IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 9, Issue 4 (Mar. - Apr. 2013), PP 80-86
www.iosrjournals.org

Peephole Optimization Technique for analysis and review of Compiler Design and Construction

Mr. Chirag H. Bhatt[1], Dr. Harshad B. Bhadka[2]
[1] *Research Scholar (Computer Science), RK University, and Lecturer, Atmiya Institute of Technology & Science, Gujarat Technological University,*
[2]*Asst. Prof. and Director, C. U. Shah College of Master of Computer Applications, Gujarat Technological University,*

Abstract : *In term of Compiler optimization, classical compilers implemented with an effective optimization technique called Peephole optimization. In the early stage of the implementation of this technique basically applied using string pattern matching performed on the regular expression and which are known as classical peephole optimizers. In the classical optimizers, the string pattern matching approach is considered as a processing of the input of the syntax of assembly code but its meaning gets lost. This paper explores prior research and current research issues in term of 'Optimizing' compilers with the implementation of Peephole optimization using different pattern matching approach that focuses on regular expression. In several research discussed below targets the implementation of pattern matching approach based on rule application strategy to achieve optimization of assembly code syntax for specific machine or retargetable [machine independent] in context of structured programming, procedure oriented programming and Object Oriented Programming.*

Keywords - *Intermediate Code, Optimization Rules, Pattern Matching Techniques, Peephole Size, Retargetable Peephole Optimizer*

I. Introduction

In compiler optimization theory, the compiler optimization basically refers to the program optimization to achieve performance in the execution. Program optimization refers to the three aspects (i) frontend: a programming language code, (ii) intermediate code: an assembly language code generated by the compiler appropriate to the programming language and (iii) backend: the specific machine or object code generated from the assembly language code for the actual execution by the compiler. The Peephole Optimization is a kind of optimization technique performed over a very small set of instructions in a segment of generated assembly code [Intermediate code]. The set of instructions is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster set of instructions to achieve speed or performance in the execution of the instruction sequences [1]. Basically Peephole Optimization is a method which consists of a local investigation of the generated object code means intermediate assembly code to identify and replace inefficient sequence of instructions to achieve optimality in targeted machine code in context of execution or response time, performance of the algorithm and memory or other resources usage by the program [2].

II. Common Techniques Applied in Peephole Optimization

Common techniques applied in peephole optimization [29].

- Constant folding - Assess constant sub expressions in advance.

  E.g. **r2 := 3 × 2** **becomes** **r2 := 6**

- Strength reduction - Faster Operations will be replaced with slower one.

  E.g. **r1:= r2 × 2** **becomes** **r1 := r2 + r2** **then** **r1 := r2<<1**
  r1 := r2/2 **becomes** **r1 := r2>>1**

- Null sequences – Operations that are ineffective will be removed.

  E.g. **r1 := r1 + 0** **or** **r1 := r1 × 1** **has no effect**

- Combine Operations - Replacement of the few operations with similar effective single operation.

  E.g. **r2 := r1 × 2**
  r3 := r2 × 1 **becomes** **r3 := r1 + r1**

- Algebraic Laws - Simplification and reordering of the instructions using algebraic laws.

E.g. **r1 := r2**
r3 := r1 **becomes** **r3 := r2;**

- Special Case Instructions - Use instructions designed for special operand cases.

  E.g. **r1 := r1 + 1** **becomes** **inc r1**

- Address Mode Operations - Simplification of the code using address modes.

  E.g. **r2 := var** **becomes** **r2 := 0x500**

III. Prior Research

3.1. Machine Specific Peephole Optimizers

William McKeeman has invented and peephole optimization technique in 1965 [2]. It is very simple but effective optimization technique. It was noted that when a program gets compiled, the code emitted from the code generators contained many redundancies around borders of basic blocks like chains of jump instructions. It becomes complicated case analysis to reduce these redundancies during the code generation phase. So it was appropriate to define a separate phase that would deal with them. The concept was described as follow:
A peephole, a small window which consisting no more than two assembly code instructions, is passed over the code. Whenever redundant instructions are found in the sequence, they are replaced by shorter or faster instruction sequences. For that the peephole optimizer uses the simple hand written pattern rules. These are first matched with the assembly instructions for the applicability of testing, and if the match found, the instructions are replaced. Therefore, a typical pattern rule consists of two parts a match part and a replacement part. The pattern set is usually small as this is sufficient for fast and efficient optimization.

