Fifty years of peephole optimization

Pinaki Chakraborty*

Division of Computer Engineering, Netaji Subhas Institute of Technology, New Delhi 110 078, India

In 1965, William M. McKeeman introduced the concept of peephole optimization. This article provides a brief review of the major peephole optimization techniques identified so far and the methodologies used to implement them. Topics for further research on peephole optimization have been also identified.

Keywords: Code generators, compilers, instruction sequences, peephole optimization.

McKeeman[1] presented a code optimization technique 'that consists of a local inspection of the object code to identify and modify inefficient sequences of instructions'. He also suggested a name that very well describes this technique – *peephole optimization*.

Code generators in most compilers operate locally producing at the best locally optimal object code fragments which may not be efficient when juxtaposed[2]. The situation can be improved by peephole optimization. This is typically applied on the object code late in the compilation process. A window or *peephole* is slid over the object program replacing instruction sequences within the peephole by equivalent but shorter and/or faster instruction sequences. The peephole typically consists of 2–3 contiguous instructions. Peephole optimization can be used for almost all types of programing languages and machine architectures. It takes care of machine-independent as well as machine-specific issues. Peephole optimization is simple, efficient and easy to implement. Unlike most other code optimization techniques, peephole optimization does not make the compiler much heavier or slower. Moreover, the logic behind it can be appreciated by all computer scientists. Consequently, peephole optimization is covered in most textbooks on compiler construction[3,4], taught to millions of students in universities around the world, has been used in many research-purpose and production-quality compilers, and serves as an important topic for research even 50 years after its invention.

Common peephole optimization techniques

McKeeman[1] originally discussed only a few optimization techniques. Later researchers like Bagwell Jr[5], Wulf *et al.*[6], and Tanenbaum *et al.*[7] identified many more such peephole optimization techniques (Table 1). These can be grouped as follows.

Elimination of redundant instructions

Code generators often produce redundant instructions which can be deleted without any side effect. There may be load and store instructions that act on the same data unit even before its value has been modified. Such redundant load and store instructions may be deleted. An unlabelled instruction sequence following an unconditional jump instruction is unreachable and may be deleted too. A test or a compare instruction not followed by a conditional branch instruction may also be deleted. An inconsequential instruction sequence, such as addition of zero or two consecutive negations, may be deleted too.

Improvements in flow of control

An unconditional or a conditional jump instruction produced by a code generator may have another jump instruction as its target. The flow of control of the program can be improved by suitably changing the target address of such a jump instruction, and in some cases the second jump instruction may be even deleted. Moreover, if a conditional jump instruction is preceded by a compare or a negation instruction, then the two instructions may be replaced by a new conditional jump instruction that preserves the logic.

Algebraic simplifications

If a program uses an expression consisting of two or more constants and no variables, then that expression may be evaluated at the compile time only. Some arithmetic instructions can be replaced by much simpler instructions on the occurrence of some specific operands. For example, x^2 can be implemented as $x * x$ and if a square root instruction is available, then $x^{0.5}$ may be implemented as sqrt(x). Similarly, multiplication by a power of 2 may be implemented as left-shift, division by a power of 2 may be implemented as right-shift, addition of 1 may be implemented as increment, subtraction of 1 may be implemented as decrement, and multiplication or division by –1 may be implemented as negation. Logic expressions may be simplified at the compile time using the concepts of Boolean algebra like the De Morgan's laws. Additionally,

*e-mail: pinaki_chakraborty_163@yahoo.com

Table 1. Common peephole optimization techniques

Optimization	McKeeman[1]	Bagwell Jr[5]	Wulf *et al.*[6]	Tanenbaum *et al.*[7]
Elimination of redundant instructions				
Removal of redundant load and store instructions	✓		✓	
Removal of unreachable instructions			✓	
Removal of useless test and compare instructions			✓	
Removal of inconsequential instruction sequences (null sequences)	✓	✓	✓	✓
Improvements in flow of control				
Elimination and coalescing of jump instructions			✓	
Modification of comparisons				✓
Algebraic simplifications				
Evaluation of constant expressions (constant folding)	✓	✓		✓
Modifying instructions to simpler forms (operator strength reduction)		✓	✓	✓
Simplification of logic expressions		✓		
Reordering arithmetic instructions	✓			✓
Use of machine idioms				
Addressing optimizations			✓	✓
Using special instructions	✓			✓

arithmetic instructions may be reordered taking advantage of the commutative and associative properties of the operations to facilitate further algebraic simplifications.

