The Case for Using Guix to Enable Reproducible RISC-V Software & Hardware

Christopher Batten[1], Pjotr Prins[2], Efraim Flashner[2], Arun Isaac[2]
Jan Nieuwenhuizen[3], Ekaitz Zarraga[4], Tuan Ta[1], Austin Rovinski[1], Erik Garrison[2]

[1] School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2] The University of Tennessee Health Science Center, Memphis, TN
[3] Joy of Source, The Netherlands, [4] ElenQ Technology, Spain

ABSTRACT

Reproducible research is a serious challenge in the field of computer architecture. RISC-V can potentially help improve reproducibility through the use of a standard, open instruction-set architecture along with open-source RISC-V software stacks and open-source RISC-V hardware models. But the diversity of the RISC-V ecosystem can raise new issues complicating reproducible computer architecture research. In this paper, we present our on-going work to use the Guix package management system to help enable reproducible RISC-V software and hardware. We describe our recent work on porting RISC-V software stacks to Guix including bootstrapping through GNU Mes and integrating modern C++ cross compilers with RISC-V support. We also describe our recent work packaging RISC-V hardware models in Guix including the Spike functional-level model, gem5 cycle-level model, and Ariane RTL model. Finally, we present a case study that illustrates a simple, reproducible experiment to cross-compile a Smith-Waterman sequence alignment workload for RISC-V and then run this workload on Spike, gem5, and Ariane.

1 INTRODUCTION

The field of computer architecture has a research reproducibility problem [3]. A specific computer architecture research project can require *complex software stacks* including compilers, runtimes, and workloads running on *complex hardware models* each targeting different system components (i.e., processors, networks, memory systems), abstraction levels (i.e., functional-, cycle-, or register-transfer-level), and/or metrics (i.e., execution time, area, energy). Different projects often use different hardware models and are often based on different instruction set architectures (ISAs) with bespoke ISA extensions. Software stacks and/or hardware models might be widely used over decades or specially designed for a single research project. Software stacks and/or hardware models can use different abstractions and/or programming languages, complicating their integration. Software stacks, hardware models, and/or ISAs might be proprietary or closed-source. It is no wonder that artifact evaluation still struggles to gain traction in computer architecture conferences [4].

The growing adoption of RISC-V within the computer architecture research community can help improve reproducibility through the use of a standard, open ISA along with open-source RISC-V software stacks and open-source RISC-V hardware models. However, one of the key strengths of RISC-V is its ability to support a diverse set of standard ISA variants in addition to bespoke instruction set extensions, and this strength can also lead to a combinatorial explosion of customized software stacks combined with hardware model options. Targeting the RV32I, RV32E, or RV64I standard base ISAs along with the M, A, F, D, Q, C standard extensions results in over 150 possible distinct ISAs not to mention the emerging L, V, B, T, and P extensions along with every research group's own bespoke extensions. We believe that the software and hardware package management system is critical to managing this diversity and thus enabling reproducible RISC-V software and hardware.

The RISC-V Software Packaging Problem – Building RISC-V software stacks can be challenging since native compilation on RISC-V platforms is not widely available. RISC-V stacks usually must be cross-compiled meaning researchers must build a complete cross-compilation tool-chain for each target architecture. Researchers might also need to build an emulator (e.g., QEMU) to test these workloads before moving to a hardware model. Researchers might need to ensure the workloads only use static libraries and do not call any unsupported syscalls for bare metal execution. All of these issues are complicated by the diversity in the RISC-V ISA. Completely different cross compilers, compiler libraries, compiler optimizations, and/or compiler back-ends can be required for each ISA variant. This means building RISC-V software has complex build-time dependencies on the compilation tool-flow. Given these challenges, it is not surprising that the RISC-V community has settled on a single RISC-V ISA variant (i.e., RV64GC) as the "standard" for RISC-V software packaging across all major Linux distributions, and then these Linux distributions are often used when building containers for reproducible RISC-V software.

