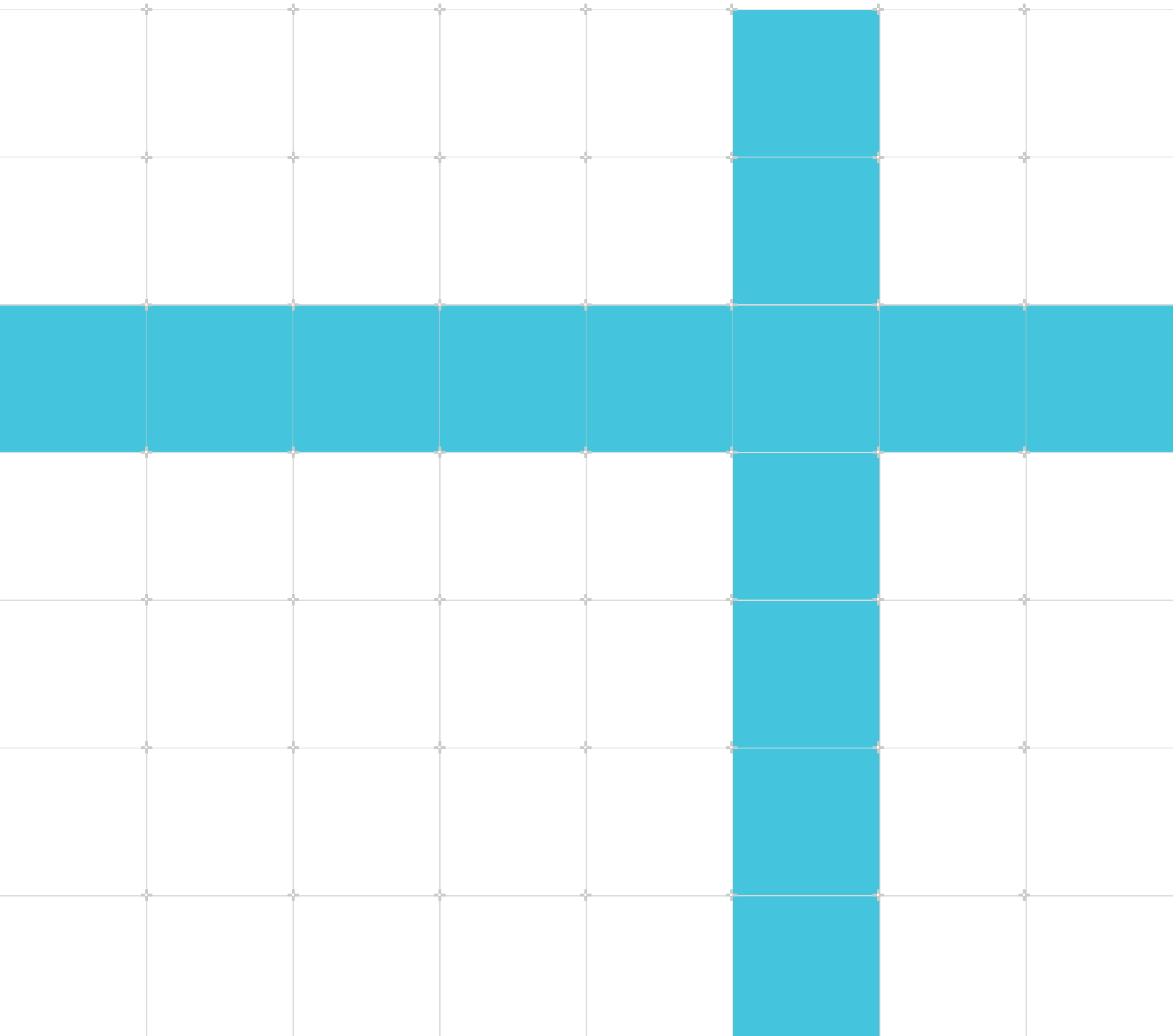
Arm Instruction Emulator

Version 2500

Developer and Reference Guide

Issue 00
102190_25.0_00_en



Arm Instruction Emulator Developer and Reference Guide

This document (102190_25.0_00_en) was issued on 2025-07-25. There might be a later issue at https://developer.arm.com/documentation/102190

The product version is 2500.

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

Users of Arm Instruction Emulator

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.

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.

Contents

1. Get started....5
1.1 Install Arm IE....5
1.2 Get started with Arm IE....7
1.2.1 Compile and run SVE code....7
1.2.2 Compile and run an example application....8
1.2.3 Compile, vectorize, and run an application with SVE code....9
1.3 Specifying Features to be Emulated....10
1.4 Troubleshoot using -s....11

2. Tutorials....13
2.1 Running SME applications with Arm IE....13
2.2 Analyze SVE applications with Arm IE....17
2.2.1 About analyzing SVE programs....17
2.2.2 Analyze an SVE program....18
2.2.3 Analyze an application with SVE code....18
2.2.4 Analyze the effect of the vector length on the number of AArch64 and emulated SVE instructions....20
2.2.5 Analyze Regions Of Interest....22
2.2.6 Dynamic instruction counts....25
2.2.7 Examine memory access behavior....26
2.3 Build an emulation-aware instrumentation client....27
2.4 Build custom instrumentation....32
2.5 About instrumentation clients....38
2.6 View the drrun command....41

3. Reference....43
3.1 Arm IE command reference....43
3.2 Emulation functions reference....44

4. Security and Privacy....46
4.1 Dynamic binary modification....46

5. Further resources....47

5.1 Arm IE resources..... 47
5.2 SVE resources..... 47
5.3 SME resources..... 48

Proprietary notice..... 49

Product and document information..... 51
Product status..... 51
Revision history..... 51
Conventions..... 52

Useful resources..... 54

1. Get started

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

Arm Instruction Emulator (referred to in this document as Arm IE, with the executable name `armie`) is an emulator that runs on any Arm®v8-A-based or Arm®v9-A-based AArch64 platform and emulates Scalable Vector Extension (SVE) and Scalable Matrix Extension (SME) instructions. Arm IE lets you develop SVE or SME code without needing access to SVE or SME-enabled hardware.

1.1 Install Arm IE

Follow these steps to download and install Arm IE.

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 IE package for your Linux host platform. To download Arm IE, see the Arm IE downloads page on the Arm Developer website.
2. Extract the downloaded package:

   ```
   $ tar -xvzf <package_name>.tgz
   ```

   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. This can be done as a privileged user:

   ```
   $ sudo ./arm-instruction-emulator_25.0*.sh
   ```

   which will install to `/opt/arm/<package_name>`, or as a non-privileged user with the `--install-to <install-dir>` option which will install to the specified `<install-dir>`.

   ```
   $ ./arm-instruction-emulator_25.0*.sh --install-to <install-dir>
   ```

   The installation script supports the following options:

   `-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.


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 IE 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 IE environment module, add the Arm IE installation's modulefiles directory to the list of locations searched:

     ```
     $ module use <install-dir>/modulefiles/
     ```

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

     ```
     $ module avail
     ```

  3. Load the appropriate Arm IE 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 armie25/25.0
     ```


Tip

Add the `module use` and `module load` commands to your `.profile` to run them automatically every time you log in.

