Declarative Peephole Optimization Using String Pattern Matching *†

Diomidis Spinellis
Department of Information & Communication Systems
University of the Aegean
GR-83200 Karlobasi, Greece
dspin@aegean.gr

Abstract

Peephole optimisation as a last step of a compilation sequence capitalises on significant opportunities for removing low level code slackness left over by the code generation process. We have designed a peephole optimiser that operates using a declarative specification of the optimisations to be performed. The optimiser's implementation is based on string pattern matching using regular expressions. We used this approach to prototype an optimiser to convert target machine instruction sequences containing conditional execution of instructions inside loop bodies into code that adaptively executes the optimum branch instructions according to the program's branch behaviour.

Keywords

Peephole optimization; branch prediction; regular expressions.

1 Introduction

Peephole optimisation [ASU85, p. 554–558] as a last step of a compilation sequence capitalises on significant opportunities for removing low level code slackness left over by the code generation process. In addition, as peephole optimisers work on low level (target) code they can perform transformations based on specific characteristics of the target architecture. The declarative specification and implementation of the peephole optimisation process can offer a number of advantages over the procedural coding of the same specification:

- the implementation cost of the optimiser is only that of the specification of the optimisations to be performed,
- the correctness of the resulting implementation only depends on the correctness of the initial specification, and
- reasoning, documentation, and maintenance of the optimiser can be performed at the specification level.

The emergence of versatile text processing languages and high quality regular expression libraries has allowed us to experiment with a peephole optimiser design based on regular expression string pattern matching. In the following sections we describe a prototype implementation of a flow-of-control optimiser based on the declarative specification of optimisation patterns and their dynamic compilation into regular expression replacement sequences. We have used this optimiser to implement a software-based dynamic jump prediction scheme.

The remainder of this paper is structured as follows: Section 2 introduces the optimiser's design; Section 3 describes our motivating example based on a software-based dynamic branch prediction scheme, while Section 4 details the optimiser's implementation. The complete listing of the optimiser prototype implementation is provided as an Appendix.

2 Optimiser Design

The optimiser's design is based on pattern matching [AG85] of the target code against declarative specifications as illustrated by the dataflow diagram in Figure 1. The optimiser initially converts the declarative — domain specific [SG97] — optimisation specification into a string regular expression that can be applied to an appropriate representation of the target code. As a trivial example Figure 2 contains a declarative optimisation specification for the elimination of jumps to jump instructions. The specification is written as a pattern to be recognised followed by the corresponding replacement code. The optimiser reads the target code, converts it into a string, storing relevant ancillary information into a lookup table, and then continuously applies the regular expression to the string representation of the target code performing the specified pattern-based substitutions. The repetitive nature of the regular expression application can result in higher order optimisations where the output code of one optimisation step is further optimised by the next application. Finally, the string representing the optimised target code is converted back into its normal representation based on the lookup table.

ACM SIGPLAN Notices, 34(2):47-51, February 1999.

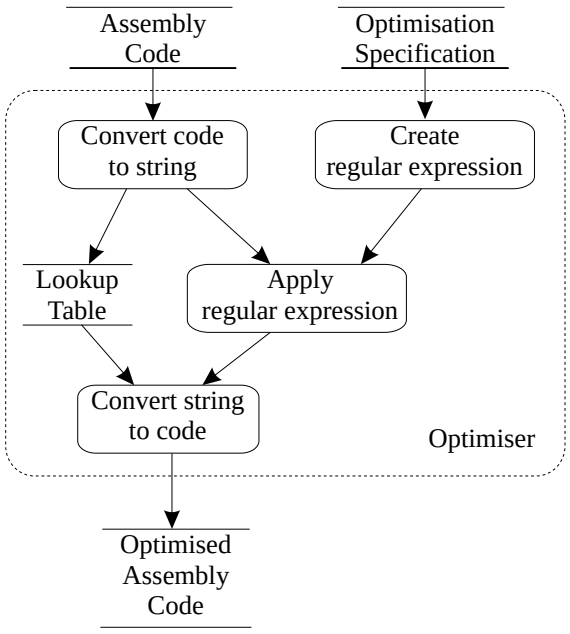

Figure 1: The peephole optimiser dataflow diagram.

```
# Input pattern
        jmpL1
        S1*
L1:     jmpL2
=>
# Output specification
        jmpL2
        S1*
L1:     jmpL2
```

Figure 2: Simple declarative specification for eliminating jumps to jump instructions.

3 Optimising Dynamic Branch Prediction

We tested the design of the optimiser using as a motivating example the optimisation of branch instructions. Modern processors make extensive use of overlapped instruction execution — *pipelining* — to increase their performance. This performance gain is not realised to its full potential due to the existence of *hazards*: conditions that prevent the execution of parallel instructions. *Control* hazards arise from the pipelining of instructions such as branches that change the program counter (PC). In this case the fetching and execution of instructions that follow a conditional branch instruction are stalled until the branch location is determined [HP90, p. 257]. It is possible to reduce pipeline stalls by predicting the program flow and arranging for the appropriate instructions to be fetched. In our example we use a loop code generation technique that results in code that adaptively follows a predicted branch path [Kra94].

