

# CoreMark Benchmarking for ARM<sup>®</sup> Cortex<sup>®</sup> Processors

## Application Note 350



# CoreMark Benchmarking for ARM Cortex Processors

## Application Note 350

Copyright © 2013 ARM Limited. All rights reserved.

### Release Information

**Table 1 Change history**

| Date      | Issue | Confidentiality  | Change        |
|-----------|-------|------------------|---------------|
| July 2013 | A     | Non-Confidential | First release |

### Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

### 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.

### Web Address

<http://www.arm.com>

# 1 Introduction

Processors in embedded systems are becoming more complex. As processor complexity increases, we require more sophisticated benchmarks to properly exercise and analyze those processors.

CoreMark is a modern, sophisticated benchmark that lets you accurately measure processor performance.

This application note addresses the following questions:

- What is CoreMark?
- Why does ARM recommend CoreMark over Dhrystone?
- Where can I obtain the CoreMark benchmark?
- How do I compile CoreMark, optimizing for either speed or code size?
- How do I run CoreMark to obtain stable, reproducible results?
- How do I interpret benchmark results and code size metrics?

## 2 About CoreMark

Developed in 2009 by the *Embedded Microprocessor Benchmark Consortium* (EEMBC), CoreMark is a freely available, easily portable benchmark program that measures processor performance.

CoreMark is intended to replace the older Dhrystone benchmark. ARM recommends using CoreMark in preference to Dhrystone.

Benchmarking modern processors with Dhrystone, developed in 1984, has a number of issues. These issues include the following:

- **Workload realism**

Dhrystone is a synthetic benchmark. This means that the workload it performs to exercise processors does not correspond well to real world applications.

CoreMark uses a more realistic workload. The CoreMark workload comprises several commonly used algorithms, including:

- Matrix manipulation, to exercise common math operations.
- Linked list manipulation, to exercise the common use of pointers.
- State machine operation, to exercise data dependent branches.
- *Cyclic Redundancy Check* (CRC), because CRC is a common function in embedded systems.

- **Compiler optimizations**

Dhrystone is susceptible to compilers being able to optimize work away. When this happens, Dhrystone becomes an unreliable processor benchmark.

With CoreMark, every operation in the benchmark derives values that are not available at compile time. This ensures that while compilers can still make optimizations, they cannot pre-compute results to optimize the work away completely.

- **Library calls**

Dhrystone contains library calls within the timed portion of the benchmark, which can account for a significant portion of the benchmark time. This makes it difficult to compare results where different libraries have been used.

CoreMark is designed so that it does not make any library calls during the timed portion of the benchmark.

- **Version control**

Dhrystone has no official source, so several different versions are in use. If the Dhrystone version is undisclosed, it is difficult to compare benchmark results.

CoreMark is available from the CoreMark web site, <http://www.coremark.org/>.

- **ANSI C compliance**

The original Dhrystone C code is not ANSI C compliant.

CoreMark code is ANSI C compliant.

- **Results reporting**

Dhrystone does provide guidelines on how to run the benchmark, but these are not universally known or enforced. As a result, Dhrystone results cannot be certified or verified. In addition, there is no standardization of how to report results. Dhrystone results might be reported in Dhrystones per second, DMIPS, or DMIPS/MHz.

CoreMark has a standard format for reporting results, and results can be uploaded to the CoreMark web site for certification.

ARM quotes figures for CoreMark 1.0. The format of CoreMark benchmark results is as follows:

CoreMark 1.0 : N / C [/ P] [/ M]

where:

N displays the number of iterations per second with seeds 0,0,0x66,size=2000.

C displays the compiler version and flags specified when compiling the benchmark.

P displays parameters such as data and code allocation specifics.

- This parameter can be omitted if all data was allocated on the heap in RAM.
- This parameter cannot be omitted when reporting CoreMark/MHz.

M shows the type of parallel algorithm execution (if used) and number of contexts.

- This parameter can be omitted if parallel execution is not used.

For example:

CoreMark 1.0 : 256.344527 / ARM C/C++ Compiler, 5.03 [Build 24] -O3 -Otime / STACK

For CoreMark benchmark results to be valid, the CoreMark code must run for at least 10 seconds.

Generally, longer runs are better. ARM recommends averaging multiple runs of at least 10000 iterations or 30 seconds, whichever is longer. See [Running CoreMark on page 10](#) for more information.

CoreMark stipulates a number of additional rules for valid benchmarks. You can find the complete set of rules in the `readme.txt` file included with the CoreMark source.

This application note describes how to build and run CoreMark on bare-metal systems.

