Arm® Instruction Emulator

Version 22.0

Developer and Reference Guide

Non-Confidential

Issue 00

102190_22.0_00_en

Arm® Instruction Emulator

Developer and Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
2010-00	21 August 2020	Non-Confidential	First release for Arm Instruction Emulator version 20.1
2100-00	30 March 2021	Non-Confidential	Update for Arm Instruction Emulator version 21.0
2200-00	31 March 2022	Non-Confidential	Update for Arm Instruction Emulator version 22.0

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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

List of Figures

List of Tables

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace` <u>`underline`</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 Get started

This section describes how to install and get started with Arm® Instruction Emulator.

Arm Instruction Emulator (`armie`) is an emulator that runs on any Armv8-A-based AArch64 platform and emulates Scalable Vector Extension (SVE) instructions. Arm Instruction Emulator lets you develop SVE code without needing access to SVE-enabled hardware.

2.1 Install Arm Instruction Emulator

Follow these steps to download and install Arm® Instruction Emulator.

Before you begin

Ensure that either Environment Modules or the Lmod Environment Modules system are installed on your machine. Some information on how to install Environment Modules is available in the Arm Allinea Studio environment configuration documentation.

Procedure

1. Download the appropriate Arm Instruction Emulator package for your Linux host platform. To download Arm Instruction Emulator, see the Arm Instruction Emulator downloads page on the Arm Developer website.
2. Extract the downloaded package:

```
tar -xvzf <package_name>.tar.gz
```

   replacing `<package_name>` with the full name of the downloaded package.
3. To see the extracted files, change to the extracted package directory:

```
cd <package_name>
```

4. Run the installation script as a privileged user:

```
sudo ./arm-instruction-emulator-22.0*_aarch64-linux-rpm.sh <option> <option>
```

   where <option> are options supported by the installation script. Supported options include:

   -a, --accept
   : Automatically accept the EULA (the EULA still displays).

   -i, --install-to <location>
   : Install to the given directory.

     Use this option if you do not have sudo rights to install to /opt/arm or another central location.

   -f, --force
   : Force an install attempt to a non empty directory.

   -h, --help
   : Display this table in the form of a help message in the terminal.

   If no options are supplied, and you run a default installation, the packages are unpacked to /opt/arm/<package_name>. If you use the -i (or --install-to) option to specify a custom install location, such as <install-dir>:

```
./<package_name>.sh --install-to <install_dir>
```

   The package will be installed to the <install_dir> that you pass to -i (or --install-to).

> **Note**
>
> If you use the --install-to option, you need to manually make the installation and module files available to other users, if they require them.

---

5. Unless you have included the -a (or --accept) option, the installer displays the EULA and prompts you to agree to the terms. To agree, type 'yes' at the prompt. For more information about the release contents, see the release notes, located in the <install-dir>/<package_name> directory.

Results

Arm Instruction Emulator is installed on your system.

Next steps

- Configure your Linux environment:
  1. To see which environment modules are available on your system, run:

```
module avail
```

  2. If you do not see the Arm Instruction Emulator environment module, configure the MODULEPATH environment variable to include the Arm Instruction Emulator installation directory:

```
export MODULEPATH=$MODULEPATH:<install-dir>/modulefiles/
```

     Re-check which which environment modules are now available on your system:

```
module avail
```

  3. Load the appropriate Arm Instruction Emulator module for the processors in your system, and for the compiler you are using:

```
module load armie<major-version>/<package-version>
```

     where <package-version> is <major-version>.<minor-version>{.<patch-version>}.

     For example:

```
module load armie22/22.0
```

     Tip: Add the module load command to your .profile to run it automatically every time you log in.

  4. Check your environment by examining the PATH variable. It should contain the appropriate Arm Instruction Emulator bin directory from <install-dir>/:

```
echo $PATH /opt/arm/arm-instruction-emulator-22.0_Generic-
AArch64_RHEL-8_aarch64-linux/bin64:...
```

- To learn how to use Arm Instruction Emulator, refer to Get started with Arm Instruction Emulator.
- For information about environment variables used by the Arm-provided suite of server and High Performance Computing (HPC) tools, see the Environment variables reference topic.
- To uninstall Arm Instruction Emulator, run the uninstall.sh script located in <install-dir>/arm-instruction-emulator-<version>_<microarch>_<OS>-<OS_Version>_aarch64-linux/uninstall.sh

2.2 Get started with Arm Instruction Emulator

This tutorial uses a couple of simple examples to demonstrate how to compile Scalable Vector Extension (SVE) code and run the resulting binary with Arm® Instruction Emulator.

Before you begin

- This task uses Arm Compiler for Linux (part of Arm Allinea Studio) as the compiler. Alternatively, you can use GCC for the compilation steps.

  If you want to use Arm Compiler for Linux, download and install Arm Compiler for Linux for your platform.
- Load the Arm Instruction Emulator module for your platform:

  ```
  module load armie<major-version>/<package-version>
  ```

  where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

  For example:

  ```
  module load armie22/22.0
  ```

  To check that your environment is now configured to run Arm Instruction Emulator, examine the `PATH` variable and confirm that it contains the appropriate Arm Instruction Emulator `bin` directory from your installation location `<install-dir>`:

  ```
  echo $PATH /<install-dir>/arm-instruction-emulator-22.0_Generic-
  AArch64_RHEL-8_aarch64-linux/bin:...
  ```

Procedure

1. Compile your source code and generate an executable binary.
2. Run the binary with Arm Instruction Emulator. Either:
   a. Invoke Arm Instruction Emulator and specify the vector length to use:

   ```
   armie -msve-vector-bits=<length> ./<binary>
   ```

   b. Invoke Arm Instruction Emulator with an instrumentation (`-i`) or emulation (`-e`) client, and specify the vector length to use:

   ```
   armie -msve-vector-bits=<arg> -e <emulation_client> -i
    <instrumentation_client> -- ./<binary>
   ```

   Instrumentation and emulation clients enable you to extract data on the execution of your binary.

Example: Compile and run a 'Hello World' application

In this example you will write a simple 'Hello World' application in C, compile it with Arm Compiler for Linux, and then run it using Arm Instruction Emulator. The example does not contain SVE code.

1. Load the Arm Compiler for Linux module for your platform:

   ```
   module load acfl/<package-version>
   ```

   where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

   For example:

   ```
   module load acfl22/22.0
   ```

2. Create a simple 'Hello World' C application and save it as a file named `hello.c`.

   ```
   /* Hello World */
   #include <stdio.h>
   int main()
   {
       printf("Hello World\n");
       return 0;
   }
   ```

3. To generate an executable binary, compile your application with Arm C/C++ Compiler.

   ```
   armclang -O3 -march=armv8-a+sve -o hello hello.c
   ```

   The `-O3` option ensures the highest optimization level with auto-vectorization is enabled. The `-march=armv8-a+sve` option targets hardware with the Armv8-A architecture, and generates SVE instructions in the executable binary.

   Note

   In this example, no SVE code is used. However, it is good practice to enable the highest level of auto-vectorization and target an SVE-enabled architecture when compiling any code to be run using Arm Instruction Emulator.

4. Run the generated binary `hello` using Arm Instruction Emulator:

```
armie -msve-vector-bits=256 ./hello
```

Which returns:

```
Hello World
```

For this simple 'Hello World' example, Arm Instruction Emulator runs the code on an emulated SVE-enabled architecture without using SVE instructions.

To use Arm Instruction Emulator to its full potential, that is, to emulate SVE instructions, we must look at a more complex application. An example of an application containing SVE code is available in the next section of this tutorial.

Example: Compile, vectorize, and run an application with SVE code

This example compiles and vectorizes some C code that targets an SVE-enabled Armv8-A architecture, then uses Arm Instruction Emulator to run the SVE binary.

1. Load the Arm Compiler for Linux module for your platform:

```
module load acfl/<package-version>
```

where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

For example:

```
module load acfl/22.0
```

2. Create a file called `example.c`, containing the following code:

```
// example.c
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
for (int i = 0; i < ARRAYSIZE; i++)
{
    a[i] = b[i] - c[i];
}
}
int main() {
for (int i = 0; i < ARRAYSIZE; i++)
{
  // Generate a random number between 200 and 300
  b[i] = (rand() % 100) + 200;
  // Generate a random number between 0 and 100
  c[i] = rand() % 100;
}
subtract_arrays(a, b, c);
printf("i \ta[i] \tb[i] \tc[i] \n");
```

```
printf("=============================\n");
for (int i = 0; i < ARRAYSIZE; i++)
{
    printf("%d \t%d \t%d \t%d\n", i, a[i], b[i], c[i]);
}
}
```

This C code subtracts corresponding elements in two arrays, and writes the result to a third array. The three arrays are declared using the `restrict` keyword, which indicates to the compiler that they do not overlap in memory.

3. Compile the C code with Arm C/C++ Compiler:

```
armclang -O3 -march=armv8-a+sve -o example example.c
```

4. Run the binary with Arm Instruction Emulator:

```
armie -msve-vector-bits=256 ./example
```

The application returns:

```
i       a[i]    b[i]    c[i]
=============================
0       197     283     86
1       262     277     15
2       258     293     35
\...
1021    165     234     69
1022    232     295     63
1023    204     235     31
```

The SVE architecture extension specifies an `IMPLEMENTATION DEFINED` vector length. The -`msve-vector-bits` option lets you specify the vector length for Arm Instruction Emulator to use. The vector length must be a multiple of 128 bits, with a maximum of 2048 bits. To list all valid vector lengths, use the `-mlist-vector-lengths` option :

```
armie -mlist-vector-lengths
```

Which returns:

```
128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
```

Next Steps

To learn how to analyze your application using the emulation and instrumentation clients available for Arm Instruction Emulator, see Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator.