Use of machine idioms

To obtain an efficient object program, the machine-specific features of the target machine need to be exploited at some stage. A typical target machine supports several addressing modes. Choosing an appropriate addressing mode can help in optimizing data transfer instructions, and arithmetic and logical instructions. Target machines often provide some kind of special instructions. For example, a target machine may provide an instruction to duplicate the data unit at the top of the stack. Such special instructions should be used to obtain an efficient object program.

Hand-coding of peephole optimization

Peephole optimization was initially supposed to be hand-coded into compilers. A compiler developer identifies inefficient instruction sequences and also suggests equivalent instruction sequences that should replace them. These replacement rules are hand-coded to form the peephole optimizer of a compiler. Several compilers have been implemented using this methodology.

The first of these compilers was developed by McKeeman[1] himself (Box 1). He used a hand-coded peephole optimizer in the compiler of a simple procedural language called Gogol[8]. The compiler ran on, and produced object code for, PDP-1 machines. He found that peephole optimization led to considerable improvement in the object code.

Wulf *et al.*[6] used peephole optimization in the FINAL phase of the Bliss/11 compiler. They presented several new peephole optimization techniques and improved the existing ones. Additionally, they presented a data structure that can be used to efficiently implement peephole optimization (Figure 1). The object program is stored in a doubly linked list. Each node in this list is either a code cell or a label cell. A code cell stores an instruction along with its operands, while a label cell marks a position in the object program. A label cell also has a sublist of nodes called reference cells. If a code cell has an instruction that makes a jump to a given label, then the label cell corresponding to that label has a reference cell that points back to that code cell. This data structure allows reading the object program in both forward and backward directions, easy insertion and deletion of instructions, and other operations that help in implementing peephole optimization. This data structure and its variants have been used by several later researchers[9,10].

Although peephole optimization was actually proposed for object code, some researchers have employed it on intermediate code and obtained encouraging results. Tanenbaum *et al.*[7] used peephole optimization on a stack machine-based intermediate language. They identified more than a 100 replacement rules. This list of replacement rules remains till date the largest repository of its type and is often used as a benchmark. Tanenbaum *et al.*[7] also found that using peephole optimization produces object programs that on an average have 16% fewer instructions and are 14% smaller in size. This peephole optimizer was used as the third of the seven phases in the Amsterdam Compiler Kit[11]. Interestingly, this peephole optimizer was followed by a machine-independent global optimizer as the fourth phase and a machine-specific target optimizer as the sixth phase in the Amsterdam

Box 1. Story behind the invention of peephole optimization.

In the summer of 1964, William M. McKeeman, then a graduate student at the Stanford University, USA, went to work with John McCarthy. McCarthy was at the time working on time-sharing using a PDP-1 computer with 4K 18-bit words of memory. McCarthy suggested McKeeman to use an available arithmetic simplification program to calculate the Einstein-Ricci-Christoffel tensor. However, that did not work out because the simplification program could not handle the complexity of the problem. So, McKeeman decided to write a fast compiler for PDP-1 instead.

McKeeman designed a simple procedural language focusing on arithmetic. The language was first named 'go' denoting that it is fast, but the name was later changed to Gogol in the memory of Nikolai Gogol. When McKeeman started implementing the Gogol compiler, the small memory of PDP-1 motivated him to invent peephole optimization. Dense object code meant bigger problems could be solved. The compiler itself had to be small. In fact, the compiler was limited to about 2K 18-bit words because it needed the other 2K words to place the object code it produced.

When McKeeman showed the Gogol compiler to McCarthy he said 'Well, you surely wasted your summer'. Some of McKeeman's contemporaries at Stanford at that time thought that it was too simple an idea to merit publication. Nevertheless, McKeeman's paper on peephole optimization was published in 1965 and the concept was soon adopted by many compiler developers.

