Contents

Vehave User Guide

[[*TOC*]]

Introduction

Vehave is a native functional emulator. It was born out of necessity of being able to functionally test the generated vector code on top of a RISC-V Linux system.

It has since deemed useful to obtain trace-level information for analysis and modelling. However vehave itself does not provide any analysis or modelling mechanism. These tasks have to be carried out using other tools.

How does it work?

Running a program with vector instructions will cause an illegal instruction exception in a CPU that does not support the V-extension. In a Unix system, such as Linux, this is materialised by means of a `SIGILL` signal.

Upon starting, Vehave sets a signal handler for SIGILL signals. This handler decodes the instruction using the LLVM MC layer and emulates it. The vector architectural state is represented in the memory of the program handled by vehave. Once the instruction has been emulated, the execution proceeds to the original process.

Multithread support

Vehave supports multithreading. This impacts how tracing works. See the tracing section for details.

Parameters

Currently vehave is fixed to these parameters:

- ELEN is 64-bit (i.e. the maximum element width for a vector is 64-bit)
- VLEN is by default 4096 bits. This means that with a SEW (single element width) of 64-bit, we can represent 64 elements.

It is possible to change the VLEN default of 4096 bits to some other value using the environment variable VEHAVE_VECTOR_LENGTH.

Running a program under vehave

Once you have built vehave, you'll have a libvehave.so library. You either need to link against it or LD_PRELOAD. LD_PRELOAD may be more convenient as it does not need to change the build scripts of the application being executed.

```
$ LD_PRELOAD=<build-dir>/libvehave.so ./my-vector-program args-if-any...
```

Debug information

By default vehave prints debug information about the execution of the emulated instructions. Debug information is controled via the environment variable VEHAVE_DEBUG_LEVEL.

VEHAVE_DEBUG_LEVEL	Meaning
0	Disables all debug output

VEHAVE_DEBUG_LEVEL	Meaning
1	Default debug. This is the default if VEHAVE_DEBUG_LEVEL is not set.
2	Register contents debug. This is the default debug plus extra information about the contents of the registers read and written by the instructions

Other levels might be defined in the future.

Tracing

Tracing is enabled by default. Check the description of the trace output for more details. Tracing can be disabled setting the environment variable VEHAVE_TRACE to 0.

Vehave has two tracing modes depending on whether the tracing is single thread or multithread. The default is multithread.

Single thread tracing is selected by setting the environment variable VEHAVE_TRACE_SINGLE_THREAD to a non-empty string (e.g. 1).

- Under single-thread tracing, the trace default output filename is vehave.<pid>.trace, where <pid> is the PID of the running process. If you want to override this filename use the environment variable VEHAVE_TRACE_FILE.
- Under multithread tracing, the trace default output filename is vehave.<pid>_<tid>.trace, where <pid> is the PID of the running process and <tid> is the Linux thread-id (as obtained by the gettid() system call). If the environment variable VEHAVE_TRACE_FILE is set to a non-empty value then the trace is output in a filename <VEHAVE_TRACE_FILE>_<tid>.

Merging multithread traces

Use the script <prefix>/share/merge-traces/merge-traces.py to merge the different traces into a single .csv trace that can then be converted into a Paraver trace.

See the Installation section to know about the installation <prefix>.

Conversion to Paraver traces

Traces generated by vehave can be converted to Paraver using the <prefix>/share/vehave2prv/vehave2prv script.

See the Installation section to know about the installation <prefix>.

Common issues you may encounter

Unimplemented instructions

It may happen that a vector instruction is still not implemented by vehave, in that case the instruction will be handled as a regular SIGILL. The debug output should make clear the faulting instruction.

If the instruction is known to LLVM (i.e. it can disassemble it correctly) but not implemented, feel free to raise an issue in vehave issue tracker.

The instruction must be known to LLVM MC first before we can handle it with vehave. So new instructions (e.g. as extensions of the V-extension itself) will have to be added first in LLVM, do not raise an issue in this case. Better ask by email.

Segmentation fault during the emulation

It may happen that a memory vector instruction causes an incorrect memory access. In such case the simulator will crash. The debug output should help identify the emulated instruction causing the fault.

Segmentation fault outside the emulation

As a bonus, when running with a VEHAVE_DEBUG_LEVEL greater than 0, vehave will decode the faulting instruction and finish the application. This might help you to pinpoint the offending instruction.

Controlling vehave from the application

It is possible to control the vehave emulation using the header `vehave-control.h`. This file provides the following macros which translate into the described instructions. `vehave-control.h` is installed in `<prefix>/include/vehave`, so make sure you add that path to `-I` if you plan to use it in your program.

Macro	Behaviour	Instruction
`vehave_disable_debug()`	Disable debug. `vehave` will act as if `VEHAVE_DEBUG_LEVEL` were set to zero after this instruction	`csrr x0, 3106`
`vehave_enable_debug()`	Enables debug. Restores the debug level before the previous `vehave_disable_debug()`	`cssr x1, 3106`
`vehave_disable_tracing()`	Disables tracing after this instruction.	`cssr x2, 3106`
`vehave_enable_tracing()`	Enables tracing after this instruction. Does not have effect if tracing wasn't enabled in the first place due to `VEHAVE_TRACE=0`	`cssr x3, 3106`
`vehave_start_mark()`	Use this as a start marker. This is a `nop`.	`cssr x4, 3106`
`vehave_stop_mark()`	Use this as a stop marker. This is a `nop`.	`cssr x5, 3106`
`vehave_trace(type, value)`	Use this to emit a `.trace type, value` instruction in the CSV trace. This is useful when converting the CSV file to Paraver trace	Several instructions that set the type and the value and then commit it.

This is useful so you can filter the records of the trace with the phases of the program of your interest or disable/enable debugging in specific parts of the program.

Building vehave

> **Note:** If you're working in `hifive01`, we already deploy this binary there so you don't need to download it.

Other requirements to build vehave are:

- a Linux RISC-V host with a RISC-V native `g++` installed
- `cmake`
- `ninja` or `make`

> **Note:** if you are in `hifive01` the requirements above are already fulfilled.

On a Linux RISC-V host checkout the code

```
$ git clone git@repo.hca.bsc.es:EPI/System-Software/vehave.git vehave-src
```

Create a build directory and configure the build

```
$ mkdir vehave-build
$ cd vehave-build
$ cmake -G Ninja ../vehave-src/ -DLLVM_CONFIG=<llvm-native-install-dir>/bin/llvm-config
```

> **Note:** In `hifive01`, `<llvm-native-install-dir>` is `/apps/llvm-toolchain/EPI/development`

Now build

```
$ ninja
```

This will create a shared object called `libvehave.so`.

Installation

If you want to install `vehave` to some `<prefix>` location you will have to pass `-DCMAKE_INSTALL_PREFIX=<prefix>`. Then run `ninja install`.

Testing

You can run the tests using `ninja check`. Testing uses lit, so you need to have it installed first.

> You don't need to install `lit` in `hifive01` as it is already there.

The easiest way to install lit is using `pip` inside a sandbox directory (in this example we call it `sandbox`).

```
$ python3 -m venv sandbox
$ ./sandbox/bin/pip install lit
```

And in the `cmake` invocation pass `-DLIT=<full-path-to>/sandbox/bin/lit`.