These techniques have been implemented in GOGOL, a translator written by the author for the PDP-1 time sharing system at Stanford. The performance and limitation on how well the optimizer will work seems to depend primarily on how much time and space are available for recognizing redundant sequences of instructions [2].

Example:

Sample Code:

```
X: = Y;
Z: = Z + X;
```

Compiled Code:

```
LDA Y   Load the accumulator from Y
STA X   Store the accumulator in X
LDA X   Load the accumulator from X
ADD Z   Add the content of Z
STA Z   Store the accumulator in Z
```

In the above given simple code [2], if the store instruction is nondestructive, the third instruction is redundant and may be discarded. The action may be accomplished by having the subroutine which emits object code check every time it produces an LDA to see if the previous instruction was the corresponding STA and behave accordingly.

Lamb [3] has defined the implementation of a pattern-driven peephole optimizer and published in 1981. The Lamb's Peephole optimizers consists of three components, a collection of pattern rules, a translator that works as a parser, and a pattern-matcher framework, The code generator generates assembly code in the form of a doubly-linked list of nodes so that the peephole optimizer can easily operate on it. For the purpose of analyzing the code the optimizer uses a compiled version of the patterns, which is then simplified for testing applicability for set of instruction sequences wherever possible. The patterns are considered a generalization over specific cases; so, it contains abstract symbols or variables that are instantiated during pattern matching. The form of the optimization rules:

< Input pattern line 1 [condition (var)] >
……
< Input pattern line n >
=
< Replacement pattern line 1 >
……
< Replacement pattern line n >

In the above form pattern matching rule, condition (var) is optional (and so that written with square brackets); it can only occur in the matching portion of the pattern rule. In an optimization rule a condition over some variable(s) occurring in the pattern is referred as an Escape by Lamb [3] that must be met. If all the instructions in the preconditions part can be applied to adjacent assembly code instructions with any particular conditions over variables evaluate to true then and only then the pattern match is successful. In the replacement part of the pattern Escapes can also be embedded, where they represent an additional action to be performed rather than describing a condition over the variables as in the matching part. For example, it is difficult to represent such an action like negating a variable, as assembly code, whereas it is more preferable to invoke a subroutine that does the job. Lamb [3] performs matching using backwards strategy, means the optimization starting with last instructions in the precondition part of the matching rule up to first one to finish, not forwards as it was first originally approached. Most peephole optimizers adopted this strategy. Forwards refers to the order of the instructions which are dealt with the compilers and backwards refers to the testing part of the matching procedure. The advantage through this procedure we can get is that the further optimizations on the newly inserted instructions and their predecessors can instantly be performed. By interacting with the matching rules it is very useful for those, new opportunities for optimization arise after replacement. So that the general strategy that analyses the optimality of the current instruction against its predecessors is reasonable to utilize. As McKeeman [1] has originally adopted the other approach, until more improvement can be made, one would have to go through the code several times. Since the code has to be tested once only, the backward strategy is surely better and effective.

The above discussed style of matching optimization pattern rule and backward strategy are also used by Fraser's peephole optimizer Copt [4] through being a retargetable optimizer.

The work discussed in this paper adopts the above mention style of pattern matching rule for the regular expression and the generic backward strategy, which is very applicable to classical peephole optimizer implementation.

3.2. Machine Independent Peephole Optimizer

The classical peephole optimizers were machine dependent and so they could only specific to a single machine. They were directly worked assembly code. For the purpose to use with different machine they, they had to be modified and rewritten completely. Retargetable optimizers are different then the classical ones in their capability of being machine independent and therefore easily retargeted to other machines. Retargetable optimizers are slower in performance than their counterparts.