Related information

armie command reference on page 49
Troubleshoot: Use -s on page 17
Learn more about Arm Instruction Emulator

DynamoRIO dynamic binary instrumentation tool platform
DynamoRIO API
DynamoRIO API Usage Tutorial
Porting and Optimizing HPC Applications for Arm SVE guide

2.3 Troubleshoot: Use -s

Describes how you can use the `-s` option to better understand what the emulation commands and files Arm® Instruction Emulator uses, and what to send to Arm Support if you require further assistance.

The `-s` and `--show-drrun-cmd` options

To show how Arm Instruction Emulator used DynamoRIO's `drrun` command to emulate and instrument an SVE binary, invoke the `-s` (or `--show-drrun-cmd`) option.

For example, in the following command line, `libsve_512.so` is the SVE emulation client and `libinscount_emulated.so` is the instrumentation client:

```
armie -s -msve-vector-bits=512 -i libinscount_emulated.so -- ./example_sve
```

Which returns:

```
/path/to/armie/bin64/drrun -client /path/to/armie/lib64/release/libsve_512.so 0 "" -
client /path/to/armie/samples/bin64/libinscount_emulated.so 1 "" -max_bb_instrs 32 -
max_trace_bbs 4 -- ./example_sve
Client inscount is running
. . .
```

The `-s` option allows you to understand how Arm Instruction Emulator uses DynamoRIO, and can be used to pass parameters and debug options to DynamoRIO's `drrun` command. For example, the inscount client has an `-only_from_app` option which only counts the application instructions and ignores libraries. Passing the `-only_from_app` option using the `drrun` command:

```
/path/to/install/bin64/drrun -client /path/to/install/lib64/release/libsve_512.so
 0 "" -client /path/to/install/samples/bin64/libinscount_emulated.so 1 "-
only_from_app" -max_bb_instrs 32 -max_trace_bbs 4 -- ./example_sve
```

returns:

```
Client inscount is running
955 instructions executed of which 709 were SVE instructions
```

which shows that the application used 955 non-SVE instructions, compared to 118497 when also counting library instructions.

The preferred method to pass command line arguments to instrumentation clients is to use the `-a` or `--arg-iclient` option. For more information, see armie command reference. The preceding method, which uses the `drrun` command, is useful in cases where both the command line arguments to instrumentation clients are required, as well as the parameters and debug options to DynamoRIO's `drrun` command.

Contact Arm Support

In the event of a program crash, the operating system kernel creates a core dump file. The location and name of this core dump file depends on your system's core dump configuration. If your configuration specifies that core dump filenames include the name of the crashed binary, note that this is the name of the executable being emulated rather than the Arm Instruction Emulator binary name (`armie`).

Core dump files should be sent to Arm support along with the output of `armie --version`. However, if you have confidentiality concerns regarding sensitive data in the core dump file, do not send the core dump to Arm. However, without a core dump file, the Arm Support team might not be able to investigate your issue.

To request technical support, Contact Arm Support.

3 Tutorials

Learn how to build instrumentation clients and custom analysis instrumentation for Arm® Instruction Emulator, and how to use Arm Instruction Emulator to analyze your Scalable Vector Extension (SVE) applications.

3.1 Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator

Describes how to use the instrumentation and emulation clients and run your applications with Arm® Instruction Emulator.

You can use Arm Instruction Emulator without any instrumentation or emulation clients, as described in Get started with Arm Instruction Emulator, to verify that the code you have developed can run on SVE hardware. However, if you are developing high-performance applications and want to gain insights into their execution behavior, runtime analysis is required . Runtime analysis enables you to identify heavily-used loops and instruction sequences, so that improvements can be made to execution speed and memory access.

To emulate and instrument SVE binaries on AArch64 hardware, Arm Instruction Emulator uses DynamoRIO . DynamoRIO is a publicly available Dynamic Binary Instrumentation (DBI) tool platform which supports x86 and Arm binaries. DynamoRIO provides an API which enables you to write your own binary-level runtime instrumentation and supply some example instrumentation. Each Arm Instruction Emulator release integrates a stable version of DynamoRIO.

Arm Instruction Emulator also provides a set of instrumentation clients which can be used to analyze SVE binaries at runtime. In this context, 'instrumentation client' refers to how Arm Instruction Emulator uses DynamoRIO to work as an analysis tool and an emulator.

Note

Before looking at an example of an instrumentation client for emulated binaries using Arm Instruction Emulator, Arm recommends that you understand the basic principles of instrumenting binaries using the DynamoRIO API. For more information, see the DynamoRIO API usage tutorial.

For example, one Arm Instruction Emulator instrumentation feature is called Regions-of-Interest (RoI). Sometimes, when analyzing large, complex, and long running applications, it is necessary to limit the amount of runtime data collected (such as memory traces, instruction, and opcode counts) to specific parts of code. You can use the RoI feature to collect runtime data for regions of the code that are marked with RoI markers. Before you can add RoI markers and build the application, you must have access to the source code under analysis. To mark a RoI, use start and stop macros in the source. These RoI markers are described in an example below.

Note

There are restrictions to the use of RoI markers in source code. RoIs must not be nested and they must not overlap. Violating these restrictions will result in undefined behavior.

To emulate and analyze an SVE binary, invoke Arm Instruction Emulator with an instrumentation client and the SVE binary. The client is a shared object file which uses the DynamoRIO API to capture and process wanted runtime events.

Before you begin

- Ensure you have loaded the Arm Instruction Emulator environment module for your platform:

  ```
  module load armie<major-version>/<package-version>
  ```

  where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.
- Ensure you have already compiled your application binary.

Procedure

1. Invoke Arm Instruction Emulator with an instrumentation (`-i <instrumentation_client>`) or emulation (`-e <emulation_client>`) client and the binary, use:

   ```
   armie -msve-vector-bits=<arg> -e <emulation_client> -i <instrumentation_client>
    -- ./<binary>
   ```

2. Analyze the results provided by the clients.

Example: Analyze an application with SVE code

The following example demonstrates how to count native AArch64 as well as emulated SVE instructions.

`event_bb_analysis()` is the function which counts instructions in the sample client :file::*<install-dir>/arm-instruction-emulator/samples/inscount_emulated.cpp*.

```
    /* Count instructions */
bb_counts.native_instrs = bb_counts.emulated_instrs = 0;
bool is_emulation = false;
for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
    next_instr = instr_get_next(instr);
    if (drmgr_is_emulation_start(instr)) {                  ←[1]
        bb_counts.emulated_instrs++;
        is_emulation = true;
        /* Data about the emulated instruction can be extracted from the
         * start label using drmgr_get_emulated_instr_data().
         */
        emulated_instr_t emulated;
        drmgr_get_emulated_instr_data(instr, &emulated);    ←[2]
        dr_printf("SVE: %p\t", emulated.pc);
        int *sveinstr;
        sveinstr = ((int *)instr_get_raw_bits(emulated.instr));
        dr_printf("0x%08x\n", *sveinstr);
        continue;
    }
```

```
    if (drmgr_is_emulation_end(instr)) {                    ←[3]
        is_emulation = false;
        continue;
    }
    if (is_emulation)
        continue;
    if (!instr_is_app(instr))
        continue;
    bb_counts.native_instrs++;
}
 /* Insert clean call */
 dr_insert_clean_call(drcontext, bb, instrlist_first_app(bb),
                      (void *)inscount, false /* save fpstate */, 2,
                      OPND_CREATE_INT64(bb_counts.native_instrs),
                      OPND_CREATE_INT64(bb_counts.emulated_instrs))
```

The count instructions example function is inserted at the end of each basic-block, at transformation time, and iterates over each instruction in a basic-block, at execution time.

Note

The difference between *transformation* and *execution* is described in the **Code Transformation and code Execution** section of About instrumentation clients.

In the count instructions example function:

- `bb_counts.native_instrs` and `bb_counts.emulated_instrs`, increment depending on if the instruction is emulated or not.

  The count instructions example function distinguishes between emulated and native instructions using the `drmgr_is_emulation_start()` ([1]) and `drmgr_is_emulation_end()` ([3]) functions of DynamoRIO.
- Where an instruction is the start of a sequence of instructions that emulate an SVE instruction, `drmgr_is_emulation_start()` returns `true`.

  The `drmgr_is_emulation_start()` instruction also contains data about the instruction being emulated. The instruction data can be extracted using `drmgr_get_emulated_instr_data()` ([2]).
- Where an instruction is the last instruction of a sequence of instructions that emulate an SVE instruction, `drmgr_is_emulation_end()` returns `true`.

Note

The reference documentation for these functions is available on the DynamoRIO web site. For a full description of these functions, see

- drmgr_is_emulation_start()
- drmgr_is_emulation_end()
- drmgr_get_emulated_instr_data()
- emulated_instr_t

To extract useful information about the instruction being emulated, you can use the `drmgr_get_emulated_instr_data()` function, the PC address, and the instruction encoding.

1. Run Arm Instruction Emulator with the `libinscount_emulated.so` instrumentation client on your example code:

```
armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./example_sve
```

Which returns:

```
Client inscount is running
SVE: 0x000000000040053c 0x04a0e3ef
SVE: 0x0000000000400554 0x04a14001
SVE: 0x000000000040055c 0x25aa1fe0
SVE: 0x0000000000400560 0x05a039e0
SVE: 0x0000000000400570 0xe5494101
SVE: 0x0000000000400574 0x04b0e3e9
SVE: 0x0000000000400578 0x04a00021
SVE: 0x000000000040057c 0x25aa1d20
SVE: 0x0000000000400570 0xe5494101
SVE: 0x0000000000400574 0x04b0e3e9
SVE: 0x0000000000400578 0x04a00021
SVE: 0x000000000040057c 0x25aa1d20
SVE: 0x00000000004005a8 0x25ac1fe0
SVE: 0x00000000004005b4 0xa5494100
SVE: 0x00000000004005b8 0xa54941a1
SVE: 0x00000000004005bc 0x85604140
SVE: 0x00000000004005c0 0x04a10000
SVE: 0x00000000004005c4 0xe5494160
SVE: 0x00000000004005c8 0x04b0e3e9
SVE: 0x00000000004005cc 0x25ac1d20
SVE: 0x00000000004005b4 0xa5494100
SVE: 0x00000000004005b8 0xa54941a1
SVE: 0x00000000004005bc 0x85604140
SVE: 0x00000000004005c0 0x04a10000
SVE: 0x00000000004005c4 0xe5494160
SVE: 0x00000000004005c8 0x04b0e3e9
SVE: 0x00000000004005cc 0x25ac1d20
120827 instructions executed of which 709 were emulated instructions
```

2. To convert the encodings output by `dr_printf("0x%08x\n", *sveinstr)` to instruction mnemonics, use the example helper script `/<install-dir>/arm-instruction-emulator/bin64/enc2instr.py`. `enc2instr.py` shows the use of the `enc2instr()` function and can be copied and modified for your own output transformations.