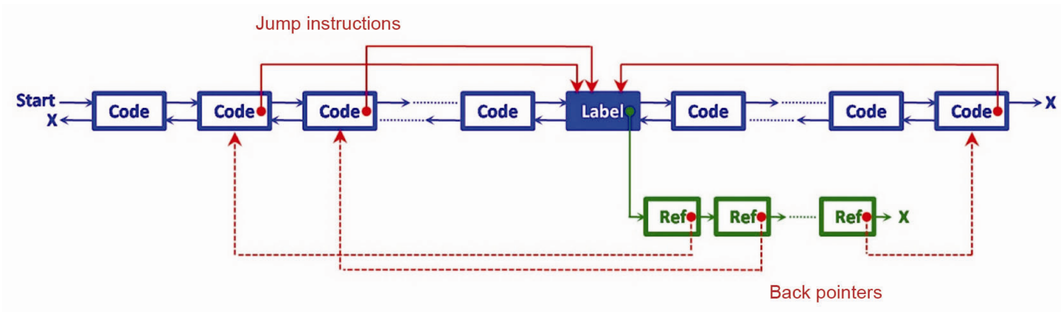

Figure 1. Data structure used by Wulf *et al.*[6] to implement peephole optimization.

Compiler Kit. Later, McKenzie[12] improved this peephole optimizer using a buffering technique. Chakraborty[13] also used peephole optimization on intermediate code in the iXC85 cross compiler that has a simple C-like language as its source language and produces object programs for Intel 8085 machines.

Automatic inference of replacement rules

Peephole optimization can be also implemented using a tool to automatically infer replacement rules from a symbolic description of the target machine. The replacement rules may be inferred either at the compile–compile time or at the compile time. There are two major advantages of this method. First, using a large number of automatically inferred replacement rules results in a much thorough optimization. Second, the compiler may be easily retargetted to another target machine by editing the machine description.

Davidson and Fraser[2] developed the first peephole optimizer using automatically inferred replacement rules and named it PO. PO takes as input a symbolic description of the target machine and an object program. It then checks each pair of adjacent instructions in the object program and, if possible, replaces them with an equivalent single instruction. PO is mostly machine-independent. It can be modified to replace three consecutive instructions by a single one. This makes PO slow but not more complex. PO has been also modified to replace some inefficient nonadjacent pairs of instructions[14]. An instruction that sets a data unit and the next instruction modifying that data unit may be sometimes replaced by a single instruction. PO can be also augmented with a compile–compile time training system to improve its speed[15].

The method of automatically inferring replacement rules has been followed by several later researchers. Lamb[9] used this method to implement the peephole optimizer of a compiler for a subset of Preliminary Ada. Kessler[16] used the method to implement a peephole optimizer, called Peep, for a Portable Standard Lisp Compiler. Peep can also reduce nonadjacent instructions. Later, Kessler *et al.*[17] used automatically inferred replacement rules in a peephole optimizer that serve as the last of the seven phases in the Experimental Portable Standard Lisp Compiler. In this compiler, two sets of replacement rules are inferred at the compile–compile time. The first set of replacement rules takes into account the flag register of the target machine, while the second set ignores it. The second set is typically much larger. At the compile time, flow analysis performed determines if the flag register is significant for a particular instance of an instruction.

