Application Note 26

Benchmarking, Performance Analysis and Profiling

Document Number: ARM DAI 0026A
Issued: December 1995

ARM
Advanced RISC Machines

Proprietary Notice

Change Log

Issue	Date	By	Change
A	Dec 95	AP/EH	Created

Table of Contents

1 Introduction

This application note explains how to run benchmarks on the ARM processor, and shows you how to use the profiling facilities to help improve the size and performance of your code. It makes extensive use of the ARM Software Development Toolkit's example programs, and contains a number of practical exercises for you to follow. You should therefore have access to the toolkit's `examples` directory, and the ARM software tools themselves, while working through it.

When developing application software or comparing the ARM with another processor, it is often useful to measure:

- code and data sizes
- overall execution time
- time spent in specific parts of an application

Such information allows you to:

- compare the ARM's performance against other processors in benchmark tests
- make decisions about the required clock speed and memory configuration of a projected system
- pinpoint where an application can be streamlined, leading to a reduction in the system's memory requirements
- identify performance-critical sections of code which you can then optimize, either by using a more efficient algorithm, or by rewriting in assembler

In this document we show you how to measure code size and execution time, and how to generate an execution profile to discover where the time is being spent in your application.

2 Measuring Code and Data Size

To measure the code size of an ARM image, use one of the following armlink options:

`-info sizes` which gives a breakdown of the code and data sizes of each object file or library member making up an image

`-info totals` which gives a summary of the total code and data sizes of all object files and all library members making up an image

The information provided by these options can be broken down into:

- code (or read-only) segment
- data (or read-write) segment
- debug data

Code (or read-only) segment

`code size` gives the code size, excluding any data which has been placed in the code segment (see `inline data`)

`inline data` reports the size of the read-only data included in the code segment by the compiler

Typically, this data contains the addresses of variables which are accessed by the code, plus any floating-point immediate values or immediate values that are too big to load directly into a register. It does not include inlined strings, which are listed separately (see `inline strings`).

`inline strings` shows the size of read-only strings placed in the code segment

The compiler puts such strings here whenever possible, because this reduces run-time RAM requirements.

`const` lists the size of any variables explicitly declared as `const`

These variables are guaranteed to be read-only and so are placed in the code segment by the compiler.

Data (or read-write) segment

`RW data` gives the size of read-write data

This is data which is read-write and which also has an initializing value. Read-write data consumes the displayed amount of RAM at runtime, but also requires the same amount of ROM to hold the initializing values which are copied into RAM on image startup.

`0-init data` shows the size of read-write data which is zero-initialized at image startup

Typically this contains arrays which are not initialized in the C source code. Zero-initialized data requires the displayed amount of RAM at runtime but does not require any space in ROM, since its initializing value is 0.

Note that at release 2.0 of the toolkit, only data items larger than 8 bytes are included in zero-initialized data.

Debug data

`debug data` reports the size of any debugging data if the files are compiled with the `-g` option

Note *There are totals for the debug data, even though the code has not been compiled for source level debugging because the compiler automatically adds information to an AIF file to allow no frame pointer debugging. See* *4.1 Compiler options on page 13.*

2.1 Calculating ROM and RAM requirements

Calculate the ROM and RAM requirements for your system as follows:

ROM `Code size` + `inline data` + `inline strings` + `const data` + `RW data`

RAM `RW Data` + `0-init data`

In more complex systems, you may require the code segment to be downloaded from ROM into RAM at runtime. Although this increases the system's RAM requirements, this could be necessary if—for example—RAM access times are faster than ROM access times and the execution speed of the system is critical.

2.2 Code and data sizes example : Dhrystone

The Dhrystone application is located in the `Examples` subdirectory of the ARM Software Development Toolkit. Copy the files into your working directory.

If you are using the command-line tools:

1 Compile the Dhrystone files, without linking:

```
armcc -c -DMSC_CLOCK dhry_1.c dhry_2.c
```

The compiler will produce a number of warnings, which you may ignore or suppress using the `-w` option. These are caused by the Dhrystone application being coded in K&R style C rather than ANSI C.

2 Perform the link stage, with the `-info totals` option to give a report on the total code and data sizes in the image, broken into separate totals for the object files and library files:

```
armlink -info totals dhry_1.o dhry_2.o armlib.32l -o dhry
```

(If `armlib.32l` is not in the current directory, you will need to refer to it by its full pathname.)