3.1 Branch Prediction Methods

Branch prediction methods can be broadly distinguished between *static* methods where the probable direction of program flow is determined before the program execution, and *dynamic* methods where branches are predicted during the program's execution by some measure of previous branch history [HP90, 307–314]. Static branch prediction methods compile branches assuming that a given branch will be taken or not taken according to various heuristics. The scheme can be improved by using profiling data from previous runs of the code [MH86, p. 401]. Dynamic schemes depend on hardware memory which is used to store the behaviour of branch instructions in their previous executions. Given the finite amount of storage that can be used to store the behaviour of all branch instructions in a program, hashing and LRU-buffer techniques are used to store predictions about a subset of the branch instructions [Smi81]. In the case of branches that are usually not taken a one-bit prediction scheme will result in two wrong guesses when a branch is taken. In order to avoid this situation an additional bit can be used to provide a second order prediction behaviour. A branch prediction buffer can be further extended into a branch target buffer by storing in the buffer the actual branch target instruction, thus gaining an instruction fetch cycle. A comparison of software and hardware schemes for reducing the cost of branches can be found in [HCC89].

3.2 Adaptive Code

Instruction sequences containing conditional execution of instructions inside loop bodies can be compiled into code that adaptively executes the optimum branch instructions according to the program's branch behaviour. A loop body containing an *if-then-else* conditional statement is compiled into two loop sequences: one sequence compiled for optimum behaviour when the *then* branch is taken, and one compiled for optimum behaviour when the *else* branch taken. For both sequences the code body for the part which is predicted not to be executed is replaced with a branch to the corresponding code of the other sequence. Thus, the code in Figure 3 is compiled using branches with a high *taken* penalty as shown in

```
LOOP
        statement-1
        IF condition THEN
                statement-2
        ELSE
                statement-3
        FI
        statement-4
END LOOP
```

Figure 3: Sample loop code.

```
L1:     statement-1
        test condition
        branch if not condition L4
L2:     statement-2
        statement-4
        branch L1

L3:     statement-1
        test condition
        branch if condition L2
L4:     statement-3
        statement-4
        branch L3
```

Figure 4: Dynamic loop compilation sequence.

Figure 4. The technique is applicable to many different loop code sequences containing conditional branches.

4 Implementation

In order to test out our approach we implemented a prototype optimiser and used it to implement the dynamic branch prediction scheme described above.

The optimiser first reads the compiler's target language output and converts all lines into strings composed of three token classes:

labels which are converted into the character *L* followed by an integer identifying the label,

branch instructions which apart from the target label are not modified, and

other code which is replaced by the character S followed by an integer identifying that code.

A lookup table is built to convert the *L* and *S* tokens back to the original target code labels and instructions. The following is a formatted sample fragment of tokenised target code:

```
S0025 L0000: S0026 S0027 jeL0001 S0028
jmpL0002 S0029 L0001: S0030 L0002: S0031
jmpL0000 S0032 S0033 S0034 S0035
```

The optimiser then reads the declarative specification of the optimisations to be performed. Figure 5 contains the optimisation

```
# Input pattern
L1:
        S1*
        jeL2
        S2*
        jmpL3
        S5*
L2:
        S3*
L3:
        S4*
        jmpL1
=>
# Output specification
N1:
        S1
        je N2
N4:
        S2
        S4
        jmp N1

N3:
        S1
        jne N4
N2:
        S3
        S4
        jmp N3
```

Figure 5: Example of optimiser input pattern and output specification.

specification for the one-bit loop optimisation presented in Section 3.2. The specification is written as a pattern to be recognised and the replacement code.

The pattern to be recognised can consist of:

label specifications names starting with *L* followed by digits,

jump instructions written as they appear in the target code output, and

code place-holders for all other code represented by names starting with *S* followed by digits.

Sn place-holders can be followed by a trailing Kleene star (*) to specify an arbitrary number of instructions.

The replacement text is separated from the code pattern by a '=>' symbol and consists of:

new label specifications names starting with *N* followed by digits; these are replaced by unique label identifiers,

literal output written as it will appear in the target code output, and

pattern specification place-holders and labels names starting with *S* or *L* followed by digits; these are copied from the matched pattern.

The input pattern is converted into an extended string regular expression with the appropriate tokens replaced with back-references to the tokens in the previous part of the expression. Thus the pattern specification in Figure 5 is translated into the following regular expression ($\backslash d+$ means any number of digits and $\backslash$ *number* is a back-reference to a previous bracketed expression):

```
(L\d+):(S\d+)*je(L\d+)(S\d+)*jmp(L\d+)
(S\d+)*\3:(S\d+)*\5:(S\d+)*jmp\1
```

the corresponding replacement specification ($*number* is a reference to a bracketed part of the regular expression):

```
$N1:    $2
        je $N2
$N4:    $4
        $8
        jmp $N1
$N3:    $2
        jne $N4
$N2:    $7
        $8
        jmp $N3
```

and code to increment the label variables $N.