Jack W. Davidson and Christopher W. Fraser developed the first retargetable peephole optimizer, PO [5, 6] and published in 1979 (Fraser C. W) and 1980 (Davidson J. W.). Before the code generation, this PO is brought into action, which allows a machine independent analysis of the code. PO generally takes two inputs, one is assembly code of the program and second the target machine's descriptions on which the program suppose to be executed proceeding as follows. First, it finds out the effects of each assembly code instruction that can be generated for the specified machine, which are represented as register transfer patterns. This information forms a bi-directional translation grammar between the assembly code instruction and the register transfers. Then the each pair of adjacent instructions in the assembly code is analyzed and converts them to equivalent instantiated register transfer patterns by means of the bi-directional grammar from the previous analysis. Afterwards, these patterns are simplified by PO. Finally, it finds the best possible single assembly code instruction that corresponds to the simplified pattern, by which the original instruction in the input assembly program is then replaced.

As this mechanism defined a machine-independent approach to peephole optimizers that is retargetable to other machines was simply achieved by invoking PO on a different set of machine descriptions. Operating on register transfers rather than performing this technique on assembly code also ensured that all possible optimizations were found without exhaustive case analysis. In the 1980's, many research works have been done and published in connection with Retargetable peephole optimization and PO [5, 6, 7, 8, 9, 10]. The GNU C compiler GCC is implemented with the peephole optimizer which is heavily inspired by PO.

Ralph E. Jonson and Carl McConnell have defined the RTL System using the technique of register transfers as an intermediate code for compilers [11]. Their work provides an object-oriented framework for optimizing compilers to implement peephole optimizers.

A different approach of using peephole optimization on intermediate code was introduced by Tanenbaum et al. [12] in 1982. They attempted by applying peephole optimization to intermediate code instead of to object code: where they supposed to have a peephole optimization routine that was independent of the different compiler front and back ends involved. The optimization is performed on the intermediate code to maintain the portability of the compilers and it is performed with a set of common hand-written pattern matching rules. Their design does not make use of any specific or refined strategies. The results described in their work suggest their system to be faster than Davidson and Fraser's retargetable approach. In their work,

Tanenbaum et al. certainly gets the advantage of having optimizations at the intermediate code level in the case like, flags and register allocation issues do not need to be considered at that stage yet. But some optimization do not possible at the intermediate code level until code generation performed so this becomes the disadvantage of intermediate code level optimization.

Davidson and Fraser built a fast peephole optimizer that works at compile time using PO [13, 14] in 1984. Their work focused on automating the development of retargetable optimizers, i.e. PO is applied at compile time and a 'training set' of patterns is obtained, to achieve automatically finding patterns for optimization rules. As discussed above, Lamb's optimization rules that approximately similar to patterns to implement HOP. Without using string pattern matching or tree manipulation, HOP efficiently utilizes hashing to perform matching and replacement [15]. This is possible here because a fixed format for the specification of patterns is used by optimization rules. Thus, separating the framework of an instruction from its context-sensitive parts means operands is clear-cut. In order to perform a match of a sequence of input instructions with pattern rules, the input instructions and pattern rules are stored in separate hash table; the addresses of the skeleton patterns are compared. If the match is successful, the context sensitive information of the input is analyzed for consistency with the optimization rule. Using this way, HOP stay away from dealing with strings and enhances matching speed to that of byte-to-byte comparison.

Robert R. Kessler has introduced an architectural description driven peephole optimizer known as Peep and published in 1984 that is being adapted for the use in the Portable Standard Lisp compiler. In which tables of optimizable instructions are generated prior at the stage of compilation from the architecture description of the target machine. Peep then performs global flow analysis of the target machine code and optimizes instructions as defined in the global flow analysis allows optimization across basic blocks of instructions, and the use of tables created at compiler generation time minimizes the overhead of discovering optimizable instructions [16]. Kessler P. B. has also developed a compiler construction tool [17] for discovering machine specific code improvement and published in 1986 which automates much of the case analysis required to develop special purpose instruction on a target machine, such analysis identifies suitable instruction sequences that are equivalent to single instruction. During code generation phase such equivalence instruction set can be used to avoid inefficiency. This approach for instance does not require a training set. It simply utilizes target machine descriptions in order to find optimization opportunities.

