Specification for x86-64 Peephole Optimizer Overhaul

By

J. Gareth “Kit” Moreton

Introduction

The primary motivation behind overhauling the x86-64 Peephole Optimizer, and the i386 version by extension, is noting how the Peephole Optimizer goes over a single block of assembly language at least five times. This implementation appeared overly inefficient, especially for large procedures that may contain many hundreds of individual instructions.

Pascal has always had a history of being a fast, single-pass compiler. While being single-pass is largely impractical in the modern day of object orientation and heavily intertwined source code, the spirit of this tradition drove the desire to reduce the number of passes in the Peephole Optimizer as much as possible.

Background

As of 20th February 2019, the default Peephole Optimizer performs at least five individual passes over a block of code. Normally a block consists of a single subroutine, but under some circumstances, may be smaller (e.g. if there's an inline assembly block inside said subroutine, the block is split either side of it, since assembly blocks are not optimised). For large subroutines, said block may consist of many hundreds of instructions interspersed with labels and markers.

Pre-Peephole Pass

This pass, only performed once, attempts to restructure certain commands to be friendlier to the first pass, such as changing multiplications by 1 into straightforward moves. Normally such changes can be easily placed into Pass 1, but doing so could potentially remove later optimisations if Pass 1 is only run once, as is done under -O1 and -O2.

Pass 1

Under -O1 and -O2 settings, this pass is performed once, while on -O3 and -O4, it is performed as many times as necessary (until there are no more changes). However, regardless of the optimisation level, Pass 1 is then run again at least once – the reason for this is primarily for -O1 and -O2, since many of the first optimisations remove superfluous instructions like MOV instructions whose source and destination registers are identical, or jumps to destinations that immediately succeed them (in this article, termed a "zero distance jump"), and as a result, unlock potential optimisations that involve the instructions either side, but which will otherwise be missed because the pass has already analysed the first instruction.

All in all, at least two runs of Pass 1 are performed.

Pass 2

This pass performs optimisations that are only executed after Pass 1 is complete and are normally more complicated in nature, hence they are more likely to be missed in Pass 1 due to suboptimal instructions appearing elsewhere, or otherwise not as well-optimised as they could be. For example, one Pass 2 optimisation attempts to turn a conditional jump followed by a number of MOV operations into CMOV operations instead. This will be performed suboptimally if the MOVs aren't analysed first and may be only partially optimised in such a way that it cannot be completed on a subsequent pass.

Post-Peephole Pass

The final pass performs speed and space saving changes that might easily jeopardise earlier optimisations due to changing sub-registers or even the entire instruction – for example, changing `MOV EAX, 0` to `XOR EAX, EAX`, for example. Ultimately, it converts instruction forms that are convenient for the Peephole Optimizer into versions that are smaller and faster when actually compiled and executed, or may just reverse changes done in the Pre-peephole Pass.

Design Restrictions

When designing the overhaul, one rule that must be followed is that both performance and efficiency must not be made worse. Under -O3, this is not too difficult because Pass 1 is run as many times as necessary. Under -O1 and -O2, this is more difficult, because to reduce the number of passes, it requires making the individual operations more in-depth so optimisations are not missed.

Testing and Verification

To test the changes made to the Peephole Optimizer, numerous projects, including the compiler itself, are compiled with switches enabled to dump the assembler text. After compiling the projects with and without the overhaul changes applied, for each optimisation level, the assembler files are compared side-by-side for differences (while accommodating for the compiler's source files that have actually been changed, making note of the names of the modified procedures). If the code appears to be worse as a result of the changes, it is considered a bug insomuch that it doesn't adhere to the design restrictions.

Most important though are the regression tests, which must be run under both i386 and x86_64. Any additional failure caused by the overhaul, even if the compiler appears to function correctly, is indicative of bad machine code being generated.

After correctness is ensured, the compilation of Lazarus (a known large project) is performed with and without the changes, and the total compilation time recorded and an average taken over at least three compilations. Individual changes were enabled and disabled to test their effect on the final time – anything that was substantially slower than the trunk was debugged and, if that failed, discarded.

Ultimately, the changes implied a speed improvement of approximately 5% to 10% under -O1 and approximately 10% to 15% under -O2 and -O3. The fact that the improvement seems to be more profound on -O2 compared to -O1 is notable, given the overhaul Peephole Optimizer only goes through one loop under those settings, but this can be explained by the fact that optimisations performed earlier in the compilation process allow for smaller assembler blocks.