If you are using the Windows toolkit:

1. Load the Dhrystone project file `dhry.apm` into the ARM Project Manager.
2. Modify the project setting to give a release, little-endian, build using the ARM tools (rather than the Thumb tools). See the ARM Software Development Toolkit Windows Toolkit Guide (ARM DUI 0022) for details of how to modify a project file.
3. Click on the toolbar's **Rebuild All** icon. This compiles and links the project, automatically generating a summary of the total code and data sizes in the image.

Results

The results are shown in the following table:

	code size	inline data	inline strings	const data	RW data	zero-init data	debug data
Object totals	2268	28	1448	0	48	10200	64
Library totals	34400	400	736	128	700	1176	416
Grand totals	36668	428	2184	128	748	11376	480

Table 1: Code and data sizes results

Notes *You may obtain slightly different figures, depending on the version of the compiler, linker and library in use.*

3 Performance Benchmarking

3.1 Cycle counting

The ARMulator built into the debugger tracks the number of cycles consumed by the instructions executed during a program. This information can be read from two debugger variables:

`$statistics`	which gives the total number of each type of cycle consumed since execution of the application started
`$statistics_inc`	which gives the number of each type of cycle consumed since the previous time that `$statistics` or `$statistics_inc` was displayed by the debugger

3.2 Cycle counting example : Dhrystone

In this example, we determine the number of instructions executed by the main loop of the Dhrystone application, plus the number of cycles consumed. A suitable place to break within the loop is the invocation of function `Proc_5`.

If you are using the command-line tools:

1 Load the executable, produced in *2.2 Code and data sizes example : Dhrystone* on page 4, into the debugger:

```
armsd dhry
```

2 Set a breakpoint on the first instruction of `Proc_5`:

```
break @Proc_5
```

When prompted, request at least two runs through Dhyrstone.

3 Once the breakpoint at the start of `Proc_5` has been reached, display the system variable `$statistics` (which gives the total number of instructions and cycles taken so far) and restart execution:

```
print $statistics
go
```

4 When the breakpoint is reached again, you can obtain the number of instructions and cycles consumed by one iteration:

```
print $statistics_inc
```

If you are using the Windows toolkit:

1 If you have not already done so, build the Dhrystone project as described in *2.2 Code and data sizes example : Dhrystone* on page 4.

2 Click on the **Debug** icon on the ARM Project Manager toolbar.

3 Locate function `Proc_5` by choosing **Low Level Symbols** from the View menu.

4 Double click on `Proc_5` to open the Disassembly window.

5 Toggle the breakpoint on `Proc_5` in the Disassembly window by clicking on the instruction, then clicking on **Toggle Breakpoint** on the toolbar.

6 Click on **Go** to begin execution.

When prompted, request at least two runs through Dhrystone.

7 When the breakpoint set at `main` is reached, click on **Go** again to begin execution of the main application.

8 Once the breakpoint at `Proc_5` is reached, choose **Debugger Internals** from the View menu.

9 Double click on the `statistics_inc` field to open the Information window.

10 Click on **Go**. When the breakpoint at `Proc_5` is reached again, the contents of the `statistics_inc` field will be updated to reflect the number of instructions and cycles consumed by one iteration of the loop.

Results

The results are shown in the following table:

Instructions	S-cycles	N-cycles	I-cycles	C-cycles	F-cycles
358	427	188	64	0	0

Table 2: Cycle counting results

S-cycles	sequential cycles. The CPU requests transfer to or from the same address, or from an address which is a word or halfword after the preceding address
N-cycles	nonsequential cycles. The CPU requests transfer to or from an address which is unrelated to the address used in the preceding cycle
I-cycles	internal cycles, ie. the CPU does not require a transfer because it is performing an internal function
C-cycles	coprocessor cycles
F-cycles	fast clock cycles for cached processors (FCLK)

Note *You may obtain slightly different figures, depending on the version of the compiler, linker and library in use.*

3.3 Real time simulation

The ARMulator also provides facilities for real time simulation. To carry out such a simulation, the ARMulator needs to know:

- the type and speed of the memory attached to the processor
- the speed of the processor

As it executes your program, the ARMulator counts the total number of clock ticks taken. This allows you to determine how long your application would take to execute on real hardware.