Warfield et al. [18] developed an expert system and published in 1988 that learns optimization rules and by this way it simplifies the implementation of a retargetable peephole optimizer that can be implemented with different machines. The research of an optimizer tool described by Whitfield and Soffa in 1991 that is basically using two tools, an optimization specification language, and an optimizer generator, which automatically generate global optimizers [19]. It was designed for both traditional and parallelizing optimizations, which require global dependence conditions. In their research and experiments they found that the cost benefit ratio of some optimizations is quite large and it can be reduced in some cases by specifying optimization carefully by different implementations.

Davidson and Fraser has enhanced the functionalities of the original version of PO in 1984 added with the CSE and divided the new version of the PO in three phases (i) Cacher, (ii) Combiner and (iii) Assigner where Cacher deals with Common Sub expression Eliminations (CSEs), the Combiner simplifies register transfers and the Assigner transfers these to assembly code [7, 8].

Ganapathi et al. [20], has given a totally different approach of peephole optimization in compiler construction. They rather implements tree matching pattern using tree manipulation language called twig. For describing target machine instructions they utilize attribute grammar parsing techniques Instead of using register transfers. String pattern matching typically applied here but moreover that context free production consists of a set of attribute evaluations which forms the Pattern rules. The pattern rules having a preconditions and a replacement part as we discussed in the earlier work of the peephole optimizers. These rules improve the code while it is being parsed into a tree-like format.

3.3. Combining Optimization with Code Generation

From review of the above research works we can understand that the retargetablity and platform independency of the peephole optimizers in compiler construction had been successfully achieved in early 1980's but speed is always remained an issue. In order to overcome this, the overall execution time for the code generation and optimization phases (implementation of pattern matching strategy) had to be reduced. As mentioned at the beginning of this section, allowing for optimization in code generation itself would have resulted in complicated case analysis. So the goal here was to maintain the efficiency of both phases without focusing the individual algorithms too much.

The concept of combining two phases of code generation and optimization in to one phase has been described by Fraser and Wendt in 1986 [15]. Using this concept they extended the initial implementation of HOP. They described a 'general rule-based rewriting system' which is basically a new version of HOP that

usually performing by pattern matching through hashing. Lamb's concept of escapes has also been added to HOP replacement pattern rules. Fraser and Wendt introduced a 'recycling' system in which system utilizes and saves the time be achieving efficient code generation in one phase rather that in separate phase.
A similar rule based rewriting system is introduced by Ancona [10] in 1995. A simple approach to the integration of peephole optimization with retargetable code generation is presented. The method is based on an intermediate code optimizer that is programmable, i.e., driven by user-defined optimization and translation rules. Optimization and translation rules, specified in form of macros, are written in a simple high-level language. Three algorithms, intermediate code optimization, target code generation and target code optimization are unified in a single process, a rule based, pattern matching and replacement algorithm. Three programs implement the method. The first two are used during the development and test phase of the optimization and translation rules. The third program generates, from a tested set of rules, an efficient code optimizer and generator, specialized for a specific target machine, and to be included in a production compiler.

Other research works [7, 9] presents that the peephole optimization can greatly improve code given by a naïve code generation. Fraser and Hanson's described the above mentioned idea of combining two phases, code generation and optimization in lcc [21, 22] in 1989 and 1991. Their retargetable C compiler lcc does not describe optimizing compiler however optimization implemented or rather then 'hard-coded' in to code generation rules wherever required.

In another lcc-based project called lcc-win32 introduced by Jacob Navia in 1995 – 2005 [23], which is only targets the windows platform only. Lcc-win32 introduced with a separate and additional peephole optimizer which is not included in the earlier version of lcc.