The optimiser works by continuously matching the input regular expressions against the tokenised program and replacing the matched part with the specified output code. When no more matches can be performed the tokenised program is converted back into normal target code using the lookup table built in the first phase to derive the optimised target code.

The declarative specification of the optimisation operations, and the use of string regular expressions for code pattern matching resulted in a flexible and compact implementation. The optimiser is currently a 127 line program written in Perl [SCP96]. It has been tested on Intel iAPX86 assembly output generated by the Microsoft C compiler.

5 Conclusions

So far the approach and the optimiser have only been tested on small example programs, a single optimisation specification, and a limited number of flow-of-control optimisations. However, the modest implementation effort invested and the flexibility offered by the parametric, declarative specification of the optimisation operations convinced us that this approach can be particularly suitable for rapid prototyping and experimentation with new optimisation methods, novel target architectures such as the Java VM, and applications of higher order optimisations.

References

[AG85] A. V. Aho and M. Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, January 1985.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[HCC89] W.W. Hwu, T.M. Conte, and P.P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 224–233, June 1989.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[Kra94] Andreas Krall. Improving semi-static branch prediction by code replication. *ACM SIGPLAN Notices*, 29(6):97–106, June 1994. Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI).

[MH86] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 1986.

[SCP96] Randal L. Schwartz, Tom Christiansen, and Stephen Potter. *Programming Perl*. O'Reilly & Associates, second edition, 1996.

[SG97] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *USENIX Conference on Domain-Specific Languages*, pages 67–76, Santa Monica, CA, USA, October 1997. USENIX.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 135–148, Minneapolis, USA, May 1981.

Appendix: Optimiser Code Listing

```
# Optimise a program with branches based on a
# declarative specification

($#ARGV == 1) || die "usage: $0 rules code\n";
strprog($ARGV[1]);
mksubst($ARGV[0]);
print "subst=$subst\n" if ($debug);
eval($subst);
$_ = $prog;
unstrprog();
print;

# Convert the change specification into the
# regular expression substitution program $subst
sub mksubst
{
    my($in) = @_;

    open(IN, $in) || die "$in: $!\n";
    for ($i = 0; $i < 10; $i++) {
```

```
        $subst .= "\$N$i = \"\\\$BO${i}0000\";\n";
    }
    $n = 1;
    $subst .= 'while ($prog =~ s/';
    while (<IN>) {
        last if (/^\=\>/);
        next if (/^#/);
        chop;
        if (m/(([LS])\d+)/) {
            $name = $1;
            $type = $2;
            if ($new = $pre{$name}) {
                $new =~ s/\$/\\/;
                s/$name/$new/;
            } else {
                $pre{$name} = "\$$n";
                $n++;
                s/$name/($type\\d+)/;
            }
            s/\s+//g;
        }
        $subst .= $_;
    }
    $subst .= '/';
    undef %N;
    while (<IN>) {
        next if (/^#/);
        if (s/(N(\d+))/\$$1/) {
            $N{$2} = 1;
        }
        if (m/(([LS])\d+)/) {
            $name = $1;
            if (!$pre{$name}) {
                printf STDERR "Unknown re-
place name $name\n";
                exit (1);
            }
            s/$name/$pre{$name}/;
        }
        $subst .= $_;
    }
    $subst .= "/g) {\n";
    foreach $i (keys %N) {
        $subst .= "\t\t\$N$i++;\n";
    }
    $subst .= "
        }
    ";
}

# Convert program into the string $prog
sub strprog
{
    my($in) = @_;

    open(IN, $in) || die "$in: $!\n";
    $S = $L = 0;
    $s = $l = '0000';
    while (<IN>) {
        # $L{internal} = program
        # $l{program} = internal

        # labels (start with a $)
        if (m/(\$\w+)/) {
            $name = $1;
            if ($old = $l{$name}) {
                $name =~ s/([^\w])/\\$1/g;
                s/$name/$old/e;
            } else {
                $new = $l{$name} = "L$l";
                $L{"L$l"} = $name;
                $name =~ s/([^\w])/\\$1/g;
                s/$name/$new/;
                $L++;
                $l = sprintf('%04d', $L);
            }
        }
        # Skip branches (start with j) and
        # labels (end with :)
        if (m/^\tj/ || m/:$/) {
            chop;
            s/SHORT//;
            s/\s+//g;
        } else {
        # Other statements
            $name = $_;
            $new = $s{$name} = "S$s";
            $S{"S$s"} = $name;
            $_ = $new;
            $S++;
            $s = sprintf('%04d', $S);
        }
        $prog .= $_;
    }
}

# Convert $_ back to the original program
sub unstrprog
{
    while (m/(L\d+)/) {
        $name = $1;
        s/$name/$L{$name}/;
    }
    while (m/(S\d+)/) {
        $name = $1;
        s/$name/$S{$name}/;
    }
}
```