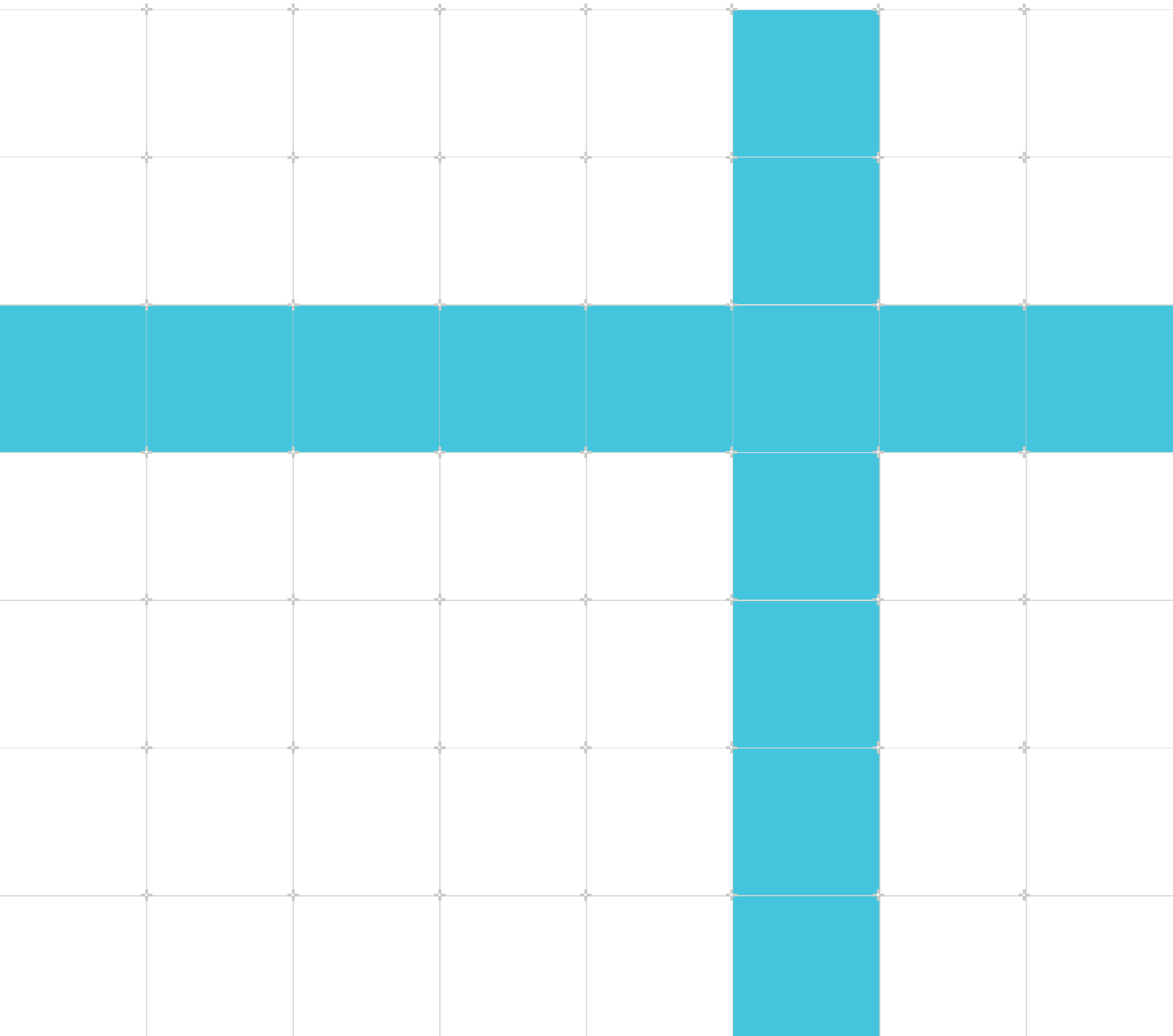
# Arm Compiler optimization

Version 1.0



**Issue 02**
102654_0100_02_en



# Arm Compiler optimization



## Release information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0100-02 | 17 August 2021 | Non-Confidential | First release |

## Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.



Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

## Product Status

The information in this document is Final, that is for a developed product.

## Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

## Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

# Contents

# 1. Overview

The Arm Compiler, `armcc`, optimizes your code for small code size and high performance. This tutorial introduces the main optimization techniques performed by the compiler, and explains how to control compiler optimization.

This tutorial assumes you have installed and licensed Arm DS-5 Development Studio. For more information, see Getting Started with Arm DS-5 Development Studio.

# 2. Overview of optimizations

The compiler performs optimizations common to other optimizing compilers as well as a range of optimizations specific to Arm architecture-based processors.