There are approximately 20 optimization rules using which lcc-win32's generated code is improved and lcc-win32 is utilized as a tool only in the peephole optimizer. Classical approach of optimization is adopted by the lcc-win32 rather than the modern approach of retargetable optimizers. This method was selected as is more suitable and efficient to the objective of this work. Lcc-win32 just provides a skeleton that this system can be used and tested with. The optimizer is not intended to replace Navia's optimizer in any way as the optimizer's rule set is partial in particular.
The pattern matching strategies and its surrounding information to the techniques that are in concern with the framework of peephole optimization is described in the next segment.

3.4. Optimization Rules (Pattern) Matching Strategies

Much work has been done in the field of pattern matching is done because it is having applicability to many areas of computer science. The Knuth-Maris-Pratt (KMP) and the Boyer-Moore (BM) algorithms are classical in context of string pattern matching. Usually, string pattern matching is associated with finding string pattern in text. Both text and string pattern containing a specific sequence of characters and optionally variables. The goal of using variables here is to resolve, i.e. associate any variables software that the text portion and pattern become equal.

Patterns are represented as regular expressions classically, which provide a format for expressing the sequence of characters to look for. This format allows abstraction over simple characters. If the string represents a valid occurrence of the pattern then a pattern successfully matches an input string and if at any point during matching, the input string does not satisfy the requirement of pattern then the pattern fails to match a string.
The earlier pattern matching strategies were basically regular expression base. The grep command of Linux is a pattern matcher implementing the same strategy for text files and Perl programming language also supports the same regular expression based pattern matching.

Spinellis D. [24] designed a peephole optimizer written in Perl programming language in Feb-1999, which performs optimization using a declarative specification of optimization and optimization rules applied regular string pattern matching, targeting specifically at branch prediction. This approach to pattern matching allows a simple specification and fast processing of the rules. Capability of grouping regular expressions and back referencing them, simplifying the reuse of parts of match provides an advantage to pattern matching. This mechanism saves the programmer from storing matches input in temporary variables which often results in unnecessarily complicated code. This approach is also used in the regular expression based strategy.
This string-based pattern matching approach can be considered very effective, but the problem is that it just processes the syntax of the input of the assembly code. It is not observed by the optimizer whether a register, a constant, or a label definition is parsed or analyzed. There is no semantic conception of the input at all. But pattern matching should not be considered as just mapping of strings. For the purpose of matching a pattern of any kind, it has to be identified in the input. Therefore, the input, which represents some form of information, has to be processed completely. From the above discussion we can say that a pattern matching is an information processing problem, in which variables have to be bound and constriction have to be satisfied. In-built facilities for pattern matching are often provided by procedural or functional programming languages with implicit use of this approach in procedures or rules. The implementation of above mentioned idea in object-oriented languages

like Java or C++ is different, because in object-oriented languages, an object represents a smallest unit of information, so everything is put together around objects. Objects are characterized by their data members and operations or functions to be performed on them. So when programmer defines an object, we can say he defines the meaning or concept of object. And so that for applying pattern matching with strings and regular expressions as mentioned above, the programmer has to work on lower level of abstraction.

The absence of pattern-matching in object-oriented programming languages is felt especially when tackling source code processing problems. Visser J. described an approach to support pattern matching in mainstream object-oriented languages without language extension in 2006 [25]. In this approach a pattern is created and be passed as an argument in a form of a first class entity which provides methods invocation just like other objects. This approach is implemented by performing matching of the objects in object graph. The pattern is an object graph that contains variable objects, indicated by empty dashed circles. After matching, these variables are bound to corresponding objects in the term, indicated by the dotted arrows.