3.3.1 Reading the simulated time

When performing a simulation, the ARMulator keeps track of the total time elapsed. This value may be read either by the simulated program or by the debugger.

Reading the simulated time from assembler

To read the simulated clock from an assembly language program use SWI 0x61 (`SWI_Clock`).

Reading the simulated time from C

From C, use the standard C library function `clock()`, which returns the number of elapsed centiseconds.

Reading the simulated time from the debugger

The internal variable `$clock` contains the number of microseconds since simulation started. To display this value, use the command:

```
Print $clock
```

if you are using armsd, or choose **Debugger Internals** from the View menu if you are using the ARM Debugger for Windows.

3.3.2 MAP files

The type and speed of memory in a simulated system is detailed in a *map* file. This defines the number of regions of attached memory, and for each region:

- the address range to which that region is mapped
- the data bus width in bytes
- the access time for the memory region

armsd expects the map file to be in the current working directory under the name `armsd.map`.

The ARM Debugger for Windows will accept a map file of any name, provided that it has the extension `.map`. You must add the map file to the project with which it is associated using the ARM Project Manager—see the ARM Software Development Toolkit Windows Toolkit Guide (ARM DUI 0022).

Format of a MAP file

The format of each line is:

`start size name width access read-times write-times`

where:

`start` is the start address of the memory region in hexadecimal, eg. `80000`.

`size` is the size of the memory region in hexadecimal, eg. `4000`.

name	is a single word which can be used to identify the memory region when the memory access statistics are displayed. This name is of no significance to the debugger, so you can use any name, but to ease readability of the memory access statistics, give a descriptive name such as `SRAM`, `DRAM`, `EPROM`.
width	is the width of the data bus in bytes (ie. `1` for an 8-bit bus, `2` for a 16-bit bus or `4` for a 32-bit bus)
access	describes the type of access which may be performed on this region of memory. The `r` is for read-only, `w` for write-only, `rw` for read-write, or `-` for no access. The character '*' may be appended to the access to describe a Thumb-based system which uses a 32-bit data bus but which has a 16-bit latch to latch the upper 16-bits of data so that a subsequent 16-bit sequential access may be fetched directly out of the latch.
read-times	describes the nonsequential and sequential read times in nanoseconds. These should be entered as the nonsequential read access time followed by `/` (slash), followed by the sequential read access time. Omitting the `/` and using only one figure indicates that the nonsequential and sequential access times are the same. **Note:** The times entered should not simply be the speed quoted on top of a memory chip, but should have a 20–30 ns signal propagation time added to them.
write-times	describes the nonsequential and sequential write times. The format is identical to that of read times.

Examples are given below.

Example 1

```
0 80000000 RAM 4 rw 135/85 135/85
```

This describes a system with a single contiguous section of RAM from 0 to 0x7fffffff with a 32-bit data bus, read-write access and N and S access times of 135ns and 85ns respectively.

This is typical of a 20MHz PIE (Platform Independent Evaluation) card. Note that the N-cycle access time is one clock cycle longer than the S-cycle access time. For a faster system a smaller N-cycle access time should be used. For example, for a 33MHz system the access times would be defined as `115/85 115/85`.

Example 2—clock speed 20MHz

```
0 80000000 RAM 1 rw 150/100 150/100
```

This describes a system with the same single contiguous section of memory, but with an 8-bit external data bus and slightly faster access times.

Note *If you are using release 2.0.0 or 2.0.1 of the toolkit, ensure that you specify access times which are multiples of the clock period when describing 8-bit or 16-bit memory regions. For example, with a 20MHz clock, the access times need to be a multiple of 50ns (ie. 1/20 x 10^6). Similarly, at 33MHz the access times must be a multiple of 30ns. This means that example 2 above might actually represent, for instance, a system with real access times of 120/70. You will therefore need to modify the map file if you change the clock speed when using 8-bit or 16-bit memory.*

Example 3—clock speed 20MHz

```
00000000 8000     SRAM  4 rw 1/1 1/1
00008000 8000     ROM   2 r  100/100 100/100
00010000 8000     DRAM  2 rw 150/100 150/100
7fff8000 8000     Stack 2 rw 150/100 150/100
```

This describes a system with four regions of memory:

- A fast region of memory from 0 to 0x7fff with a 32-bit data bus.
- A slower region from 0x8000 to 0xffff with a 16-bit data bus. This is labelled ROM and contains the image code, and so is marked as read-only.
- Two sections of RAM, one from 0x10000 to 0x17fff which will be used for image data and one from 0x7fff8000 to ox7fffffff which will be used for stack data (the stack pointer is initialized to 0x80000000).