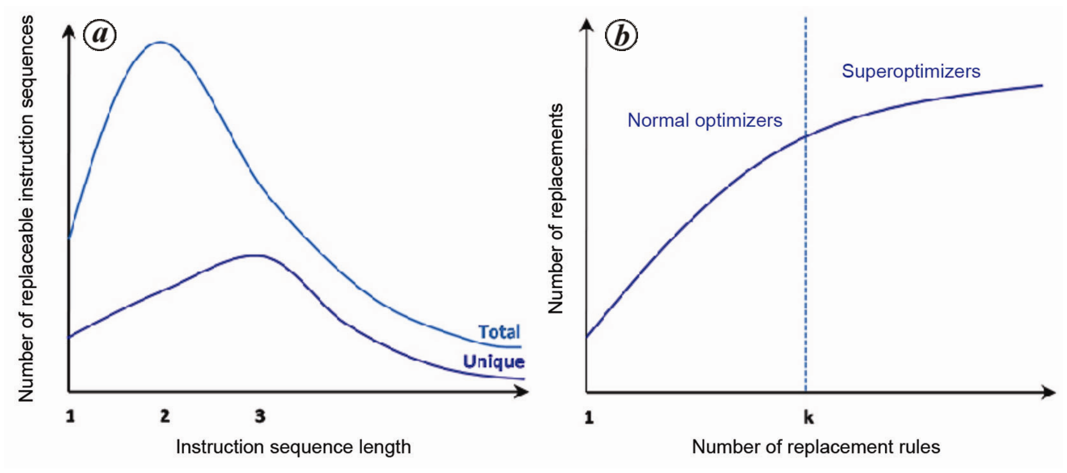

Figure 2. Performance of peephole optimization at (***a***) compile–compile time and (***b***) compile time.

Since the flag register can be ignored in most cases, the second set of replacement rules is used more often resulting in better optimization. Warfield and Bauer III[18] implemented a peephole optimizer as an expert system. This expert system identifies inefficient pairs of instructions, replaces them with equivalent single instructions, infers new replacement rules and saves them for future use. Several researchers[10,19,20] used automatically inferred replacement rules to implement peephole optimizers for C compilers, while Lambright[21] used automatically inferred replacement rules to implement a peephole optimizer for Java bytecode. Several of these researchers observed that a small number of replacement rules is sufficient to get satisfactorily efficient object programs. In fact, Lamb[9], Davidson and Whalley[19], and Lambright[21] used only 53, 39 and 25 replacement rules respectively.

Bansal and Aiken went a step forward and developed a peephole superoptimizer[22,23]. Unlike normal optimizers, superoptimizers use brute force optimization using thousands of replacement rules. The optimizer uses training programs to infer the replacement rules and stores them in a database. The superoptimizer has been successfully used on C programs. Superoptimizers produce object programs of high quality, but are slower than normal optimizers and hence should be used to compile production-quality software only.

Performance issues

The effectiveness of peephole optimization depends on several factors like the nature of the source language, the parsing and code generation techniques used in the compiler, and the specifications of the target machine. The performance of peephole optimization can be measured at compile–compile time, compile time and runtime. At the compile–compile time, a compiler developer identifies the replacement rules or they are automatically inferred from a description of the target machine. However, many of these replacement rules are variants of each other just using different registers. The number of unique replacement rules is generally much less than the total number of replacement rules. The replaceable instruction sequences typically consist of 1 to 5 instructions. In most compilers, maximum of the replaceable instruction sequences have two instructions, while maximum of the unique replaceable instruction sequences have three instructions (Figure 2 *a*). At the compile time, the number of replacements initially rises with an increase in the number of replacement rules. However, the number of replacements becomes saturated after a threshold is reached (Figure 2 *b*). It is wise to use more replacement rules than this threshold only in a superoptimizer. Several researchers have measured the performance of peephole optimization at the runtime. It has been observed that peephole optimization generally leads to 10–15% decrease in the size and execution time of the object programs.

Concluding remarks

Peephole optimization will certainly continue to be used in compilers both in hand-coded form and with automatically inferred replacement rules. Additionally, new topics related to peephole optimization as listed below may be also investigated.

1. The concept of peephole optimization can be extended from general purpose programing languages to domain-specific languages. In fact, peephole optimization has been successfully used for hardware description languages[24] and database languages[25]. In future, peephole optimization may be applied to other domains as well.
2. Object-oriented programing is an effective technique for implementing large and complex pieces of software. This technique can be used to efficiently implement compilers as well. It will be interesting to see

how object oriented programing can be used to realize peephole optimization.
3. Peephole optimization can be applied in just-in-time compilers too. A suitably designed peephole optimizer can help a just-in-time compiler to produce efficient executable codes on the fly.
4. Peephole optimization can also be used in compilers for parallel computers. Specialized replacement rules for parallel architectures may be inferred.
5. With the advent of the energy-efficient paradigm of computing, compilers are supposed to produce energy-efficient object programs. Peephole optimization may be used to replace more energy consuming instruction sequences by equivalent but less energy-consuming instruction sequences.

