Setup and Evaluation Documentation for R-Visor

This document details the system setup required to run R-Visor. R-Visor is a Dynamic Binary Instrumentation (DBI) tool developed for open Instruction Set Architectures (ISAs) such as RISC-V. The tool uses Just-In-Time execution to execute target application binaries in a sandboxed environment, while running user-defined instrumentation routines. It allows for instrumentation of applications compiled in the Executable And Linkable Format (ELF), while maintaining instruction transparency and equivalent control flow to the original binary. R-Visor contains an instrumentation API which enables users to write custom instrumentation tools for a wide variety of purposes such as optimization, program analysis, binary patching, instruction emulation, etc. To tackle the extensibility requirement for DBI, R-Visor leverages ArchVisor, a Domain-Specific Language (DSL) that allows users to concisely write specifications for new ISA components that are subsequently integrated into R-Visor for instrumentation.

This codebase is made publicly available at https://doi.org/10.5281/zenodo.15300670 and is made reusable through an MIT license.

Getting Started

Requirements

System Specifications

- Host OS: Ubuntu 20.04 (recommended) or any recent Linux distribution
- RAM: 8GB recommended
- Disk space: At least 1GB

Dependencies

- QEMU v7.0.0
- qemu-riscv64 version 6.2.0
- RISC-V GCC toolchain
- RISC-V newlib toolchain
- CMake v3.22.1

Installing Dependencies

RISC-V Toolchain

```
sudo apt update
sudo apt install gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu gawk te
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
git checkout rvv-0.8
git submodule update
sudo mkdir -p /opt/riscv_imafd
sudo chown -R $(echo "$USER") /opt/riscv_imafd
./configure --prefix=/opt/riscv_ima --with-arch=rv64imafd --with-abi=lp64
sudo make linux -j $(nproc --ignore=1)
export PATH=$PATH:/opt/riscv_ima/bin
sed -i "$ a export PATH=\$PATH:/opt/riscv_ima/bin" ~/.bashrc
cd ..
```

QEMU

```
sudo apt update
sudo apt install libglib2.0-dev libpixman-1-dev ninja-build git make gcc
git clone https://github.com/qemu/qemu
cd qemu
git checkout v7.0.0
./configure --target-list=riscv64-softmmu
make -j $(nproc --ignore=1)
sudo make -j $(nproc --ignore=1) install
cd ..
```

Next install qemu user

```
sudo apt install qemu-user
```

Installing R-Visor

With Docker

A Docker image with all dependencies pre-installed is available as `rvisor_artifact_image.tar.gz`. This can be set up using the following commands:

```
docker load < rvisor_artifact_image.tar.gz
docker run --rm -it rvisor_artifact /bin/bash
```

Alternatively, since the dockerfile is included in the project, thus the image can be recreated using the following commands:

```
docker build -t rvisor_artifact .
docker save rvisor_artifact | gzip > rvisor_artifact_image.tar.gz
```

Using Compressed Archive

While alternatives such as Ninja exist for building projects that contain a CMakeLists.txt, we show the build instructions using CMake. First unzip the archive then run the following commands:

```
cd rvisor
cmake .
make
```

The command builds the R-Visor binaries which are stored in `bin/` .

FPGA Evaluation

The R-Visor performance evaluations are done by executing the R-Visor binaries on an FPGA which hosts a Trireme RISC-V core. We load the operating system together with the DynamoRio and R-Visor binaries as a gzip file which we have included in the artifact (`initramfs.cpio.gz`). After flashing the FPGA, the DynamoRIO binary can be accessed at `build_riscv64/bin64/` . We use the command `/build_riscv64/bin64/drrun -- <binary>` to execute benchmarks without instrumentation and we use `/build_riscv64/bin64/drrun -c /build_riscv64/api/bin/libbbcount.so -- <binary>` to use basic block counting routine.

R-Visor binaries are located at `usr/bin/railbins/` . The binaries used for evaluation are located at `usr/bin/benchmarks/embench` .