Example: Analyze the effect of the vector length on the number of AArch64 and emulated SVE instructions

This example uses the same instrumentation client that was used in the preceding example, `libinscount_emulated.so`. However, in this example we show how you can use `libinscount_emulated.so` to investigate the effect that vector length has on the number of SVE instructions. For example, to minimize them and help reduce time spent in execution.

1. Invoke Arm Instruction Emulator with an instrumentation client named `libinscount_emulated.so` and run the `example` binary:

```
armie -msve-vector-bits=128 -i libinscount_emulated.so -- ./example_sve
```

Which returns:

```
Client inscount is running
```

```
SVE: 0x00000000004006c8 0x25a91fe0
SVE: 0x00000000004006d0 0xa54842a0
SVE: 0x00000000004006d4 0xa54842c1
SVE: 0x00000000004006d8 0x04a10400
SVE: 0x00000000004006dc 0xe54842e0
SVE: 0x00000000004006e0 0x04b0e3e8
SVE: 0x00000000004006e4 0x25a91d00
SVE: 0x00000000004006d0 0xa54842a0
SVE: 0x00000000004006d4 0xa54842c1
SVE: 0x00000000004006d8 0x04a10400
SVE: 0x00000000004006dc 0xe54842e0
SVE: 0x00000000004006e0 0x04b0e3e8
SVE: 0x00000000004006e4 0x25a91d00
i       a[i]    b[i]    c[i]
=============================
0       197     283     86
1       262     277     15
2       258     293     35
3       194     286     92
. . .
1019    243     290     47
1020    185     261     76
1021    165     234     69
1022    232     295     63
1023    204     235     31
2134094 instructions executed of which 1537 were emulated instructions
```

Notice the difference in output from the preceding example shown in Get started with Arm Instruction Emulator (see section **Compile, vectorize, and run an application with SVE code**) which did not use `-i libinscount_emulated.so`. The additional information is what the instrumentation client, `libinscount_emulated.so`, outputs as part of its analysis of the example binary as it runs:

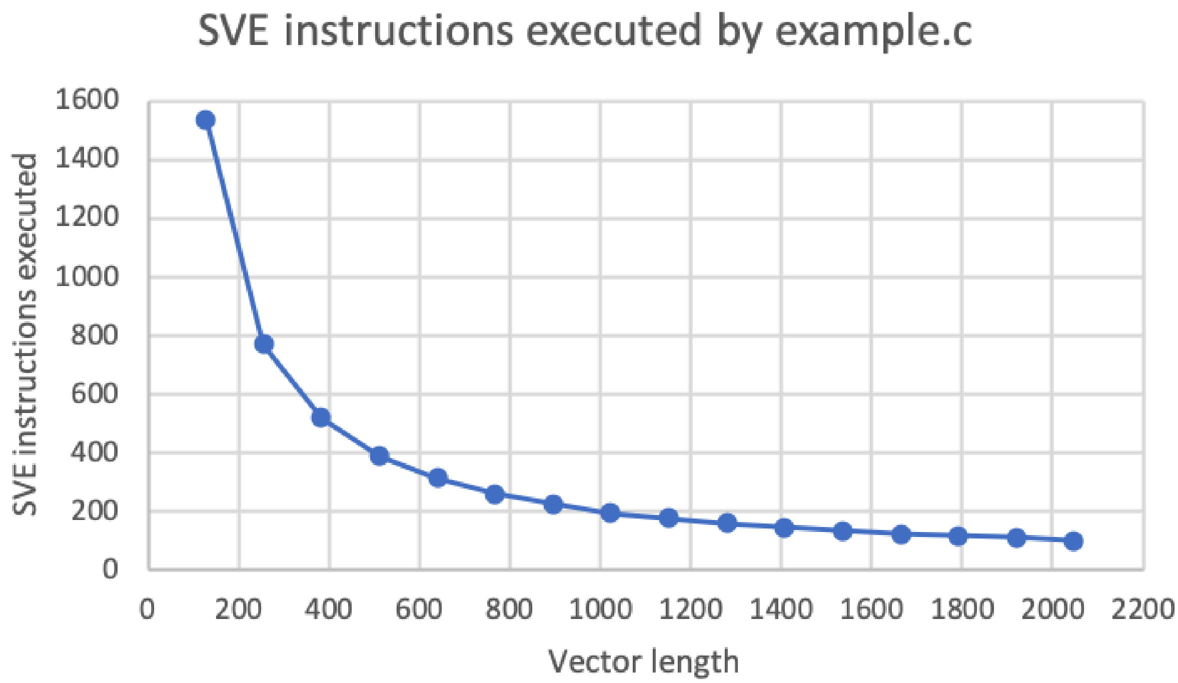
```
Client inscount is running
SVE: 0x00000000004006c8 0x25a91fe0
...
2134094 instructions executed of which 1537 were emulated instructions
```

2. Run the example binary with each vector length and tabulate the results:

Vector Length	128	256	384	512	640	768	896	1024	1152	1280	1408	1536	1664	1792	1920	2048
SVE Instructions	1537	769	517	385	313	259	223	193	175	157	145	133	121	115	109	97

3. Plot the results on a line graph:

Figure 3-1: Plot of SVE Instructions



The graph shows us that the largest reduction in SVE instructions executed occurs between 128 and about 512 bits. This type of analysis of the runtime behavior of an application can be used with other types of analysis. For example, to study the impact of vector length on performance.

Example: Analyze Regions-of-Interest (RoI)

To avoid large trace files and focus on trace behavior of specific sections of code, you can insert start and stop trace macros into the source code being analyzed:

```
#define __START_TRACE() {asm volatile (".inst 0x2520e020");}
#define __STOP_TRACE() {asm volatile (".inst 0x2520e040");}
```

These start and stop macros instruct Arm Instruction Emulator to start and stop collecting trace data, which allows you to focus your analysis on specific areas of code, instead of analyzing the entire application. Focussing on specific sections of code makes the analysis of large long-running applications much easier and less time-consuming.

The code in this example illustrates the use of the `libinscount_emulated.so` client, an instrumentation client that allows you to limit the amount of runtime data collected to specific parts of code. Limiting the amount of runtime data is particularly useful when analyzing large, complex, or long-running applications.

The application used in this example, `loops`, contains two loops. This example uses the RoI feature to limit instruction counting to a single loop. First, the first loop is investigated, then the second is investigated and compared. The initial source code for `loops` is:

```
#define N 42
int a[N], b[N], c[N];
int main(void) {
  a[0] = 0;
  b[0] = 1;
  c[0] = a[0] + b[0];
for(int i=0; i<N; ++1)
  c[i] = i;
for(int i=0; i<N; ++i)
  a[i] = b[i] + b[c[i]];
}
```

1. Build and run the example `loops` application with the `libinscount_emulated.so` client:

   ```
   armie -msve-vector-bits=512 -i libinscount_emulated.so ./loops
   ```

   which returns:

   ```
   Client inscount is running
   89539 instructions executed of which 36 were emulated instructions
   ```

   All executed instructions are counted.

2. To limit instruction counting to a specific area of code, or the region of interest (RoI), add RoI markers to the `loops` source:

   - To indicate where to start counting, add the `__START_TRACE()` marker.
   - To indicate where to stop counting, add the `__STOP_TRACE()` marker.

   For example, to wrap the first loop of the `loops` code in RoI markers, use:

   ```
   #define N 42
   int a[N], b[N], c[N];
   #define __START_TRACE() { asm volatile (".inst 0x2520e020"); }
   #define __STOP_TRACE() { asm volatile (".inst 0x2520e040"); }
   int main(void) {
     __START_TRACE();
     a[0] = 0;
     b[0] = 1;
     c[0] = a[0] + b[0];
   for(int i=0; i<N; ++i)
       c[i] = i;
     __STOP_TRACE();
     for(int i=0; i<N; ++i)
       a[i] = b[i] + b[c[i]];
   }
   ```

3. Build the new binary and call it `first_loop`.

4. Run `first_loop` with the `libinscount_emulated.so` client:

   ```
   armie -msve-vector-bits=512 -i libinscount_emulated.so -a -roi ./first_loop
   Client inscount is running
   ```

```
31 instructions executed of which 16 were emulated instructions
```

The results are different to the `loops` run:

- Only the first loop has been instrumented and as a result fewer executed instructions have been counted at runtime.
- The `armie` command includes the `-a -roi` option to inform the `libinscount_emulated.so` client. `a roi` informs the client to enable and disable instruction counting, based on the `__START_TRACE()` and `__STOP_TRACE()` macros. Without the `-a -roi` option, the client ignores the macros and counts all instructions producing the same output as for the `loops` run above:

  ```
  armie -msve-vector-bits=512 -i libinscount_emulated.so ./first_loop
  Client inscount is running
  89539 instructions executed of which 36 were emulated instructions
  ```

  The `-a` option enables you to pass command line arguments to instrumentation clients. In this case, the argument is `-roi` but it can be any string which the client can use to adjust its behavior at execution time. For a description of the `-a` option, run `armie --help` or, see the armie command reference section.