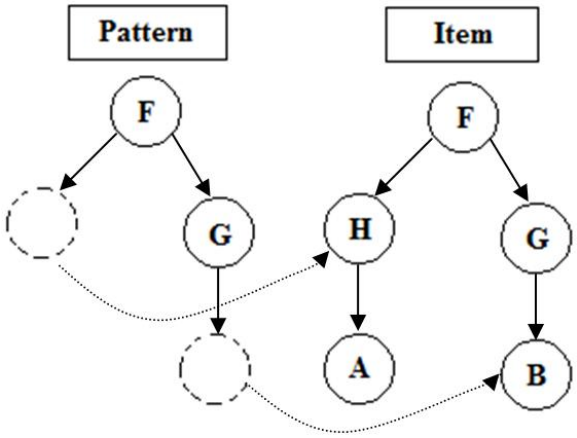

Figure-1 Pattern matching for objects in object graph [25]

JMatch [26] is another attempt to introduce pattern matching in Java, though by language extension. Mostly, in the case of peephole optimization, one not only wishes to identify patterns, but to replace them as well. So pattern matching is also a rewriting problem.

Visser E. et al [27] described and published Strategic pattern matching in 1999, and also introduces Stratego in 2000-2005 [28], which a term rewriting language that provides elasticity in strategy application by separating rule definitions from strategy specifications. Hence a suitable rewriting strategy can be selected for applications with a base set of rules according to the input. Peephole optimizers have always implicitly supported this design: Usually there is a separate rules file and a peephole optimizer skeleton in classical peephole optimizers. But as for strategically aspects of rule application, only the backwards strategy has been applied so far.

IV. Research Issues

- As per the earlier research work, Size of peephole is only capable to cover a smaller set of instructions of the intermediate code for the optimization investigation and so the issue is that, Can Peephole size is increased that can covers bigger set of intermediate code instructions generated from the longer instruction sequences of Higher Level Language? And how it is possible?
- As per the study of different Peephole Optimizer, it is also important to choose an efficient pattern matching strategy for the replacement part. Earlier Peephole Optimizers have commonly implemented string based pattern matching approach and latter on other pattern matching strategies have been identified like tree manipulation or object based pattern matching. A question arises from the above study is that, Can other pattern matching strategy be implemented rather than string based for replacement part?
- Peephole Optimizers were generally focusing on pattern matching strategies and replacement rules to map and replace redundancy from the code and so that the meaning of the code gets lost. So the issue suggests Can PO be achieved even more successfully by utilizing the meaning of the code? And how?
- PO is an old technique from the 1980s, and from the observation of above study, we found that the concept of Peephole Optimization has already applied with structured or procedure oriented programming approaches to optimize object code but a question arises is that can it be combined with newer concepts & approaches like OOP with more efficiency for optimizing byte code?
- Optimization Overhead is a major issue to the work of optimizing compiler to be concerned as the number of scan performed over intermediate code until no more improvement found.

V. Conclusion

This paper describes the foundation of the Peephole Optimization technique and its implementation for the construction and design of optimizing compilers along with the optimization rules that suppose to be matched to investigate and replace redundant instructions of intermediate code. Different pattern matching approaches like string based, tree manipulation and object based pattern matching of pattern rules have also been discussed.

As the main section of this paper (Section 3) is divided in to four parts in which first section focuses on the machine dependent peephole optimizers which were capable to work with object code instruction and to the specific machine and pattern rules are hard coded. The second part describes the foundation of retargetable peephole optimizers which were generally machine independent and that are also capable to work upon register transfers rather than assembly code to avoid exhaustive case analysis. It has also describes the pattern matching techniques using hashing mechanism. In this section few researches has also covered the concept of automatic generators of optimization rules means pattern rules can be automated and machine driven. The third part describes the integration of code generation and optimization in to one phase to achieve speed and efficiency for optimization. And the last part describes different pattern matching strategies implemented over different peephole optimizers.

As per the study of all the section of this paper we can also understand the current state and research issues regarding the peephole optimization in construction and design of optimizing compilers.