4. The Arm IE `<install-dir>/bin64` directory path should now be in the `PATH` environment variable. Run the `armie` command and the help text will be displayed:

```
$ echo $PATH
/opt/arm/arm-instruction-emulator-25.0_RHEL-9/bin64:...
$ armie
. . .
```

- To learn how to use Arm IE, refer to Get started with Arm IE.
- To uninstall Arm IE, run the `uninstall.sh` script located in `<install-dir>/arm-instruction-emulator-<version>_<OS>-<OS_Version>/uninstall.sh`

1.2 Get started with Arm IE

This tutorial shows how to build Scalable Vector Extension (SVE) applications and run them with Arm IE.

1.2.1 Compile and run SVE code

The first example program is a simple 'Hello World' C program and is intended to show basic build and run commands. The second example is more complex and focuses on the type of source code which the compiler can autovectorise using SVE instructions.

Before you begin

- This task uses GCC as the compiler. Alternatively, you can use Clang.
- Load the Arm IE 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 armie25/25.0
```

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

```
$ echo $PATH
/<install-dir>/arm-instruction-emulator-25.0_RHEL-9/bin:...
```

Procedure

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

      ```
      $ armie -mvl=<length> ./<binary>
      ```

   b. Invoke Arm IE with an instrumentation client (`-i`) and specify the vector length in bytes to use:

      ```
      $ armie -mvl=<arg> -i <instrumentation_client> -- ./<binary>
      ```

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

1.2.2 Compile and run an example application

This example uses a simple 'Hello World' C program, and shows you how to compile it with GCC, and then run it using Arm IE. The example does not contain Scalable Vector Extension (SVE) code.

Procedure

1. 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;
   }
   ```

2. To generate an executable binary, compile your application with GCC.

   ```
   $ gcc -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 allows SVE instructions in the generated 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 IE.

3. Run the generated binary `hello` using Arm IE:

```
$ armie -mvl=32 ./hello
Hello World
```

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

To use Arm IE 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 Compile, vectorize, and run an application with SVE code.

Next steps

To learn how to analyze your application using the emulation and instrumentation clients available for Arm IE, see Analyze SVE applications with Arm IE.

1.2.3 Compile, vectorize, and run an application with SVE code

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

Procedure

1. 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 array parameters use the `restrict` keyword, which indicates to the compiler that they do not overlap in memory.

2. Compile the C code with GCC:

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

3. Run the binary with Arm IE:

```
$ armie -mvl=32 ./example
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 `-mvl` option lets you specify the vector length in bytes for Arm IE to use. The vector length must be a power of 2 between 16 and 256 bytes. To list all valid vector lengths, use the `-mlist-vector-lengths` option :

```
$ armie -mlist-vector-lengths
16 32 64 128 256 (bytes)
```

Next steps

To learn how to analyze your application using the instrumentation clients available for Arm IE, see Analyze SVE applications with Arm IE.

1.3 Specifying Features to be Emulated

The Armv8-A architecture specifies features which describe CPU capabilities. Examples are FEAT_FP16 which specifies support for half-precision floating-point arithmetic and FEAT_SVE2 specifies support for version 2 of the Scalable Vector Extension.

Arm IE enables users to select which features to emulate regardless of the host CPU's capabilities. By default, Arm IE will emulate those features not supported by the host CPU. For more information, see Arm IE command reference.

Note

Selective feature support is provided for advanced users. Typically, the majority of users will not need to specify which feature(s) to emulate, relying on default behaviour.

1.4 Troubleshoot using -s

This page describes how you can use the `-s` option to better understand what emulation commands and files Arm IE uses, and what to send to Arm Support if you require further assistance.

The -s and -show-drrun-cmd options

To show how Arm IE uses DynamoRIO's `drrun` command to emulate and instrument an SVE binary, use the `-s` (or `--show-drrun-cmd`) option.

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

```
$ armie -s -mvl=64 -i libinscount_emulated.so -- ./example
```

This command returns:

```
$ /path/to/armie/bin64/drrun -client /path/to/armie/lib64/release/libarmie.so
 0 64,32,0x2010,0x755ff3fddffcde -client /path/to/armie/samples/bin64/
libinscount_emulated.so 1 "" -max_bb_instrs 32 -max_trace_bbs 4 -- ./example
Client inscount is running
. . .
2421990 instructions executed of which 322 were emulated instructions
```

The `-s` option allows you to understand how Arm IE 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/armie/bin64/drrun -client /path/to/armie/lib64/release/libarmie.so
 0 64,32,0x2010,0x755ff3fddffcde -client /path/to/armie/samples/bin64/
libinscount_emulated.so 1 "-only_from_app" -max_bb_instrs 32 -max_trace_bbs 4 -- ./
example
Client inscount is running
. . .
39511 instructions executed of which 322 were emulated instructions
```

This example shows that the application used 39511 instructions, compared to 2421990 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 Arm IE 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 failure, error messages or some other form of output, (for example a core dump file), will appear depending on the nature of the failure. To request support, post a message on the Arm Community compilers and libraries support forum with details.


Note

When creating a new post on the forum, make sure to enter the `Arm Instruction Emulator` tag in the `Tags` section of the post.

In the case of a crash the operating system kernel may create 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 IE binary name (`armie`).

Core dump files should be uploaded to the support forum as part of the post. If you have confidentiality concerns regarding sensitive data in the core dump file, do not upload the file. However, without a core dump file, the support team might not be able to investigate your issue.

2. Tutorials

Learn how to build instrumentation clients for Arm IE, and how to use Arm IE to analyze your Scalable Vector Extension and Scalable Matrix Extension applications.

2.1 Running SME applications with Arm IE

Arm IE allows you to execute SME applications on non-SME hardware, allowing you to test SME applications before SME hardware is available. This tutorial shows how to build a simple SME application and run it under Arm IE. It also shows how to collect runtime information about the application using instrumentation clients.

Before you begin

This tutorial is based on the Single precision matrix-by-matrix multiplication tutorial. It expands on the tutorial by adding the required C code and showing how to run the application with Arm IE. However this tutorial does not attempt to explain SME concepts. See the SME Programmer's Guide for an introduction to SME.

This application has been built on an AArch64 machine with the `clang 17.0.1` toolchain. A recent toolchain is required in order to generate SME instructions.

You can install clang-17 with the commands:

```
$ sudo wget https://apt.llvm.org/llvm.sh
$ sudo chmod u+x llvm.sh
$ sudo ./llvm.sh 17
```

About this task

Matrix manipulations are an integral part of Artificial Intelligence (AI), particularly in machine learning, neural networks, and computer vision. The ability to perform matrix operations in a performant manner decreases the cost of AI, or increases the possibilities of what can be achieved on your hardware.

In this task we will build an SME application, observe that it cannot run on current hardware and then see how to run it and instrument it with Arm IE.

Procedure

1. Copy the SME example application code at `/path/to/armie/samples/sme_example_app/` to your home directory and build it. For example:

```
$ cp -r /opt/arm/arm-instruction-emulator-25.0_Generic-
AArch64_Ubuntu-22.04_aarch64-linux/samples/sme_example_app ~
$ cd ~/sme_example_app
$ clang-17 -O2 -fno-tree-vectorize -march=armv8.5-a+sme2+sve2-bitperm+bf16+sme-
i16i64+sme-f64f64 preprocess_l.s matmul_opt.s matmul_opt.c -o matmul_fp32
```


Note

The installation directory is not writable for users and you would have to change the permissions of that directory in order to build the application there. It is considered better practise to work in your home directory.

Try running the application natively:

```
$ ./matmul_fp32
```

Assuming that you are running the application on non-SME hardware, the following output is displayed:

```
Input Left Matrix:
 7.1  8.3  5.3  4.2  8.9
 3.9  8.2  4.3  3.7  5.6
 6.5  1.7  9.6  3.5  3.7
 5.7  7.7  9.3  9.4  9.7
 8.5  9.1  5.9  5.3  6.9
 9.7  6.8  3.4  3.9  3.8
 6.7  9.3  7.3  9.9  5.5
 1.9  4.7  5.8  9.5  2.3
Illegal instruction (core dumped)
```

2. Run the application using Arm IE, using a Vector Length of 16 bytes and a Streaming Vector Length of 64 bytes:

```
$ armie -mvl=16 -msvl=64 -- ./matmul_fp32
Input Left Matrix:
 7.1  8.3  5.3  4.2  8.9
 3.9  8.2  4.3  3.7  5.6
 6.5  1.7  9.6  3.5  3.7
 5.7  7.7  9.3  9.4  9.7
 8.5  9.1  5.9  5.3  6.9
 9.7  6.8  3.4  3.9  3.8
 6.7  9.3  7.3  9.9  5.5
 1.9  4.7  5.8  9.5  2.3

Processed Left Matrix:
 7.1  3.9  6.5  5.7  8.5  9.7  6.7  1.9  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 8.3  8.2  1.7  7.7  9.1  6.8  9.3  4.7  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 5.3  4.3  9.6  9.3  5.9  3.4  7.3  5.8  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 4.2  3.7  3.5  9.4  5.3  3.9  9.9  9.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 8.9  5.6  3.7  9.7  6.9  3.8  5.5  2.3  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

Input Right Matrix:
 7.8  5.2  3.0  9.2  8.0  4.1  2.8  6.1  3.6  9.3
 5.0  1.7  2.4  6.6  8.0  3.4  6.7  9.4  6.0  3.3
 5.7  4.1  1.8  6.0  3.5  8.5  5.9  8.6  2.5  2.8
 3.6  1.8  7.0  6.9  1.7  6.2  3.5  2.6  8.5  1.7
 9.3  4.4  8.9  9.8  9.5  9.7  5.7  7.9  1.3  2.4

Output Matrix:
225.0 119.5 159.4 268.1 233.4 214.8 172.2 248.1 135.9 136.8
161.3  83.2 114.9 196.2 171.3 157.7 136.1 191.7 112.7  95.1
160.9  98.6  98.3 189.0 140.3 171.6 119.6 176.5  92.2 107.8
260.0 140.5 204.4 319.0 247.9 281.0 210.6 288.2 182.5 143.7
228.7 123.8 156.5 277.9 236.0 215.7 177.5 256.4 154.0 151.2
178.4  99.7 112.7 218.7 186.6 152.8 128.1 192.5 122.3 137.9
227.2 122.6 173.8 289.0 222.6 235.9 190.1 260.3 189.5 143.5
127.0  68.9 114.4 171.4 111.1 154.3 117.4 148.5 133.3  71.1
```


Note

`Left Matrix` is transposed first, converting columns to rows to allow for efficient memory loading during the multiplication. The rows are padded with zeros to fill the Streaming Vector register. The Stream Vector Length (SVL) is 64 bytes and each number uses 4 bytes (single precision float) so each Streaming Vector register can hold 16 elements.

3. Try changing the SVL to 32:

```
$ armie -mvl=16 -msvl=32 -- ./matmul_fp32
--- cut ---

Processed Left Matrix:
 7.1  3.9  6.5  5.7  8.5  9.7  6.7  1.9  8.3  8.2  1.7  7.7  9.1  6.8  9.3  4.7
 5.3  4.3  9.6  9.3  5.9  3.4  7.3  5.8  4.2  3.7  3.5  9.4  5.3  3.9  9.9  9.5
 8.9  5.6  3.7  9.7  6.9  3.8  5.5  2.3  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

--- cut ---
```

The processed matrix is no longer padded with zeros and only half the memory is used.


Note

This simple C program does not adjust the amount of memory available for the Processed Left Matrix or the dimensions of the printed matrix.

4. Run Arm IE with an instrumentation client.
The sample instruction trace client, `instrace_emulated`, can be used to log the instructions run.

```
$ armie -mvl=16 -msvl=64 -i libinstrace_emulated.so -- ./matmul_fp32
Data file path/to/armie/samples/sme_example_app/
instrace.matmul_fp32.3919134.0000.log created

--- cut ---
```

A file with a name like `instrace.matmul_fp32.3919134.0000.log` is generated. If you inspect the file you can see that it is a simple list of instructions executed:

```
$ more instrace.matmul_fp32.3919134.0000.log
Format: <instr address>,;<opcode>,;<is memory>
0xffff86b4c6d0;0xd280001d;0
0xffff86b4c6d4;0xd280001e;0
0xffff86b4c6d8;0xaa0003e5;0
0xffff86b4c6dc;0xf94003e1;1

--- cut ---
```

For each instruction the current program counter, the instruction encoding and a flag to indicate if the instruction reads or writes to memory is logged.

Emulated instructions have their fields delimited with a `,` rather than a `;`.


Note

The `emulated` version of the `instrace` client will take into account which instructions have been emulated and will log the original instructions, so the output will include SME instructions.

5. Process the instruction log.
To aid processing of raw instruction data a Python script, `enc2instr.py` is provided that disassembles the instruction encodings. It is located in the same directory as the Arm IE executable and uses the LLVM disassembler `llvm-mc`. Set the environment variable `LLVM_MC` to point to the `llvm-mc` executable before running `enc2instr.py`. `llvm-mc` version 17 should be found in the same directory as `clang-17`:

```
$ readlink -f "$(which clang-17)"
/usr/lib/llvm-17/bin/clang

$ export LLVM_MC=/usr/lib/llvm-17/bin/llvm-mc
```

Check that you have pointed `LLVM_MC` to a recent version of `llvm-mc`:

```
$ ${LLVM_MC} --version
LLVM (http://llvm.org/):
LLVM version 17.0.1
```

To disassemble the instruction encodings in the above trace file:

```
$ tail -n +2 instrace.matmul_fp32.3919134.0000.log | \
> cut -d',' -f2 | \
> cut -d';' -f2 | \
> /path/to/enc2instr.py -mattr=armv8.5-a,+sme2,+sve2-bitperm,+bf16,+sme-i16i64,
+sme-f64f64 > \
> instrace.matmul_fp32.disassembled.txt
```

- The `tail` command outputs all the lines in the log file starting at line 2, missing out the column headers.
- The first `cut` command splits each line on the `,` delimiter and outputs the second field (the instruction encoding).
- The second `cut` command does this again using the `;` delimiter.
- The `enc2instr.py` script then uses the `llvm-mc` disassembler to disassemble each instruction.

The output file contains the original SME instructions:

```
$ cat instrace.matmul_fp32.disassembled.txt | grep fmopa
0x80820820 : fmopa za0.s, p2/m, p0/m, z1.s, z2.s
0x80820ca2 : fmopa za2.s, p3/m, p0/m, z5.s, z2.s
0x80832821 : fmopa za1.s, p2/m, p1/m, z1.s, z3.s
0x80832ca3 : fmopa za3.s, p3/m, p1/m, z5.s, z3.s

--- cut ---
```

2.2 Analyze SVE applications with Arm IE

This section describes how to use instrumentation clients and run your Scalable Vector Extension applications with Arm IE.


Note

The same instrumentation methods and principles apply to Scalar Matrix Extension applications.

2.2.1 About analyzing SVE programs

You can use Arm IE without any instrumentation clients 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.

For more information, see Get started with Arm IE.

To emulate and instrument SVE binaries on AArch64 hardware, Arm IE uses DynamoRIO. DynamoRIO is a publicly available Dynamic Binary Instrumentation tool which supports x86 and Arm platforms. DynamoRIO provides an API that enables you to write your own binary-level runtime instrumentation, and supplies some example instrumentation clients. Each Arm IE release integrates a stable version of DynamoRIO. Version 25.0 of Arm IE is based on version 11.0 of DynamoRIO.

Arm IE also provides a set of instrumentation clients that can be used to analyze SVE or SME binaries at runtime. In this context, 'instrumentation client' refers to how Arm IE uses DynamoRIO to work as an analysis tool as well as an emulator.


Note

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

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

Related information

- Build custom instrumentation
- Optimizing C/C++ code with SVE and SVE2

- Arm IE

2.2.2 Analyze an SVE program

This topic shows the basic steps for analyzing an Scalable Vector Extension (SVE) program.

About this task

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

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

  In this command, `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.
- Ensure that you have already compiled your application binary.

Procedure

1. To invoke Arm IE with an instrumentation client and the binary use:
   ```
   $ armie -mvl=16 -i <instrumentation_client> -- ./<binary>
   ```
2. Analyze the results provided by the clients.

Next steps

See more examples for analysis:

- Analyze an application with SVE code
- Analyze the effect of the vector length on the number of AArch64 and emulated SVE instructions
- Analyze Regions Of Interest

2.2.3 Analyze an application with SVE code

The following example shows how to count native AArch64 instructions as well as emulated Scalable Vector Extension (SVE) instructions.

Before you begin

`event_bb_analysis()` is the function which counts instructions in the sample client `<install-dir>/arm-instruction-emulator-25.0_Generic-AArch64_<OS>_aarch64-linux/samples/inscount_emulated.cpp`.

A simplified version of the function, with added comments, is shown below.

```
instr_t *instr, *next_instr;
instr_counts bb_counts;

/* bb_counts.native_instrs and bb_counts.emulated_instrs increment depending on
 * whether the instruction is emulated or not.
```

```
   */
  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);

    /* The count instructions example function distinguishes between emulated and
     * native instructions using the drmgr_is_emulation_start() and
     * drmgr_is_emulation_end() functions of DynamoRIO.
     */
    if (drmgr_is_emulation_start(instr)) {
        bb_counts.emulated_instrs++;
        is_emulation = true;

        /*
         * Data about the instruction being emulated can be extracted using
         * drmgr_get_emulated_instr_data() e.g:
         * emulated_instr_t emulated;
         * emulated.size = sizeof(emulated);
         * drmgr_get_emulated_instr_data(instr, &emulated);
         * unsigned int *sveinstr;
         * sveinstr = ((unsigned int *)instr_get_raw_bits(emulated.instr));
         */

        continue;
    }
    if (drmgr_is_emulation_end(instr)) {
        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 event_bb_analysis() function is called during DynamoRIO's transformation stage, once for each basic block. It totals up the different types of instructions in the basic block and then inserts a clean call into the code to update the totals during application execution. A clean call is C function call where the application state is saved before the call and restored when it completes.


Note

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


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 program counter address, and the instruction encoding.

Procedure

Run Arm IE with the `inscount_emulated` instrumentation client on your example code:

```
$ armie -mvl=16 -i libinscount_emulated.so -a"-instr_types" -- ./example
```

The `-a"-instr_types"` argument is passed to the client and makes it output information about the types of instructions executed.

This command returns:

```
Client inscount is running
i    a[i]    b[i]    c[i]
============================
0    197     283     86
1    262     277     15
2    258     293     35
3    194     286     92
4    228     249     21
5    235     262     27
6    231     290     59
7    237     263     26
8    214     240     26
9    236     272     36
. . .
1020     185     261     76
1021     165     234     69
1022     232     295     63
1023     204     235     31
2422144 instructions executed of which 642 were emulated instructions
SVE instruction count: 642
SVE2 instruction count: 0
SME instruction count: 0
SME2 instruction count: 0
Floating Point instruction count: 13397
```

2.2.4 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, `inscount_emulated`. However, in this example we show how you can use `inscount_emulated` to investigate the effect that vector length has on the number of Scalable Vector Extension (SVE) instructions.

Procedure

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

```
$ armie -mvl=16 -i libinscount_emulated.so -- ./example
Client inscount is running
```

```
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 1282 were emulated instructions
```

Notice the difference in output from the example shown in Get started with Arm IE (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, `inscount_emulated`, outputs as part of its analysis of the example binary as it runs.

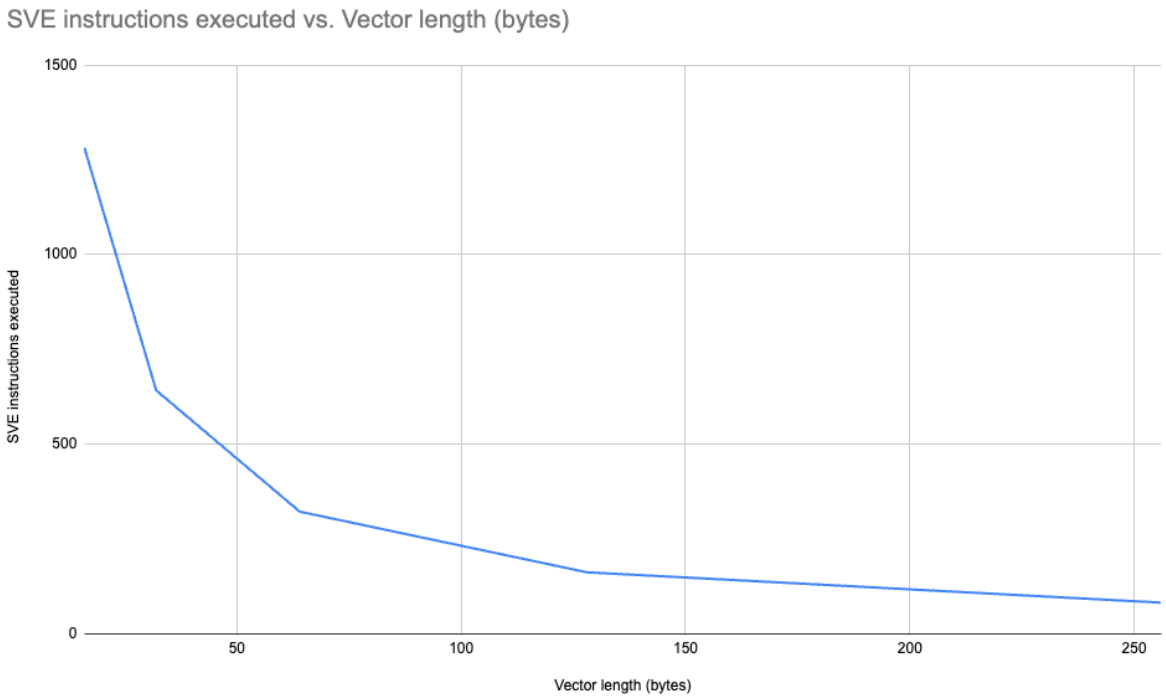
2. Run the example binary with each vector length and tabulate the results:

Table 2-1: Binary vector lengths

Vector Length (bytes)	16	32	64	128	256
SVE Instructions	1282	642	322	162	82

3. Plot the results on a line graph:

Figure 2-1: Plot of SVE Instructions



The graph shows us that as vector length doubles the number of SVE instructions executed halves, leading to improved performance.