QEMU Evaluation

In the absence of FPGA or RISC-V hardware for performance evaluation, we make provision for evaluation on QEMU through busybox. Although the performance figures do not exactly replicate the FPGA results, the trend is similar with R-Visor showing lower performance and memory overhead on average. Due to the size restriction, we only include the QEMU evaluation files on the Zenodo archive.

This evaluation setup is contained in `busybox_eval.zip` (Available in the Zenodo archive). In order to use this setup first unzip the folder and use the following script to start the emulator:

```
cd operating_systems
cd initramfs
./recompile
cd ..
./qemu_run
```

This follows a similar folder structure to the **FPGA Evaluation** with the binaries and benchmarks in the same locations indicated.

Step-By-Step Instructions

This section details the steps required to set up and test the various instrumentation routines demonstrated in the R-Visor LCTES submission. Reproducing the performance results (from the No Instrumentation and Basic Block Counting routines) in the paper would require the evaluator to set up the Trireme core or any equivalent RISC-V core. The Dynamic Instruction Compression and Basic Block frequency routines will yield the same results regardless of the execution platform.

Artifact Folder Structure

The holder structure for the r-visor and archvisor codebase is shown in the table below. Additionally we include the benchmarks and results from the performance evaluations and use cases.

File / folder	Description
`bin/`	R-Visor instrumentation binaries
`comparison/`	ArchVisor lines-of-code comparison results
`embench/`	Benchmarks from the Embench suite used for evaluation
`eval/`	Evaluation results (raw data and plots)
`helpers/`	Helper libraries shared by instrumentation passes
`routines/`	Individual R-Visor instrumentation routines
`src/`	R-Visor and ArchVisor source code
`22.pdf`	Camera-ready version of the submitted paper
`busybox_eval.zip`	Busybox setup for performance evaluation on QEMU (Available on Archive)

File / folder	Description
`CMakeLists.txt`	CMake build script for R-Visor
`Dockerfile`	Docker file for building R-Visor
`initramfs.cpio.gz`	Compressed archive for FPGA evaluation
`LICENSE.txt`	MIT License for R-Visor tool
`rvisor_artifact_image.tar.gz`	Docker image for R-Visor artifact (Available on Archive)

Benchmarks Used for Evaluation

Our evaluation of R-Visor makes use of the Embench benchmark suite. Due to the highlighted limitations of R-Visor all benchmarks used must be compiled with the RISC-V newlib toolchain (`riscv64-unknown-elf`) using static libraries (`-s` flag). We have included the precompiled embench suite in `embench/` .

Performance Evaluations

In our publication we evaluate the performance of r-visor using two routines. The no instrumentation routine and basic block counting routine. We perform the evaluation by executing these routines on the `embench` suite using a command of the format:

```
time -v ./<rvisor-binary> <benchmark>
```

We execute this 5 times on the FPGA for each benchmark and average the results. The reported profiles obtained from executing `time -v` gives us both the **execution time** and the **max resident set size**(RSS). We base the performance overhead on the execution time values (Section 8.2.1) and we use the max RSS to calculate the memory overhead (Section 8.2.2). We include the execution logs in `eval\performance` .

No Instrumentation Routine

The *no instrumentation* routine is used to evaluate the performance of R-Visor while executing the target binary just-in-time. The source file for this routine is located in `routines/tlrvisor.c` . This routine has the trace linking optimization enabled.

To use this routine, ensure that the line

```
add_executable(tlrvisor ${ROUTINESDIR}/tlrvisor.c ${HEADER_FILES})
```

is uncommented in the `CMakeLists.txt` before running `make`.

After running `make`, the routine can be tested by executing it with the binary name as the first argument. For example:

```
./bin/tlrvisor embench/aha-mont64
```

The logs for this routine are stored in `tlrvisor_logs.txt`. This file should be empty after running the routine since the routine carries out no action.

Basic Block Counting

The *basic block counting* routine is used to evaluate the performance of R-Visor while counting the number of basic blocks executed by the target binary. The source file for this routine is located in `routines/tlrvisor_bb.c`. This routine has the trace linking optimization enabled.