5. Next, analyze the second loop. Move the `__START_TRACE()` and `__STOP_TRACE` markers to surround the second `for` loop:

```
#define N 42
int a[N], b[N], c[N];
#define __START_TRACE() { asm volatile (".inst 0x2520e020"); }
#define __STOP_TRACE() { asm volatile (".inst 0x2520e040"); }
int main(void) {
  a[0] = 0;
  b[0] = 1;
  c[0] = a[0] + b[0];
 for(int i=0; i<N; ++i)
    c[i] = i;
  __START_TRACE();
  for(int i=0; i<N; ++i)
    a[i] = b[i] + b[c[i]];
  __STOP_TRACE();
}
```

6. Build the new binary and call it `second_loop`.
7. Run and analyze the `second_loop` binary:

```
armie -msve-vector-bits=512 -i libinscount_emulated.so -a -roi ./second_loop
```

Which returns:

```
Client inscount is running
31 instructions executed of which 20 were emulated instructions
```

In the `second_loop` run, more SVE instructions are executed than in the `first_loop` run. More instructions are run because of the extra vector load and arithmetic instructions in the second loop.

Note

The example source code is in the `samples` directory of your Arm Instruction Emulator installation. You can modify these clients for your own custom analysis requirements.

Traces can be used in post-processing to prune any non-SVE accesses outside the RoI.

In addition to the `libinscount_emulated` client, the following clients also support `__START_TRACE` and `__STOP_TRACE`: `memtrace_emulated`, `instrace_emulated`, `meminstrace_emulated`, and `opcodes_emulated`.

To enable RoIs, all these clients accept the `-a -roi` Arm Instruction Emulator option. If you do not use the `-a -roi` option, RoIs are ignored and all instructions are counted or traced.

Example: Count the dynamic instruction counts

Dynamic instruction counts, or in other words, counting instructions executed by a binary at runtime, is a useful way of assessing the performance-related behavior of an application. An instruction count client, `libinscount.so`, is supplied as an example of how to use the DynamoRIO API with SVE emulation. The client source code is available as a DynamoRIO example in `api/samples/inscount.cpp`. Use the `-i` (or `--iclient`) option to run the client with `armie`, for example:

```
armie -msve-vector-bits=512 -i libinscount.so -- ./example_sve
```

Which returns:

```
Client inscount is running
Instrumentation results: 106384 instructions executed
```

To compare the number of SVE instructions to the number of native AArch64 instructions executed, use the `libinscount_emulated.so` client, for example:

```
armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./example_sve
```

Which returns:

```
Client inscount is running
106384 instructions executed of which 22 were emulated instructions
```

The source code is available in `samples/inscount_emulated.cpp`.

Another useful way of assessing the performance-related behavior of an application is to count instructions executed by opcode type. Such a count can give you more detailed insights into execution behavior than a total instruction count. For an example, see the Emulating SVE on Armv8 using DynamoRIO and ArmIE blog.

Example: Examine memory access behavior

The memory access behavior of an executable is another useful aspect of performance. Memory trace emulation clients for all vector lengths, `libmemtrace_sve_<vector length>.so` are supplied to work with the DynamoRIO instrumentation client, `libmemtrace_emulated.so`. To trace memory accesses, use the `-e` and `-i` options of `armie`. For example:

```
armie -e libmemtrace_sve_512.so -i libmemtrace_emulated.so -- ./example_sve
```

This command creates two trace files in the current directory: a non-SVE AArch64 trace from `libmemtrace_emulated.so`, and an SVE trace from `libmemtrace_sve_512.so`. For example:

```
head memtrace.example_sve.10120.0000.log
0: 0, 0,  0,  8, 0xffffe31ea730, 0x40043c
1: 0, 0,  0,  8, 0x400460, 0x400448
2: 0, 0,  0,  8, 0x400468, 0x40044c
3: 0, 0,  0,  8, 0x400470, 0x400450
4: 0, 0,  0,  8, 0x420000, 0x400404
5: 0, 0,  1, 16, 0xffffe31ea720, 0x4003e0
6: 0, 0,  0,  8, 0x41fff8, 0x4003e8
7: 0, 0,  1, 16, 0xffffe31ea5c0, 0x400610
8: 0, 0,  1, 16, 0xffffe31ea5d8, 0x400618
```

```
head sve-memtrace.example_sve.10120.log
27, -1, 0, 1, 0, (nil), (nil)
40, 0, 0, 0, 64, 0x4200d8, 0x4005e4
41, 0, 0, 0, 64, 0x420030, 0x4005e8
42, 0, 3, 0, 4, 0x420030, 0x4005ec
43, 0, 2, 0, 4, 0x420034, 0x4005ec
44, 0, 2, 0, 4, 0x420038, 0x4005ec
45, 0, 2, 0, 4, 0x42003c, 0x4005ec
46, 0, 2, 0, 4, 0x420040, 0x4005ec
47, 0, 2, 0, 4, 0x420044, 0x4005ec
48, 0, 2, 0, 4, 0x420048, 0x4005ec
. . .
86, 0, 2, 0, 4, 0x4200c8, 0x4005ec
87, 0, 2, 0, 4, 0x4200cc, 0x4005ec
88, 0, 6, 0, 4, 0x4200d0, 0x4005ec
89, 0, 0, 0, 36, 0x420200, 0x4005f4
90, -2, 0, 1, 0, (nil), (nil)
```

The SVE trace includes start and stop trace entries to delimit the chosen Region-of-Interest (RoI):

```
start -> xx, -1, 0, 1, 0, (nil), (nil)
stop  -> xx, -2, 0, 1, 0, (nil), (nil)
```

For an explanation on RoI, see the previous example.

The sequence number of the SVE trace is delimited by a comma. The sequence number of a non-SVE trace is delimited by a colon.

To enable you to analyze memory trace files, utilities are provided. For example, the merge utility produces one file with each trace, in chronological order, from a non-SVE AArch64 trace file and an SVE trace file:

```
merge memtrace.example_sve.10120.0000.log sve-memtrace.example_sve.10120.log >
 merged-memtrace.log
```

Memory tracing format

The memory trace uses a comma-separated-value format with the following fields:

```
sequence, tid, bundle, isWrite, size, addr, pc
```

Where:

sequence
: Sequence number which orders the load/stores across multiple trace files.

tid
: Thread id

bundle
: Support bundling of multiple mem_refs for gather/scatter/strided accesses.

isWrite
: `true` if store, `false` if load.

size
: Number of bytes that are stored or loaded.

addr
: Load or store address.

pc
: Instruction address.

Next steps

- Further instrumentation clients are available, that provide different insights, including:
  - `inscount_emulated.cpp`
  - `instrace_emulated.c`
  - `meminstrace_emulated.c`
  - `memtrace_emulated.c`
  - `opcodes_emulated.cpp`

  These are RoI-capable and their source code is in the Arm Instruction Emulator installation `samples` directory:

  ```
  /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/
  samples/
  ```

  You can modify and enhance these clients for your specific analysis requirements. For examples and guidance on how to modify and enhance clients, see Building custom analysis instrumentation.
- For more advanced analysis examples of a real-world application, see Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE. The blog includes use-case examples of `libopcodes_emulated.so` and `libmemtrace_simple.so`.

Related information

Building custom analysis instrumentation on page 37
Porting and Optimizing HPC Applications for Arm SVE
Arm Instruction Emulator

3.2 Build an emulation-aware instrumentation client

The ability to instrument emulated applications is a recent addition to the DynamoRIO API. Therefore, most of the samples which come with DynamoRIO (and Arm® Instruction Emulator) are not capable of interpreting emulated instructions. This tutorial demonstrates how to modify existing native-only clients to also handle emulated instructions, and how to write your own emulation aware clients.

Before you begin

- This tutorial assumes that you have a good working knowledge about the DynamoRIO API. Documentation is available at:

  https://dynamorio.org/files.html

  and includes the event driven usage model of DynamoRIO and example clients, from which the following clients are derived:

  - `samples/inscount_emulated.cpp`
  - `samples/instrace_emulated.c`
  - `samples/memtrace_simple.c`
  - `samples/memtrace_emulated.c`
  - `samples/meminstrace_emulated.c`
  - `samples/opcodes_emulated.cpp`
- Understand the About instrumentation clients.
- Understand how to run a pre-built instrumentation client. For more information on running instruction clients, see Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator.

About this task

Arm Instruction Emulator allows developers to use the API of DynamoRIO API to write instrumentation clients, which run alongside emulation clients, to analyze emulated binaries at runtime.

The following emulation aware functions can be used in an instrumentation client:

- `bool drmgr_is_emulation_start(instr_t *instr)`
- `bool drmgr_is_emulation_end(instr_t *instr)`
- `bool drmgr_get_emulated_instr_data(instr_t *instr, emulated_instr_t *emulated)`

```
typedef struct _emulated_instr_t {
    size_t size;
    app_pc pc;
    instr_t *instr;
} emulated_instr_t;
```

Procedure

1. Run the pre-built `libbbcount.so` client with Arm Instruction Emulator, which counts the number of basic blocks executed by an application:

   ```
   armie -msve-vector-bits=128 -i libbbcount.so -- ./example
   ```

   Which returns:

   ```
   Client bbcount is running
   i        a[i]    b[i]    c[i]
   ```

```
=============================
0       197     283     86
1       262     277     15
. . .
1021    165     234     69
1022    232     295     63
1023    204     235     31
Instrumentation results:
449561 basic block executions
  1971 basic blocks needed flag saving
     0 basic blocks did not
```

We will change the code to write both native and emulated basic block execution counts to `stdout`.

