Internal documentation on the peephole optimizer from the Amsterdam Compiler Kit

1. Introduction

Part of the Amsterdam Compiler Kit is a program to do peephole optimization on an EM program. The optimizer scans the program to match patterns from a table and if found makes the optimization from the table, and with the result of the optimization it tries to find yet another optimization continuing until no more optimizations are found.

Furthermore it does some optimizations that can not be called peephole optimizations for historical reasons, like branch chaining and the deletion of unreachable code.

The peephole optimizer consists of three parts

1) A driving table
2) A program translating the table to internal format
3) C code compiled with the table to make the optimizer proper

In this document the table format, internal format and data structures in the optimizer will be explained, plus a hint on what the code does where it might not be obvious. It is a simple program mostly.

2. Table format

The driving table consists of pattern/replacement pairs, in principle one per line, although a line starting with white space is considered a continuation line for the previous. The general format is:

```
optimization : pattern ':' replacement '\n'

pattern : EMlist optional_boolean_expression

replacement : EM_plus_operand_list
```

Example of a simple one

```
loc stl $1==0 : zrl $2
```

There is no real limit for the length of the pattern or the replacement, the replacement might even be longer than the pattern, and expressions can be made arbitrarily complicated.

The expressions in the table are made of the following pieces:

- Integer constants
- $*n*, standing for the operand of the *n*'th EM instruction in the pattern, undefined if that instruction has no operand.
- w, standing for the wordsize of the code optimized.
- p, for the pointersize.
- defined(expr), true if expression is defined
- samesign(expr,expr), true if expressions have the same sign.
- sfit(expr,expr), ufit(expr,expr), true if the first expression fits signed or unsigned in the number of bits given in the second expression.
- rotate(expr,expr), first expression rotated left the number of bits given by the second expression.
- notreg(expr), true if the local with the expression as number is not a candidate to put in a register.

- rom(*n*,expr), contents of the rom descriptor at index expr that is associated with the global label that should be the argument of the *n*'th EM instruction. Undefined if such a thing does not exist.

The usual arithmetic operators may be used on integer values, if any operand is undefined the expression is undefined, except for the defined() function above. An undefined expression used for its truth value is false. All arithmetic on local label operands is forbidden, only things allowed are tests for equality. Arithmetic on global labels makes sense, i.e. one can add a global label and a constant, but not two global labels.

In the table one can use five additional EM instructions in patterns. These are:

lab	Stands for a local label
LLP	Load Local Pointer, translates into a **lol** or into a **ldl** depending on the relationship between wordsize and pointersize.
LEP	Load External Pointer, translates into a **loe** or into a **lde**.
SLP	Store Local Pointer, **stl** or **sdl**.
SEP	Store External Pointer, **ste** or **sde**.

There is only one peephole optimizer, so the substitutions to be made for the last four instructions are made at run time before the first optimizations are made.

3. Internal format

The translating program, *mktab* converts the table into an array of bytes where all patterns follow unaligned. Format of a pattern is:

1) One byte for high byte of hash value, will be explained later on.
2) Two bytes for the index of the next pattern in a chain.
3) An integer[*], pattern length.
4) The list of pattern opcodes, one per byte.
5) An integer expression index, 0 if not used.
6) An integer, replacement length.
7) A list of pairs consisting of a one byte opcode and an integer expression index.

The expressions are kept in an array of triples, implementing a binary tree. The *mktab* program tries to minimize the number of triples by reusing duplicates and even reverses the operands of commutative operators when doing so would spare a triple.

4. A tour through the sources

Now we will walk through the sources and note things of interest.

4.1. The header files

The header files are the place where data structures and options reside.

4.1.1. alloc.h

In the header file alloc.h several defines can be used to select various kinds of core allocation schemes. This is important on small machines like the PDP-11 since a complete procedure must be in core at the same space, and the peephole optimizer should not be the limiting factor in determining the maximum size of procedures if possible. Options are:

- USEMALLOC, standard malloc() and free() are used instead of the own core allocation package. Not recommended unless the own package does not work on some bizarre machine.
- COREDEBUG, prints large amounts of information about core management. Not recommended unless the code is changed and it stops working.