This would be typical of an embedded system with 32Kb on-chip memory, 32Kb external 16-bit ROM and 32Kb external DRAM which will be used for both the image data and the stack data. This is described above as two regions of memory, although in the final hardware these would be combined. This does not make any difference to the accuracy of the simulation.

Note that the SRAM region is given access times of 1nS. In effect this means that each access will take 1 clock cycle, as armsd rounds this up to the nearest clock cycle. However, specifying it as 1nS allows the same map file to be used for a number of simulations with differing clock speeds.

Note *To ensure accurate simulations, take care that all areas of memory which the image you are simulating is likely to access are described in the memory map.*

To ensure that you have described all areas of memory you think the image should access, you can define a single memory region which covers the entire address range as the map file's last line.

For example, to the above description you could add the line:

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

You can then detect if any reads or writes are occurring outside the regions of memory you expect using the `print $memory_statistics` command. This can prove a very useful debugging tool.

Reading the memory statistics

To read the memory statistics use the command:

```
Print $memory_statistics
```

The statistics will be reported in the following form

```
address  name  w acc R(N/S)  W(N/S) reads(N/S) writes(N/S) time (ns)

00000000 Dummy 4 -   1/1     1/1    0/0        0/0         0
7FFF8000 Stack 4 rw  135/85  135/85 2852/829   829/1456    833214
00008000 RO    4 r   70/70   70/70  12488/38069 38069/902   5788907
00000000 SRAM  4 rw  135/85  135/85 27/0       0/0         4050
```

`Print $memstats` is a shorthand version of `Print $memory_statistics`.

Processor clock speed

The debugger also needs details of the clock speed of the processor being simulated. In armsd, this is set by the command-line option `-clock` *`value`*. The value is presumed to be in Hz unless Mhz is specified.

In the ARM Debugger for Windows, the clock speed is set by configuring the debugger. This is done by choosing **Configure Debugger -> Armulator** from the Options menu. The value entered in the dialog box should be specified in Mhz.

3.4 Real time simulation example : Dhrystone

To work through this example, you need to create a map file. (If one exists in the files you copied from the toolkit directory, edit it to match the one shown here.) Call it `armsd.map`.

```
00000000 80000000 RAM 4 RW 135/85 135/85
```

This describes a system with a single contiguous section of memory 0x80000000 bytes in length, labelled as RAM, starting at address 0x0, with a 32-bit (4-byte) data bus, with both read and write access, and read and write access times of 135 nanoseconds nonsequential and 80 nanoseconds sequential.

If you are using the command-line tools:

1. Load the executable produced in *2.2 Code and data sizes example : Dhrystone* on page 4 into the debugger, telling the debugger that the processor is clocked at 20Mhz:

   ```
   armsd -clock 20Mhz dhry
   ```

   As the debugger loads, you will be able to see the information about the memory system that the debugger has obtained from the `armsd.map` file.

2. Begin execution:

   ```
   go
   ```

3. When requested for the number of dhrystones, enter 30000.

4. When the application completes, record the number of Dhrystones per second reported. This is your performance figure.

If you are using the Windows toolkit:

You first need to associate the `armsd.map` file with the Dhrystone project:

1. Choose **Edit** from the Project menu.
2. Add `armsd.map` to the project by clicking on **Add** and selecting **map files** from the **List files of type** field.
3. Choose **armsd.map** and click on **OK**.

The association is now set up, and you can run the program.

1. Start up ARM Debugger for Windows by clicking on the **Debug** icon. If a dialog box prompts you to save the changes to the project file, click on **Yes**.
2. Set up the debugger to run at the required clock speed by choosing **Configure Debugger -> ARMulator** from the Options menu.
3. Change the clock speed to read 20Mhz and click on **OK**.
4. Click on the **Reload** icon on the Toolbar.
5. Click on the **Go** button to begin execution, and again when the breakpoint on `main` is hit.
6. When requested for the number of dhrystones, enter 30000.
7. When the application completes, record the number of Dhrystones per second reported. This is your performance figure.

