NSP
Natural Sciences Publishing Cor.