Specification for Free Pascal Compiler

Jump Optimisations

By

J. Gareth “Kit” Moreton

Introduction

There is a continual drive to improve and evolve the Free Pascal Compiler in order to increase the execution speed of produced binaries while retaining a fast compilation speed. While many optimisations are specific to particular platforms, there are a number that are common among multiple architectures, and improving the compiler in these areas gives benefits across the board.

Testing and Verification

Since this change is just an optimisation to jumps and jump chains, the main tests are that the generated binaries behave in exactly the same way when viewed as a black box. Internally there may be subtle differences, but ultimately the outputs must be identical for given inputs. In other words, the criteria for refactoring.

Most important are the regression tests, which must be run under both i386 and x86_64. Any new failure caused by the changes, even if the compiler appears to function correctly, is indicative of bad machine code being generated. Additionally, the use of "`make fullcycle`" must verify that all cross-platform compilations are successful.

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. Slower compilation times may not be avoidable, but should be minimised – this may be improved later on with code refactoring or optimisation checks that are proven to make no difference, or are otherwise redundant.

Implementation Specifics

Cross-Platform Optimisations

Overview

One of the more complex aspects of the Peephole Optimizer 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, and short-cutting these 'jump chains' can yield large performance gains.

"condition_in"

An extension of "conditions_equal", this function takes two conditions and determines if the first one is a subset of the second. It is used in the context of destination tracing (via `GetFinalDestination`) and simplifying jump chains to see if a conditional branch is deterministic based on known information. For example, under i386, if a JA instruction branches, and the instruction after the label is JNE, it can be determined that the JNE instruction will always branch in this situation because "jump if above (zero)" is a subset of "jump if not equal (to zero)" – that is, if a value is above zero, it is certainly not equal to zero as well. With this knowledge, the label specified in the JA instruction can be changed to the one specified in the JNE instruction.

Different architectures have different sets of conditions, so while this function exists on all platforms that support conditional jumps, its implementation is platform-specific. As a failsafe, the function returns False (i.e. not a subset) if not enough information is known about a platform's condition flags, so no jump shortcuts will be performed in this instance.

"condition_in" will always return True if the conditions are an exact match or if the second condition is C_None (i.e. a conditional jump is always a subset of an unconditional jump, since the unconditional jump will always branch no matter the condition).

Collapsing Zero Distance Jumps

A zero-distance jump is where a jump's destination label immediately succeeds the instruction. Individual platforms tended to have their own routines to perform this optimisation when they look ahead in the instruction stream, but by promoting such an optimisation to a cross-platform method, it allows all builds to benefit and also reduces code maintenance due to platform-specific routines being removed and otherwise wrapped into a single all-in-one algorithm. To facilitate this, a new method in the ancestral optimiser class will perform this optimisation, 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` (see "x86 Specifics" below) to be removed.

Removing Redundant Alignment Hints

When stripping dead labels, 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.

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 method, 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.

Label Stripping

While stripping dead labels is handled by many of the jump optimisations as an afterthought, there are times where it can be done almost for free during the main pass of the Peephole Optimizer. A new version of the `UpdateUsedRegs` routine, named `UpdateUsedRegsAndOptimize`, has been developed to facilitate this as part of Pass 1 of the Peephole Optimizer. 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).

x86 Specifics

Removal of Redundant Functionality

One of the tests in `OptPass2Jcc` sought to remove a zero-distance jump with a `Jcc` instruction. This has been removed since the functionality has been wrapped into the `CollapseZeroDistJump` method and hence will never get triggered.

Jcc → RET Optimisation

One set of opcodes that occasionally appeared was `Jcc @Lbl1; JMP @Lbl2` followed by some miscellaneous code, and then immediately after `@Lbl1` is a `RET` instruction to exit the current procedure. When this sequence of instructions is detected, the condition is inverted and the destination branch changed to `@Lbl2` (and the reference count for `@Lbl1` decremented). Meanwhile, the `JMP` opcode is replaced with another `RET` instruction.

Normally this would be a fine cross-platform optimisation, but different architectures handle procedure exits differently and not all may have a straightforward `RET` opcode, so currently it is specific to the `OptPass2Jcc` routine of the x86 optimiser.

Miscellaneous Compiler Performance Improvements

Destination Trace

Given how simplifying jumps is so central to the Peephole Optimizer, the `GetFinalDestination` method, which shortcuts jump chains, 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 and labels on the fly.

Future Improvements

More Cross-Platform Optimisation Routines

There are a number of operations that are relatively agnostic across different architectures, but which have platform-specific implementations due to the varied different ways they have to be performed. Two such operations are the detection and placement of an 'exit function' opcode (e.g. `RET` on x86) or transmuting a conditional branch into an unconditional jump. Discussion would be required for each specific function as well as a contingency for when a platform does not support a particular feature (e.g. the Java Virtual Machine does not have conditional jumps in the traditional sense, but have what are effectively if-statements as opcodes, while the AVR platform has more complex conditions for its jumps).