1. McKeeman, W. M., Peephole optimization. *Commun. ACM*, 1965, **8**, 443–444.
2. Davidson, J. W. and Fraser, C. W., The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 1980, **2**, 191–202.
3. Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2007, 2nd edn.
4. Fischer, C. N., Cytron, R. K. and LeBlanc Jr, R., *Crafting a Compiler*, Addison-Wesley, 2010.
5. Bagwell Jr, J. T., Local optimizations. *ACM SIGPLAN Not.*, 1970, **5**, 52–66.
6. Wulf, W., Johnson, R. K., Weinstock, C. B., Hobbs, S. O. and Geschke, C. M., *The Design of an Optimizing Compiler*, Elsevier, 1975.
7. Tanenbaum, A. S., van Staveren, H. and Stevenson, J. W., Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, 1982, **4**, 21–36.
8. McKeeman, W. M. and Wirth, N., Gogol, Technical report, Stanford time-sharing project, Memo no. 24, Stanford University, USA, 1964.
9. Lamb, D. A., Construction of a peephole optimizer. *Software: Pract. Exp.*, 1981, **11**, 639–647.
10. Zhang, H. G. and Lan, X.-Z., Design and implementation of peephole optimization based on extensible template. In Proceedings of the First International Workshop on Education Technology and Computer Science, 2009, vol. 2, pp. 95–98.
11. Tanenbaum, A. S., van Staveren, H., Keizer, E. G. and Stevenson, J. W., A practical tool kit for making portable compilers. *Commun. ACM*, 1983, **26**, 654–660.
12. McKenzie, B. J., Fast peephole optimization techniques. *Software: Pract. Exp.*, 1989, **19**, 1151–1162.
13. Chakraborty, P., Design and implementation of a cross compiler. *J. Multidiscip. Eng. Technol.*, 2009, **3**, 6–15.
14. Davidson, J. W. and Fraser, C. W., Register allocation and exhaustive peephole optimization. *Software: Pract. Exp.*, 1984, **14**, 857–865.
15. Davidson, J. W. and Fraser, C. W., Automatic inference and fast interpretation of peephole optimization rules. *Software: Pract. Exp.*, 1987, **17**, 801–812.
16. Kessler, R. R., Peep: an architectural description driven peephole optimizer. *ACM SIGPLAN Not.*, 1984, **19**, 106–110.
17. Kessler, R. R., Peterson, J. C., Carr, H., Duggan, G. P. and Knell, J., EPIC – a retargetable, highly optimizing Lisp compiler. *ACM SIGPLAN Not.*, 1986, **21**, 118–130.
18. Warfield, J. W. and Bauer III, H. R., An expert system for a retargetable peephole optimizer. *ACM SIGPLAN Not.*, 1988, **23**, 123–130.
19. Davidson, J. W. and Whalley, D. B., Quick compilers using peephole optimization. *Software: Pract. Exp.*, 1989, **19**, 79–97.
20. Spinellis, D., Declarative peephole optimization using string pattern matching. *ACM SIGPLAN Not.*, 1999, **34**, 47–50.
21. Lambright, H. D., Java bytecode optimizations. In Proceedings of the Forty-second IEEE International Computer Conference, 1997, pp. 206–210.
22. Bansal, S. and Aiken, A., Automatic generation of peephole superoptimizers. *ACM SIGPLAN Not.*, 2006, **41**, 394–403.
23. Bansal, S., Peephole Superoptimization, PhD thesis, Stanford University, USA, 2008.
24. Keutzer, K. and Wolf, W., Anatomy of a hardware compiler. *ACM SIGPLAN Not.*, 1988, **23**, 95–104.
25. Derr, M. A., Morishita, S. and Phipps, G., The glue-nail deductive database system: design, implementation, and evaluation. *VLDB J.*, 1994, **3**, 123–160.

ACKNOWLEDGEMENTS. I thank Prof. W. M. McKeeman, Dartmouth College, USA for sharing information about the events that led to the invention of peephole optimization and for reading a draft of this manuscript.

Received 4 November 2014; accepted 22 March 2015