Contents

Run RISC-V Vectorial application step by step **1**
Prepare the environment . . . 1
Compile QEMU (tested on Ubuntu 16.04 and 20.04) . . . 2
Download fedora . . . 2
Create a shared folder . . . 2
Download RISC-V BSC-tools . . . 2
Boot Fedora and mount the shared folder . . . 3
Compile and run an application using the vector extension . . . 4
Download and build a simple example . . . 4
Prepare and build your kernels/applications . . . 4
Run a binary emulating vector instructions . . . 5
Simulate traces . . . 5
Step 0: Download and install MUSA . . . 5
Step 1: Configure the simulation . . . 6
Step 1: Run the simulation . . . 6
Analyze traces . . . 6
Download Paraver (on x86 system) . . . 6
Open a trace . . . 6

Test case **7**
Paraver analysis . . . 7

Run RISC-V Vectorial application step by step

Prepare the environment

The following diagram represents the workflow of this environment. We will guide you with the setup of the virtual machine and the usage of the tools used to execute riscv codes.

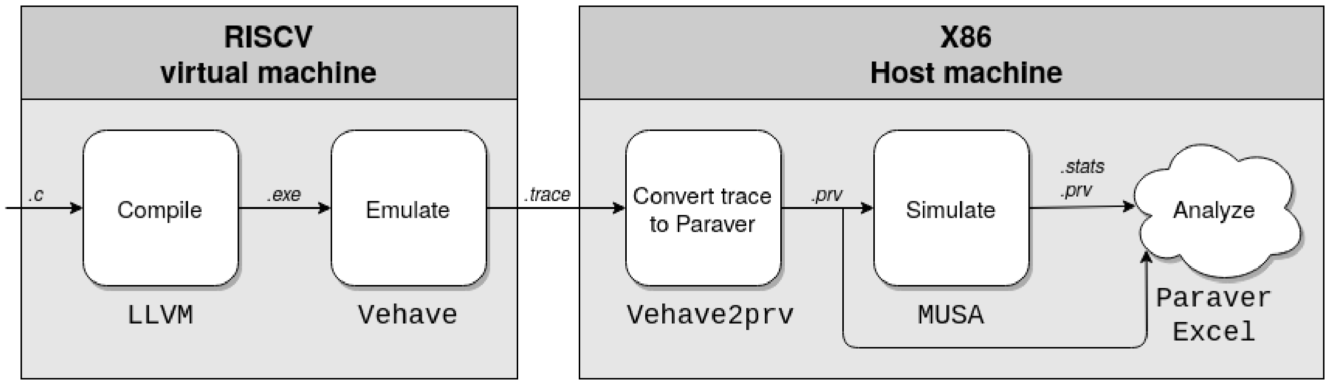

Figure 1: Demo_epi_future

Throughout this guide you will find many code snippets. The prompt of each command tells in which host the command is executed.

- If the prompt is `x86$` the command is executed in your x86 laptop or server;
- If the prompt is `riscv$` the command is executed inside qemu virtual machine.

Compile QEMU (tested on Ubuntu 16.04 and 20.04)

First, get the following packages:

```
x86$ sudo apt-get install build-essential pkgconf libglib2.0-dev libpixman-1-dev
    libcap-ng-dev libattr1-dev
```

After that, install QEMU (tested version 5.0.1). Change `${QEMU_INSTALLDIR}` for your desired installation path.

```
x86$ wget https://download.qemu.org/qemu-5.0.1.tar.xz
x86$ tar xf qemu-5.0.1.tar.xz
x86$ cd qemu-5.0.1
x86$ ./configure  --enable-virtfs --target-list=riscv64-softmmu --prefix=${
    QEMU_INSTALLDIR}
x86$ make -j8
x86$ make install
x86$ cd ..
```

Download fedora

```
x86$ wget https://ssh.hca.bsc.es/epi/ftp/vm/Fedora-Developer-Rawhide-20200108.n.0-
    fw_payload-uboot-qemu-virt-smode.elf
x86$ wget https://ssh.hca.bsc.es/epi/ftp/vm/Fedora-Developer-Rawhide-20200108.n.0-
    sda.raw.xz
x86$ unxz Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz
```

Create a shared folder

We create a folder that can be accessed from inside the virtual machine. If you create a file inside the virtual machine, you can open or modify from your native filesystem and vice versa.

```
x86$ mkdir shared_folder
x86$ cd shared_folder
```