To use this routine, ensure that the line

```
add_executable(tlrvisor ${ROUTINESDIR}/tlrvisor_bb.c ${HEADER_FILES})
```

is uncommented in the `CMakeLists.txt` before running `make`.

After running `make`, the routine can be tested by executing it with the binary name as the first argument. For example:

```
./bin/tlrvisor_bb embench/aha-mont64
```

The logs for this routine are stored in `tlrvisor_bb_logs.txt`. This file contains the basic block count for the executed binary.

Dynamic Instruction Compression

The `Dynamic Instruction Compression` routine demonstrates a use case where a binary's full-width instructions are converted to their compressed counterparts during basic block allocation. This instrumentation routine uses the `Group` parameter provided by ArchVisor. To use this routine we first have to assign an ArchVisor group to all the instructions that have a compressed equivalent. For our experiment we use a new ArchVisor source located in `src/decodeFileDBT.rdec`. In this file, all instructions to be translated are assigned the `DBT` group. This group is referenced within the dynamic instruction compression routine (`routines/dbt.c`), where we implement logic to compress the instructions if the correct condition is met.

The source file for this routine is located in `routines/dbt.c`.

To use this routine, first rebuild \sys using with the new ArchVisor specifications by executing the command:

```
make parse-dbt
```

Subsequently, ensure that the line

```
add_executable(dbt ${ROUTINESDIR}/dbt.c ${HEADER_FILES})
```

is uncommented in the `CMakeLists.txt` before running `make`.

After running `make`, the routine can be tested by executing it with the binary name as the first argument. For example:

```
./bin/dbt embench/aha-mont64
```

The logs for this routine are stored in `rvisor_dbt_logs.txt`. This file contains the calculated reduction in size after using compressed instructions.

To obtain the results shown in the paper, we executed the `dbt` routine on the benchmarks and plotted the results obtained. The logs are located in `eval/dbt/out` and the results were plotted using `eval\dbt\run.ipynb`.

Basic Block Frequency

The *Basic Block Frequency* routine demonstrates a use case where the execution frequency of basic blocks within a target binary is recorded to determine hot paths.

\noindent The source file for this routine is located in `routines/bb_frequency.c`.

\noindent To use this routine, ensure that the line

```
add_executable(bb_frequency ${ROUTINESDIR}/bb_frequency.c ${HEADER_FILES}
```

is uncommented in the \lstinline|CMakeLists.txt| before running `make`.

After running `make`, the routine can be tested by executing it with the binary name as the first argument. For example:

```
./bin/bb_frequency embench/aha-mont64
```

The logs for this routine are stored in `bb_frequency_logs.txt` . This file contains the calculated reduction in size after using compressed instructions.

In the paper, we demonstrated this routine by executing it on `embench/sha` and we plot the results. The logs are located in `eval/bb_frequency/bb_frequency_logs` and these logs were plotted using `eval/bb_frequency/bb_frequency_plot.py`

Evaluating ArchVisor

Our paper demonstrates the utility of ArchVisor through the DBT use case. Additionally we also evaluate ArchVisors ability to enhance extensibility by comparing the lines of code (LOC) required to support the floating point and compressed extensions from RISC-V.

The code used for this comparison is stored in the `comparison` folder, with `comparison\compressed` containing the code required for the implementation of the compressed instruction set and `comparison\float` containing the LOC comparison for the single precision floating-point ISA extension for RISC-V. For reference, these numbers are used in Table 1 of the paper.

Artifact Claims

Claims Supported by Artifact

Paper Claim	How the Artifact Demonstrates It
Lower overhead relative to DynamoRIO	Performance evaluations
Extensible instrumentation	LOC analysis (`src/comparison`)
Utility through dynamic instruction compression	Use case with DBT binary
Utility through basic block frequency measurement	Use case with Basic Block Frequency routine
Code-cache optimizations	Implicit through performance experiments

Claims not Supported

Unsupported claim	Reason
Exact resource utilisation on QEMU	Performance evaluations were done on an FPGA with a RISC-V core

Unsupported claim	Reason
Compatibility with all RISC-V toolchains	**R-Visor** currently supports only the `newlib` toolchain