The RISC-V Hardware Packaging Problem – Building RISC-V hardware models for simulation and eventual FPGA or ASIC implementation can be just as challenging as packaging RISC-V software stacks. For example, the gem5 simulator [2] is a complex piece of software with numerous build- and run-time dependencies including a modern C++ compiler, SCons, Boost, and Python. The gem5 simulator has numerous compile-time options to experiment with different ISAs, coherence protocols, and/or accelerators. As another example, hardware RTL models for RISC-V processors often use complex RTL generators, and the actual process of building hardware RTL for simulation is far less standardized than building software. The specific subset of RTL supported by open-source simulators is constantly changing creating fragile build-dependencies on open-source simulation frameworks such as Verilator and Icarus Verilog, and then of course these simulation frameworks have their own build-time dependencies on native compiler tool-chains. All of these issues are again complicated by the diversity in the RISC-V

ISA. A different hardware RTL configuration or even an alternative implementation is required for each RISC-V ISA variant. Given these challenges, it is again, not surprising that the RISC-V community has settled on a single RISC-V ISA variant (i.e., RV64GC) as the "standard" for Linux-cable RISC-V cores [1, 8, 10], and usually provide detailed step-by-step manual installation instructions that try to capture all of the build- and run-time dependencies.

Current solutions to the RISC-V software and hardware packaging problem essentially undermine one of the key strengths of RISC-V which is its ability to facilitate transformative computer architecture research through the use of diverse ISA extensions. An ideal RISC-V software and hardware packaging solution could potentially enable reproducible RISC-V research while at the same time maintaining the diversity of the RISC-V software and hardware ecosystem. Such an ideal packaging solution would be:

- **Transparent** – enable researchers to understand the entire development environment for RISC-V software stacks and hardware models including the exact source code and build configuration for every dependency.
- **Lightweight** – enable researchers to easily integrate RISC-V software stacks and hardware models into standard development environments and workflows without requiring cumbersome, heavyweight binary containers.
- **Flexible** – enable researchers to easily switch between different development environments to experiment with new version of RISC-V software stacks, hardware models, and/or dependencies.
- **Isolated** – enable researchers to easily isolate the entire development environment to prevent accidentally "leaking" a user's environment into an experiment and to enable creating binary containers when necessary.
- **Portable** – enable researchers to easily build software stacks for both native execution on traditional ISAs and RISC-V variants and also easily enable cross compilation for RISC-V.
- **Fast** – enable researchers to easily use precompiled binary packages when one can prove the resulting package is bit-exact to what would be produced when recompiling from source.
- **Distribution Agnostic** – enable researchers to use the Linux distribution of their choice.
- **Extensible** – enable researchers to extend the package management system by modifying existing packages, adding new packages, or implementing new functionality using a general-purpose programming language.

This paper makes the case for using Guix to enable reproducible RISC-V software and hardware. Guix is a mature functional cross-platform package manager that is transparent, lightweight, flexible, isolated, portable, fast, distribution agnostic, and extensible. Section 2 provides a brief description of the Guix package manager. Section 3 describes our on-going efforts to port RISC-V software stacks to Guix including bootstrapping through GNU Mes and integrating modern C++ cross compilers with RISC-V support, and Section 4 describes our on-going efforts to package RISC-V hardware models in Guix including the Spike functional-level model, gem5 cycle-level model, and Ariane RTL model. Section 5 describes a case study that illustrates how our efforts enable: (1) easily installing QEMU, Spike, gem5, and Ariane; (2) easily cross compiling a bioinformatics workload for RISC-V; and (3) running this workload on all hardware models. Section 6 discusses related software and hardware package management systems. Our goal is to introduce the RISC-V computer architecture community to the Guix package manager and hopefully spark interest in contributing to our efforts to enable reproducible RISC-V software and hardware.

2 GUIX BACKGROUND

GNU Guix is a "functional" package management tool. Even though it is used to distribute the GNU software system it can also be used on its own on top of any common Linux distribution, including Debian and CentOS. In fact Debian and derived distributions have `apt-get install guix`. Guix makes it easy for unprivileged users to install, upgrade, or remove software packages, to roll back to a previous package set, to build packages from source, and generally assists with the creation and maintenance of software environments. Some features are transactional installs, reproducible software deployment, and lightweight containers. Each on its own deserves proper inspection, but here we are focused on easy RISC-V cross compilation and RISC-V emulation.

There is a plethora of package management and software deployment frameworks. Distributions have their own package managers, including `dpkg` and `rpm`. Even so, many modern computer languages include their own package manger including JVM `jar`, Javascript `npm`, Python `pip`, Ruby `gems`, and Rust `cargo`. The use of these package managers, however, breaks down when having to deal with multiple languages and complex ecosystems. Other high-level packagers including `conda`, `brew`, and `spack` try to address such concerns, but fall short in reproducibility and depend on the underlying distribution to "bootstrap" the build process. Also, similar to the main distros, they do not escape dependency "hell" and have no uniform way of managing mixed software versions (i.e., requiring an older version of a library). GNU Guix and its sibling Nix, sidestep many of the issues and provide next-generation reproducible package management for everyone.