* An integer is encoded as a byte when less than 255, otherwise as a byte containing 255 followed by two bytes with the real value.

- SEPID, defining this will add an extra procedure that will go through a lot of work to scrape the last bytes together if the system won't provide more. This is not a good idea if memory is scarce and code and data reside in the same spaces, since the room used by the procedure might well be more than the room saved.
- STACKROOM, number of shorts used in stack space. This is used if memory is scarce and stack space and data space are different. On the PDP-11 a UNIX process starts with an 8K stack segment which cannot be transferred to the data segment. Under these conditions one can use a lot of the stack space for storage.

4.1.2. assert.h

Just defines the assert macro. When compiled with -DNDEBUG all asserts will be off.

4.1.3. ext.h

Gives external definitions of variables used by more than one module.

4.1.4. line.h

Defines the structures used to keep instructions, one structure per line of EM code, and the structure to keep arguments of pseudos, one structure per argument. Both structures essentially contain a pointer to the next, a type, and a union containing information depending on the type. Core is allocated only for the part of the union used.

The *struct line* has a very compact encoding for small integers, they are encoded in the type field. On the PDP-11 this gives a line structure of only 4 bytes for most instructions.

4.1.5. lookup.h

Contains definition of the struct used for symbol table management, global labels and procedure names are kept in one table.

4.1.6. optim.h

If one defines the DIAGOPT option in this header file, for every optimization performed a number is written on stderr. The number gives the number of the pattern in the table or one of the four special numbers in this header file.

4.1.7. param.h

Contains one settable option, LONGOFF. If this is not defined the optimizer can only optimize programs with wordsize 2 and pointersize 2. Set this only if it must be run on a Z80 or something pathetic like that.

Other defines here should not be touched.

4.1.8. pattern.h

Contains defines of indices in a pattern, definition of the expression triples, definitions of the various expression operators and definition of the result struct where expression results are put.

This header file is the main one that is also included by *mktab*.

4.1.9. proinf.h

This one contains definitions for the local label table structs and for the struct where all information for one procedure is kept. This is in one struct so it can be saved easily when recursive procedures have to be resolved.

4.1.10. tes.h

Contains the data structure used by the top element size computation.

4.1.11. types.h

Collection of typedefs to be used by almost all modules.

4.2. The C code itself.

The C code will now be the center of our attention. We will make a walk through the sources and we will try to follow the sources in a logical order. So we will start at

4.2.1. main.c

The main.c module contains the main() function. Here nothing spectacular happens, only thing of interest is the handling of flags:

-L This is an instruction to the peephole optimizer to perform one of its auxiliary functions, the generation of a library module. This makes the peephole optimizer write its output on a temporary file, and at the end making the real output by first generating a list of exported symbols and then copying the temporary file behind it.

-n Disables all optimization. Only thing the optimizer does now is filling in the blank after the *END* pseudo and resolving recursive procedures.

The place where main() is left is the call to getlines() which brings us to

4.2.2. getline.c

This module reads the EM code and constructs a list of *struct line* records, linked together backwards, i.e. the first instruction read is the last in the list. Pseudos are handled here also, for most pseudos this just means that a chain of argument records is linked into the linked line list but some pseudos get special attention:

exc This pseudo is acted upon right away. Lines read are shuffled around according to instruction.

mes Some messages are acted upon. These are:

- ms_err The input is drained, just in case it is a pipe. After that the optimizer exits.
- ms_opt The do not optimize flag is set. Acts just like -n on the command line.
- ms_emx The word- and pointersize are read, complain if we are not able to handle this.
- ms_reg We take notice of the offset of this local. See also comments in the description of peephole.c

pro A new procedure starts, if we are already in one save the status, else process collected input. Collect information about this procedure and if already in a procedure call getlines() recursively.

end Process collected input.