Download RISC-V BSC-tools

```
# Vehave
x86$ wget https://ssh.hca.bsc.es/epi/ftp/vehave-EPI-0.7-development-latest.tar.bz2
x86$ tar xf vehave-EPI-0.7-development-latest.tar.bz2
x86$ rm vehave-EPI-0.7-development-latest.tar.bz2

# llvm-EPI toolchain
```

```
x86$ wget https://ssh.hca.bsc.es/epi/ftp/llvm-EPI-0.7-development-toolchain-native-
    latest.tar.bz2
x86$ tar xf llvm-EPI-0.7-development-toolchain-native-latest.tar.bz2
x86$ rm llvm-EPI-0.7-development-toolchain-native-latest.tar.bz2

x86$ cd ..
```

Boot Fedora and mount the shared folder

To boot fedora, execute the following command, changing `${QEMU_INSTALLDIR}` with the path where you installed QEMU on the first step.

```
x86$ ${QEMU_INSTALLDIR}/bin/qemu-system-riscv64 \
    -daemonize \
    -machine virt \
    -smp 4 \
    -m 4G \
    -virtfs local,path=shared_folder,mount_tag=host0,security_model=passthrough,id=
        host0 \
    -kernel Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.
        elf \
    -object rng-random,filename=/dev/urandom,id=rng0 \
    -device virtio-rng-device,rng=rng0 \
    -device virtio-blk-device,drive=hd0 \
    -drive file=Fedora-Developer-Rawhide-20200108.n.0-sda.raw,format=raw,id=hd0 \
    -device virtio-net-device,netdev=usernet \
    -netdev user,id=usernet,hostfwd=tcp::10000-:22

qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release.
    Please use the -bios option to   avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
[...]
```

If you get the error message `could not read keymap file: 'en-us'`, replace the flag `-daemonize` for the flag `-nographic`.

You also may get some warnings that you can disregard.

Once the boot has finished, open another terminal and ssh into the machine

```
x86$ ssh -p 10000 -o UserKnownHostsFile=/dev/null  -o StrictHostKeyChecking=no
    riscv@localhost
```

- **User** `riscv`
- **Password** `fedora_rocks!`

After you have logged in for the first time, upgrade the system and mount the shared folder.

Note: the upgrade can take hours, depending on your system

```
# Upgrade package of Linux distribution
riscv$ sudo dnf upgrade
# Create mounting point for shared_folder
riscv$ sudo mkdir /shared_folder
# Mount shared_folder
riscv$ sudo  mount -t 9p -o trans=virtio host0 /shared_folder/ -oversion=9p2000.L
```

Compile and run an application using the vector extension

In this section, we present an example of how to compile and run a vectorized kernel step by step. If you want to skip this part and try a complete test-case that is run with a script that we provide, jump to the last section of this document.

Download and build a simple example

First, download a basic matrix multiplication kernel that we provide.

```
riscv$ wget https://ssh.hca.bsc.es/epi/ftp/example/riscv_example.tar.gz
riscv$ tar -xzf riscv_example.tar.gz
```

Then, compile it using the Makefile:

```
riscv$ cd matmul
riscv$ make
```

Prepare and build your kernels/applications

In order to adapt your code, to use vector instructions we can proceed in two different ways:

- Auto-Vectorization. For example:

```
static void matmul(int n, double (*c)[n], double (*a)[n], double (*b)[n]) {
  for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
  #pragma clang loop vectorize(enable)
  for (int k = 0; k < n; k++) {
    c[i][j] += a[i][k] * b[k][j];
  }
}
  }
}
```

- Intrinsics. For example: "'c void axpy_intrinsics(double a, double *dx, double* dy, int n) { int i;

long gvl = __builtin_epi_vsetvl(n, __epi_e64, __epi_m1); __epi_1xf64 v_a = __builtin_epi_vbroadcast_1xf64(a, gvl);

for (i = 0; i < n;) { gvl = __builtin_epi_vsetvl(n - i, __epi_e64, __epi_m1); __epi_1xf64 v_dx = __builtin_epi_vload_1xf64(&dx[i], gvl); __epi_1xf64 v_dy = __builtin_epi_vload_1xf64(&dy[i], gvl); __epi_1xf64 v_res = __builtin_epi_vfmacc_1xf64(v_dy,

v_a, v_dx, gvl); __builtin_epi_vstore_1xf64(&dy[i], v_res, gvl); i += gvl; } } “‘
If you like to write/compile your vector codes using the EPI intrinsics, the reference is here.