2.2.5 Analyze Regions Of Interest

This section describes how to use macros to define Regions of Interest (RoI) in your target application.

Sometimes, when analyzing large and complex 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 results in undefined behavior.

2.2.5.1 Create Regions of Interest

This section describes how to use hint instructions to define Regions of Interest (RoI) in your target application.

About this task

Insert hint instructions into the source code using these macros:

```
#define __START_TRACE() {asm volatile ("hint 0x40");}
#define __STOP_TRACE()  {asm volatile ("hint 0x41");}
```

These start and stop macros instruct Arm IE to start and stop collecting trace data, which allows you to focus your analysis on specific areas of code. This can reduce the amount of noise in the trace data you need to analyse.

The code in this example shows the use of the `inscount_emulated` client, an instrumentation client that counts the number of emulated instructions executed.

The application used in this example, `loops`, contains two loops. This example uses RoI 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; ++i)
        c[i] = i;
```

```
    for(int i=0; i<N; ++i)
        a[i] = b[i] + b[c[i]];
}
```

Procedure

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

   ```
   $ gcc -O3 -march=armv8-a+sve -o loops loops.c
   $ armie -mvl=64 -i libinscount_emulated.so ./loops
   Client inscount is running
   86414 instructions executed of which 32 were emulated instructions
   ```

   All executed instructions are counted.
2. To limit instruction counting to a specific area of code or the 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 ("hint 0x40");}
   #define __STOP_TRACE()  {asm volatile ("hint 0x41");}

   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 `inscount_emulated` client:

   ```
   $ armie -mvl=64 -i libinscount_emulated.so -a -roi ./first_loop
   Client inscount is running
   28 instructions executed of which 13 were emulated instructions
   ```

   The results are different to the `loops` run:
   - Only the first loop is instrumented and as a result fewer executed instructions are counted at runtime.
   - The `armie` command includes the `-a -roi` option to inform the client to enable and disable instruction counting, based on the `__START_TRACE()` and `__STOP_TRACE()` hint

instructions. Without the `-a -roi` option, the client ignores the hint instructions and counts all instructions, producing the same output as for the `loops` run.

The `-a` option enables you to pass command-line arguments to instrumentation clients. For a description of the `-a` option, run `armie --help` or, see the Arm IE command reference section.

5. 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 ("hint 0x40");}
#define __STOP_TRACE()  {asm volatile ("hint 0x41");}

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 -mvl=64 -i libinscount_emulated.so -a -roi ./second_loop
Client inscount is running
30 instructions executed of which 19 were emulated instructions
```

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


Note

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

Next steps

- In addition to the `inscount_emulated` client, the following clients also support `__START_TRACE` and `__STOP_TRACE`:
  - `instrace_emulated.c`
  - `meminstrace_emulated.c`
  - `opcodes_emulated.cpp`

To enable RoIs, all these clients accept the `-a -roi` Arm IE option. If you do not use the `-a -roi` option, RoIs are ignored and all instructions are counted or traced. Their source code is in the Arm IE installation `samples` directory:

```
/path/to/your/arm-instruction-emulator-<xx.y>_<OS>/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 Build custom instrumentation.

- For more advanced analysis examples of a real-world application, see Emulating SVE on existing Armv8-A hardware using DynamoRIO and Arm IE. The blog includes use case examples of the `opcodes_emulated` and `inscount_emulated` clients.

2.2.6 Dynamic instruction counts

Dynamic instruction counts show the number of instructions that are executed by a binary at runtime. These counts are a useful way of assessing the performance-related behavior of an application. An instruction count client, `inscount`, is supplied as an example of how to use the DynamoRIO API with Scalable Vector Extension (SVE) emulation. The client source code is available as a DynamoRIO example in the Arm IE installation `samples` directory. Use the `-i` (or `--iclient`) option to run the client with `armie`, for example:

```
$ armie -mvl=64 -i libinscount.so -- ./example
Client inscount 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: 2421990 instructions executed
```

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

```
$ armie -mvl=64 -i libinscount_emulated.so -- ./example
Client inscount 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
2421990 instructions executed of which 322 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 Arm IE blog.

2.2.7 Examine memory access behavior

The memory access behavior of an executable is another useful aspect of performance. A memory trace emulation client, `memtrace_emulated` is supplied. To trace memory accesses, use the `-i` option of `armie`. For example:

```
$ armie -i libmemtrace_emulated.so -- ./example
```

This command creates a trace file in the current directory logging all memory accesses. For example:

```
$ head memtrace.example.3448421.0000.log
Format: <sequence number>: <thread ID>, <is bundle>, <is write>, <data size>, <data
 address>, <pc>
0: 3448421, 0,  0,  8, 0xffffd83a5f30, 0xffffb4fdf828
1: 3448421, 0,  0,  8, 0xffffb4fefff0, 0xffffb4fdf838
2: 3448421, 0,  0,  8, 0xffffb4feffe8, 0xffffb4fdf840
3: 3448421, 0,  0,  8, 0xffffb4feffc8, 0xffffb4fdf848
4: 3448421, 0,  0,  8, 0xffffb4feff90, 0xffffb4fdf694
5: 3448421, 0,  1, 16, 0xffffd83a5da0, 0xffffb4fdf970
6: 3448421, 0,  1, 16, 0xffffd83a5db0, 0xffffb4fdf978
7: 3448421, 0,  1, 16, 0xffffd83a5dc0, 0xffffb4fdf984
8: 3448421, 0,  1, 16, 0xffffd83a5dd0, 0xffffb4fdf998
```

Memory tracing format

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

```
sequence, tid, bundle, write, 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.

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

`size`
: Number of bytes that are stored or loaded.

addr
: Load or store address.

pc
: Instruction address.

2.3 Build an emulation-aware instrumentation client

This tutorial shows how to modify existing native-only clients to also handle emulated instructions, and how to write your own emulation aware clients.

Before you begin

- To do this tutorial you must have a good working knowledge of 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 page.
- Understand how to run a pre-built instrumentation client. For more information on running instruction clients, see Analyze SVE applications with Arm IE.

About this task

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 IE) are not capable of interpreting emulated instructions.

Tne new emulation aware functions in the DynamoRIO API allow developers to write instrumentation clients which can distinguish between emulation instructions generated by Arm IE and the original application instructions.

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;
```

```
    dr_emulate_options_t flags;
} emulated_instr_t;
```