2. Add the emulated instruction counter variable. Copy the `bbcount.cpp` file to `bbcount_tut2.cpp` in: `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples`.
   Where `bbcount.cpp`, is:

```
/* we only have a global count */
static int global_count;
#ifdef SHOW_RESULTS
/* some meta-stats: static (not per-execution) */
static int bbs_eflags_saved;
static int bbs_no_eflags_saved;
#endif
static void
event_exit(void)
{
#ifdef SHOW_RESULTS
    char msg[512];
    int len;
    len = dr_snprintf(msg, sizeof(msg) / sizeof(msg[0]),
                      "Instrumentation results:\n"
                      "%10d basic block executions\n"
                      "%10d basic blocks needed flag saving\n"
                      "%10d basic blocks did not\n",
                      global_count, bbs_eflags_saved, bbs_no_eflags_saved);
    DR_ASSERT(len > 0);
    NULL_TERMINATE(msg);
    DISPLAY_STRING(msg);
#endif /* SHOW_RESULTS */
    drx_exit();
    drreg_exit();
    drmgr_exit();
}
```

   Edit `bbcount_tut2.cpp` to add a global emulation counter variable:

```
/* we have global native and emulated counts */
static int native_count;
static int emulated_count;
#ifdef SHOW_RESULTS
/* some meta-stats: static (not per-execution) */
static int bbs_eflags_saved;
static int bbs_no_eflags_saved;
#endif
static void
event_exit(void)
{
#ifdef SHOW_RESULTS
    char msg[512];
    int len;
```

```
    len = dr_snprintf(msg, sizeof(msg) / sizeof(msg[0]),
                      "Instrumentation results:\n"
                      "%10d native basic block executions\n
                      "%10d emulated basic block executions
                      "%10d basic blocks needed flag saving
                      "%10d basic blocks did not\n",
                      native_count, emulated_count,
                      bbs_eflags_saved, bbs_no_eflags_saved
    DR_ASSERT(len > 0);
    NULL_TERMINATE(msg);
    DISPLAY_STRING(msg);
#endif /* SHOW_RESULTS */
    drx_exit();
    drreg_exit();
    drmgr_exit();
}
```

3. Add the basic block emulation counting function. Modify the instrumentation callback function `event_app_instruction()` to look for at least one emulated instruction in a block, and if found, increment `emulated_count` when the block is executed.

   `bbcount.c`:

```
static dr_emit_flags_t
event_app_instruction(void *drcontext, void *tag, instrlist_t *bb, instr_t *inst,
                      bool for_trace, bool translating, void *user_data)
{
#ifdef SHOW_RESULTS
    bool aflags_dead;
#endif
   /* By default drmgr enables auto-predication, which predicates all
 instructions with
    * the predicate of the current instruction on ARM.
    * We disable it here because we want to unconditionally execute the following
    * instrumentation.
    */
    drmgr_disable_auto_predication(drcontext, bb);
    if (!drmgr_is_first_instr(drcontext, inst))
        return DR_EMIT_DEFAULT;
#ifdef VERBOSE
    dr_printf("in dynamorio_basic_block(tag=" PFX ")\n", tag);
#    ifdef VERBOSE_VERBOSE
    instrlist_disassemble(drcontext, tag, bb, STDOUT);
#    endif
#endif
#ifdef SHOW_RESULTS
    if (drreg_are_aflags_dead(drcontext, inst, &aflags_dead) == DRREG_SUCCESS
 && !aflags_dead)
        bbs_eflags_saved++;
    else
        bbs_no_eflags_saved++;
#endif
    /* racy update on the counter for better performance */
    drx_insert_counter_update(drcontext, bb, inst,
                              /* We're using drmgr, so these slots
                              * here won't be used: drreg's slots will be.
                              */
                              SPILL_SLOT_MAX + 1,
                              IF_AARCHXX_(SPILL_SLOT_MAX + 1) & global_count, 1,
 0);
#if defined(VERBOSE) && defined(VERBOSE_VERBOSE)
    dr_printf("Finished instrumenting dynamorio_basic_block(tag=" PFX ")\n",
 tag);
    instrlist_disassemble(drcontext, tag, bb, STDOUT);
#endif
    return DR_EMIT_DEFAULT;
```

```
}
```

`bbcount_tut2.c`:

```
static dr_emit_flags_t
event_app_instruction(void *drcontext, void *tag, instrlist_t *bb, instr_t *inst,
                      bool for_trace, bool translating, void *user_data)
{
    instr_t *instr, *next_instr;
#ifdef SHOW_RESULTS
    bool aflags_dead;
#endif
    /* By default drmgr enables auto-predication, which predicates all
 instructions wi
    * the predicate of the current instruction on ARM.
    * We disable it here because we want to unconditionally execute the following
    * instrumentation.
    */
    drmgr_disable_auto_predication(drcontext, bb);
    if (!drmgr_is_first_instr(drcontext, inst))
        return DR_EMIT_DEFAULT;
#ifdef VERBOSE
    dr_printf("in dynamorio_basic_block(tag=" PFX ")\n", tag);
#    ifdef VERBOSE_VERBOSE
    instrlist_disassemble(drcontext, tag, bb, STDOUT);
#    endif
#endif
#ifdef SHOW_RESULTS
    if (drreg_are_aflags_dead(drcontext, inst, &aflags_dead) == DRREG_SUCCESS
 && !aflags_dead)
        bbs_eflags_saved++;
    else
        bbs_no_eflags_saved++;
#endif
    for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
        next_instr = instr_get_next(instr);
        if (drmgr_is_emulation_start(instr)) {
            drx_insert_counter_update(drcontext, bb, inst,
                        SPILL_SLOT_MAX + 1,
                        IF_AARCHXX_(SPILL_SLOT_MAX + 1) & emulated_count, 1, 0);
            return DR_EMIT_DEFAULT;
        }
    }
    /* racy update on the counter for better performance */
    drx_insert_counter_update(drcontext, bb, inst,
                              /* We're using drmgr, so these slots
                              * here won't be used: drreg's slots will be.
                              */
                              SPILL_SLOT_MAX + 1,
                              IF_AARCHXX_(SPILL_SLOT_MAX + 1) & native_count, 1,
 0);
#if defined(VERBOSE) && defined(VERBOSE_VERBOSE)
    dr_printf("Finished instrumenting dynamorio_basic_block(tag=" PFX ")\n",
 tag);
    instrlist_disassemble(drcontext, tag, bb, STDOUT);
#endif
    return DR_EMIT_DEFAULT;
}
```

There are three things to note about this code change:

a) The `for()` loop uses `instrlist_first()` and `instr_get_next()` to look at each instruction in a block. Using `instrlist_first()` and `instr_get_next()` to look at each instruction in a block is a standard DynamoRIO method used in many clients.

b) The drmgr_is_emulation_start() function is used to detect if an instruction is the start of a sequence of instructions which are emulating a non-native instruction. There is also a drmgr_is_emulation_end() function which detects the end of the sequence but it is not required in this client as we only want to know if there is at least one emulated instruction in the block. See opcodes_emulated.cpp as an example of how drmgr_is_emulation_start() and drmgr_is_emulation_end() are used together.

Note

The reference documentation for these functions is not yet available at the DynamoRIO web site. See Emulation functions reference for a full description of these functions.

---

c) Instead of using dr_insert_clean_call(), as in opcodes_emulated.cpp, the client uses drx_insert_counter_update() to increment native_count and emulated_count. The difference is that dr_insert_clean_call() inserts a user-defined function, which is run when the block is executed. Whereas, drx_insert_counter_update() inserts its own code to increment a variable, which is run when the block is executed. See the DynamoRIO API reference documentation for more details.

4. Download the files bbcount.c and bbcount_tut2.c and compare them with a diff viewer to look at the modifications in full.

5. To build the modified client, add bbcount_tut2.c to /<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/CMakeLists.txt:

```
. . .
add_sample_client(bbcount      "bbcount.c"       "drmgr;drreg;drx")
add_sample_client(bbcount_tut2 "bbcount_tut2.c" "drmgr;drreg;drx")
add_sample_client(bbsize       "bbsize.c"        "drmgr")
. . .
```

6. Run cmake.

Note

The current version of Arm Instruction Emulator (22.0) requires that clients are built with GCC version 7.1.0 or higher:

```
cmake .
```

which returns:

```
-- The C compiler identification is GNU 7.1.0
-- The CXX compiler identification is GNU 7.1.0
-- Check for working C compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/cc
-- Check for working C compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/c++
```

```
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/c++ -- works
-- Detecting CXX compiler ABI info -- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features -- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples
```

7. Run make:

```
make
```

Which returns:

```
. . .
Scanning dependencies of target bbcount_tut2
[ 46%] Building C object CMakeFiles/bbcount_tut2.dir/bbcount_tut2.c.o
[ 48%] Linking C shared library bin/libbbcount_tut2.so
Usage: pass to drconfig or drrun: -c /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin/
libbbcount_tut2.so
[ 48%] Built target bbcount_tut2
. . .
```

8. Copy the built client from `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin` to `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin64`:

```
cp bin/libbbcount_tut2.so ./bin64/
file bin64/libbbcount_tut2.so bin64/libbbcount_tut2.so: ELF 64-bit LSB shared
 object, ARM aarch64, version 1 (SYSV), dynamically linked, not stripped
```

9. Run the modified client:

```
armie -msve-vector-bits=128 -i libbbcount_tut2.so -- ./example
```

The output now includes a count for blocks which contain at least one emulated instruction:

```
Client bbcount is running
i       a[i]    b[i]    c[i]
=============================
0       197     283     86
1       262     277     15
2       258     293     35
. . .
1021    165     234     69
1022    232     295     63
1023    204     235     31
Instrumentation results:
449306 native basic block executions
   256 emulated basic block executions
  1971 basic blocks needed flag saving
     0 basic blocks did not
```

Results

The output now includes a count for blocks which contain at least one emulated instruction.

Example 3-1: Examples

For examples of typical usage, see:

- `samples/inscount_emulated.cpp`
- `samples/instrace_emulated.c`
- `samples/memtrace_simple.c`
- `samples/memtrace_emulated.c`
- `samples/meminstrace_emulated.c`
- `samples/opcodes_emulated.cpp`

and the examples described in Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator.