### 3 Downloading the source

The official CoreMark source is available from the CoreMark web site,  
<http://www.coremark.org>.

## 4 Compiling CoreMark with ARM Compiler

CoreMark consists of the following C source and header files:

- coremark.h
- core\_main.c
- core\_list\_join.c
- core\_matrix.c
- core\_state.c
- core\_util.c
- simple/core\_portme.c
- simple/core\_portme.h

You must modify the source files in the `simple/` directory to target CoreMark to your particular platform.

### 4.1 Library functions required by CoreMark

CoreMark requires the following C library functions:

- `printf()` to report the benchmarking results.
- A function for measuring elapsed time appropriate to your platform, to compute the execution time of the benchmark.  
Typically this is the C library `clock()` function. However you could use cycle counters to directly implement the `start_time()` and `stop_time()` functions in `core_portme.c` instead.

These functions are contained in the following libraries:

- The standard C library, included with the ARM Compiler toolchain and the Keil™ MDK toolkit.
- An alternative, size-optimized C library called `microlib`, included with the Keil MDK toolkit.

When compiling CoreMark for a bare-metal system you must provide the startup code and the retargeted `printf()` and `clock()` functions.

Most modern ARM processors include performance counters. You can program these performance counters to count the number of processor cycles, and then accurately compute the elapsed time. To use this feature, you must retarget the `clock()` function. [Appendix A on page 14](#) shows a sample implementation of a retargeted `clock()` function for a Cortex-M4 processor.

In the absence of performance counters and a retargeted `clock()` function, you can use the semihosted `clock()` function from the standard C library. You must connect a debugger capable of supporting semihosting, such as the DS-5 Debugger.

#### ———— Note ————

Using semihosting can introduce extra delay, because the target system must communicate with the host. This extra delay can be unpredictable, because of variations in transmission and process scheduling. To reduce the effects of variable semihosting overheads in your benchmark results, you can increase the number of CoreMark iterations.

As with the `clock()` function, you can use a semihosted `printf()` function with DS-5 Debugger. MDK-ARM also allows the use of UART for `printf()` on microcontroller boards.

## 4.2 Memory layout

Systems containing ARM processors can have a wide variety of memories and memory hierarchies. Common memories include:

- On-chip Tightly Coupled Memory (TCM).
- L1 and L2 instruction and data caches.
- Flash memory.
- ROM.

You must compile your application to use memory correctly and efficiently. ARM compiler supports scatter-loading, which lets you match code and data with the most appropriate type of memory. See the *ARM Compiler toolchain Linker Reference* for more information.

## 4.3 Command-line options

Using the ARM Compiler, armcc, you can build CoreMark to run in a bare-metal or semihosted environment.

To optimize for execution speed, use the -Otime, -O3, and --loop\_optimization\_level=2 options.

To reduce code size, you can use microlib instead of the standard C library.

### Speed-optimized

```
armcc -c -W <cflags> --cpu=name -O3 -Otime --loop_optimization_level=2
-I./ -Isimple -DITERATIONS=0 -DSEED_METHOD=SEED_ARG
-DCOMPILER_FLAGS=\"<cflags> --cpu=name -O3 -Otime --loop_optimization_level=2\"
core_main.c core_list_join.c core_matrix.c core_state.c core_util.c
simple/core_portme.c
armlink core_main.o core_list_join.o core_matrix.o core_state.o core_util.o
core_portme.o -o coremark.axf
```

The compiler switches used here are:

- c                Performs the compilation and link steps separately.
- W                Disables all warnings. This is optional.
- <cflags>          Any additional compiler flags you require.
- cpu=name       Specifies the name of the target processor, for example --cpu=Cortex-M4.
- O3               Applies maximum optimization.
- Otime            Optimizes for time.
- I./ -Isimple     Lists the directories to search for source files.
- loop\_optimization\_level=2  
                  Specifies that the compiler performs high-level optimization, including aggressive loop optimization. This option is usually best for performance.
- DSEED\_METHOD=SEED\_ARG  
                  Specifies that randomization seeds are passed as command-line arguments at runtime, rather than being automatically generated.

`-DITERATIONS=0`

Specifies that the number of iterations is passed as a command-line argument at runtime.

———— **Note** ————

If you prefer, you can specify the number of iterations here rather than passing the iteration count as an argument. See [Running CoreMark on page 10](#) for more information.

`-DCOMPILER_FLAGS=\"<cflags> --cpu=name -O3 -Otime --loop_optimization_level=2\"`

Specifies the compiler flags, as provided to the compiler earlier on the command line. This allows the compiler flags to be included in the output of the CoreMark results.