Procedure

1. Run the pre-built `bbcount` client with Arm IE, which counts the number of basic blocks executed by an application:

```
$ armie -mvl=16 -i libbbcount.so -- ./example
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
```

   Next, 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_emulated.cpp` in: `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_<OS>/samples`.
   Where `bbcount.cpp`, is:

```
/* we only have a global count */
static int global_count;

/* code here omitted to save space */

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);
```

   Edit `bbcount_emulated.cpp` to add a global emulation counter variable and use it in the call to `dr_snprintf()`:

```
/* we have global native and emulated counts */
static int native_count;
static int emulated_count;

/* code here omitted to save space */

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\n"
                      "%10d basic blocks needed flag saving\n"
                      "%10d basic blocks did not\n",
                      native_count, emulated_count,
                      bbs_eflags_saved, bbs_no_eflags_saved);

    DR_ASSERT(len > 0);
```

3. Add the loop which checks for emulated instructions in each basic block. 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

/* code here omitted to save space */

#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);

/* code here omitted to save space */
}
```

`bbcount_emulated.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

/* code here omitted to save space */
```

```
#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);

/* code here omitted to save space */
}
```

There are three things to note about this code change:

- 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.
- 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.

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.

- Instead of using `dr_insert_clean_call()`, as in `opcodes_emulated.cpp`, the client uses `drx_insert_counter_update()` to increment the `native_count` and `emulated_count` variables. The difference is that `dr_insert_clean_call()` inserts a call to a user-defined function, which is run when the block is executed, while `drx_insert_counter_update()` inserts 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_emulated.c and compare them with a diff viewer to look at the modifications in full.

5. To build the modified client, add `bbcount_emulated.c` to `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_<OS>/samples/CMakeLists.txt`:

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

6. Run `cmake`.


Note

The current version of Arm IE (25.0) requires that clients are built with GCC version 11.4 upwards.

```
$ cmake .
-- 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_SUSE-12/bin/cc
-- Check for working C compiler: /opt/arm/gcc-7.1.0_SUSE-12/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_SUSE-12/bin/c++
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_SUSE-12/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>_<OS>/samples
```

7. Run make:

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

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

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

9. Run the modified client:

```
$ armie -mvl=16 -i libbbcount_emulated.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 2-1: 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`

See also the examples described in Analyze SVE applications with Arm IE.

Related information

Build custom instrumentation on page 32
Emulation functions reference on page 44
About instrumentation clients on page 38
Arm IE

2.4 Build custom 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 IE.

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 SVE applications with Arm IE.
- Understand the About instrumentation clients, `opcodes_emulated` and its implementation in the file `opcodes_emulated.cpp`.

Procedure

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

```
$ armie -mvl=64 -i libopcodes_emulated.so -- ./example
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:
<count> : <opcode>
       1 : b
       1 : blr
       1 : cbnz
       1 : ldrb
       1 : movk
       1 : nop
       1 : strb
       1 : ubfm
       3 : cbz
       8 : ret
      11 : ldp
      11 : stp
    1035 : movz
    2048 : msub
    2048 : smaddl
    2050 : sbfm
    2050 : str
    2050 : sub
    2051 : subs
    2055 : orr
    2115 : bcond
    3076 : br
    3079 : bl
    3095 : adrp
    6158 : ldr
    6238 : add
7 unique emulated instructions written to undecoded.example.2853678.log
```

   The file `undecoded.example.2853678.log` contains:

```
64 : 0xe5404040
64 : 0xa54040a0
64 : 0xa5404081
64 : 0x25a10c00
64 : 0x04a10400
1  : 0x25a10fe0
1  : 0x04a0e3e3
```

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 `opcodes_emulated` 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>_<OS>/samples`
3. Edit `opcodes_emulated_tut1.cpp` to merge `opcount()` and `record_emulated_inst()` into one function:
   `opcodes_emulated.cpp`:

```
static void
count_emulated_op(uint opcode)
{
    emulated_ops_map[opcode]++;
}

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_ops_map[opcode]++;
}
```

4. Update the `dr_insert_clean_call()` calls which insert calls to `opcount()`:
   `opcodes_emulated_tut1.cpp`:

```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
   /* Lines not shown */

   bool is_emulation = false;
   for (instr = instrlist_first(bb);
         instr != NULL;
         instr = instr_get_next(instr)) {
      if (drmgr_is_emulation_start(instr)) {
            /* Lines not shown */

            if (enable_count &&
               (*sveinstr != __START_TRACE_INSTR &&
```

```
                *sveinstr != __STOP_TRACE_INSTR)) {
                uint encoding;
                instr_encode_to_copy(drcontext, emulated.instr, (byte *)&encoding,
                                     instr_get_app_pc(emulated.instr));

                /* Updated clean call */
                dr_insert_clean_call(drcontext, bb, instr,
                   (void *)opcount, false, 2,
                   OPND_CREATE_INT32(encoding),
                   OPND_CREATE_INT(1));
             }
             is_emulation = true;
             continue;
       }
       /* Lines not shown */

       if (enable_count) {
             /* Updated clean call */
             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 a call to `opcount()`, must now define two input parameters, rather than one. The `opcount()` function call must now 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_tut1.cpp`:

```
static void
event_exit(void)
{
    /* Lines not shown */

    for(iter2=ranks.rbegin(); iter2!=ranks.rend(); ++iter2) {
        /* Line below updated */
        dr_printf("  %9lu : 0x%08x\n", iter2->first, iter2->second);
    }

    fclose(file);
    emulated_ops_map.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>_<OS>/samples/CMakeLists.txt`:

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

```
add_sample_client(stl_test    "stl_test.cpp"    "")
. . .
```

7. Run `cmake`.


Note

The current version of Arm IE (25.0) requires that clients are built with GCC version 11.4 upwards.

```
$ cmake .
-- 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_SUSE-12/bin/cc
-- Check for working C compiler: /opt/arm/gcc-7.1.0_SUSE-12/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_SUSE-12/bin/c++
-- Check for working CXX compiler: /opt/arm/gcc-7.1.0_SUSE-12/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>_<OS>/samples
```

8. Run `make`:

```
$ make
. . .
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>_<OS>/samples/bin/libopcodes_emulated_tut1.so
[ 9%] Built target opcodes_emulated_tut1
. . .
```

9. Copy the built client from:

   `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_<OS>/samples/bin`

   to

   `/<path/to/your/installation>/arm-instruction-emulator-<xx.y>_<OS>/samples/bin64`

   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 -mvl=64 -i libopcodes_emulated_tut1.so -- ./example
. . .
1022 232 295 63
```

```
1023 204 235 31
<count> : <opcode>
       1 : b
       1 : blr
       1 : cbnz
       1 : ldrb
       1 : movk
       1 : nop
       1 : strb
       1 : ubfm
       3 : cbz
       8 : ret
      11 : ldp
      11 : stp
    1035 : movz
    2048 : msub
    2048 : smaddl
    2050 : sbfm
    2050 : str
    2050 : sub
    2051 : subs
    2055 : orr
    2115 : bcond
    3076 : br
    3079 : bl
    3095 : adrp
    6158 : ldr
    6238 : add
7 unique emulated instructions written to /home/phiram01/tasks/ossrcvm-2538/
example/undecoded.example.2907061.log
      64 : 0xe5404040
      64 : 0xa54040a0
      64 : 0xa5404081
      64 : 0x25a10c00
      64 : 0x04a10400
       1 : 0x25a10fe0
       1 : 0x04a0e3e3
```

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 IE releases.