The phrase "process collected input" is used twice, which brings us to

4.2.3. process.c

This module contains the entry point process() which is called at any time the collected input must be processed. It calls a variety of other routines to get the real work done. Routines in this module are in chronological order:

symknown Marks all symbols seen until now as known, i.e. it is now known whether their scope is local or global. This information is used again during output.

symvalue Runs through the chain of pseudos to give values to data labels. This needs an extra pass. It cannot be done during the getlines pass, since an **exc** pseudo could destroy things. Nor can it be done during the backward pass since it is impossible to do good fragment numbering backward.

checklocs Checks whether all local labels referenced are defined. It needs to be sure about this since otherwise the semi global optimizations made cannot work.

relabel	This routine finds the final destination for each label in the procedure. Labels followed by unconditional branches or other labels are marked during the peephole fase and this leeds to chains of identical labels. These chains are followed here, and in the local label table each label has associated with it its replacement label, after this procedure is run. Care is taken in this routine to prevent a loop in the program to cause the optimizer to loop.
cleanlocals	This routine empties the local label table after everything is processed.

But before this can all be done, the backward linked list of instructions first has to be reversed, so here comes

4.2.4. backward.c

The routine backward has a number of functions:

- It reverses the backward linked list, making two forward linked lists, one for the instructions and one for the pseudos.
- It notes the last occurrence of data labels in the backward linked list and puts it in the global symbol table. This is of course the first occurence in the procedure. This information is needed to decide whether the symbols are global or local to this module.
- It decides about the fragment boundaries of data blocks. Fragments are numbered backwards starting at 3. This is done to be able to make the type of an expression containing a symbol equal to its fragment. This type can then not clash with the types integer and local label.
- It allocates a rom buffer to every data label with a rom behind it, if that rom contains only plain integers at the start.

The first thing done after process() has called backward() and some of its own little routines is a call to the real routine, the one that does the work the program was written for

4.2.5. peephole.c

The first routines in peephole.c implement a linked list for the offsets of local variables that are candidates for a register implementation. Several patterns use the notreg() function, since it is forbidden to combine a load of that variable with the load of another and it is not allowed to take the address of that variable.

The routine peephole hashes the patterns the first time it is called after which it doesn't do much more than calling optimize. But first hashpatterns().

The patterns are hashed at run time of the optimizer because of the **LLP**, **LEP**, **SLP** and **SEP** instructions added to the instruction set in this optimizer. These are first replaced everywhere in the table by the correct replacement after which the first three instructions of the pattern are hashed and the pattern is linked into one of the 256 linked lists. There is a define CHK_HASH in this module that can be set if the randomness of the hashing function is not trusted.

The attention now shifts to optimize(). This routine calls basicblock() for every piece of code between two labels. It also notes which labels have another label or a branch behind them so the relabel() routine from process.c can do something with that.

Basicblock() keeps making passes over its basic block until no more optimizations are found. This might be inefficient if there is a long basicblock with some deep recursive optimization in one part of it. The entire basic block is then scanned a lot of times just for that one piece. The alternative is backing up after making an optimization and running through the same code again, but that is difficult in a single linked list.

It hashes instructions and calls trypat() for every pattern that has a full hash value match, i.e. lower byte and upper byte equal. Longest pattern is tried first.

Trypat() checks length and opcodes of the pattern. If correct it fills the iargs[] array with argument values and calculates the expression. If that is also correct the work shifts to tryrepl().

Tryrepl() generates the list of replacement instructions, links it into the list and returns true. Why then the name tryrepl() if it always succeeds? Well, there is a mechanism in the optimizer, unused until today that makes it possible to do optimizations that cannot be described by the table. It is possible to give a number as a replacement which will cause the optimizer to call a routine special() to do some work. This routine might decide not to do an optimization and return false.

The last routine that is called from process() is putline() to write the optimized code, bringing us to

4.2.6. tes.c

Contains the routines used by the top element size computation phase, which is run after the peephole-optimisation. The main routine of tes.c is tes_instr(). This looks at an instruction and decides the size of the element on top of the stack after the instruction is executed. When a label is defined or used, the size of the top element is remembered for later use. When the information in consistent throuhout the procedure, it is passed to the code generator by means of an ms_tes message.

4.2.7. putline.c

The major part of putline.c is the standard set of routines that makes EM compact code. The extra functions performed are:

- For every occurence of a global symbol it might be necessary to output a **exa**, **exp**, **ina** or **inp** pseudo instruction. That task is performed.
- The **lin** instructions are optimized here, **lni** instructions added for **lin** instructions and superfluous **lin** instructions deleted.