Related information

Building custom analysis instrumentation on page 37
Emulation functions reference on page 51
About instrumentation clients on page 44
Arm Instruction Emulator

3.3 Building custom analysis instrumentation

Using the DynamoRIO API, you can change existing instrumentation clients or write your own from scratch. This tutorial describes how to modify the instrumentation of an existing client for your own purposes and build and execute the modified client with Arm® Instruction Emulator.

Before you begin

- You need a good working knowledge about the DynamoRIO API. DynamoRIO documentation is available and includes DynamoRIO's event driven usage model example clients, from which `inscount_emulated.cpp`, `opcodes_emulated.cpp`, and `memtrace_simple.c` are derived.
- To learn how to run a pre-built instrumentation client, work through Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator.
- Understand the About instrumentation clients, `libopcodes_emulated.so` and its implementation in the file `opcodes_emulated.cpp`.

Procedure

1. Use the following command to run Arm Instruction Emulator, with the pre-built instrumentation client, `libopcodes_emulated.so`. This client writes native AArch64 opcode counts to stdout and emulated counts to a file:

```
armie -msve-vector-bits=128 -i libopcodes_emulated.so -- ./example
```

Which returns:

```
Client opcodes_emulated is running
i       a[i]    b[i]    c[i]
=============================
0       197     283     86
1       262     277     15
. . .
1022    232     295     63
1023    204     235     31
Opcode execution counts in AArch64 mode:
      34900 : bl
      39725 : and
      41232 : csel
      44149 : ret
      54344 : ldrb
      68104 : cbnz
      73037 : ldp
      77676 : cbz
      79184 : stp
     100349 : sub
     110960 : movz
     126343 : str
     144182 : bcond
     171068 : subs
     171899 : orr
     183813 : add
     234517 : ldr
7 unique emulated instructions written to undecoded.txt
```

The file `undecoded.txt` contains:

```
256 : 0xe54842e0
256 : 0xa54842c1
256 : 0xa54842a0
256 : 0x25a91d00
256 : 0x04b0e3e8
256 : 0x04a10400
  1 : 0x25a91fe0
```

We are going to modify this instrumentation client, so that it writes both native and emulated counts to stdout in a format which makes it easier to be parsed by scripts when running and collating output from many applications, typically in an automated test environment.

Note

To correctly modify the `libopcodes_emulated.so` client, you must understand its existing implementation, `opcodes_emulated.cpp`. Refer to About instrumentation clients for a detailed description of instrumentation client structure.

2. Copy the `opcodes_emulated.cpp` file to a new file, `opcodes_emulated_tut1.cpp` and save it in the following location:

   `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples`

3. Edit `opcodes_emulated_tut1.cpp` to merge `opcount()` and `record_emulated_inst()` into one function:

   `opcodes_emulated.cpp`:

```
static void
record_emulated_inst(uint code)
{
  emulated[code]++;
}
static void
opcount(uint opcode)
{
  count[opcode]++;
}
```

   `opcodes_emulated_tut1.cpp`:

```
static void
opcount(uint opcode, int is_emulated)
{
    if (is_emulated == 0)
        count[opcode]++;
    else
        emulated[opcode]++;
}
```

4. Update the `dr_insert_clean_call()` calls which insert `opcount()`:

   `opcodes_emulated.cpp`:

```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
    instr_t *instr;
    for (instr = instrlist_first(bb);
        instr != NULL;
        instr = instr_get_next(instr)) {
        if (drmgr_is_emulation_start(instr)) {
            is_emulation = true;
            emulated_instr_t emulated;
            drmgr_get_emulated_instr_data(instr, &emulated);
            dr_insert_clean_call(drcontext, bb, instr,
                                (void *)record_emulated_inst, false, 1,
                                OPND_CREATE_INT32(
                                    instr_get_raw_word(emulated.instr, 0)));
        }
        if (drmgr_is_emulation_end(instr))
            is_emulation = false;
        if (is_emulation)
            continue;
        if (!instr_is_app(instr))
            continue;
        dr_insert_clean_call(drcontext, bb, instr,
                            (void *)opcount, false, 1,
```

```
                            OPND_CREATE_INT32(instr_get_opcode(instr)));
    }
    return DR_EMIT_DEFAULT;
}
```

`opcodes_emulated_tut1.cpp`:

```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
    instr_t *instr;
    for (instr = instrlist_first(bb);
        instr != NULL;
        instr = instr_get_next(instr)) {
        if (drmgr_is_emulation_start(instr)) {
            is_emulation = true;
            emulated_instr_t emulated;
            drmgr_get_emulated_instr_data(instr, &emulated);
            dr_insert_clean_call(drcontext, bb, instr,
                (void *)opcount, false, 2,
                OPND_CREATE_INT32(instr_get_raw_word(emulated.instr, 0)),
                OPND_CREATE_INT(1));
        }
        if (drmgr_is_emulation_end(instr))
            is_emulation = false;
        if (is_emulation)
            continue;
        if (!instr_is_app(instr))
            continue;
        dr_insert_clean_call(drcontext, bb, instr,
                            (void *)opcount, false, 2,
                            OPND_CREATE_INT32(instr_get_opcode(instr)),
                            OPND_CREATE_INT(0));
    }
    return DR_EMIT_DEFAULT;
}
```

Notice that by merging `opcount()` and `record_emulated_inst()` into one callback function, `opcount()`, the `dr_insert_clean_call()` functions, which insert `opcount()`, must now define two input parameters, rather than one. The `dr_insert_clean_call()` functions must also pass `1` for emulated instructions and `0` for native instructions.

5. Update `event_exit()` to write the emulated instruction data to stdout rather than a file:

`opcodes_emulated.cpp`:

```
static void
event_exit(void)
{
#ifdef SHOW_RESULTS
    char msg[(NUM_COUNT_SHOW + 2) * 80];
    int len, i;
    size_t sofar = 0;
    /* First, sort the counts */
    uint indices[NUM_COUNT];
    /* Initialise indices */
    for (i = 0; i < NUM_COUNT; i++)
        indices[i] = i;
    qsort(indices, NUM_COUNT, sizeof(indices[0]), compare_counts);
    len = dr_snprintf(msg, sizeof(msg) / sizeof(msg[0]),
                      "Opcode execution counts in AArch64 mode:\n");
    DR_ASSERT(len > 0);
    sofar += len;
```

```
    for (i = OP_LAST - 1 - NUM_COUNT_SHOW; i <= OP_LAST; i++) {
        if(count[indices[i]] != 0) {
            len = dr_snprintf(msg + sofar, sizeof(msg) / sizeof(msg[0]) - sofar,
                              "  %9lu : %-15s\n", count[indices[i]],
                              decode_opcode_name(indices[i]));
            DR_ASSERT(len > 0);
            sofar += len;
        }
    }
    len = dr_snprintf(msg + sofar, sizeof(msg) / sizeof(msg[0]) - sofar,
          "%u unique emulated instructions written to undecoded.txt\n",
          emulated.size());
    DR_ASSERT(len > 0);
    sofar += len;
    NULL_TERMINATE(msg);
    DISPLAY_STRING(msg);
#endif /* SHOW_RESULTS */
    map<uint,long>::iterator iter;
    multimap<long,uint>::reverse_iterator iter2;
    for(iter=emulated.begin(); iter!=emulated.end();++iter) {
        ranks.insert(make_pair(iter->second,iter->first));
    }
    for(iter2=ranks.rbegin(); iter2!=ranks.rend(); ++iter2) {
        fprintf(file, "%9lu : 0x%08x\n", iter2->first, iter2->second);
    }
    fclose(file);
    emulated.clear();
    if (!drmgr_unregister_bb_app2app_event(event_basic_block))
      DR_ASSERT(false);
    drmgr_exit();
}
```

`opcodes_emulated_tut1.cpp`:

```
static void
event_exit(void)
{
#ifdef SHOW_RESULTS
    char msg[(NUM_COUNT_SHOW + 2) * 80];
    int len, i;
    size_t sofar = 0;
    /* First, sort the counts */
    uint indices[NUM_COUNT];
    /* Initialise indices */
    for (i = 0; i < NUM_COUNT; i++)
        indices[i] = i;
    qsort(indices, NUM_COUNT, sizeof(indices[0]), compare_counts);
    len = dr_snprintf(msg, sizeof(msg) / sizeof(msg[0]),
                      "Opcode execution counts for AArch64 instructions:\n");
    DR_ASSERT(len > 0);
    sofar += len;
    for (i = OP_LAST - 1 - NUM_COUNT_SHOW; i <= OP_LAST; i++) {
        if(count[indices[i]] != 0) {
            len = dr_snprintf(msg + sofar, sizeof(msg) / sizeof(msg[0]) - sofar,
                              "  %9lu : %-15s\n", count[indices[i]],
                              decode_opcode_name(indices[i]));
            DR_ASSERT(len > 0);
            sofar += len;
        }
    }
    len = dr_snprintf(msg + sofar, sizeof(msg) / sizeof(msg[0]) - sofar,
          "Instruction execution counts for %u emulated instructions:",
          emulated.size());
    DR_ASSERT(len > 0);
    sofar += len;
    NULL_TERMINATE(msg);
    DISPLAY_STRING(msg);
#endif /* SHOW_RESULTS */
```

```
     map<uint,long>::iterator iter;
     multimap<long,uint>::reverse_iterator iter2;
     for(iter=emulated.begin(); iter!=emulated.end();++iter) {
         ranks.insert(make_pair(iter->second,iter->first));
     }
     for(iter2=ranks.rbegin(); iter2!=ranks.rend(); ++iter2) {
         dr_printf("  %9lu : 0x%08x\n", iter2->first, iter2->second);
     }
     fclose(file);
     emulated.clear();
     if (!drmgr_unregister_bb_app2app_event(event_basic_block))
       DR_ASSERT(false);
     drmgr_exit();
}
```

Download the files for `opcodes_emulated.cpp` and `opcodes_emulated_tut1.cpp` and compare them with a diff viewer to view the modifications in full.