Run a binary emulating vector instructions

Now you are ready to execute the kernel, following this procedure: (If you want to run your binary change `./matmul` to `./your_binary`)

```
riscv$ export VEHAVE_TRACE_SINGLE_THREAD=1
# Set simulator vector length to 16384 bits(same Testchip). Feel free to change
    other values.
riscv$ export VEHAVE_VECTOR_LENGTH=16384
# Run using vehave emulator; VEHAVE_TRACE_FILE define the name of output trace;
    VEHAVE_DEBUG_LEVEL define verbosity possible values {0,1,2}
riscv$ VEHAVE_DEBUG_LEVEL=0 VEHAVE_TRACE_FILE=matmul.trace /shared_folder/vehave-
    EPI-0.7-development-2020-12-01-2200/bin/vehave ./matmul
riscv$ mv matmul.trace /shared_folder/.
```

This will generate a file called `matmul.trace`, containing information about the execution of the vector instructions. Move this trace to the shared folder to study it outside the virtual machine.

Outside the virtual machine, convert it to a Paraver-compliant file with the following command:

```
x86$ shared_folder/vehave-EPI-0.7-development-2020-12-01-2200/share/vehave2prv/
    vehave2prv matmul.trace
```

Note: vehave2prv also works inside the virtual machine, but it is recommended to run it outside since it will be faster.

For more information about Vehave look in this wiki

If you have any trouble with the Vehave tools, contact sdv-support@bsc.es.

Simulate traces

MUSA is a tool that simulates the vehave trace in a parameterizable machine, where the user can specify a set of parameters such as the memory bandwidth, the size and shape of the cache and the number of instructions being executed at the same time, among many other parameters.

Step 0: Download and install MUSA

```
x86$ cd shared_folder
x86$ wget https://ssh.hca.bsc.es/epi/ftp/musa_redistributable.tar.gz
x86$ tar -xvf musa_redistributable.tar.gz
x86$ cd musa_redistributable
x86$ ./install.sh
x86$ cd ..
```

Step 1: Configure the simulation

The file `musa_redistributable/tasksim/etc/conf/riscv_reference.conf` contains the parameters of the simulation. You can leave them unchanged, but feel free to experiment with them and study their effect on the simulation.

For riscv, only the first level of cache and the main memory are used in the simulation. Other cache levels are ignored.

For more information about the tool and its parameters, contact sdv-support@bsc.es.

Step 1: Run the simulation

MUSA requires the input trace consisting of three files, `.prv .row .pcf` that we previously generated with `vehave2prv`. After running, it generates an output trace.
To run the simulation, run this command adjusting the paths to your environment.

```
# Usage: musa_redistributable/tasksim/bin/tasksim.sh <tasksim_config_file> <
    output_trace> <trace>
x86$ musa_redistributable/tasksim/bin/tasksim.sh musa_redistributable/tasksim/etc/
    conf/riscv_reference.conf matmul_sim matmul
```

Where:

- The input traces must be named `matmul.prv matmul.pcf matmul.row` (last argument)
- The output traces will be named `matmul_sim.prv matmul_sim.pcf matmul_sim.row` (second to last argument)

- A file named `matmul_sim.stats` will be generated with information about the simulation

Analyze traces

There are two main ways to analyze the traces generated by the simulator. The simpler way consists of looking at the information saved in the `.stats` file.

The other way uses BSC's tool Paraver. It can analyze either initial or simulated traces, outside the QEMU virtual machine.

Download Paraver (on x86 system)

```
x86$ wget https://ftp.tools.bsc.es/wxparaver/wxparaver-4.9.0-Linux_x86_64.tar.bz2
tar xf wxparaver-4.9.0-Linux_x86_64.tar.bz2
```

Open a trace

```
x86$ ./wxparaver-4.9.0-Linux_x86_64/bin/wxparaver matmul_sim.prv
```

Test case

We have prepared an easy to use example to show the potential of these tools. This example runs a vectorized matrix multiplication kernel of size 64x64 two times, with two different maximum vector lengths (16 and 32), and then simulates the effect of various cache sizes.

It could run with bigger matrixs and vectors, but running in a virtual environment is slow and the intent of this test is to be an initial and fast approach to the environment.

First, run You can execute it with the following commands inside the QEMU virtual machine. It can take around 10 minutes to finish all the simulations.

```
riscv$ cd /shared_folder
riscv$ wget https://ssh.hca.bsc.es/epi/ftp/example/riscv_example.tar.gz
riscv$ tar -xzf riscv_example.tar.gz
riscv$ cd matmul
riscv$ ./execute.sh
```

After the script ends, a new folder called `study` will be generated. Inside, one can find another folder with the vehave traces.

To study the traces, exit the virtual machine and execute the file `study.sh` in the same folder in your x86 host. This will run various simulations with different cache sizes.

```
x86$ cd shared_folder/matmul
x86$ ./study.sh
```

The simulates traces and their `.stats` are located inside the `study` folder. There's also a file named `cycles.csv` containing the number of cycles for each *(vector length,cache size)* configuration.

