A Tour of the New Peephole Optimizer

B. J. McKenzie

1. Introduction

The peephole optimizer consists of four major parts:

a) the table describing the optimization to be performed

b) a program to parse these tables and build input and output routines to interface to the library and a dfa based routine to recognize patterns and make the requested replacements.

c) common routines for the library that are independent of the table of a)

d) a stand alone version of the optimizer.

The library conforms to the *EM_CODE(3)* module interface but with routine names of the form ***C***_xxx replaced by names like ***O***_xxx. Furthermore there is also no routine *O_getid* and no variable *O_tmpdir* in the module. The library module results in calls to the usual *EM_CODE(3)* module. It is possible to write a front end so that it can call either the normal *EM_CODE(3)* module or this new module by adding **#define PEEPHOLE** before the line **#include <em.h>** This will map all calls to the routine ***C***_xxx into a call to the routine ***O***_xxx.

We shall now describe each of these major parts in some detail.

2. The optimization table

The file *patterns* contains the patterns of EM instructions to be recognized by the optimizer and the EM instructions to replace them. Each pattern may have an optional restriction that must be satisfied before the replacement is made. The syntax of the table will be described using extended BNF notation used by *LLGen* where:

```
[...]   - are used to group items
|       - is used to separate alternatives
;       - terminates a rule
?       - indicates item is optional
*       - indicates item is repeated zero or more times
+       - indicates item is repeated one or more times
```

The format of each rule in the table is:

```
rule    : pattern global_restriction? ':' replacement
        ;
```

Each rule must be on a single line except that it may be broken after the colon if the next line begins with a tab character. The pattern has the syntax:

```
pattern         : [ EM_mnem [ local_restriction ]? ]+
                ;
EM-mnem : "An EM instruction mnemonic"
        | 'lab'
        ;
```

and consists of a sequence of one or more EM instructions or *lab* which stands for a defined instruction label. Each EM-mnem may optionally be followed by a local restriction on the argument of the mnemonic and take one of the following forms depending on the type of the EM instruction it follows:

```
local_restriction   : normal_restriction
                    | opt_arg_restriction
                    | ext_arg_restriction
                    ;
```

A normal restriction is used after all types of EM instruction except for those that allow an optional argument, (such as *adi*) or those involving external names, (such as *lae*) and takes the form:

```
normal_restriction      : [ rel_op ]? expression
                        ;
rel_op      : '=='
        | '!='
        | '<='
        | '<'
        | '>='
        | '>'
        ;
```

If the rel_op is missing, the equality == operator is assumed. The general form of expression is defined later but basically it involves simple constants, references to EM_mnem arguments that appear earlier in the pattern and expressions similar to those used in C expressions.

The form of the restriction after those EM instructions like *adi* whose arguments are optional takes the form:

```
opt_arg_restriction     : normal_restriction
                        | 'defined'
                        | 'undefined'
                        ;
```

The *defined* and *undefined* indicate that the argument is present or absent respectively. The normal restriction form implies that the argument is present and satisfies the restriction.

The form of the restriction after those EM instructions like *lae* whose arguments refer to external object take the form:

```
ext_arg_restriction     : patarg  offset_part?
                        ;
offset_part             : [ '+' | '-' ] expression
                        ;
```

Such an argument has one of three forms: a offset with no name, an offset form a name or an offset from a label. With no offset part the restriction requires the argument to be identical to a previous external argument. With an offset part it requires an identical name part, (either empty, same name or same label) and supplies a relationship among the offset parts. It is possible to refer to test for the same external argument, the same name or to obtain the offset part of an external argument using the *sameext* , *samenam* and *offset* functions given below.

The general form of an expression is:

```
expression  : expression binop expression
            | unaryop expression
            | '(' expression ')'
            | bin_function '(' expression ',' expression ')'
            | ext_function '(' patarg ',' patarg ')'
            | 'offset' '(' patarg ')'
            | patarg
            | 'p'
            | 'w2'
            | 'w'
            | INTEGER
            ;

bin_function      : 'sfit'
            | 'ufit'
            | 'samesign'
            | 'rotate'
            ;

ext_function: 'samenam'
            | 'sameext'
            ;
patarg            : '$' INTEGER
            ;
binop       : "As for C language"
unaryop           : "As for C language"
```

The INTEGER in the *patarg* refers to the first, second, etc. argument in the pattern and it is required to refer to a pattern that appears earlier in the pattern The *w* and *p* refer to the word size and pointer size (in bytes) respectively. The *w2* refers to twice the word size. The various function test for:

sfit	the first argument fits as a signed value of the number of bit specified by the second argument.
ufit	as for sfit but for unsigned values.
samesign	the first argument has the same sign as the second.
rotate	the value of the first argument rotated by the number of bit specified by the second argument.
samenam	both arguments refer to externals and have either no name, the same name or same label.
sameext	both arguments refer to the same external.
offset	the argument is an external and this yields it offset part.