Until then, as a workaround, a helper script is provided with Arm IE, `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>_<OS>-<OS-version>/llvm-
bin/llvm-mc
$ echo 0xe54842e0 | /<install-dir>/arm-instruction-emulator-<xx.y>_<OS>/bin64/
enc2instr.py
0xe54842e0 : st1w {z0.s}, p0, [x23, x8, lsl #2]
```

Next steps

- Build an emulation-aware instrumentation client

Related information

Arm IE
Analyze SVE applications with Arm IE on page 17

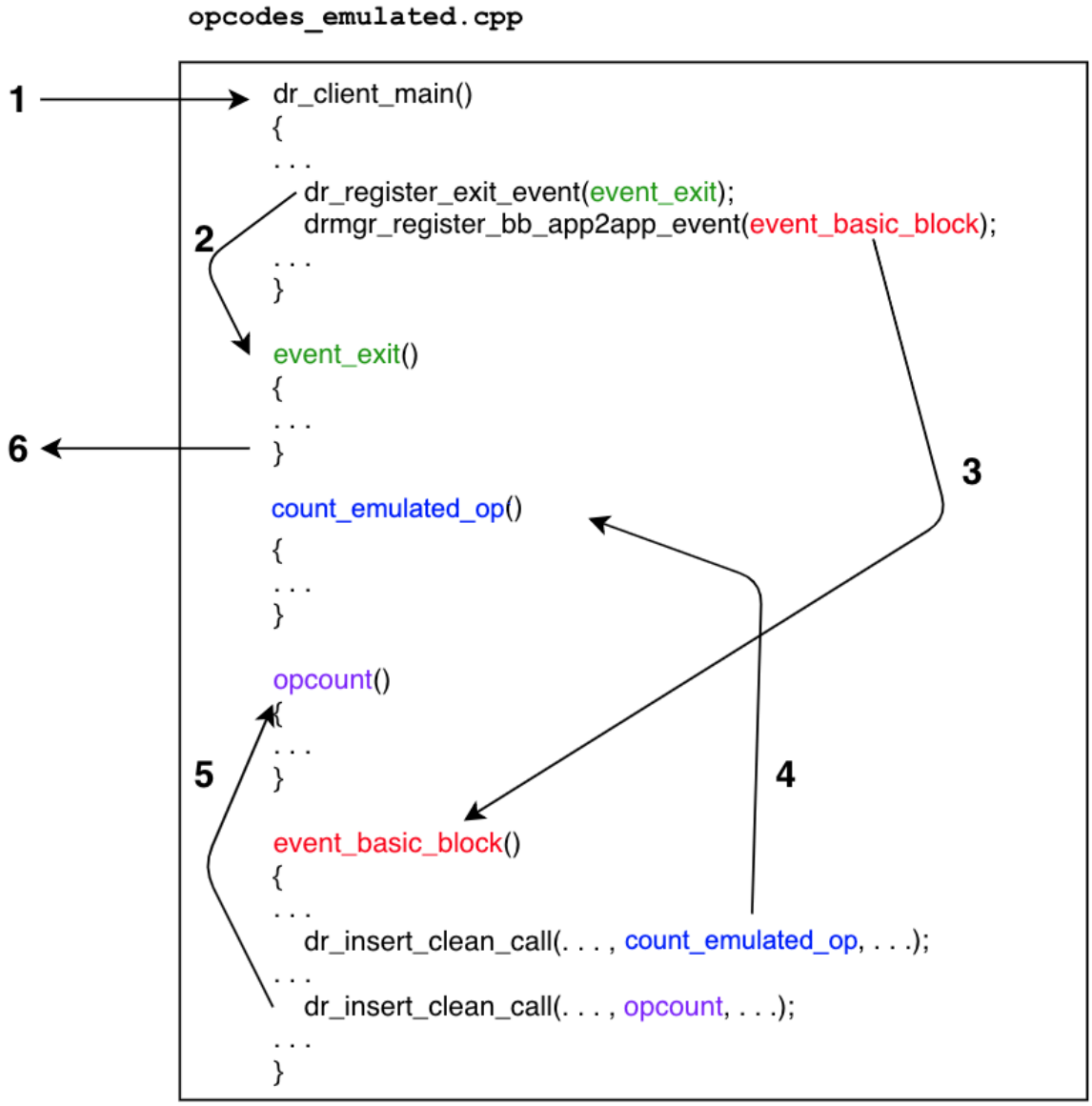
2.5 About instrumentation clients

This topic describes the basic structure of an instrumentation client, including the common event handlers that a client implements.

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

To correctly modify the `opcodes_emulated` 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 2-2: 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()`. A function that is registered for an event is usually referred to as a 'callback function'.
3. In `dr_client_main()`, the client registers a callback function that will be called as each block of code in the application is prepared before being executed.

4. In `event_basic_block()`, the client inserts a call to a function, `count_emulated_op()`, for each emulated instruction which appears in the code of the application. The `count_emulated_op()` increments the counter for the instruction opcode.
5. In `event_basic_block()`, the client inserts a call to a function, `opcount()`, for each native instruction which appears in the code of the application. The `opcount()` increments the counter for the instruction opcode.
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 inserts a call to a function in your client.
- `drmgr_register_bb_app2app_event()`, which registers a callback function which is called at the start of every basic block to be executed.

Code Transformation and Code Execution

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

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 that was inserted during transformation.

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

In this earlier 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 a particular block of code is run at execution time, those functions are called, to increment the instruction 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

Build custom instrumentation on page 32
Analyze SVE applications with Arm IE on page 17
Emulation functions reference on page 44
Arm IE

2.6 View the drrun command

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

About this task

The `-s` option is provided to show how the DynamoRIO `drrun` command is being used to run Arm IE. This allows the command line to be modified as required with DynamoRIO specific options.

Procedure

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

```
$ armie -mvl=16 -s -- ./example
v8.2 hardware detected
Setting VL = 16, SVL = 32
/<path/to/your/installation>/bin64/drrun -max_bb_instrs 32 -max_trace_bbs
 4 -c /<path/to/your/installation>/lib64/release/libarmie.so
 16,32,2199023524874,16519036908076670 -- ./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 `libarmie.so` to run the example binary.

2. If an instrumentation client is specified:

```
$ armie -mvl=16 -s -i libinscount$_emulated.so -- ./example
v8.2 hardware detected
Setting VL = 16, SVL = 32
/<path/to/your/installation>/bin64/drrun -client /<path/to/your/installation>/
lib64/release/libarmie.so 0 16,32,2199023524874,16519036908076670 -client /<path/
to/your/installation>/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 `libarmie.so` and the instrumentation client `libinscount_emulated.so` to count instructions executed by the example application.

3. The `-only_from_app` option for the `inscount_emulated` 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>/bin64/drrun -client /<path/to/your/installation>/
lib64/release/libarmie.so 0 16,32,2199023524874,16519036908076670 -client /<path/
to/your/installation>/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
40855 instructions executed of which 1282 were emulated instructions
```

Notice that the native AArch64 instruction count has dropped to 40855, from 2134094, due to the exclusion of library instructions.

