The Case for Using Guix to Solve the gem5 Packaging Problem

Christopher Batten[1], Pjotr Prins[2], Efraim Flashner[2], Arun Isaac[2], Ekaiz Zarraga[3], Erik Garrison[2], Tuan Ta[1]

[1] School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2] The University of Tennessee Health Science Center, Memphis, TN [3] ElenQ Technology

This talk will first describe the gem5 packaging problem before making the case for using Guix, a mature functional cross-platform package manager, for building, distributing, installing, and managing the gem5 ecosystem.

1. The gem5 Packaging Problem

The gem5 simulator has become the defacto standard for cycle-level simulation, and gem5 now supports evaluating a diverse set of workloads [1]. Unfortunately, both the gem5 simulator and gem5 workloads still lack a compelling software packaging solution to simplify building, distributing, installing, and managing the gem5 ecosystem. In this section, we describe the gem5 simulator and workload packaging problems before sketching an ideal software packaging solution.

The gem5 Simulator Packaging Problem – The gem5 simulator is a complex piece of software with numerous build- and run-time dependencies including a modern C++ compiler, SCons, Boost, and Python. To mitigate dependency issues, the gem5 installation instructions strongly recommend using specific versions of Ubuntu. The gem5 simulator has numerous compile-time options to experiment with different ISAs, coherence protocols, and/or accelerators, and this in turn makes providing a single precompiled binary difficult. Even so, the gem5 community does point new researchers to a small number of precompiled Docker images. Given these challenges, the gem5 community has chosen not to support any kind of packaging for the gem5 simulator. Almost all researchers individually manage dependencies and recompile the gem5 simulator from source.

The gem5 Workload Packaging Problem – Building workloads to run on the gem5 simulator using syscall emulation can be just as challenging as building the gem5 simulator itself. These workloads must be cross-compiled meaning researchers must build a complete cross-compilation toolchain for each target architecture. Researchers might also need to build an emulator (e.g., QEMU) to test these workloads before moving to cycle-level simulation. Researchers must ensure the workloads only use static libraries and do not call any unsupported syscalls. Given these challenges, the gem5 community directly included precompiled binaries as part of the gem5 source distribution for many years. More recently, the community has migrated to precompiled binaries as part of the gem5 resources project [2].

An ideal software packaging solution would be: *reproducible* – easily duplicate precisely specified development environments; *transparent* – understand entire development environment including exact build configuration and version of every dependency; *composable* – easily integrate the gem5 simulator and workload into standard development environments without needing cumbersome, heavyweight containers; *fast* – leverage precompiled packages; *distribution agnostic* – enable researchers to use the Linux distribution of their choice; *unified* – same packaging solution can be used for both the gem5 simulator and workloads; *portable* – easily build gem5 workloads for native execution and/or target multiple ISAs for cycle-level simulation without manually managing multiple cross-compilation toolchains; and *flexible* – easily switch between development environments, modify existing packages, add new packages, produce reproducible workflows, and/or generate containers.

2. Using Guix for gem5

Guix is a mature functional cross-platform package manager with hundreds of committers and over 20K packages. In this section, we briefly describe our on-going efforts to use Guix for packaging both the gem5 simulator and gem5 workloads.

Using Guix to Package gem5 Simulators – We have developed a proof-of-concept Guix package for the gem5 simulator that handles all build- and run-time dependencies and installation[1]. The package builds gem5 for six ISAs, ensures builds are reproducible by eliminating non-deterministic use of `__DATE__` and `__TIME__`, patches the build environment to work with SCons, and performs a well-structured install of the gem5 simulator binaries and example configurations. The package is: *reproducible* through the use of isolated and deterministic environments for experimenting with the gem5 simulator; *transparent*, since the complete recursive dependency graph is precisely specified; *composable*, since the gem5 simulator is installed just like any other tool without the need for a container; and *distribution agnostic*, since the package can be installed on Ubuntu, RHEL, SUSE, or even Guix System which is an entire distribution based exclusively on Guix. Derived packages could enable easily providing packages for different compile-time configurations. When merged upstream into the main Guix package repository, this package will be part of the Guix build farm enabling binary package substitution for *fast* installation.

Using Guix to Package gem5 Workloads – Guix already includes packages for QEMU and cross-compilation toolchains for commercial ISAs. We are contributing to the RISC-V port of Guix including packaging the RISC-V cross-compilation toolchain. We identified a simple Smith-Waterman sequence alignment Guix package as an interesting gem5 workload and developed a derived Guix package that patches the standard build process to produce a statically linked binary[2]. In addition to being *reproducible*, *transparent*, *composable*, and *distribution agnostic*, using Guix is also *portable*, since we can easily cross-compile the package for ARM or RISC-V, and *flexible*, since it requires only eight lines of Guile code to create a new derived package supporting static compilation.

The attached appendix describes step-by-step commands for a case study that: installs QEMU, gem5, and cross-compilers for x86, ARM, and RISC-V in an isolated environment; cross-compiles and runs Smith-Waterman for all three ISAs; and runs this benchmark on the in-order and out-of-order timing models.

Acknowledgments

This work was supported by NSF PPoSS Award #2118709 and NLNet awards for GNUMes-RISCV and Guix-Riscv64.

References

[1] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[2] B. R. Bruce et al. Enabling Reproducible and Agile Full-System Simulation. *ISPASS*, Mar 2021.