Additionally, the use of "`make fullcycle`" must verify that all cross-platform compilations are successful.

Implementation Specifics

Collapsing 4 Passes into 1

Besides just removing calls to `PrePeepHoleOpts`, `PeepHoleOptPass2` and the additional call to `PeepHoleOptPass1` (but keeping `PostPeepHoleOpts`), all of the methods that are called within the x86 Peephole Optimizer's `PeepHoleOptPass2Cpu` and `PrePeepHoleOptsCpu` routines are moved to the case block within `PeepHoleOptPass1Cpu` instead. Where there are clashes, as with `OptPass1MOV` and `OptPass2MOV`, the optimisations within are merged. Nevertheless, `OptPass1MOV` requires special care due to its complexity and size. This is described in the "OptPass1MOV Refactor" section.

Making Helper Functions Public

There are a number of subroutines in `compiler/aoptobj.pas` that are useful utility functions, such as `IsJumpToLabel`, but are not declared in the interface section, so cannot be used outside of that specific unit. These functions are moved to `compiler/aoptutils.pas` and defined in the interface section so they can be called by platform-specific units without having to duplicate code.

The following functions have been moved to `compiler/aoptutils.pas` and publicised:

`JumpTargetOp` - Returns the operand that contains a branch's destination symbol (under x86, this is always the first operand). This function was also made inline.

`IsJumpToLabel` - Returns True if the given instruction is any kind of branch to a label. Since the conditions are relatively simple, this function was also made inline.

`IsJumpToLabelUncond` - Returns True if the given instruction is specifically an unconditional branch to a label (under x86, this is JMP)

There is also a new helper function called `SetAndTest` that sets a variable of type TAi (ancestor class for an individual assembler instruction or label etc.) to another, and returns True if it's not nil. While this inlined function is simply to the effect of "`var2 := var1; Result := Assigned(var1);`", it is more convenient when used as part of a large set of conditions and also improves an inefficient hack (as self-admitted in the comments) in the `GetFinalDestination` method that calls `SkipLabels` twice just to set one variable to another while still in the if-block (the second call to `SkipLabels`, which returns false if the next instruction is nil, is replaced with `SetAndTest` that sets another variable to the next instruction as returned by `SkipLabels`).

Jump Optimisation

Overview

One of the more complex aspects of Pass 1 is optimising branches. There are numerous aspects to this system, from collapsing zero distance jumps, simplifying certain jump combinations to tracing the jumps to see where they lead, since the first instruction after a destination label is often times another jump.

Collapsing Zero Distance Jumps

The collapsing of zero distance jumps is one of the main reasons why Pass 1 originally needed at least two runs, because removing the jump and the label often led to an optimisation with the two instructions either side of them. To remove the need for a second pass, the collapsing of a zero distance jump, if present, has to become part of the individual optimisation routines when they look ahead in the instruction stream. To facilitate this, the code that removes the jump and the label is moved into a new method in the ancestral optimiser class named `CollapseZeroDistJump`. Besides collapsing the jump, an additional step is performed that strips 'dead labels' (labels with a zero reference count) between the jump and the destination label. Functions such as `GetNextInstruction` normally skip over these automatically, but by actually removing them from the list and destroying their objects, subsequent passes are made slightly faster and memory is freed earlier.

Additionally, the check for zero distance jumps now also includes conditional branches, which allows for that specific check in `OptPass2Jcc` (which became `OptPass1Jcc` in the overhaul) to be removed.

Destination Trace

Given how simplifying jumps is so central to the Peephole Optimizer, the `GetFinalDestination` method has been overhauled to be more efficient by using temporary variables to hold destination labels, as opposed to constantly dereferencing the symbol in the jump instruction's operand, and has also been improved so it strips dead code (code between an unconditional jump and the next label that will never be called under any circumstances) and dead labels on the fly.

Additionally, when stripping dead labels, which also occurs under `CollapseZeroDistJump` above, memory alignment hints are also removed if they are associated with a dead label. Besides shrinking the resultant code by removing unnecessary padding, it also catches zero distance jumps that were previously missed because of said alignment hints in between the jump and the label, and with those new jumps removed, further optimisations can be performed elsewhere.