6. To build the modified client, add `opcodes_emulated_tut1.cpp` to `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/CMakeLists.txt`:

```
. . .
add_sample_client(opcodes "opcodes.c" "drmgr;drreg;drx")
add_sample_client(opcodes_emulated "opcodes_emulated.cpp" "drmgr;drreg")
add_sample_client(opcodes_emulated_tut1 "opcodes_emulated_tut1.cpp"
 "drmgr;drreg")
add_sample_client(stl_test "stl_test.cpp" "")
. . .
```

7. Run `cmake`.

Note

The current version of Arm Instruction Emulator (22.0) requires that clients are built with GCC version 7.1.0 or higher:

```
cmake .
```

Which returns:

```
-- The C compiler identification is GNU 7.1.0
-- The CXX compiler identification is GNU 7.1.0
-- Check for working C compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/cc
-- Check for working C compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/cc -- works
-- Detecting C compiler ABI info -- Detecting C compiler ABI info - done --
 Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/c++
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_Generic-
AArch64_SUSE-12_aarch64-linux/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features -- Detecting CXX compile features - done
-- Configuring done
-- Generating done
```

```
-- Build files have been written to: /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples
```

8. Run `make`:

```
make
```

Which returns:

```
. . .
Scanning dependencies of target opcodes_emulated_tut1
[ 7%] Building CXX object CMakeFiles/opcodes_emulated_tut1.dir/
opcodes_emulated_tut1.cpp.o
[ 9%] Linking CXX shared library bin/libopcodes_emulated_tut1.so
Usage: pass to drconfig or drrun: -c /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin/
libopcodes_emulated_tut1.so
[ 9%] Built target opcodes_emulated_tut1
. . .
```

9. Copy the built client from:

For example:

```
cp bin/libopcodes_emulated_tut1.so ./bin64/
file ./libopcodes_emulated_tut1.so ./libopcodes_emulated_tut1.so: ELF 64-bit LSB
 shared object, ARM aarch64, version 1 (SYSV), dynamically linked, not stripped
```

10. Run the modified client. Now, the emulated instruction output is written to stdout and the `undecoded.txt` file is not created:

```
armie -msve-vector-bits=128 -i libopcodes_emulated_tut1.so -- ./example
```

Which returns:

```
. . .
1022 232 295 63
1023 204 235 31
Opcode execution counts for AArch64 instructions:
 34900 : bl
 39725 : and
 41232 : csel
 44149 : ret
 54344 : ldrb
 68104 : cbnz
 73037 : ldp
 77676 : cbz
 79184 : stp
 100349 : sub
 110960 : movz
 126343 : str
 144182 : bcond
 171068 : subs
 171899 : orr
 183813 : add
 234517 : ldr
Instruction execution counts for 7 emulated instructions:
 256 : 0xe54842e0
 256 : 0xa54842c1
 256 : 0xa54842a0
```

```
256 : 0x25a91d00
256 : 0x04b0e3e8
256 : 0x04a10400
1 : 0x25a91fe0
```

Results

Notice that the emulated instructions appear as raw encodings rather than mnemonics. This is a reflection of the current state of emulation support in the Public DynamoRIO API. Arm is working to improve such emulated instrumentation features and more comprehensive features will be available in the public API for future Arm Instruction Emulator releases.

Until then, as a workaround, a helper script is provided with Arm Instruction Emulator, `enc2instr.py`, which can be used to disassemble the encodings in your own post-processing scripts:

```
export LLVM_MC=/<install-dir>/arm-linux-compiler-<xx.y>_Generic-AArch64_<OS>-<OS-version>_aarch64-linux/llvm-bin/llvm-mc
echo 0xe54842e0 | /<install-dir>/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux//bin64/enc2instr.py 0xe54842e0 : st1w {z0.s}, p0, [x23, x8, lsl #2]
```

Next steps

- Build an emulation-aware instrumentation client

Related information

Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator on page 19
Arm Instruction Emulator

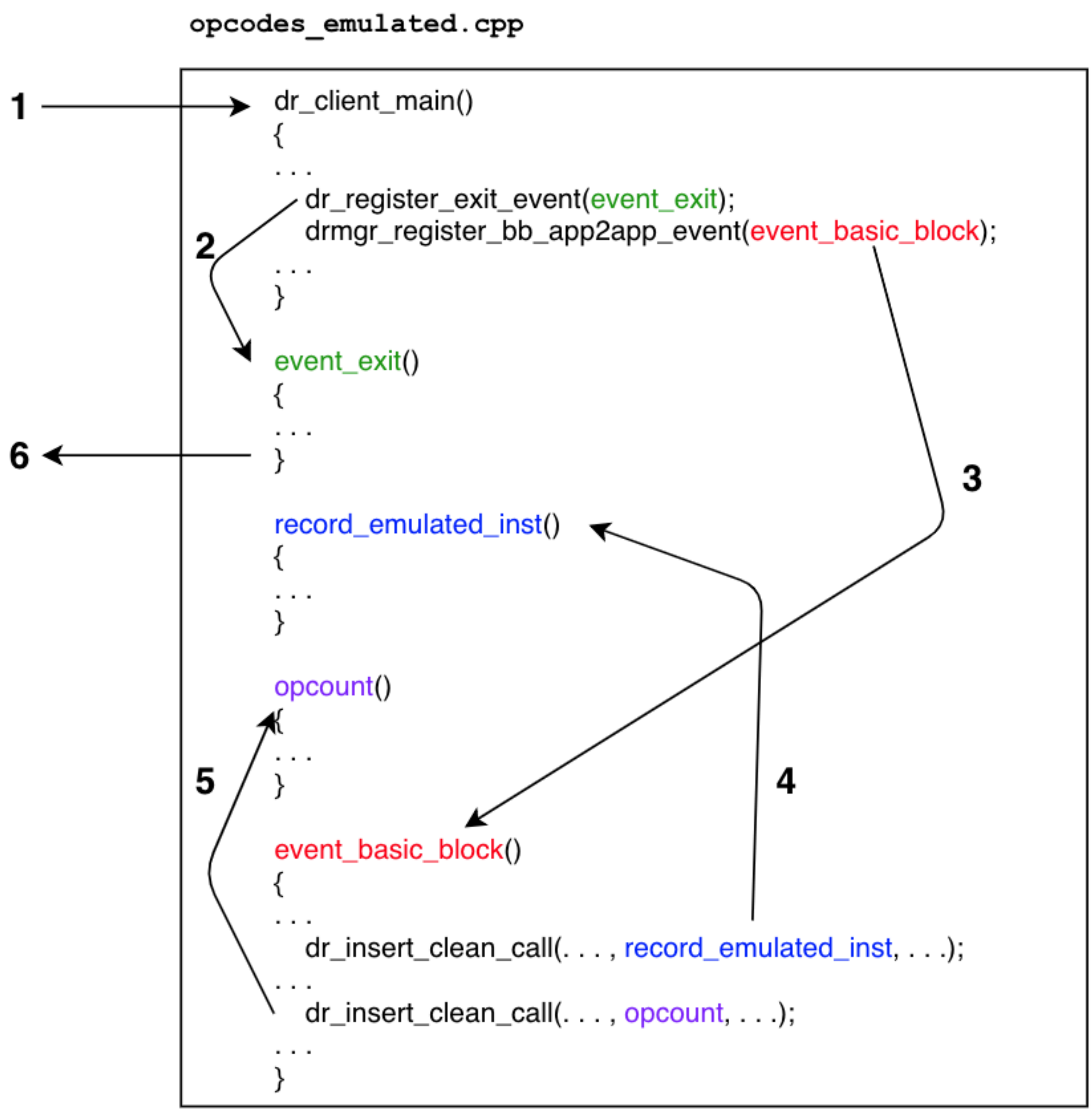
3.4 About instrumentation clients

This topic describes the basic structure of an instrumentation client, including the main events which occur during execution and what is typically done in each event.

Arm® Instruction Emulator provides a set of instrumentation clients which can be used to analyze SVE binaries at runtime. The term 'instrumentation client' in this context refers to how Arm Instruction Emulator uses DynamoRIO to work as an analysis tool as well as an emulator. Arm Instruction Emulator is invoked with an instrumentation client and the SVE binary to be emulated and analyzed. The client is simply a shared object file which uses the DynamoRIO API to capture and process wanted run-time events.

To correctly modify the `libopcodes_emulated.so` client, you must understand its existing implementation, `opcodes_emulated.cpp` (download opcodes_emulated.cpp). The diagram below shows the key functions in `opcodes_emulated.cpp` and how they relate to each other.

Figure 3-2: Diagram showing the key functions in opcodes_emulated.cpp



The easiest way to understand the client is to think of it as event-driven. Each function is called as a result of events which occur as the application is running:

1. DynamoRIO loads and runs the client, calling `dr_client_main()`, before beginning to execute the application.
2. In `dr_client_main()`, the client registers a function which is called just before the client stops running, `event_exit()`. Registering such a function for an event is usually referred to as a 'callback function'.
3. In `dr_client_main()`, the client registers a callback function as each block of code in the application is prepared before being executed.

4. In `event_basic_block()`, the client registers a callback function which is executed for each emulated instruction which appears in the code of the application, `record_emulated_inst()`. The `record_emulated_inst()` function is the instrumentation which is the purpose of the client.
5. In `event_basic_block()`, the client registers a callback function which is executed for each native instruction which appears in the code of the application, `opcount()`. The `opcount()` function is the instrumentation which is the purpose of the client.
6. The application stops running and DynamoRIO calls `event_exit()`.

The preceding information is a simplified explanation of how a client operates. For a more detailed information, read the `opcodes_emulated.cpp` file, which can be downloaded from the Arm Developer website, and refer to details of key functions in the DynamoRIO functions reference manual, especially:

- **`dr_insert_clean_call()`, which implements the instrumentation you** want.
- **`drmgr_register_bb_app2app_event()`, which defines where the** instrumentation must be inserted.