This example uses UNIX-style escaped quotes. On Windows, use

`-DCOMPILER_FLAGS=\"\"<cflags> --cpu=name...\"\"`

### Speed-optimized, with microlib

```
armcc -c -W <cflags> --cpu=name -O3 -Otime --loop_optimization_level=2
--library_type=microlib -I./ -Isimple -DITERATIONS=0 -DSEED_METHOD=SEED_ARG
-DCOMPILER_FLAGS=\"<cflags> --cpu=name -O3 -Otime --loop_optimization_level=2
--library_type=microlib\"
core_main.c core_list_join.c core_matrix.c core_state.c core_util.c
simple/core_portme.c
armlink core_main.o core_list_join.o core_matrix.o core_state.o core_util.o
core_portme.o -o coremark.axf
```

The compiler and linker switches used here are the same as in the speed-optimized example, with the following difference:

`--library_type=microlib`

Links with the size-optimized library.

### Size-optimized, with microlib

```
armcc -c -W <cflags> --cpu=cuname -O3 -Ospace -I./ -Isimple -DITERATIONS=0
-DSEED_METHOD=SEED_ARG -DCOMPILER_FLAGS=\"<cflags> --cpu=cuname -O3 -Ospace\"
--library_type=microlib --split_sections core_main.c core_list_join.c
core_matrix.c core_state.c core_util.c simple/core_portme.c
armlink core_main.o core_list_join.o core_matrix.o core_state.o core_util.o
core_portme.o -o coremark.axf
```

The compiler and linker switches used here are the same as in the speed-optimized example, with the following differences:

`--library_type=microlib`

Links with the size-optimized library.

`-Ospace` Optimizes for size.

`--split_sections`

Generates one ELF section for each function in the source file.

Compilers usually collect functions and data together and emit one section for each category. The linker can only eliminate a section if it is entirely unused. The `--split_sections` option ensures that each function is in a separate section, so that the linker can eliminate all unused functions.

## 5 Running CoreMark

CoreMark is a small benchmark that must be run multiple times to obtain reproducible numbers.

To minimize the variation caused by inconsistent processor states, ARM recommends you perform two validation runs followed by at least ten profile runs, then calculate the average for the profile runs.

CoreMark requires a minimum execution time of 10 seconds for a valid result. ARM recommends a minimum execution time of 30 seconds. Execution time varies with the processor frequency. As guidance, 15 000 iterations on a Cortex-M4 processor running at 168MHz yields an execution time of about 30 seconds.

ARM recommends the following process for deciding on a suitable iteration count:

1. Set the iteration count to 10000 iterations initially.
2. If this iteration count produces an execution time for a single profile run less than 30 seconds, increase the number of iterations until the CoreMark execution time is at least 30 seconds.
3. Run ten profile runs with this iteration count, and calculate the average CoreMark score.
4. Increase both the iteration count and the number of profile runs until the Coremark score converges to a value. That is, subsequent runs show a small standard deviation.

Table 2 shows an example of this process.

**Table 2 Increasing iterations and number of runs until results converge**

| Number of iterations | Single profile run time | Number of profile runs | Average CoreMark score | Notes                                                                |
|----------------------|-------------------------|------------------------|------------------------|----------------------------------------------------------------------|
| 10000                | 25s                     | 1                      | -                      | Check run time: minimum not met.                                     |
| 12000                | 32s                     | 1                      | -                      | Check run time: minimum OK.                                          |
| 12000                | -                       | 10                     | 230                    | First profile run.                                                   |
| 12000                | -                       | 10                     | 290                    | Run again. Results not yet converged.                                |
| 14000                | -                       | 12                     | 320                    | Increase iterations and number of runs. Results still not converged. |
| 16000                | -                       | 14                     | 321                    | Results have converged.                                              |

The arguments used for the runs are as follows:

- One validation run with arguments: `0x0 0x0 0x66 ITER_VALIDATION 7 1 2000`
- One validation run with arguments: `0x3415 0x3415 0x66 ITER_PROFILE 7 1 2000`
- 10 profile runs with arguments: `0x0 0x0 0x66 ITER_PROFILE 7 1 2000`

Where:

- `ITER_VALIDATION` is the number of iterations for a validation run. For example, 2000.
- `ITER_PROFILE` is the number of iterations for a profile run. For example, 20000.

---

**Note**

---

If you prefer, you can specify the number of iterations at compile time rather than passing the iteration count as an argument. Use the compiler options `-DITERATION=iterations` `-DPERFORMANCE_RUN=1`, where *iterations* is the required number of iterations.