GNU Guix is especially suitable for bootstrapping, cross compilation and emulating different architectures in a reproducible way. What does "reproducible" really mean? It means that when one researcher builds a software stack or hardware model with all its dependencies on one system, another person can regenerate the same binary dependency "graph" faithfully on another system. It also means researchers can share binaries ("substitutes" in Guix terminology) over the network and trust they are bit-exact compared to the binaries built from source. Because Guix installs software under a hash value, all packages are isolated. So if Apache is built with SSL 1.1.1 and another with SSL 1.0.3, they can both co-exist on one system and even run at the same time. Likewise, on a Guix system we can cross-compile packages to different architectures, both dynamically and statically linked, and thanks to Linux `binfmt` support we run them transparently on QEMU in emulation on one single system. GNU Guix is not a small software distribution. At last count there are over 20K packages including 1800 Python packages and over 1900 R packages. In March 2022, over 70 people committed changes to the Guix package tree with 800 git commits.

3 GUIX FOR PACKAGING RISC-V SOFTWARE STACKS

In this section, we describe our on-going efforts to port RISC-V software stacks to Guix including bootstrapping through GNU Mes and integrating modern C++ cross compilers with RISC-V support. Bootstrapping software for novel architectures is a challenge. Typical software distributions, such as Debian, use a single binary 'blob' that is larger than 250Mb. It takes a lot of know how and it is non-trivial to bring such bootstraps to new platforms.

The GNU Mes project aims to bootstrap from, so called, Stage0 to a C compiler capable of compiling GCC, with only the explicit requirement of a single 1 KB binary or less. Today GNU Guix for AMD64 and ARM64 bootstraps from this tiny binary Stage0 starting point! GNU Mes provides the TinyCC C compiler written in a simple Scheme interpreter consisting of about 5,000 lines of C. With these, GNU Mes now supports a full open source self-hosted bootstrap that compiles GCC 4.6.4. Through GNU Mes, the GNU Guix operating system is successfully bootstrapped from source on a regular basis on the GNU Guix build farms.

The next phase is to complete bootstrapping of RISC-V targets with GNU Mes. The GNU Compiler Collection (GCC) added support for RISC-V with version 7.5, making migration of RISC-V software possible. To close the bootstrapping gap the NLNet-funded GNU Mes-RISCV project is back-porting the RISC-V support to the GCC 4.6.4 C compiler, a version that is able to compile C++ and, thus, GCC 7.5 and on-wards. This effort in combination with a port of the GNU Mes to RISC-V, will provide a full bootstrap system for RISC-V machines. So far, the core of RISC-V has been back-ported from GCC 7.5 into GCC 4.6.4. The latter therefore cross-compiles to RISC-V, and some simple programs compile to assembler. More work is required to support associated libraries, specifically `libgcc` for some complex operations and, perhaps, `libatomic`. Once these are migrated, testing will be easier and we will aim to build GCC 7.5 using the back-ported compiler.

Next we will continue with porting TinyCC to RISC-V so that it can compile GCC 4.6.4 directly for RISC-V. Once the GNU Mes RISC-V bootstrap is functional it means that bootstrapping any RISC-V target (including exotic research RISC-V variants!) should become fairly trivial by adapting Stage0.

Another NLNet-funded project is porting GNU Guix packages to RISC-V. The following software is working as Guix installable software for RISC-V: the GNU stack, including gcc, gfortran and gccgo; mesa graphics; clang, ruby and R. Java support will be patched in with icedtea-2 (java 7). There is no Rust for RISC-V yet, but that's a particular focus of the NLNet grant and it targeted for the coming year.

By getting a GNU Mes bootstrap and GNU Guix packages ported to RISC-V we are creating a full software deployment ecosystem that can be used by anyone on any RISC-V platform. In the near future the build farms for the GNU Guix project will have have RISC-V RV64GC machines. That will allow distribution of binary substitutes. There is also the option of creating substitutes using QEMU emulation. Currently, GNU Guix provides packages for POWERPC 64le using QEMU emulation. Note that, because of its reproducible packaging, anyone can distribute binaries, and we are sharing compiled binaries between our own machines.

4 GUIX FOR PACKAGING RISC-V HARDWARE MODELS

In this section, we describe our on-going efforts to port RISC-V hardware models to Guix including the Spike functional-level model, gem5 cycle-level model [2], and Ariane RTL model [10].

