ISSN:0975-9646

A Proposed Collaborative Approach for Pattern Matching and Replacement Policies for the design of Peephole Optimizer

Chirag H. Bhatt[#], Dr. Harshad B. Bhadka[*]

[#]*Assistant Professor, Dept. of MCA, Atmiya Institute of Technology & Science, Rajkot, India,*

Ph.D. Scholar, School of Computer Science, RK University, Rajkot, India

[*]*Professor and Director – MCA, C. U. Shah University, Surendranagar, India*

Abstract— **In the perspective of peephole optimization, this paper describes a framework that approaches a collaborative fashion for the pattern matching and replacement strategies on the basis of formal exploration of the pattern matching strategies that have been implemented. In the first section, the framework of this paper is set up: Peephole optimization is observed from an information-processing viewpoint; the distinct components that are involved in this process are presented and formally defined. Then in the next sections the strategically modules are illustrated, which are the different pattern matching, rule application and replacement policies that have been evaluated in this work.**

Keywords— **Peephole Optimizer, Pattern Matching Strategies, Collaborative Fashion, Replacement Policies.**

I. INTRODUCTION

This paper is about three issues: pattern matching, Replacement policies for optimization rules application and peephole optimization. What is pattern matching? The pattern matching problem (in this instance within peephole optimization), can be observed as an information processing problem if generalized. Figure 1.2 shows the modules that are involved in this process. What exactly take place in this technique; what kind of information is processed? The input to this system is some assembly code to be optimized this is the entering information. Within the system, the peephole optimizer uses its information base – the optimization rules, to replace portions of the code. The dealings between the assembly code and the rules are where the soul of the task is, where information processing happens. If the match is successful, the code is replaced, otherwise it remains unchanged. The assembly code is returned as the output of the system, once the application of rules is accomplished. This is basically what happens in a peephole optimizer [1], [2].

II. PREVIOUS WORK

A. *Flow of the Peephole Optimizer*

In the phases of compilation the Peephole Optimizer performs the task of improving the suboptimal input ASM code as per the flow shown in the figure II.1. As per the flow the suboptimal assembly (ASM) code is given as input to the Peephole Optimizer along with the set of optimization rules. This describes the conditions which may occur in the ASM code and based on the matching the suboptimal portion of the code would be replaced with the optimal one in the set. In the next part the peephole optimizer provides the interface and interaction between the input ASM code and the Pattern Matching and Replacement policies based on which the optimization can take place along with the operation of Parsing. And in the next step the Approach which includes the specific mechanism to deal with the optimization rules and the ASM code where algorithm or strategy can be applied to obtain results in the form of equivalent and optimized ASM code as output [2], [3].

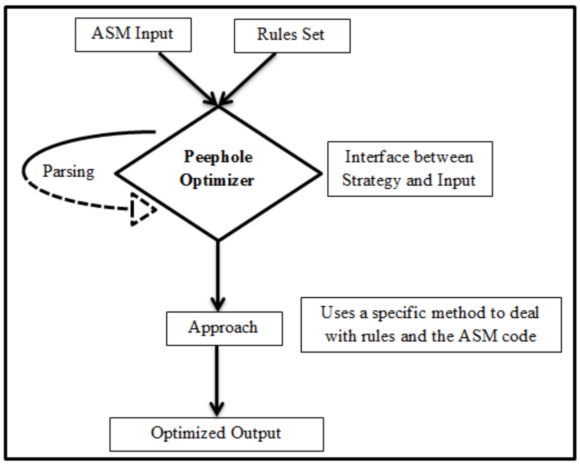
ASM Input
Rules Set
Parsing
Peephole Optimizer
Interface between Strategy and Input
Approach
Uses a specific method to deal with rules and the ASM code
Optimized Output

Fig. 1: Flow of the Peephole Optimizer [3]

B. *Design of Peephole Optimizer*

The Design of Peephole Optimizer in figure 2 displays a collaborative approach for the peephole optimizer. Based on the flow of peephole optimizer described above, the framework also takes suboptimal code as input along with optimization rules set. And within the peephole optimizer it consists of two major tasks called pattern matching technique and Replacement Policies. In this work the first task is related with the decision making of how to match a single rule, on the basis of more than one pattern matching mechanism available or suitable for specific snippet of ASM code and hence the combination of multiple pattern matching can overcome the limitations of the implementation of single pattern matching algorithm. And the other major task called Replacement Policies decides how to apply several rules based on multiple rule application approaches and it replaces the optimal code with unoptimal code in the input ASM code if the successful matches found, otherwise it would be remain as it is.