[1] https://git.genenetwork.org/guix-bioinformatics/guix-bioinformatics/src/branch/master/gn/packages/virtualization.scm

[2] https://git.genenetwork.org/guix-bioinformatics/guix-bioinformatics/src/branch/master/gn/packages/static.scm

A. Case Study

To reproduce this case study, a researcher first must download and install Guix[3].

A.1. Add new channel

By default, Guix includes its own main package repository, but users can also create their own "channels" that include third-party packages. We need to add such a channel to get access to the gem5 simulator package and the derived Smith-Waterman package.

```
 % cd $HOME/.config/guix
 % cat > channels.scm \
<<'END'
(use-modules (guix ci))
(list
 (channel
  (name 'gn-bioinformatics)
  (url (string-append "https://git.genenetwork.org/"
        "guix-bioinformatics/guix-bioinformatics.git"))
  (branch "master"))
 (channel-with-substitutes-available
  %default-guix-channel "https://ci.guix.gnu.org"))
END
```

A.2. Update Guix and install Smith-Waterman

We use `guix pull` to download all of the package descriptions from the main package repository along with any third-party packages. We then install the default Smith-Waterman package and run it natively. Here we use the default "profile", but we could also install this package in a dedicated Guix "profile", similar to Python's virtual environment.

```
% mkdir -p $HOME/tmp/misc/test-guix
% cd $HOME/tmp/misc/test-guix
% guix pull
% guix install smithwaterman
% smithwaterman -p TGATTGTACCAAA TGATCATGTACCA
```

A.3. Install QEMU and gem5

We now install both the QEMU and gem5 packages for all architectures in the same profile.

```
% guix install qemu
% guix install gem5
```

A.4. Build and run Smith-Waterman for x86_64 ISA

We use `guix build -target=x86_64-linux-gnu` to cross-compile most Guix packages for x86_64. Here we cross-compile the derived package for Smith-Waterman which produces a statically linked executable that we then run on both QEMU and gem5.

```
% cd $HOME/tmp/misc/test-guix
% DIR=$(guix build \
         --target=x86_64-linux-gnu smithwaterman-static)
% ln -sf $DIR/bin/smithwaterman sw-x86_64
% qemu-x86_64 ./sw-x86_64 -p TGATTGTACCAAA TGATCATGTACCA
% gem5-x86.opt \
    $GUIX_PROFILE/share/gem5/configs/example/se.py \
    --cmd=./sw-x86_64 \
    --options="-p TGATTGTACCAAA TGATCATGTACCA"
```

[3] https://guix.gnu.org/en/download

A.5. Build and run Smith-Waterman for ARM ISA

We use `guix build -target=aarch64-linux-gnu` to cross-compile most Guix packages for ARM. Here we cross-compile the derived package for Smith-Waterman which produces a statically linked executable that we then run on both QEMU and gem5.

```
% cd $HOME/tmp/misc/test-guix
% DIR=$(guix build \
         --target=aarch64-linux-gnu smithwaterman-static)
% ln -sf $DIR/bin/smithwaterman sw-aarch64
% qemu-aarch64 ./sw-aarch64 -p TGATTGTACCAAA TGATCATGTACCA
% gem5-arm.opt \
    $GUIX_PROFILE/share/gem5/configs/example/se.py \
    --cmd=./sw-aarch64 \
    --options="-p TGATTGTACCAAA TGATCATGTACCA"
```

A.6. Build and run Smith-Waterman for RISC-V ISA

We use `guix build -target=riscv64-linux-gnu` to cross-compile most Guix packages for RISC-V. Here we cross-compile the derived package for Smith-Waterman which produces a statically linked executable that we then run on both QEMU and gem5.

```
% cd $HOME/tmp/misc/test-guix
% DIR=$(guix build \
         --target=riscv64-linux-gnu smithwaterman-static)
% ln -sf $DIR/bin/smithwaterman sw-riscv64
% qemu-riscv64 ./sw-riscv64 -p TGATTGTACCAAA TGATCATGTACCA
% gem5-riscv.opt \
    $GUIX_PROFILE/share/gem5/configs/example/se.py \
    --cmd=./sw-riscv64 \
    --options="-p TGATTGTACCAAA TGATCATGTACCA"
```

A.7. Run experiment on RISC-V ISA

Once we have used Guix to install the gem5 simulator and the gem5 workload packages, we can easily perform a computer architecture research experiment. Here we compare the performance of running Smith-Waterman on an in-order vs. out-of-order RISC-V processor model.

```
% cd $HOME/tmp/misc/test-guix

% gem5-riscv.opt \
    --outdir=m5out-minor-sw \
    $GUIX_PROFILE/share/gem5/configs/example/se.py \
    --cmd=./sw-riscv64 \
    --options="-p TGATTGTACCAAA TGATCATGTACCA" \
    --cpu-type=MinorCPU --ruby

% gem5-riscv.opt \
    --outdir=m5out-o3-sw \
    $GUIX_PROFILE/share/gem5/configs/example/se.py \
    --cmd=./sw-riscv64 \
    --options="-p TGATTGTACCAAA TGATCATGTACCA" \
    --cpu-type=O3CPU --ruby

% grep system.cpu.numCycles m5out-minor-sw/stats.txt
% grep system.cpu.numCycles m5out-o3-sw/stats.txt
```