Related information

Build custom instrumentation on page 32
Arm IE
Get started with Arm IE on page 7
Analyze SVE applications with Arm IE on page 17

3. Reference

This section contains reference information for the `armie` command and the emulation functions included with Arm IE.

3.1 Arm IE command reference

The `armie` command runs a compiled binary using Arm IE. Arm IE is an emulator that can execute AArch64 Scalable Vector Extension (SVE) and Scalable Matrix Extension (SME) instructions on any Armv8-A-based hardware.

Usage

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

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

Options

Option	Description
-mvl=<uint>	Vector length in bytes, a power of two in the range 16-256. The default is the current vector length on SVE hardware, otherwise 16.
-msvl=<uint>	Streaming vector length in bytes, a power of two in the range 16-256. The default is the current vector length on SME hardware, otherwise 32.
-mlist-vector-lengths	List the accepted vector lengths.
-i, –iclient <client>	Use an instrumentation client based on the DynamoRIO API.
-a, –arg-iclient <string>	An optional argument to be passed to the instrumentation client.
-s, –show-drrun-cmd	Writes the full DynamoRIO drrun command used to execute armie to stderr. This can be useful when debugging or developing clients.
-h, –help	Prints this help message.
-V, –version	Prints the version.

Advanced Options

Arm IE's default emulation behaviour is to emulate AArch64 features which are not available on the host machine. This will satisfy most users' requirements. However, there may be use cases which require finer control over features which are to be emulated and features which rely on the host machine. For example, a compiler developer may want to check if instructions supporting a specific architectural feature are being emitted by the compiler's code generator. This can be done by disabling emulation of that feature using -m<feature> and executing the compiled test binaries.

Option	Description
-mlist-features	List known features, with values taken from the auxiliary vector.

Option	Description
-m<feature>=<uint>	Enable or disable feature to be emulated, 1 or 0. Use lower-case for <feature>, e.g. -msve=0. By default, all features that can be emulated are enabled except FEAT_AFP, FEAT_EBF16 and FEAT_RPRES. For the full list of features see -mlist-features.
-mwrite-features=<file>	Write default features to be emulated to <file>. The resulting file can be used as input with -mfeatures=<file>. The default features to be emulated are all features that can be emulated, except that FEAT_AFP, FEAT_EBF16 and FEAT_RPRES are instead copied from the host. The file can be "-", meaning stdout.
-mfeatures=<file>	Read those features which are to be emulated from <file>.
-c<feature>=<uint>	Host features are taken from the auxiliary vector but can be overridden with this flag, for testing or if the auxiliary vector is known to be wrong.
-cwrite-features=<file>	Write host features to <file>. The resulting file can be used as input with -mfeatures=<file>. The file can be "-", meaning stdout.

Example 3-1: Example: Use -mlist-vector-lengths to list the valid vector lengths

To list all valid vector lengths, use

```
$ armie -mlist-vector-lengths
16 32 64 128 256 (bytes)
```

Example 3-2: Example: Use -mvl to specify the SVE vector size in bytes

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

```
$ armie -mvl=32 -- ./sve_program
```

Related information

Get started with Arm IE on page 7
Analyze SVE applications with Arm IE on page 17

3.2 Emulation functions reference

This topic describes the emulation functions applicable to Arm IE.

Arm IE is based on the DynamoRIO Dynamic Binary Instrumentation tool platform and allows developers to use the API of DynamoRIO to write instrumentation clients which run alongside Arm IE's SVE / SME emulation client. These instrumentation clients can allow you to analyze SVE / SME 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

Arm IE
Get started with Arm IE on page 7
API Usage Tutorial
SVE guides
SME guides

4. Security and Privacy

This section contains information about aspects of security and privacy users should be aware of in order to make informed decisions when running certain types of applications.

4.1 Dynamic binary modification

Arm IE is built on the DynamoRIO dynamic instrumentation framework, which is an example of a class of tools known as dynamic binary modifiers. A distinguishing feature of dynamic binary modifiers is their ability to access and change the register and memory state of an application at single instruction resolution. This has an impact on host applications in the context of security and privacy.

Guidelines

The following information is intended as a guide to help you decide if it is safe to run Arm IE with applications in which security and/or privacy are a significant concern. These are guidelines and should not be considered an exhaustive list.

- Arm IE does not act as a sandbox. The transparency requirements of emulation instrumentation dictate that the application can do the same things when running with Arm IE as it can without.
- In-memory secrets can be leaked if caution is not exercised when configuring and running Arm IE. Logfiles created and populated by instrumentation clients are an example of how secure data can be exposed if logging is used carelessly. As with all third-party executables, exercise caution if using third-party instrumentation clients.
- Arm IE is not intended for running production code. It is a development tool designed and implemented to execute and profile applications for which hardware is not available, typically used to prepare code to make the most of new hardware.
- The behavior of instructions supported by Arm IE has been tested using Arm's user-space architectural validation suite. This is a rigourous test set but not as rigorous as formal models or hardware. Arm IE must not be relied on for mission-critical security or safety applications.
- When set, the Data Independent Timing (DIT) bit in the processor state (PSTATE) results in the execution time of a specified selection of instructions not depending on the data they handlle. This provides a significant level of protection against attacks which exploit data controlled variation of execution time. Arm IE's emulation of these instructions does not preserve such timing behaviour even if PSTATE.DIT is set.

If you encounter a security or privacy issue which you think Arm IE did not handle correctly, please inform psirt@arm.com.

5. Further resources

Lists the additional resources available which you can to use to learn more about Arm IE, the Scalable Vector Extension and the the Scalable Matrix Extension.

5.1 Arm IE resources

This section lists some useful resources where you can read more about Arm IE.

- Arm IE
- Download Arm IE
- Release history
- Blog: DynamoRIO and Arm IE
- Blog: Optimizing HPCG for Arm SVE

5.2 SVE resources

This section lists some useful resources you can use to learn more about the Scalable Vector Extension.

- Introduction to SVE
- SVE Programming Examples
- 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

  The location for the latest documents about the Application Binary Interface for the Arm Architecture, both for source files and officially released documents.
- Arm Architecture Reference Manual for A-profile architecture

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

5.3 SME resources

This section lists some useful resources you can use to learn more about the Scalable Matrix Extension.

- Arm Scalable Matrix Extension (SME) Introduction

  A three-part detailed introduction to SME.
- SME Programmer's Guide

  An introduction to how SME can be used to efficiently work with matrices and other forms of data.
- Arm A-profile A64 Instruction Set Architecture

  A reference guide to Neon, SVE and SME instructions.
- Arm Architecture Reference Manual for A-profile architecture

  The A-profile reference manual for hardware and software developers.

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

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

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

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
2500-00	25 July 2025	Non-Confidential	Update for Arm IE version 25.0
2200-00	31 March 2022	Non-Confidential	Update for Arm IE version 22.0
2100-00	30 March 2021	Non-Confidential	Update for Arm IE version 21.0
2010-00	21 August 2020	Non-Confidential	First release for Arm IE version 20.1

Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in Document release information on page 51.

Table 2: Issue 22.0

Change	Topics affected
Initial release	-

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. 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 underline`	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**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.