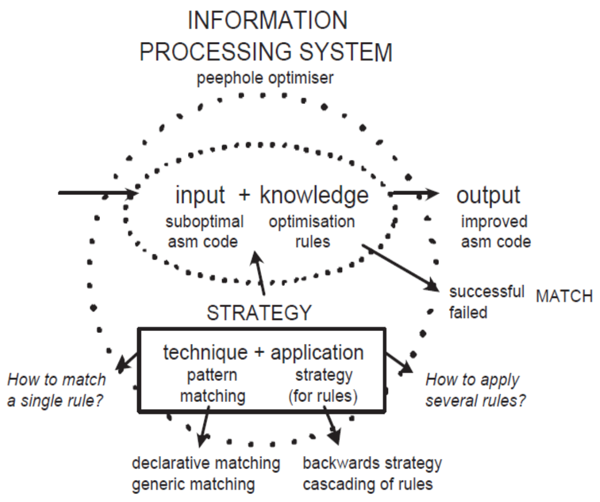

Fig. 2: Peephole optimisation as an information processing problem [11]

B.A.1 *Pattern Matching Technique*

1) Declarative Pattern Matching: In logical and functional programming languages like PROLOG, LISP, Haskell etc. pattern matching is a very leading programming feature. These languages provide functionalities driven through conditions, variables, data structures and relations between them in a descriptive style. The machine performs the computation based on the specifications are given in a declarative way. The interpreter observes the responsibility to manage this task in such programming languages. This generates result not only in less code, fewer algorithms, and in stress-free maintenance, but also in more compact picture of the logic of the program [4].

For this paradigm, the procedural and structured programming are playing corresponding role since in those languages the methods/procedures are used to represent the outcome obtained by steps that shall be carefully taken. Accordingly, in Java the lengthy code is written, both programming styles can be accepted. The objects are controlled by means of methods even though the procedural aspect is more common. Particularly the specification of the rules is expressed in a declarative style as in [4], using back-references and groups.

2) Generic Pattern Matching: A more intangible form of pattern matching that is conducted on an object level denoted by Generic Pattern Matching [5]. The declarative approach controls the pattern matching on the basis of the presentation of the optimization rules whereas The control of the generic pattern matching approach does not lie in the format of the optimization rules, but in the generic way of treating the patterns: The difference is represented with matching the objects themselves as abstract data types and the information it contains in a form of primitive types. These are mostly simple strings. Since the object contains and encapsulates the primitive one, with a conceptual significance to it, this allows more 'intelligent' matching: First the type is evaluated; only if it is capable, then the primitive information – the 'contents' – are matched. Hereafter, this higher level of abstraction permits an assembly code line to be further divided into ordinary instructions and label definitions; the previous ones further containing of the opcode part and the arguments, whereas the concluding ones typically hold label numbers. For optimization rules that involve look ahead, particularly this type of information is useful. In its place of using one more line of input for every matching attempt, the generic strategy uses stored information about assembly instruction and their elements on demand with just a few array accesses for quick retrieval. This approach is complex and expects a more operational implementation style, which is also lengthier in sense of the programming efforts.

The extended generic strategy goes one step further by probing for some overlapping optimization opportunities that might else disqualify each other. It then selects the best promising order of optimizations. Investigating with cooperating rules here becomes a very interesting issue.

B.A.2 *Replacement Policies*

1) Declarative Pattern Matching: The backwards strategy has been introduced in connection with Lamb's work [6]. It is described as a rule presentation strategy for both the declarative and generic pattern matching in its operation. As the below given figure 3 illustrates the cursor is set to top of the code line-1 where rules are being tested in forwards fashion. The Optimization rules are tested with the input in backwards fashion. The peephole window size of the optimizer is as big as the matching part of the current rule needs. Therefore, its corresponding part must fit into the presently reachable part of the assembly code input, which is defined by cursor in order for a rule to be 'testable'. This results in smaller rules being preferred over lengthier ones because they are chosen for testing ahead of time.

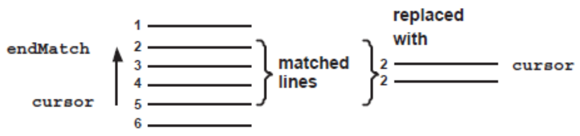

Fig. 3: Backward Strategy [6].