To remove duplicate code, the i386 version of the Peephole Optimizer has had its own version of `GetFinalDestination` removed (it wasn't even a method, but a nested function inside the `PeepholeOptPass1` method) so it uses the more in-depth, cross-platform version.

Label Clustering

Occasionally, multiple live labels get clustered together, usually because the instructions between them get removed or the labels are born from different node types that happen to fall on the same location. A new private function in `PeepHoleOptPass1`, named `CollapseLabelCluster`, is called whenever a valid jump is encountered and it checks the destination label to see if any labels (and alignment hints) immediately follow it (i.e. is part of a cluster). The destination is changed to the last label in the cluster and the reference counts updated. This increases the chance that the other labels will become dead labels and be removed, hence allowing other potential optimisations.

OptPass1MOV Refactor

Overview

Though not directly related to reducing the number of passes, there are subroutines for MOV in both Pass 1 and Pass 2, so the optimisations contained within `OptPass2MOV` must be carefully merged into `OptPass1MOV`. Additionally, MOV commands are disproportionately the most common instruction found in compiled code, averaging about 25% of all operations, so it makes sense that the `OptPass1MOV` method should be made as efficient as possible.

GetNextInstruction Checks

Other than the very first optimisation, where the routine checks to see if the source and destination registers are identical (in which case, MOV is essentially a null operation), all optimisations depend on the presence of a succeeding instruction. Currently, individual optimisations check for the presence of said instruction with the `GetNextInstruction` method. As a consequence, this method is called multiple times when the result is already known and is wasteful processing, especially since the pointer to the next instruction won't change unless a major alteration was made, in which case the re-run flag is marked and the routine exits. Therefore, the first major optimisation to this routine is to call `GetNextInstruction` right after the single-instruction optimisation and set a local Boolean variable indicating the result, and then exit if the result is false or the next instruction isn't actually an instruction (e.g. a label).

The next step is to group optimisations that share an identical next instruction, in effect 'factorising' the conditional blocks. This is performed by building a large case block

Collapsing Zero Distance Jumps

As described previously, there are situations where a jump is immediately succeeded by its destination label. This can obviously be simplified by removing the jump, and in most cases, the label's reference count will fall to zero and can be removed as well. However, there are often sequences such as the following:

```
{$ASMMODE Intel}
MOV REG1, REG2
JMP @LabelX
@LabelX:
MOV REG3, REG1
```

After removing the jump and the label, an optimisation can be made with the two MOV commands, removing the first one (if REG1 isn't used later) and changing REG1 to REG2 in the second. Nevertheless, this optimisation is invisible to the Peephole Optimizer while the jump is present, and Pass 1's loop has already gone past the first MOV instruction that would otherwise optimise the two MOV instructions. Before, the second run of Pass 1 would perform the optimisation after the first run removes the jump.

For MOV in particular, this is best solved by adding “JMP” and “Jcc” to the 'next instruction' case block and seeing if the jump can be collapsed. If it can, then the jump is removed there and then and the 'new' next instruction sought (if the label cannot be removed because another branch jumps to it, then the optimisation routine exits as normal). Other optimisation subroutines may also benefit from this specific look-ahead.

Multiple Optimisations

Because MOV has so many possible optimisations, the overhaul now encases the majority of the routine with a "`repeat...until False;`" loop, which acts as a label for "`Continue;`" calls that appear later in the routine. Many of the optimisations in `OptPass1MOV` delete or modify the succeeding instruction and set the re-run flag (setting the result to True); before, the function would exit and return control to Pass 1's routine, which will then reanalyse the current instruction and call `OptPass1MOV` again, since the current instruction wasn't actually changed. By changing "Exit;" to "Continue;", a large number of cycles can be saved by jumping back to the start of `OptPass1MOV` where the program flow would have eventually returned anyway.

Other individual optimisation functions now also have a similar configuration.

Skip Function Prologue

Once the registers have been assigned for a particular subroutine, the prologue and epilogue can be considered static and must not change. Depending on the platform, this may contain PUSH and POP operations or reserve space on the stack then write non-volatile registers to specific locations. Additionally, there may be markers related to Structured Exception Handling. Ultimately, it means that the first few instructions can be ignored completely.