Once the debugger is configured to emulate a processor of the required clock speed (in this case 20Mhz), you can repeat the simulation by clicking on **Execute** rather than **Debug** in the ARM Project Manager.

Result: 13452.9 Dhrystones per second

Note *You may obtain a slightly different figure, depending on the version of the compiler, linker and library in use.*

3.5 Reducing the time required for simulation

You may be able to significantly reduce the time taken for a simulation by dividing the specified clock speed by a factor of ten and multiplying the memory access times by the corresponding factor of ten. Take the time reported by the `clock()` function (or by `SWI_Clock`) and divide by the same factor of ten.

The reason this works is because the simulated time is recorded internally in nanoseconds, but `SWI_Clock` only returns centiseconds. Therefore, dividing the clock speed by ten shifts digits from the nanosecond count into the centisecond count, allowing the same level of accuracy but taking only one tenth the time to simulate.

4 Improving Performance and Code Size

4.1 Compiler options

The ARM C compiler has a number of command-line options which control the way in which code is generated. You can find a full list in the C Compiler chapter of the ARM Software Development Toolkit Reference Manual (ARM DUI 0020).

By default, the ARM C compiler is highly optimizing. None of the optimizations carried out are dangerous. The code produced from your source will be balanced for a compromise of code size versus execution speed. However, there are a number of compiler options which can affect the size and/or the performance of generated code. These may be used both individually or combined to give the required effect.

`-g` turns on source level code debugging. This option severely impacts the size and performance of generated code, since it turns off all compiler optimizations. Use it only when carrying out source level debugging of your code, and never enable it for a release build.

`-Ospace` optimizes for code size at the expense of performance

`-Otime` optimizes for performance at the expense of size

Note that `-Ospace` and `-Otime` are complementary. They can be used together on different parts of a build. For example, `-Otime` could be used on time-critical source files, with `-Ospace` being used on the remainder.

`-zpj0` disables crossjump optimization. Crossjump optimization is a space-saving strategy whereby common sections of code at the end of each element in a `switch()` statement are identified and commoned together, each occurrence of the code section being replaced with a branch to the commoned code section. However, this optimization can lead to extra branches being executed which may decrease performance, especially in interpreter-like applications which typically have large `switch()` statements. Use the `-zpj0` option to disable this optimization if you have a time-critical `switch()` statement.

Alternatively, you can use:

```
#pragma nooptimise_crossjump
```

before the function containing the `switch()` and:

```
#pragma optimise_crossjump
```

after it.

`-apcs /nofp`

By default, armcc generates code which uses a dedicated frame pointer register. This register holds a pointer to the stack frame and is used by the generated code to access a function's arguments. By specifying `-apcs /nofp` on the command line, you can force armcc to generate code which does not use a frame pointer, but which accesses the function's arguments via offsets from the stack pointer.

This means that function entry is simplified (saves two instructions) and a register is freed up for use as a work register, so your code should be smaller and run more quickly. However to take full advantage of this, you need to recompile your library.

Note: tcc never uses a frame pointer, so this option does not apply when compiling Thumb code.

`-apcs /noswst`

By default, armcc generates code at the head of each function which checks that the stack has not overflowed. This code can contribute several percent to the code size, so it may be worthwhile disabling this option with `-apcs /noswst`.

Again this means that function entry is simplified, saving a compare and a conditional branch per non-leaf function, and a register is freed up for use as a work register, improving both code size and execution speed. You need to recompile the library to take full advantage of this.

Be careful to ensure that your program's stack is not going to overflow, or that you have an alternative stack checking mechanism such as an MMU-based check.

Note: tcc has stack checking disabled by default.

`-pcc`

The code generated by the compiler can be slightly larger when compiling with the `-pcc` switch. This is because of extra restrictions on the C language in the ANSI standard which the compiler can take advantage of when compiling in ANSI mode.

If your code will compile in ANSI mode, do not use the `-pcc` switch. An example of this is with the Dhrystone application which, although written in old-style K&R C, compiles more efficiently in ANSI mode, even though this causes a number of warning messages to be generated.

`-ARM7T`

This option applies to armcc only.