The global restriction takes the form:

```
global_restriction : '?' expression
                   ;
```

and is used to express restrictions that cannot be expressed as simple restrictions on a single argument or are can be expressed in a more readable fashion as a global restriction. An example of such a rule is:

```
dup w ldl stf  ? p==2*w : ldl $2 stf $3 ldl $2 lof $3
```

which says that this rule only applies if the pointer size is twice the word size.

3. Incompatibilities with Previous Optimizer

The current table format is not compatible with previous versions of the peephole optimizer tables. In particular the previous table had no provision for local restrictions and only the equivalent of the global

restriction. This meant that our *'?'* character that announces the presence of the optional global restriction was not required. The previous optimizer performed a number of other tasks that were unrelated to optimization that were possible because the old optimizer read the EM code for a complete procedure at a time. This included tasks such as register variable reference counting and moving the information regarding the number of bytes of local storage required by a procedure from it *end* pseudo instruction to it's *pro* pseudo instruction. These tasks are no longer done by this module but have been moved to other modules or programs in the pipeline. The register variable reference counting is now performed by the front end. The reordering of code, such as the moving of mes instructions and the local storage requirements from the end to beginning of procedures, is now performed using the insertpart mechanism in the *EM_CODE* (or *EM_OPT*) module. The removal of dead code is performed by the global optimizer. Various *ext_functions* available in the old tables are no longer available as they rely on information that is not available to the current program. These are the *notreg* and the *rom* functions. The previous optimizer allowed the use of *LLP, LEP, SLP* and *SEP* in patterns. For example *LLP* stood for either *lol* if the pointer size was the same as the word size, or for *ldl* if the pointer size was twice the word size. In the current optimizer it is necessary to include two patterns for each such single pattern in the old table. For example for a pattern containing *LLP* there would be one pattern with *lol* and with a global restriction of the form *p=w* and another pattern with ldl and a global restriction of the form *p=2*w.*

4. The Parser

The program to parse the tables and build the pattern table dependent dfa routines is built from the files:

parser.h	header file
parser.g	LLGen source file defining syntax of table
syntax.l	Lex sources file defining form of tokens in table.
initlex.c	Uses the data in the library *em_data.a* to initialize the lexical analyzer to recognize EM instruction mnemonics.
outputdfa.c	Routines to output the dfa when it has been constructed. It outputs the files *dfa.c* and *trans.c*
outcalls.c	Routines to output the file *incalls.r* defined in the next section.
findworst.c	Routines to analyze patterns to find how to continue matching after a successful replacement or failed match.

The parser checks that the tables conform to the syntax outlined in the previous section and also makes a number of semantic checks on their validity. Further versions could make further checks such as looking for cycles in the rules or checking that each replacement leaves the same number of bytes on the stack as the pattern it replaces. The parser builds an internal dfa representation of the rules by combining rules with common prefixes. All local and global restrictions are combined into a single test to be performed are a complete pattern has been detected in the input. The idea is to build a structure so that each of the patterns can be matched and then the corresponding tests made and the first that succeeds is replaced. If two rules have the same pattern and both their tests also succeed the one that appears first in the tables file will be done. Somewhat less obvious is that if one pattern is a proper prefix of a longer pattern and its test succeeds then the second pattern will not be checked for.

A major task of the parser if to decide on the action to take when a rule has been partially matched or when a pattern has been completely matched but its test does not succeed. This requires a search of all patterns to see if any part of the part matched could be part of some other pattern. for example given the two patterns:

loc adi w loc adi w : loc $1+$3 adi w
loc adi w loc sbi w : loc $1-$3 adi w

If the first pattern fails after seeing the input:

loc adi loc

the parser will still need to check whether the second pattern matches. This requires a decision on how to fix up any internal data structures in the dfa matcher, such as moving some instructions from the pattern to the output queue and moving the pattern along and then deciding what state it should continue from. Similar decisions are requires after a pattern has been replaced. For example if the replacement is empty it is necessary to backup *n-1* instructions where *n* is the length of the longest pattern in the tables.