---

## 5.1 CoreMark output

The CoreMark output shown in [Example 1](#) is from a development board containing a Cortex-M4 processor running at 168MHz.

### Example 1 CoreMark output

---

```

2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks        : 3512687483
Total time (secs): 20.908854
Iterations/Sec     : 478.266287
Iterations         : 10000
Compiler version   : ARM C/C++ Compiler, 5.03 [Build 24]
Compiler flags     : -O3 -Otime --loop_optimization_level=2
Memory location    : STACK
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x988c
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 478.266287 / ARM C/C++ Compiler, 5.03 [Build 24] -O3 -Otime
--loop_optimization_level=2 / STACK

```

---

## 6 Measurement characteristics

The CoreMark benchmark number is the number of iterations per second. In the output shown in [Example 1 on page 11](#), the CoreMark number is 478.266287.

A commonly reported figure is CoreMarks / MHz. For [Example 1 on page 11](#), this is calculated as the CoreMark number (478.266287) divided by the processor speed in MHz (168):

$$478.266287 / 168 = 2.8468$$

### 6.1 Measuring code and image size

When measuring the code size of an executable, you must decide whether to include or exclude libraries. For example, if an embedded device runs only a single application, then the total ROM size required for the device includes both the application and libraries. However, if the device runs multiple applications then the measurement might exclude any shared libraries that the executable uses.

You can use the `fromelf` utility, provided with the ARM Compiler toolchain, to compute the code size of an executable. For example, the following command prints the code size of CoreMark, including the standard C library:

```
fromelf -z coremark.axf
```

where `coremark.axf` is the CoreMark executable compiled using the commands described in [Command-line options on page 8](#). The `-z` option prints the code and data size information. The output is shown in [Example 2](#).

#### Example 2 fromelf output for CoreMark including the C library

---

```
** Object/Image Component Sizes
Code (inc. data)  RO Data  RW Data  ZI Data  Debug  Object Name
14292      1262      724      160     65376   2548   coremark.axf
14292      1262      724      160         0     0   ROM Totals for coremark.axf
```

---

The first column shows the application code size in bytes that includes the size of inline data shown in the second column. The inline data is located in the code section and comprises literal pools, case-branch offset tables, and short strings. The third, fourth, and fifth columns show the size of the read-only data, read-write data, and zero initialized data respectively of CoreMark and the library. The read-only data comprises constant strings and constant variables, while the read-write data includes initialized variables. The zero-initialized data comprises uninitialized global variables.

This example shows the total instruction size in bytes of CoreMark including the library (code minus inline data) is:

$$14292 - 1262 = 13030$$

You can use this output to calculate the ROM and RAM requirements of CoreMark. The total ROM footprint including the library (code plus RO data plus RW data) is:

$$14292 + 724 + 160 = 15176$$

The total RAM footprint (not including the stack and heap), including the library, is (RW data plus ZI data):

$$160 + 65376 = 65536$$

To calculate the code size of CoreMark excluding the library, sum the code sizes from all object files for the application. This size includes the vector table, CMSIS code, and C startup code.

Use the following command to calculate the code size of CoreMark excluding the library code:

```
fromelf -z core_main.o core_list_join.o core_matrix.o core_state.o core_util.o
        core_portme.o
```

This produces the output shown in [Example 3](#):

### Example 3 fromelf output for CoreMark excluding the C library

---

| ** Object/Image Component Sizes |         |         |         |       |             |                  |
|---------------------------------|---------|---------|---------|-------|-------------|------------------|
| Code (inc. data)                | RO Data | RW Data | ZI Data | Debug | Object Name |                  |
| 2420                            | 924     | 193     | 42      | 0     | 140         | core_main.o      |
| 2360                            | 6       | 0       | 0       | 0     | 492         | core_list_join.o |
| 2148                            | 0       | 0       | 0       | 0     | 444         | core_matrix.o    |
| 1096                            | 18      | 128     | 64      | 0     | 152         | core_state.o     |
| 484                             | 0       | 0       | 0       | 0     | 256         | core_util.o      |
| 296                             | 68      | 0       | 48      | 4096  | 336         | core_portme.o    |

---

This example shows that the total instruction size in bytes of CoreMark excluding the library (code minus data) is:

$$(2420 + 2360 + 2148 + 1096 + 484 + 296) - (924 + 6 + 18 + 68) = 7788$$

The total instruction size in bytes of CoreMark including the library (code inc data) is:

$$(2420 + 2360 + 2148 + 1096 + 484 + 296) = 8804$$

The ROM footprint excluding the library (code plus RO Data plus RW Data) is:

$$(2420 + 2360 + 2148 + 1096 + 484 + 296) + (193 + 128) + (42 + 64 + 48) = 9279$$

The total RAM footprint excluding the library (RW Data plus ZI Data) is:

$$(42 + 64 + 48) + (4096) = 4250$$

### Comparing code size

To determine how effective the compiler is at generating code (for example, to compare it with other compilers), the following key metrics used are:

- Total ROM footprint including the library (code plus RO data plus RW data).
- Total instruction size excluding the library and related data, such as inline strings and literal pools (code minus data).
- Total instruction size in bytes of CoreMark including the library (code inc data).

