Dhrystone Benchmarking for ARM Cortex Processors

Application Note 273

Released on: 20 July 2011

ARM®

ARM DAI 0273A (ID072111)

Dhrystone Benchmarking for ARM Cortex Processors
Application Note 273

Release Information

Table 1 Change history

Date	Issue	Confidentiality	Change
July 2011	A	Non-Confidential	First release

Proprietary Notice

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

The first version of the Dhrystone benchmark program was written in 1984 by Reinhold Weicker to measure the integer performance of processors and compilers. It became popular because it was very simple, could be quickly compiled and run and computed a single result that was more representative than the MIPS numbers reported by processor vendors. In 1988, version 2 of Dhrystone was released in the Ada, C and Pascal languages.

Today, Dhrystone is no longer seen as a representative benchmark and ARM does not recommend using it. In general, it has been replaced by more complex processor benchmarks such as SPEC and CoreMark.

ARM quotes figures for the C version of Dhrystone 2.1.

Dhrystone performance is calculated using the formula:

Dhrystones per second = number of runs / execution time.

For the result to be valid, the Dhrystone code must be executed for at least two seconds, although longer is generally better, and ARM recommends at least 20 seconds.

This application note describes how to build and run Dhrystone on both bare-metal and OS-hosted systems. If you have any problems or questions about benchmarking ARM Cortex processors, contact ARM support, http://www.arm.com/support.

2 Downloading the source

The official Dhrystone source was released on the Usenet forum in 1988. There is no central repository containing it. You can access it from the following locations:

- *Dhrystone 2.1 (the original Usenet posting)*, http://groups.google.com/group/comp.arch/browse_thread/thread/b285e89dfc1881d3/068aac05d4042d54?lnk=gst&q=dhrystone+2.1#068aac05d4042d54
- *Netlib.org*, http://www.netlib.org/benchmark/dhry-c.

3 Compiling Dhrystone

Dhrystone consists of two C files and one header file: `dhry_1.c`, `dhry_2.c` and `dhry.h`.

When compiling Dhrystone, the following compiler optimizations are prohibited:

- function inlining
- multifile compilation.

3.1 Library functions required by Dhrystone

Dhrystone requires the presence of the following C library functions:

- `scanf()` and `printf()` to read the iteration count input by the user and to report back the benchmarking results
- `clock()` to compute the execution time of the benchmark
- `strcpy()` and `strcmp()` to perform string assignment and comparison.

The standard C library is included with the ARM Compiler toolchain. The Keil MDK toolkit uses the ARM Compiler toolchain and also includes an alternative, size-optimized C library called microlib.

Library functions for an OS-hosted device

When compiling Dhrystone for an OS-hosted device, all required functions can be provided by the standard C library, or you can use your own versions of the `strcpy()` and `strcmp()` functions.

Library functions for a bare-metal or semihosted system

When compiling Dhrystone for a bare-metal system then in general you must provide the startup code and retargeted `scanf()`, `printf()` and `clock()` functions. You can use the `strcmp()` and `strcpy()` functions from the standard C library or your own versions.

Most modern ARM processors include performance counters. They can be programmed to count the number of processor cycles, which can be used to accurately compute the elapsed time. The `clock()` function must be retargeted to use this feature. A sample implementation of such a retargeted `clock()` function for a Cortex-M3 processor can be found in *Appendix A* on page 11.

In the absence of performance counters and a retargeted `clock()` function, the semihosted `clock()` function from the standard C library is used. A debugger capable of supporting semihosting, such as the ARM RealView Debugger (RVD) or DS-5 Debugger must be connected.

3.2 Command-line options

Using the ARM compiler, `armcc`, you can build Dhrystone to run in a bare-metal or semihosted environment with the `-Otime` option to optimize it for execution speed. If your priority is code size rather than performance, use the `-Ospace` option instead. To further reduce code size, you can use microlib instead of the standard C library.