To aid with this, the `GetFirstInstruction` method is overridden in the x86 Peephole Optimizer to scan the beginning of a routine and return the first actual instruction that isn't part of the prologue. Under Microsoft Windows, it looks for the 'end prologue' SEH marker, otherwise it looks for an instruction that isn't modifying the stack and preserving non-volatile registers. At the same time, it calls `UpdateUsedRegs` if it runs into any register allocation markers.

Once the first true instruction has been determined, subsequent runs of the optimisation loop can skip to this instruction instead of the start of the prologue, thus saving some unnecessary calls to `OptPass1MOV`, for example (one point of note is that when the optimisation loop is re-run, the register state must be restored to what it is at the end of the prologue – failing to properly do this will cause incorrect optimisation and hence bad machine code to be generated).

Instruction Re-check Limitation

Before, if the re-run flag is set, the current instruction is checked again regardless. For MOV instructions and other operations that call long and complex optimisation routines, this can become a performance drain if it's known that nothing new will be optimised. The main loop in Pass 1 of the Peephole Optimizer will now move to the next instruction if a subroutine sets the re-run flag but the opcode of the current instruction hasn't changed (regardless of whether the current instruction is the original one or not). This is possible to do without missing potential optimisations thanks to the "`repeat...until False;`" loops in the optimisation subroutines, along with other code that calls `Continue` rather than `Exit` if the current instruction still has the same opcode – in a sense, a form of tail recursion.

Label Stripping

A final optimisation is included in the form of a new version of the `UpdateUsedRegs` routine, named `UpdateUsedRegsAndOptimize` – this routine, besides doing everything that `UpdateUsedRegs` does in order to evaluate which registers are currently in use, also strips away dead labels and alignment hints. Since `UpdateUsedRegs` steps into these entries anyway and just skips over them after verifying what they are, there is little performance loss from actively removing them.

Because `UpdateUsedRegsAndOptimize` now has the ability to remove entries from the code block, there is every possibility that the current entry (which may not be an instruction) gets removed as well – therefore, the method is designed to return the entry that appears prior to the next instruction (so "`p.Next;`" in the optimisation loop behaves as expected and also so `UpdateUsedRegsAndOptimize` can simply pass through the current instruction if nothing needs to be optimised).

Future Improvements

Minimising the Setting of the Re-run Flag

Whenever the re-run flag is set in Pass 1, the entire block is reprocessed. For large subroutines, this can amount to several hundred instructions to check, so if it can be argued that an individual optimisation case can get away with not setting the re-run flag, this amounts to a large performance gain for the compiler under -O3 and -O4. As before, to test if the change actually works, one must compare the assembler dumps of many projects for changes. Logically there should not be any improvements in this case, but it must not be made worse.

Register Virtualisation

As part of the compilation process, real registers are not actually assigned until a procedure has been fully parsed. If it is possible to move the entire Peephole Optimizer before this stage, then optimisations that might even completely remove a register will benefit the end program even further, because when the function prologue and epilogue are generated, an extra register need not be preserved. Additionally, the code in the Peephole Optimizer to skip over the function prologue and epilogue may be removed, since that code will no longer exist at this stage.

Merge Post-Peephole Stage

This one is more involved since the Post-Peephole Pass specifically makes changes that are unfriendly towards the Peephole Optimizer in general. Nevertheless, if it can be determined that no further changes will be performed on a specific section of code within a block, then the Post-Peephole changes may be performed without affecting the overall result.

A proposed implementation for this design would be to perform a specific Post-Peephole optimisation on an instruction only if no changes have been made up to that point (for -O3 – it can be done regardless for -O1 and -O2) – if the flag is set to re-run the pass, then all Post-Peephole optimisations cease. However, the final code is guaranteed to be identical only if optimisations don't look backwards in the code, which some routines do – however, such optimisations are in the minority and can be analysed individually and, if necessary, modified.

Node-Level Optimisations

To reduce pressure on the Peephole Optimizer and hence lower the chances of Pass 1 being repeated for -O3, some optimisations could potentially be moved to the first or second pass stage of the intermediate node generation. For example, if the nodes involved with multiplication could detect if the multiplicand was equal to 1 and pass through the other argument unmodified, this would remove the need to have a separate peephole optimisation for IMUL to do just this.