References

[1] Aho, A. V., Sethi, R., Ullman, J. D. Compilers: Principles, Techniques, and Tools. Massachusetts: Addison-Wesley,
[2] McKeeman, W. M. (1965) Peephole optimization. CACM 8(7):443-444.
[3] Lamb, D. A. (1981) Construction of a Peephole Optimizer. Software - Practice & Experience.
[4] Fraser, C. W. (1982) Copt, a simple, retargetable peephole optimizer. Software, available at ftp://ftp.cs.princeton.edu/pub/lcc/contrib/copt.shar.
[5] Davidson, J. W., Fraser, C. W. (1980) the design and application of a retargetable peephole optimizer. ACM TOPLAS 2(2):191-202.
[6] Fraser, C. W. (1979) A compact, machine-independent peephole optimizer. POPL'79:1-6.
[7] Davidson, J. W., Fraser, C. W. (1984) Code selection through object code optimization. ACM TOPLAS 6(4):505-526.
[8] Davidson, J. W., Fraser, C. W. (1984) Register allocation and exhaustive peephole optimization. Software - Practice & Experience 14(9):857-865.
[9] Davidson, J. W., Whalley, D. B. (1989) Quick compilers using peephole optimizations. Software - Practice & Experience 19(1):195-203.
[10] Ancona, M. (1995) an optimizing retargetable code generator. Information and Software Technology 37(2):87–101.
[11] Johnson, R. E., Mc Connell, C., Lake, J. M. (1991) the RTL System: A Framework for Code Optimization. In Code Generation – Concepts, Tools, Techniques, Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, pp. 255–274.
[12] Tanenbaum, A. S., van Staveren, H., Stevenson J. W. (1982) Using Peephole Optimization on Intermediate Code. ACM TOPLAS 4(1):21-36.
[13] Davidson, J. W., Fraser, C. W. (1987) Automatic Inference and Fast Interpretation of Peephole Optimization Rules. Software - Practice & Experience 17(11):801-812.
[14] Davidson, J. W., Fraser, C. W. (1984) Automatic generation of peephole optimizations. CC84:111-116.
[15] Fraser, C. W., Wendt, A. L. (1986) Integrating Code Generation and Optimization. Proceedings of the SIGPLAN'86 symposium on Compiler Construction, pp. 242–248.
[16] Kessler, R. R. (1984) Peep – An architectural description driven peephole optimizer. CC84:106-110.
[17] Kessler, P. B. (1986) Discovering machine-specific code improvements. CC86:249-254.
[18] Warfield, J. W., Bauer, III, H. R. (1988) an Expert System for a Retargetable Peephole Optimizer. ACM SIGPLAN Notices 23(10):123–130.
[19] Whitfield, D., Soffa, M. L. (1991) Automatic Generation of Global Optimizers. Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation.
[20] Aho, A. V., Ganapathi, M., Tjiang, S. W. K. (1989) Code Generation Using Tree Matching and Dynamic Programming. ACM TOPLAS 11(4):491-516.
[21] Fraser, C. W. (1989) A Language for Writing Code Generators. Proceedings of the SIGPLAN '89 symposium on Compiler Construction, SIGPLAN Notices 24(7):238–245.
[22] Fraser, C. W., Hanson, D. R. (1991) a Retargetable Compiler for ANSI C. SIGPLAN Notices 26(10):29–43.
[23] Navia, J. (1999-2005) lcc-win32: A Compiler system for Windows. Available at http://www.cs.virginia.edu/~lcc-win32/
[24] Spinellis, D. (1999) Declarative Peephole Optimization Using String Pattern Matching. ACM SIGPLAN Notices 34(2):47–51.
[25] Visser J. (2006) Matching Objects Without Language Extension, in Journal of Object Technology, vol. 5, no. 8, November-December 2006, pages 81–100, http://www.jot.fm/issues/issue 2006 11/article2.pdf.
[26] Liu, J., Myers, A. C. (2003) JMatch: Iterable abstract pattern matching for Java. Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages.
[27] Visser, E. (1999) Strategic Pattern Matching. RTA'99, Vol. 1631 of Lecture Notes in Computer Science, pp. 30–44.
[28] Visser, E. et al. (2000-2005) Stratego: Strategies for Program Transformation. http://www.stratego-language.org/
[29] Fischer, C.N. and LeBlanc, R.J. (1988) Crafting a Compiler, Benjamin Cummings, Menlo Park, CA