Code Transformation and code Execution

If you are new to the DynamoRIO Dynamic Binary Instrumentation (DBI) tool platform in general, and DynamoRIO in particular, ensure you understand the method by which instrumentation is added to application code.

Remember that instrumentation occurs in two phases, *transformation* and *execution*:

- Transformation - Instrumentation code is inserted into the application code.
- Execution - The application code runs, including the instrumentation code which was inserted during transformation.

DynamoRIO performs transformation and execution transparently, provided that you conform to the rules of its API.

In the preceding example, `event_basic_block()` is the transformation phase. Calls to `opcount()` or `record_emulated_inst()` are inserted for each instruction but are not called at transformation time. If or when a particular block of code is run at execution time, those functions are called, to increment and store the instruction and count.

This is a subtle distinction for new users. The best way to think of the difference is to recognize that `dr_insert_clean_call()` will be called once when a block of application code is transformed but the function it registered may be called many times when the block is executed.

Related information

Building custom analysis instrumentation on page 37
Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator on page 19
Emulation functions reference on page 51
Arm Instruction Emulator

3.5 View the drrun command

This topic describes how to use the `-s` or `--show-drrun-cmd` Arm® Instruction Emulator option to output the full DynamoRIO `drrun` command that Arm Instruction Emulator uses.

About this task

The `-s` option is provided to enable the full range of options for `drrun`, and to pass command-line arguments to clients. Without this feature, options and arguments are required to be passed through the `-a` or `-arg-iclient` options.

Procedure

1. Run Arm Instruction Emulator with the `-s` option, using the example described in Get started with Arm Instruction Emulator:

```
armie -msve-vector-bits=128 -s -- ./example
```

   Which returns:

```
/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-
AArch64_<OS>_aarch64-linux/bin64/drrun -max_bb_instrs 32 -max_trace_bbs 4
 -c /<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-
AArch64_<OS>_aarch64-linux/lib64/release/libsve_128.so -- ./example
 i       a[i]     b[i]     c[i]
=============================
0       197      283      86
1       262      277      15
. . .
1021    165      234      69
1022    232      295      63
1023    204      235      31
```

   Notice that `drrun` uses the emulation client `libsve_128.so` to run the example binary.

2. If an instrumentation client is specified:

```
armie -msve-vector-bits=128 -s -i libinscount_emulated.so -- ./example
```

   Which returns:

```
/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-
AArch64_<OS>_aarch64-linux/bin64/drrun -client /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/lib64/release/
libsve_128.so 0 "" -client /<path/to/your/installation>/arm-instruction-emulator-
<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin64/libinscount_emulated.so 1
 "" -max_bb_instrs 32 -max_trace_bbs 4 -- ./example
Client inscount is running
. . .
1022    232      295      63
1023    204      235      31
    2134094 instructions executed of which 1537 were emulated instructions
```

   Notice that `drrun` now uses two clients: the emulation client `libsve_128.so` and `libinscount_emulated.so` to run and count instructions executed by example.

3. The `-only_from_app` option for the `libinscount_emulated.so` client only counts instructions executed by the application, rather than also including linked libraries. You can copy and paste the above command and add `-only_from_app`:

```
/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_Generic-
AArch64_<OS>_aarch64-linux/bin64/drrun -client /<path/to/your/installation>/arm-
instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/lib64/release/
libsve_128.so 0 "" -client /<path/to/your/installation>/arm-instruction-emulator-
<xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin64/libinscount_emulated.so 1
 "-only_from_app" -max_bb_instrs 32 -max_trace_bbs 4 -- ./example
Client inscount is running
    . . .
    1021    165     234     69
    1022    232     295     63
    1023    204     235     31
    42902 instructions executed of which 1537 were emulated instructions
```

Notice that the native AArch64 instruction count has dropped to 42902, from 2134094, due to the exclusion of library instructions.

Related information

Building custom analysis instrumentation on page 37
Get started with Arm Instruction Emulator on page 12
Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator on page 19
Arm Instruction Emulator

4 Reference

This section contains reference information for armie command and the emulation functions included with Arm® Instruction Emulator.

4.1 armie command reference

The armie command runs a compiled binary using Arm® Instruction Emulator. Arm Instruction Emulator is an emulator that can execute AArch64 Scalable Vector Extension (SVE) instructions on any Armv8-A-based hardware.

Note

The following content is relevant for Arm Instruction Emulator versions 18.2 and later. If you are using a previous version of Arm Instruction Emulator, download the Arm Instruction Emulator v1.2.1 user guide.

Usage

To execute and provide operational instructions to the Arm Instruction Emulator, use:

```
armie  [options] -- <command to execute>
```

Options

Table 4-1: armie command options

Option	Description
-m<string> -msve-vector-bits=<uint> -mlist-vector-lengths	Architecture-specific options. -msve-vector-bits=<uint> specifies the vector length to use. <uint> must be a multiple of 128 bits, up to a maxiumum of 2048 bits. -mlist-vector-lengths lists all the valid vector lengths.
-e <client> --eclient <client>	Use a DynamoRIO API-based emulation client. The libmemtrace_sve_<width>.so SVE emulation clients (in lib64/release) can be used with the memory tracing instrumentation clients. <width> is the vector width between 128 bits and 2048 bits (in increments of 128 bits). **Note:** If an SVE emulation client is not specified, the default SVE client is used by armie.

Option	Description

`-i <client>`
 	
`--iclient <client>`
	Use a DynamoRIO API-based instrumentation client.
	
	The following instrumentation clients are provided with Arm Instruction Emulator (in `samples/bin64`):
	• `libinscount_emulated.so`
	• `libinstrace_emulated.so`
	• `libmeminstrace_emulated.so`
	• `libmemtrace_emulated.so`
	• `libopcodes_emulated.so`
	• `libemulated_regs.so`
	
	To learn how to create your own custom instrumentation client, see Building custom analysis instrumentation and Build an emulation-aware instrumentation client

`-a`
 	
`--arg-iclient <string>`
	Pass an (optional) `<string>` argument to the instrumentation client.

`-x`
 	
`--unsafe-ldstex`
	This options is DEPRECATED
	
	The `-x` and `--unsafe-ldstex` options enable a workaround to avoid an exclusive load/store bug on specific AArch64 hardware. `-x` is always enabled and is no longer set from the command line, if required.
	
	For more information about the details of the need for this workaround, see the **Known Issues** section in `RELEASE_NOTES.txt`.

`-y`
 	
`--safe-ldstex`
	Use `-y` in the unlikely event that `-x` or `--unsafe-ldstex` must be disabled.

`-s`
 	
`--show-drrun-cmd`
	Write the full DynamoRIO `drrun` command used to execute `armie` to `stderr`.
	
	`-s` can be useful when debugging or developing clients.

`-h`
 	
`--help`
	Show the command help.

`-V`
 	
`--version`
	Print the version.

Example: Use `-mlist-vector-lengths` to list the valid vector lengths

To list all valid vector lengths, use:

```
armie -mlist-vector-lengths
```

Which returns:

```
128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
```

Example: Use '-msve-vector-bits' to specify the number of vector bits

To run the compiled binary 'sve_program' with 256-bit vectors, use:

```
armie -msve-vector-bits=256 -- ./sve_program
```

Related information

Get started with Arm Instruction Emulator on page 12
Analyze Scalable Vector Extension (SVE) applications with Arm Instruction Emulator on page 19

4.2 Emulation functions reference

This topic describes the emulation functions applicable to Arm® Instruction Emulator.

Arm Instruction Emulator (ArmIE) is based on the DynamoRIO Dynamic Binary Instrumentation (DBI) tool platform and allows developers to use the API of DynamoRIO to write instrumentation clients which run alongside the SVE emulation client. These instrumentation clients can allow you to analyze SVE binaries at runtime:

- `drmgr_is_emulation_start()`: See the DynamoRIO documentation for drmgr_is_emulation_start()
- `drmgr_is_emulation_end()`: See the DynamoRIO documentation for drmgr_is_emulation_end()
- `drmgr_get_emulated_instr_data()`: See the DynamoRIO documentation for drmgr_get_emulated_instr_data()
- `emulated_instr_t`: See the DynamoRIO documentation for emulated_instr_t

Related information

Get started with Arm Instruction Emulator on page 12
Arm Instruction Emulator
API Usage Tutorial
Learn about SVE

5 Further resources

Lists the additional resources available which you can to use to learn more about Arm® Instruction Emulator or the Scalable Vector Extension (SVE).

5.1 Arm Instruction Emulator resources

This topic lists some useful resources where you can read more about Arm® Instruction Emulator.

- Arm Instruction Emulator
- Download Arm Instruction Emulator
- Release history
- Get help
- Blog: DynamoRIO and ArmIE
- Blog: Optimizing HPCG for Arm SVE

5.2 Scalable Vector Extension (SVE) resources

This topic lists some useful resources you can use to learn more about the Scalable Vector Extension (SVE).

- Porting and Tuning HPC Applications for Arm SVE

  A guide to the tools and methodologies to porting your applications to SVE-enabled hardware, or to emulate with Arm® Instruction Emulator.

- Past presentations and hackathon materials

  Past presentations at Arm events, including downloadable SVE Hackathon materials.

- White Paper: A sneak peek into SVE and VLA programming

  An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.

- White Paper: Arm Scalable Vector Extension and application to Machine Learning

  In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of a machine learning system. The examples are written using the Vector Length Agnostic (VLA) approach introduced by the Scalable Vector Extension (SVE).

- Arm C Language Extensions (ACLE) for SVE

  The SVE ACLE defines a set of C and C++ types and accessors for SVE vectors and predicates.

- DWARF for the ARM® 64-bit Architecture (AArch64) with SVE support

  This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.

- Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support

  This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.

- Arm Architecture Reference Manual for A-profile architecture

  This guide includes information that describes the Scalable Vector Extension to the Armv8-A architecture profile.