5. Structure of the Resulting Library

The major data structures maintained by the library consist of three queues; an *output* queue of instructions awaiting output, a *pattern* queue containing instructions that match the current prefix, and a *backup* queue of instructions that have been backed up over and need to be reparsed for further pattern matches. These three queues are maintained in a single fixed size buffer as explained in more detail in the next section. Also, after a successful match, a replacement queue is constructed.

If no errors are detected by the parser in the tables it output the following files if they have changed from the existing version of the file:

dfa.c — this contains the dfa encoded into a number of arrays using the technique of row displacement for compacted sparse matricies. Given an opcode and the current state, the value of *OO_base[OO_state]* is consulted to obtain a pointer into the array *OO_checknext.* If this pointer in zero or the *check* field of the addressed structure does not correspond to the curerent state then it is known there is no entry for this opcode/state pair and the *OO_default* array is consulted instead. If the check field does match then the *next* field contains the new state. After each transition the array *OO_ftrans* is consulted to see if this state corresponds to a final state (i.e. a complete pattern) and if so the corresponding function is called.

trans.c — this contains external declarations of transition routines with names like **OO_xxxdotrans** (where *xxx* is a small integer). These are called when there a transition to state *xxx* that corresponds to a complete pattern. Any tests are performed if necessary to confirm that the pattern matches and then the replacement instructions are placed on the output queue and the routine *OO_mkrepl* is called to make the replacement and if backup the amount required. If there are a number of patterns with the same instructions but different tests, these will all appear in the same routine and the tests performed in the order they appear in the original *patterns* file.

incalls.r — this contains an entry for every EM instruction (plus *lab*) giving information on how to build a routine with the name ***O***_xxx for the library version of the module. If the EM instruction does not appear in the tables patterns at all then the dfa routine is called to flush any current queued output and the the output ***C***_xxx routine is called. If the EM instruction does appear in a pattern then the instruction data structure fields are initialized and it is added onto the end of the pattern queue. The dfa routines are then called to attempted to make a transition. This file is input to the *awk* program *makefuns.awk.*

The following files contain code that is independent of the pattern tables:

main.c — this is used only in the stand alone version of the optimizer and consists of code to open the input file, read the input using the *READ_EM(3)* module and call the dfa routines. This version does not require the routines constructed from the incalls.r file described above.

nopt.c — general routines to initialize, and maintain the data structures. The file handling routines *O_open* etc are defined here. Also defined are routines for flushing the output queue by calling the *EM_mkcalls* routine from the *READ_EM(3)* module and moving instructions from the output to the backup queue. Routines to free the strings stored in instructions with types of *sof_ptyp, pro_ptyp, str_ptyp, ico_ptyp, uco_ptyp,* and also*fco_ptyp*are is extended by *Realloc* if it overflows. The strings can be thrown away on any flush that occurs when the backup queue

is empty.

mkstrct.c contains routines to build the data structure from the input ***C***_xxx routines and place the structure on the pattern queue. These routines are also used to build the data structures when a replacement is constructed.

aux.c routines to implement the external functions used in the pattern table.

The following files are also used in building the module library:

makefuns.awk
this *awk* program is used to produce individual C files with names like ***O***_xxx.c each containing a single function definition and then call the *cc* compiler to produce a single output file. This enables the loader to only load those routines that are actually needed when the library is loaded.

pseudo.r this file is like the *incalls.r* file produced by the parser but is built by hand and handles the pseudo EM instructions. It is also processed by *makefuns.awk.*

6. Miscellaneous Issues

The output, pattern and backup queues are maintained in fixed length array, *OO_buffer* allocated of size *MAXBUFFER* (a constant declared in nopt.h) at run time. It consists of an array of the *e_instr* data structure used by the *READ_EM(3)* module. At any time the pointers *OO_patternqueue* and *OO_nxtpatt* point to the beginning and end of the current pattern prefix that corresponds to the current state. Any instructions on the backup queue are between *OO_nxtpatt* and *OO_endbackup.* If there are no instructions on the backup queue then *OO_endbackup* will be 0 (zero). The size of the replacement queue is set to the length of the maximum replacement length by the tables output by the parser.

The fixed size of the buffer causes no difficulty in practice and can only result in some potential optimizations being missed. When space for a new instruction is required and the buffer is full the routine *OO_halfflush* is called to flush half the buffer and move all the data structures left. It should be noted that it is not possible to statically determine the maximum possible size for these queues as they need to be unbounded in the worst case. A study of the rule

inc dec :

with the input consisting of *N inc* and then *N dec* instructions requires an output queue length of *N-1* to find all possible replacements.