Speed-optimized

```
armcc -c -W --cpu=cpuname -O3 -Otime --no_inline --no_multifile -DMSC_CLOCK dhry_1.c dhry_2.c
armlink dhry_1.o dhry_2.o -o dhry.axf
```

The compiler switches used here are:

`-c` Performs the compilation and link steps separately.

`-W` Disables all warnings.

`--cpu=`*`cpuname`* Specifies the name of the target processor, for example `--cpu=Cortex-A8`.

`-O3` Applies maximum optimization.

`-Otime` Optimizes for time.

`--no_inline` Disables function inlining, because Dhrystone demands "No procedure merging".

`--no_multifile` Disables multifile compilation, because Dhrystone demands that the two source files be compiled independently.

`-DMSC_CLOCK` Ensures the C library function `clock()` is used for timing measurements.

`<other options>` Processor-specific tuning options.

Size-optimized

```
armcc -c -W --cpu=cpuname --thumb -O3 -Ospace --no_inline --no_multifile -DMSC_CLOCK dhry_1.c dhry_2.c
armlink dhry_1.o dhry_2.o -o dhry.axf
```

where *cpuname* is the target processor, for example `--cpu=Cortex-M3`.

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

`--thumb` Compiles for Thumb.

`-Ospace` Optimizes for size.

Size-optimized with microlib

```
armcc -c -W --cpu=cpuname --thumb -O3 -Ospace --no_inline --no_multifile -DMSC_CLOCK dhry_1.c dhry_2.c
armlink dhry_1.o dhry_2.o -o dhry.axf --library_type=microlib
```

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

`--library_type=microlib` Links with the size-optimized library.

4 Running Dhrystone

Dhrystone is a very small integer benchmark that must be run multiple times in order to obtain reproducible numbers. ARM recommends you run Dhrystone ten times with varying iteration counts and that each run takes at least 20 seconds.

In order to minimize the variation caused by inconsistent processor states, ARM recommends you disregard the first run and calculate the average for the remaining nine runs. In addition, when running Dhrystone on a bare-metal system, before each run you must initialize the processor by resetting, clearing and enabling the caches, TLBs, BTBs and other micro-architectural features such as branch prediction.

4.1 Dhrystone output

The Dhrystone output shown in Example 1 is from a development board containing a Cortex-M3 processor running at 18.5MHz.

Example 1 Dhrystone output

```
Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without 'register' attribute
Please give the number of runs through the benchmark: 1000000
Execution starts, 1000000 runs through Dhrystone
Execution ends
Final values of the variables used in the benchmark:
Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    1000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          65288
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          65288
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
```

```
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING
Microseconds for one run through Dhrystone:   24.6
Dhrystones per Second:                      40600.9
```

5 Measurement characteristics

DMIPS (Dhrystone MIPS) numbers are calculated using the formula:

DMIPS = Dhrystones per second / 1757.

The output shown in Example 1 on page 7 gives this result:

`40600.9 / 1757 = 23.11`.

A more commonly reported figure is DMIPS / MHz. For Example 1 on page 7, this is calculated as:

`23.11 / 18.5 = 1.25`.

5.1 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 should exclude libraries that are shared among the applications.

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 Dhrystone including the standard C library:

```
fromelf -z dhry.axf
```

where `dhry.axf` is the Dhrystone executable compiled using the commands described in *Command-line options* on page 5. The `-z` option prints the code and data size information. The output is shown in Example 2.

Example 2 `fromelf` output for Dhrystone including the library

```
** Object/Image Component Sizes
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
15412       2278        476         64      10536       4020   dhry.axf
15412       2278        476         64          0          0   ROM Totals for dhry.axf
```

The first column shows the application code size in bytes which 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 Dhrystone and 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 Dhrystone including the library (code minus inline data) is:

`15412 - 2278 = 13134`.

This `fromelf` output can also be used to calculate Dhrystone's ROM and RAM requirements. The total ROM footprint including the library (code plus RO data plus RW data) is:

`15412 + 476 + 64 = 15952`.