By default, armcc generates code which is suitable for running on processors that implement ARM Architecture 3 (eg. ARM6, ARM7). If you know that the code is going to be run on a processor with halfword support, you can use the `-ARM7T` option to instruct the compiler to use the ARM Architecture 4 halfword and signed byte instructions. This can result in significantly improved code density and performance when accessing 16-bit data.

4.2 Other improvements

You can make further improvements to code size and performance in addition to those achieved by good use of compiler options by altering the code to take advantage of the ARM's features.

Use of shorts

ARM cores implementing an ARM Architecture earlier than version 4 do not have the ability to directly load or store halfword quantities (or *shorts*). This has an effect on code size. Broadly speaking, code generated for Architecture 3 that makes use of shorts will be larger than equivalent code which only performs byte or word transfers. Storing a short is particularly expensive, as the ARM must make two byte stores. Similarly, loading a short requires a word load, followed by shifting out the unwanted halfword.

If your core has halfword support, tell the compiler using the `-ARM7T` option discussed in *4.1 Compiler options* on page 13. This will ensure that the resulting code contains the Architecture 4 halfword instructions.

If you are writing or porting for cores that do not have halfword support, you should ideally minimize the use of shorts. However this is sometimes impossible—for instance, when porting C programs from x86 or 68k architectures, which frequently make heavy use of them. If the code has been written with portability in mind, all you may have to do is change a `typedef` or `#define` to use `int` instead of `short`. Where this is not the case, you may have to make some functional alterations to the code.

You may be able to establish the extent of code size increase caused by using shorts by compiling the code with:

```
armcc -Dshort=int
```

which preprocesses all instances of `short` to `int`. Be aware that although it may compile and link correctly, code created with this option may well not function as expected.

Whatever your approach, you will need to weigh the change in code size against the opposite change in data size.

The program below illustrates the effect of using shorts, integers and the `-ARM7T` option on code and data size.

```
typedef short number;
#include <stdio.h>
      number array [2000];
      number loop;
      int main()
      {
            for (loop=0; loop < 2000; loop++)
                  array[loop] = loop;
            return 0
      }
```

The results of compiling the program with all three options are shown in the following table:

	code size	inline data	inline strings	const data	RW data	O-init data	debug data
short	76	8	0	0	4	4000	64
short -ARM7T	60	8	0	0	4	4000	64
int	44	8	0	0	4	8000	0

Table 1: Object code and data sizes

Floating-point

The standard ARM core does not have inbuilt floating-point hardware, and while it is possible to add floating-point hardware by making use of the coprocessor interface, this is not a common approach.

There are three standard methods for handling floating-point operations:

1 Use floating-point instructions:

```
armcc -apcs /hardfp
```

Code compiled with this option will work either with Floating Point Accelerator (FPA) hardware, or the Floating Point Emulator (FPE), which is a software emulation of the FPA.

If no FPA hardware is attached to the system, attempts to execute a floating-point instruction will cause an undefined instruction exception. This exception is intercepted by the FPE (if it is present), which then emulates the instruction.

Note: This option is not available with tcc.

For more information, refer to Application Note 23, ARM Floating-point Emulator (ARM DAI 0023).

2 Use floating-point library calls:

```
armcc -apcs /softfp
```

This option causes library function calls to be inserted in your code where the floating point instructions would normally be. This gives a speed increase of the order two to three times over the FPE and is the preferred option for use with embedded cores which are never likely to have an FPA fitted with them.

This will also typically minimize the overall size of your software, since the FPE occupies about 26Kb, regardless of how much of it is used, while only those parts of the floating point library that are used will be included in the image.

Note: This is the default option for release 2.0 of the toolkit.

3 Rewrite critical floating-point routines to use fixed point integer arithmetic.

Other changes

- Modify performance-critical C source to compile efficiently on the ARM. See “Writing Efficient C for the ARM” in the ARM Software Development Toolkit Programming Techniques guide (ARM DUI 0021).
- Port small, performance-critical routines into ARM assembler.

  Use the compiler’s `-S` option to produce assembly output, and take this as a starting point for your own hand-optimized assembly language.

  An area in which you could make significant performance improvements is the use of Load and Store Multiple instructions in memory-intensive algorithms. The compiler cannot fully exploit these because of the complexity of register allocation.
- Replace small, performance-critical functions by macros.