For a rule with a matching part of 4 lines, end-Match indicates the last line of the rule to be matched – cursor the first line which does not affect the backward strategy. If the replacement consists of 2 lines, the 4 lines are replaced and the cursor is set to the first line of the replacement. This method is required to detect all optimization opportunities further up the code that the replacement might have presented. Rescanning the assembly input becomes redundant with this approach. For the sake of ensuring that no optimization opportunities are skipped, the index to the rules file is set back and the complete set of rules is tested again upon a successful match-replacement operation. A key dissimilarity between the two forms presented is the introduction of look-ahead. An optimization rule can cover parts that match a random number of assembly code lines. Such lines are compared and only copied to the replacement without change [6].

2) Cascading of Rules: As per the above discussion, the advantage of the backwards strategy deceits in the fact that the new optimization opportunities are found promptly due to the assembly code input desires to be scanned once only. However, There is a vital weakness in this strategy: Due to the nature of the method smaller rules are favoured over longer ones. This is not always anticipated because the smaller rules mostly eliminate the longer ones that might be improving more or better. More indeed, the relationship between rules can be specified as follows:

Choice point: Given a pattern matching problem, a choice point defines a state in the assembly input C where at least two rules o_i and $o_j \in O$ are applicable to the input sequence $c_1 . . . c_n \in C$ [11].

Interaction of Rules: Given a choice point as defined, if a matching rule o_i either eliminates another matching rule o_j, or causes to succeed a rule o_j that did not match before, the rules o_i and o_j are described to interact. This relation is also valid for sets of rules $O_k = \{o_1. \;\; . \; . \; o_n\}$ and $O_l = \{o_1. \; . \; . \; o_m\}$[11].

According to the purpose one desires the optimization approach to implement, editing the rules file is a conceivable but clumsy solution to stop or enable the collaboration of rules. A far better option is to use an optimization approach that pursuits for alternative optimization orders to apply the most hopeful one. The purpose here is not to discover the best optimization sequence/s, which according to [7] is NP complete. For that cause the rule cascading strategy that has been adopted here as an addition to the generic strategy, acquires the advantages of the backwards strategy, observing a few rules ahead to apply the best available optimization sequence. The user can have investigation with the results by operating the limit of the look-ahead. For the avoidance of the scanning the complete assembly input, the maximum number of assembly lines examined after each match is also limited within a single optimization sequence. The standard for choosing the best available optimization sequence is optimization for space, i.e. the order of rules that rejects most of the code is selected. For this determination, mock choice points are formed, which define the start and end points of substitute optimization sequences. The results are maintained in so-called options [9], [10]:

Option: An Option keeps the state of a pattern matching problem between two choice points. Hence it holds the most recent information about the set of rules that have been applied, the state of the input, the labels table, and the cursor pointing to the input. Moreover, it also tracks the total sum of eliminated lines [11].

B.A.3 String-Based Matching And Replacement

The difficulty of string based pattern matching is related with finding a string substitution for a string pattern so that another string and the substitution pattern become equal [8]. Given a string wordn, the following typical 'patterns' can occur in a string matching problem within the context of peephole optimization:

$$pattern1 := word1$$
$$pattern2 := variable1$$
$$pattern3 := word1 \; variable1$$
$$pattern4 := word1 \; variable1 \; word2 \; variable2 \ldots$$

Consequently, in demand to match a wordn with one of the patterns above, it has to be split in such a way as to 'fit' into the pattern. Since the first two cases can be matched to the whole term immediately are simple and resulting in a successful or failed match. Though the Patterns 3 and 4 require the subsequent pattern character to be originate in the term (known as search string) whenever a variable is met in the pattern. One has to look ahead in both the word and the pattern in instruction to determine the length of the string to be matched with the variable.

B.A.4 Limitations of the Work

- The results are not representative due to the rules set provided by rules.txt with the use of compiler generated assembly files for input.
- Rules are only strategy oriented
- Rules with escapes are not implemented.
- The assembly parser is not implemented completely.
- Unidentified lines of input assembly files are commented to solve the temporary problems.

C. *Proposed Approach for the Peephole Optimizer*

In the Proposed approach of enhanced peehole optimizer more pattern matching techniques would be involved to achive better searching of unoptimized code of assembly provided as input file as well as more precised rules would be defined to overcome the limitations of the peephole optimizer explained above. More rule application stretegies also involved to decide more suitable replacement to be applied with the match found and if match not found then the search for the next opptortunity for optimization