We also provide a gnuplot script to visualize this csv, doing the following command:

```
x86$ gnuplot plot.gnp
```

This will generate an image called `study.png` that should resemble this one:

The scripts provided can be relatively easily adapted to fit your necessities with other programs. We encourage you to change them for your tests and email us with additional questions.

Paraver analysis

We provide some Paraver configuration files that can be useful for analyzing RISCV-V traces. They can be downloaded with the following command:

```
x86$ wget https://ssh.hca.bsc.es/epi/ftp/cfg/example_cfgs.tar.gz
```

We included these configuration files:

- `instruction_type.cfg` : Shows the number of instructions of each type.
- `PC.cfg` : Shows the Program Counter of each instruction.

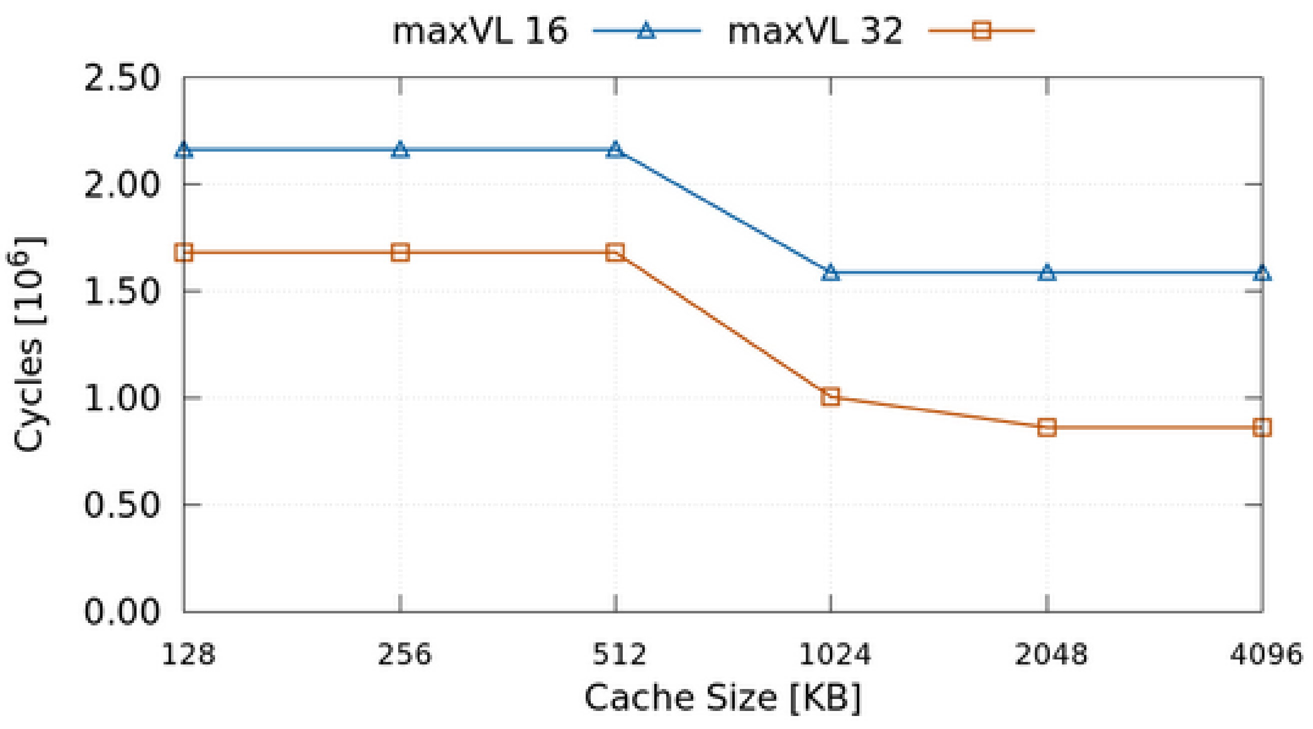

Figure 2: plot

- `VL.cfg` : Shows the vector length used by each instruction.
- `bytes.cfg` : Shows the total number of bytes (assuming data types of 64 bits) loaded and stored.
- `Hit_and_miss.cfg` : Shows the hit and miss ratio of the L1 cache for each type of memory instruction, alongside the total accesses to L1 and Memory lines.
- `timing_model.cfg` : Shows the average and total latency per instruction and the number of cycles of the execution.

It's important no note that the last two cfgs only work with simulated traces.