5 Profiling

Profiling allows the time spent in specific parts of an application to be examined. It does not require any special compile time or link time options. The only requirement is that low-level symbols must be included in the image. These will be inserted by the linker unless it is instructed otherwise by the `-Nodebug` option.

Profiling data is collected by armsd or the ARM Debugger for Windows while the code is being executed. The data is saved to a file, which is then loaded into the ARM profiler, armprof, which displays the results. The profiler in turn generates a profile report.

5.1 Collecting profile data

The debugger collects profiling data while an application is executing. You can turn data collection on and off during execution, so that only the relevant sections of code are profiled:

- If you are using armsd, use the `profon` and `profoff` commands.
- If you are using the ARM Debugger for Windows, choose **Profiling -> Toggle Profiling** from the Options menu.

The format of the execution profile obtained depends on the type of information stored in the data file:

PC sampling	provides a flat profile of the percentage time spent in each function (excluding the time spent in its children)
Function call count	provides a call graph profile showing the percentage time spent in each function, plus the percentage time accounted for by calls to the children of each function and the percentage time allocated to calls from different parents

The debugger needs to know which profiling method you require when it loads the image. The default is PC sampling. To obtain a call graph profile:

- If you are using armsd, load the image with:

  ```
  load/callgraph image-file
  ```

- If you are using the ARM Debugger for Windows, choose **Profiling -> Call Graph Profiling** from the Options menu.

Then execute the code to collect the profile data.

5.2 Saving profile data

Once collection is complete, save the data to a file:

- If you are using armsd, issue the `profwrite` command:

  ```
  profwrite data-file
  ```

- If you are using the ARM Debugger for Windows, choose **Profiling -> Write to File** from the Options menu.

5.3 Generating the profile report

The ARM profiler utility, armprof, generates the profile report using the data in the file. The report is divided into sections, each of which gives information about a single function in the program.

A section's function (called the *current function*) is indicated by having its name start at the left-hand edge of the `Name` column. If call graph profiling is used, information will also be given about child and parent functions. Functions listed below the current function are its children—functions called by it. Those listed above the current function are its parents—functions that call it.

The columns in the report have the following meanings:

`Name` displays the function names. The current function in a section starts at the column's left-hand edge: parent and child functions are shown indented.

`cum%` shows the total percentage time spent in the current function plus the time spent in any functions which it called. It is only valid for the current function.

`self%` shows the percentage time spent in a function.

- For the current function, it shows the percentage time spent in this function.
- For parent functions, it shows the percentage time spent in the current function on behalf of the parent.
- For child functions, it shows the percentage time spent in this child on behalf of the current function.

`desc%` shows the percentage time spent in a function.

- For the current function, it shows the percentage time spent in children of the current function on the current function's behalf.
- For parent functions, it shows the percentage time spent in children of the current function on behalf of this parent.
- For child functions, it shows the percentage time spent in this child’s children on behalf of the current function.

`calls` shows the number of times a function is called.

- For the current function, it shows the number of times this function was called.
- For parent functions, it shows the number of times this parent called the current function.
- For child functions, it shows the number of times this child was called by the current function.

Below is a section of the output from armprof for a call graph profile:

```
Name         cum%      self%    desc%    calls
main         96.04%    0.16%   95.88%        0
  qsort                0.44%    0.75%        1
  _printf              0.00%    0.00%        3
  clock                0.00%    0.00%        6
  _sprintf             0.34%    3.56%     1000
  check_order          0.29%    5.28%        3
  randomise            0.12%    0.69%        1
  shell_sort           1.59%    3.43%        1
  insert_sort         19.91%   59.44%        1
-------------------------------------------------------------
  main                19.91%   59.44%        1
insert_sort 79.35%    19.91%   59.44%        1
  strcmp              59.44%    0.00%   243432
-------------------------------------------------------------
```

From the `cum%` column, you can see (in the `main` section) that the program spent 96.04 percent of its time in `main` and its children. Of this, only 0.16 percent of the time is spent in `main` (`self%` column), whereas 95.88 percent of the time is spent in functions called by `main` (`desc%` column). The call count for `main` is 0 because it is the top-level function, and is not called by any other functions, whereas the section for `insert_sort` shows that it made 243432 calls to `strcmp`, and that this accounted for 59.44 percent of the time spent in `strcmp` (the `desc%` column shows 0 in this case because `strcmp` does not call any functions).