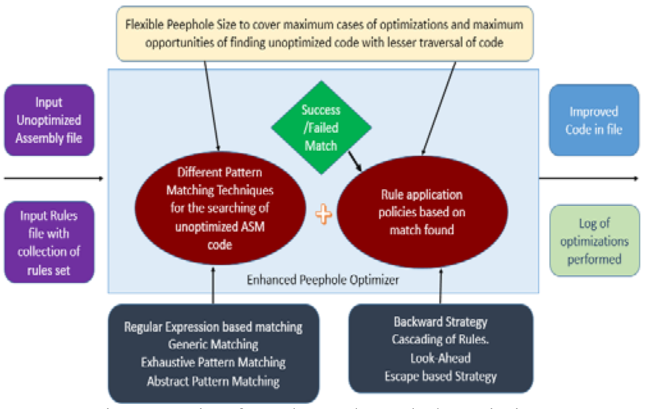
Flexible Peephole Size to cover maximum cases of optimizations and maximum opportunities of finding unoptimized code with lesser traversal of code
Input Unoptimized Assembly file
Input Rules file with collection of rules set
Success /Failed Match
Different Pattern Matching Techniques for the searching of unoptimized ASM code
Rule application policies based on match found
Enhanced Peephole Optimizer
Improved Code in file
Log of optimizations performed
Regular Expression based matching
Generic Matching
Exhaustive Pattern Matching
Abstract Pattern Matching
Backward Strategy
Cascading of Rules.
Look-Ahead
Escape based Strategy

Fig. 4: Design for enhanced peephole optimizer

The figure 4 proposes the new design for enhanced peephole optimizer with flexible peephole size that helps to cover maximum cases of optimizations and maximum opportunities of finding unoptimized code with less number of traversal of assembly code for searching for match and it may also improve the performance of the optimizer. The proposed design also include Exhaustive pattern matching and Abstract Pattern Matching techniques [12], [13] and some other pattern matching techniques can also be included letter on to improve the pattern matching process. Along with that escape based and look ahead based rules application strategies can also be adopted for alternatives that help better replacement policies. This design also suggests to generate logs of optimizations performed over input along with improved code as output which helps the enhanced peephole optimizer to learn some more optimizations in future.

III. Conclusions

This paper approach for a collaborative fashion for the Peephole Optimizer in terms of pattern matching techniques along with the strategies for rule application to conduct matching unoptimal code and to replace it with most optimal instruction sequence. Previously described strategies would have different qualities to lead the assembly code to the optimization. But they might not lead the code to the better option and they also busy with the searching of best optimization which consumes time. Rather than that the collaborative approach suggests to utilize more than one strategy to lead the code to the next optimal level.

References

[1] Aho, A. V., Sethi, R., Ullman, J. D. (1986) Compilers: Principles, Techniques, and Tools. Massachusetts: Addison-Wesley, pp. 554–558.

[2] Bhadka. H. B. Bhatt C. H., “Peephole Optimization Technique for analysis and review of Compiler Design and Construction,” IOSR Journal of Computer Engineering (IOSR-JCE), pp. 80-86, 2013.

[3] Davidson, J. W., Whalley, D. B. (1989) Quick compilers using peephole optimizations. Software - Practice & Experience 19(1):195-203.

[4] Spinellis, “Declarative Peephole Optimization Using String Pattern Matching,” ACM SIGPLAN Notices 34(2), p. 47–51, 1999.

[5] Visser, J., L¨ammel, R. (2004) Matching Objects. Available at http://www.di.uminho.pt/joost.visser/publications/MatchingObjects.pdf (up 12/02/2005).

[6] Lamb, D. A. (1981) Construction of a Peephole Optimizer. Software Practice & Experience 11(6):639–647.

[7] Grune, D., Bal, H. E., Jacobs, C. J. H., Langendoen, K. H. (2000) Modern Compiler Design. NY: John Wiley & Sons, pp. 1, 371–375.

[8] “grep.html,” [Online]. Available: http://stat.ethz.ch/R-nual/R-devel/library/base/html/grep.html.

[9] Ganapathi, M., Fischer, C. N. (1985) Affix Grammar Driven Code Generation.ACM TOPLAS 7(4):560–599.

[10] Visser, E. et al. (2000-2005) Stratego: Strategies for Program Transformation. http://www.stratego-language.org/ (up 01/05/2005).

[11] Elif Aktolga "Pattern Matching Strategies for Peephole Optimisation" August 26, 2005 MRes Dissertation in Computer Science and Artificial Intelligence, School of Science and Technology, University of Sussex

[12] Chinawat Isradisaikul and Andrew C. Myers. Reconciling exhaustive pattern matching with objects. Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'13), pp. 343–353, June 2013.

[13] Jed Liu, Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java, Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03), pp. 110–127, New Orleans, LA, Jan. 2003. LNCS 2562.