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

`64 + 10536 = 10600`.

To calculate the code size of Dhrystone excluding the library, sum the code sizes from both object files for the application. This approach is valid for Dhrystone because it has no unused functions or sections that would otherwise be removed by the linker. Additionally, in this example, the linker has not added any padding or interworking veneers.

The following command can be used to calculate the code size of Dhrystone excluding the library code:

```
fromelf -z dhry_1.o dhry_2.o
```

This produces the output shown in Example 3:

Example 3 `fromelf` output for Dhrystone excluding the library

```
** Object/Image Component Sizes
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
2660       1556          0         44      10200        360   dhry_1.o
256          16          0          0          0        388   dhry_2.o
```

This example shows that the total instruction size in bytes of Dhrystone excluding the library is:

`2660 + 256 - (1556 + 16) = 1344`.

The ROM footprint excluding the library is:

`2660 + 256 + 44 = 3004`.

The total RAM footprint excluding the library is:

`44 + 10200 = 10244`.

Comparing code size

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

- total instruction size excluding the library and related data, such as inline strings and literal pools
- ROM footprint excluding the library.

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.

6 Appendix A

A sample implementation of the `clock()` function using Cortex-M3 performance counters

```
/* Copyright (C) 2011 ARM Ltd. All rights reserved. */
/* A small implementation of the clock function using the performance counters
* available on the Cortex-M3 processor. */

/* The cpu cycle counter (SysTick) in Cortex-M3 counts down from 0xFFFFFF 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.  */

/* Set the clock frequency appropriately for your board. */
#define CPU_MHZ        (20*1000000)

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

/* SysTick registers */
/* SysTick control & status */
#define INITCPU_SYST_CSR     ((volatile unsigned int *)0xE000E010)
/* SysTick Reload value */
#define INITCPU_SYST_RVR     ((volatile unsigned int *)0xE000E014)
/* SysTick Current value */
#define INITCPU_SYST_CVR     ((volatile unsigned int *)0xE000E018)

/* SysTick CSR register bits */
#define INITCPU_SYST_CSR_COUNTFLAG (1 << 16)

#define INITCPU_SYST_CSR_CLKSOURCE (1 << 2)
#define INITCPU_SYST_CSR_TICKINT   (1 << 1)
#define INITCPU_SYST_CSR_ENABLE    (1 << 0)

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. */
    *INITCPU_SYST_RVR = 0x00ffffff;
    *INITCPU_SYST_CVR = 0;
    /* Reset the overflow counter */
    systick_overflows = 0;
}

static void start_cycle_counter(void)
{
    /* Enable the SysTick timer and enable the SysTick overflow interrupt */
    *INITCPU_SYST_CSR |=
        (INITCPU_SYST_CSR_CLKSOURCE |
         INITCPU_SYST_CSR_ENABLE |
         INITCPU_SYST_CSR_TICKINT);
}
```

```
static uint64_t get_cycle_counter(void)
{
    unsigned int overflows = systick_overflows;
    /* A systick overflow might happen here */
    unsigned int systick_count = *INITCPU_SYST_CVR;
    /* 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 since new_overflows was read */
        systick_count = *INITCPU_SYST_CVR;
        overflows = new_overflows;
    }
    /* Recall that the SysTick counter counts down. */
    return (((uint64_t)overflows << 18) + (0x00FFFFFF - systick_count));
}

static uint64_t cycle_counter_init_value;

extern clock_t clock(void)
{
    return (clock_t) (((get_cycle_counter() -
                        cycle_counter_init_value) *
                       CLOCKS_PER_SEC) / CPU_MHZ);
}

/* Microlib calls the _clock_init() function during library initialization to
 * set up the cycle_counter_init_value variable. The value of this variable is
 * then used as the baseline for subsequent calls to the clock() function. */
extern void _clock_init(void)
{
    reset_cycle_counter();
    start_cycle_counter();
    cycle_counter_init_value = get_cycle_counter();
}
```