For deeply embedded, size-constrained applications, the total ROM footprint including the library (preferably microlib) is a good metric to evaluate the effectiveness of the compilation toolchain.

## 7 Appendix A

A sample implementation of the clock() function using Cortex-M4 performance counters:

```

/* Copyright (C) 2011,2013 ARM Ltd. All rights reserved. */
/* A small implementation of the clock function using the performance counters
 * available on the Cortex-M4 processor. */

/* The cpu cycle counter (SysTick) in Cortex-M4 counts down from 0xFFFFF and an
 * overflow interrupt is generated when the value reaches 0. The clock function
 * relies on the startup code to install an interrupt handler to catch the
 * interrupt and update the overflow counter. */

#include <stdint.h>
#include <time.h>

/* Compile with -DCMSIS to use the CMSIS SysTick definition, otherwise
 * we use a compatible definition. */
#ifdef CMSIS

/* Replace this with your device-specific include file */
#include <ARMCM4.h>

#else

/* Set the clock frequency appropriately for your device. */
uint32_t SystemCoreClock = (20*1000000);

/* SysTick registers */
typedef struct
{
    volatile uint32_t CTRL; /* Offset: 0x000 (R/W) SysTick Control and Status Register */
    uint32_t LOAD; /* Offset: 0x004 (R/W) SysTick Reload Value Register */
    volatile uint32_t VAL; /* Offset: 0x008 (R/W) SysTick Current Value Register */
    const uint32_t CALIB; /* Offset: 0x00C (R/ ) SysTick Calibration Register */
} SysTick_Type;
#define SCS_BASE (0xE000E000UL)
#define SysTick_BASE (SCS_BASE + 0x0010UL)
#define SysTick ((SysTick_Type *) SysTick_BASE)

/* SysTick CSR register bits */
#define SysTick_CTRL_COUNTFLAG_Msk (1 << 16)
#define SysTick_CTRL_CLKSOURCE_Msk (1 << 2)
#define SysTick_CTRL_TICKINT_Msk (1 << 1)
#define SysTick_CTRL_ENABLE_Msk (1 << 0)

#endif

static volatile unsigned int systick_overflows = 0;

/* This function is called by the SysTick overflow interrupt handler. The
 * address of this function must appear in the SysTick entry of the vector
 * table. */
extern __irq void SysTick_Handler(void)
{
    systick_overflows++;
}

static void reset_cycle_counter(void)
{
    /* Set the reload value and clear the current value. */
    SysTick->LOAD = 0x00ffffff;
    SysTick->VAL = 0;
    /* Reset the overflow counter */
}

```

```

    systick_overflows = 0;
}

static void start_cycle_counter(void)
{
    /* Enable the SysTick timer and enable the SysTick overflow interrupt */
    SysTick->CTRL |=
        (SysTick_CTRL_CLKSOURCE_Msk |
         SysTick_CTRL_ENABLE_Msk |
         SysTick_CTRL_TICKINT_Msk);
}

static uint64_t get_cycle_counter(void)
{
    unsigned int overflows = systick_overflows;
    /* A systick overflow might happen here */
    unsigned int systick_count = SysTick->VAL;
    /* check if it did and reload the low bit if it did */
    unsigned int new_overflows = systick_overflows;
    if (overflows != new_overflows)
    {
        /* This suffices as long as there is no chance that a second
        overflow can happen because new_overflows was read */
        systick_count = SysTick->VAL;
        overflows = new_overflows;
    }
    /* Recall that the SysTick counter counts down. */
    return (((uint64_t)overflows << 0x18) + (0x00FFFFFF - systick_count));
}

extern clock_t clock(void)
{
    return (clock_t) ((get_cycle_counter() * CLOCKS_PER_SEC) / SystemCoreClock);
}

/* The C library initialization code calls _clock_init() to initialize
 * anything that is required for clock() to work. Here we do this by
 * starting the systick timer. */
extern void _clock_init(void)
{
    reset_cycle_counter();
    start_cycle_counter();
}

```