The main optimizations are:

**Common subexpression elimination**

The compiler identifies common subexpressions in the code and uses the result for each instance, rather than re-evaluating them repeatedly. For example, your code might use the expression `a+1` in several places. The compiler identifies and evaluates this expression only once, using the result for all subsequent instances.

**Loop invariant motion (expression lifting)**

The compiler identifies expressions inside loops that do not change as the loop is running. Continuous re-evaluation of these expressions would be costly, so the compiler moves the expression outside of the loop and evaluates it only once.

**Live range splitting for dynamic register allocation**

The compiler identifies the live state of variables within a program section, and allocates registers accordingly. For example, a variable might be used in one situation as a counter for a loop, then later as a working variable within a calculation. If these two uses are completely unrelated the compiler can allocate them to different registers. Additionally when a variable is no longer required, the compiler can reuse the register.

**Constant folding**

The compiler replaces constant expressions with their calculated values.

**Tailcall optimization and tail recursion**

A tailcall is a call immediately before a return. Normally this function is called, and when it returns to the caller, the caller returns again. Tailcall optimization avoids this by restoring the saved registers before jumping to the tailcall. The called function then returns directly to the caller's caller, saving a return sequence.

The compiler also supports tailcall recursion, which is possible when the tailcall is made to the same function. In this case it is possible to skip the entry and exit sequence altogether, converting the call into a loop.

**Cross jump elimination**

The compiler combines instances of identical code. For example, if multiple functions generate identical results the compiler optimizes them to a single return sequence. This optimization mainly saves space, and is disabled when optimizing for time.

**Table-driven peepholing**

The compiler identifies common code sequences and replaces them with known optimal versions. This is achieved by viewing the code through a window (of some number of instructions) called a peephole, and then replacing identified instruction sequences with a hand-crafted version. The table of peepholes is constantly growing as optimal sequences are identified and added by Arm engineers.

**Structure splitting**

The compiler can divide structures into their components and assign these to registers, for faster access. This is a particular advantage when a function returns a structure, because the whole structure can be returned in registers rather than on the stack.

**Conditional execution or branch elimination**

The compiler uses conditional execution to avoid branches. Conditional execution saves both space and execution time, as many conditional branches can be removed.

**Function inlining**

Function inlining offers a trade-off between code size and performance. By default, the compiler decides for itself whether to inline code or not. As a general rule, the compiler makes sensible decisions about inlining with a view to producing code of minimal size.

**Automatic vectorization for NEON**

The compiler analyzes loops in your code to find opportunities for parallelization using the NEON unit.

**Loop restructuring**

The compiler can unroll small loops for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. The compiler unrolls loops automatically at `-O3 -Otime`.

**Instruction scheduling**

Instruction scheduling is enabled at optimization level `-O1` and higher. Instructions are re-ordered to suit the processor that the code is compiled for. This can help improve throughput by minimizing interlocks and also making use of processor features such as dual execution.

# 3. Optimizing for code size versus speed

The compiler provides two options for optimizing for code size and performance:

`-Ospace`

Causes the compiler to optimize mainly for code size. This is the default option.

`-Otime`

This option causes the compiler to optimize mainly for speed.

# 4. Compiler optimization levels

The precise optimizations performed by the compiler depend both on the level of optimization chosen, and whether you are optimizing for performance or code size.

The compiler supports the following optimization levels:

`-O0`

Minimum optimization. Turns off most optimizations. When debugging is enabled, this option gives the best possible debug view because the structure of the generated code directly corresponds to the source code.

`-O1`

Restricted optimization. The compiler only performs optimizations that can be described by debug information. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with `--debug`, this option gives a generally satisfactory debug view with good code density.

`-O2`

High optimization. If used with `--debug`, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler may perform optimizations that cannot be described by debug information. The compiler automatically inlines functions.

`-O3`

Maximum optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

If you use `-O3` and `-Otime` together, the compiler performs extra optimizations that are more aggressive, such as:

- High-level scalar optimizations, including loop unrolling. This can give significant performance benefits at a small code size cost, but at the risk of a longer build time.
- More aggressive inlining and automatic inlining.

The `--loop_optimization_level=option` controls the amount of loop optimization performed at `-O3 -Otime`.

For extra information about the high level transformations performed on the source code at `-O3 -Otime` use the `--remarks` command-line option.


Note

Do not rely on the implementation details of these optimizations, because they might change in future releases.

By default, the compiler optimizes to reduce image size at the expense of a possible increase in execution time. That is, `-Ospace` is the default, rather than `-Otime`.


Note

`-Ospace` is not affected by the optimization level `-Onum`. That is, `-O3 -Ospace` enables more optimizations than `-O2 -Ospace`, but does not perform more aggressive size reduction.

The default optimization level is `-O2`.

# 5. Further reading

You can find out more about Arm compilers in the following guides:

Arm Compiler armcc User Guide

- Compiler Command-line Options
- Compiler Coding Practices
- Compiler optimization levels and the debug view