5.4 Profiling example : sorts

The `sorts` application can be found in the `Examples` subdirectory of the ARM Software Development Toolkit. Copy the files into your working directory.

5.4.1 PC sampling information

If you are using the command-line tools:

1. Compile the `sorts.c` example program:
   ```
   armcc -Otime -o sorts sorts.c
   ```
2. Start armsd and load the executable:
   ```
   armsd sorts
   ```
3. Turn profiling on:
   ```
   ProfOn
   ```
4. Run the program as normal:
   ```
   go
   ```
5. Once execution completes, write the profile data to a file using the `ProfWrite` command:
   ```
   ProfWrite sort1.prf
   ```

6 Exit armsd:

```
Quit
```

7 The profile for the collected data can now be generated by entering the following at the system prompt:

```
armprof sort1.prf > prof1
```

The profiler generates report and the output is sent to file `prof1`. This can then be viewed as a text file.

If you are using the Windows toolkit:

1 Load the project file `sorts.apj` into the ARM Project Manager by choosing **Open** from the Project menu.

2 Build the project by clicking on the **Rebuild-all** icon on the toolbar.

3 Load the debugger by clicking on the **Debug** icon on the toolbar.

4 Turn on profiling in the ARM Debugger for Windows by choosing **Profiling -> Toggle Profiling** from the Options menu.

5 Start the program by clicking on the toolbar's **Go** icon.

The program runs and stops at the breakpoint on `main`.

6 Click on the **Go** icon again.

The program resumes execution.

7 Once execution completes, write the profile data to the file `sort1.prf`, by choosing **Profiling -> Write to file** from the Options menu.

8 Exit the debugger and start a DOS session. Make the profile directory the current directory.

The profile for the collected profile data can now be generated by entering the following at the system prompt:

```
armprof sort1.prf > prof1
```

armprof generates the profile report and outputs it to the profile file. This can then be viewed as a text file.

5.4.2 Call graph information

If you are using the command line tools:

1 Restart the debugger:

```
armsd
```

2 Load the `sorts` program into armsd with the /`callgraph` option:

```
load/callgraph sorts
```

`/callgraph` tells armsd to prepare an image for function call count profiling by adding code that counts the number of function calls.

3 Turn profiling on:

```
ProfOn
```

4 Run the program as normal:

```
go
```

5 Once execution completes, write the profile data to a file:

```
ProfWrite sort2.prf
```

6 Exit armsd:

```
Quit
```

7 Generate the profile by entering the following at the system prompt:

```
armprof -Parent sort2.prf > prof2
```

`-Parent` instructs armprof to include information about the callers of each function. armprof generates the profile report and outputs it to `prof2`, which can then be viewed as a text file.

If you are using the Windows toolkit:

1 Reload the debugger by clicking on the **Debug** icon on the ARM Project Manager's toolbar.

2 Turn on call graph profiling by choosing **Profiling -> Call graph profiling** from the Options menu.

3 Reload the image by clicking on the Toolbar's **Reload** icon. This forces call graph profiling to take effect.

4 Turn on profiling in ARM Debugger for Windows by choosing **Profiling -> Toggle Profiling** from the Options menu.

5 Start the program by clicking on the toolbar's **Go** icon.

The program runs and stops at the breakpoint on `main`.

6 Click on the **Go** icon again.

The program resumes execution.

7 Once execution completes, write the profile data to the file `sort2.prf`, by choosing **Profiling -> Write to file** from the Options menu.

8 Exit the debugger and invoke a DOS session.

9 Generate the profile by entering the following at the DOS prompt:

```
armprof -Parent sort2.prf > prof2
```

`-Parent` instructs armprof to include information about the callers of each function. armprof generates the profile report, which is output to `prof2`. This can then be viewed as a text file.

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
England
Telephone:+44 1223 400400
Facsimile:+44 1223 400410
Email:info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone:+49 (0) 89 608 75545
Facsimile:+49 (0) 89 608 75599
Email:info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone:+81 44 850 1301
Facsimile:+81 44 850 1308
Email:info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone:+1 408 399 5199
Facsimile:+1 408 399 8854
Email:info@arm.com

World Wide Web Address: http://www.arm.com/