We have developed a Guix package for the Spike RISC-V functional-level hardware model which is now upstreamed to the main Guix package repository[1]. The package describes how to reproducibly download Spike 1.1.0 and build it from source using the standard GNU build system. The package describes a key dependency on the device tree compiler package (`dtc`) which was already packaged in Guix. The package also describes how to patch the build process to ensure that when building Spike it will refer to the version of `dtc` installed in the Guix store as opposed to any other version installed on the host system. Finally, the package captures a build-time dependency on Python which is required for some of the unit tests. Since this package is in the main Guix package repository, precompiled binaries are available through the Guix build farm enabling fast installation. The package is guaranteed to always produce the same bit-exact install on any machine.

We have developed a proof-of-concept Guix package for the gem5 RISC-V cycle-level hardware model that handles all build- and run-time dependencies and installation[2]. There is a base package that builds gem5 for six ISAs and a derived package that only builds gem5 for RISC-V to reduce build time and space usage. The package 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. Additional derived packages could enable easily providing packages for different compile-time configurations. The package is guaranteed to always produce the same bit-exact install on any machine.

We have developed a proof-of-concept Guix package for the Ariane RISC-V hardware model[3]. The original build instructions provided in the Ariane GitHub repository do not specify any kind of build-time dependencies on the native compiler tool-chain. The instructions provide both Bash shell scripts to install Verilator and the device tree compiler from source as well as instructions on how to install a specific version of Verilator and the device tree compiler package on Ubuntu. The instructions download a large precompiled binary containing a specific version of the RISC-V cross-compiler from an online SiFive archive. The precompiled binary cross-compiler is installed in a non-standard location necessitating the use of an ad-hoc environment variable to enable later build steps to locate these tools. It took two person-days to be able to successfully build the Ariane RTL simulator, cross-compile a RISC-V workload, and run this workload on the Ariane RTL simulator. Complications included managing specific versions of Verilator on non-Ubuntu distributions, undefined dependencies on native tools, an ambiguous dependency graph, and non-standard installations. The package we have developed avoids all of these issues by: precisely specifying the build-time dependencies on Spike and

[1] https://git.savannah.gnu.org/cgit/guix.git/tree/gnu/packages/virtualization.scm
[2] https://git.genenetwork.org/guix-bioinformatics/guix-bioinformatics/src/branch/master/gn/packages/virtualization.scm
[3] https://git.genenetwork.org/guix-bioinformatics/guix-bioinformatics/src/branch/master/gn/packages/riscv.scm

Verilator 4.110 regardless of the host Linux distribution; patching the build process to eliminate the use of an ad-hoc environment variable to specify the location of `libfesvr.a`; patching the source to fix a bug which prevents displaying standard output during a simulation; and wrapping the installed binary to provide a clean interface. The package is guaranteed to always produce the same bit-exact install on any machine.

5 CASE STUDY

This case study illustrates how to use Guix to reproduce a simple experiment involving both RISC-V software and hardware. The case study first installs a Smith-Waterman sequence alignment workload for native execution. The case study then installs QEMU for RISC-V emulation, Spike for RISC-V functional-level hardware modeling, gem5 for RISC-V cycle-level hardware modeling, and Ariane for RISC-V RTL hardware modeling (using the Verilator simulator). Finally, the case study uses a RISC-V cross-compiler to compile the Smith-Waterman workload for RISC-V and then runs this workload on QEMU, Spike, gem5, and Ariane. This case study uses RISC-V RV64GC, but Guix provides the mechanisms to enable similar experiments on other RISC-V architectures. To reproduce this case study, a researcher first must download and install Guix[4].

5.1 Add a new channel

In addition to Guix's main package repository, users can create their own "channels" that include third-party packages. We need to add such a channel to get access to the gem5 package, Ariane 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
```

5.2 Update Guix and install Smith-Waterman

We use `guix pull` to download package descriptions from the main and third-party package repositories. We then install 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
```

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

5.3 Install QEMU, Spike, gem5, and Ariane

We now install QEMU, Spike, gem5, and Ariane. Guix takes care of ensuring all dependencies are installed. For example, Guix will not just ensure the C++ compiler for Spike is installed, but it will also ensure the C compiler used to bootstrap building that C++ compiler is installed. Guix will install SCons, Boost, Python, Verilator, and dtc. Guix will automatically download bit-exact substitutes when available (e.g., QEMU, Spike) and compile packages from source otherwise (e.g., gem5, Ariane).

```
 % guix install qemu spike gem5-riscv cva6
```

5.4 Build and run Smith-Waterman on QEMU

We can use `guix build –target=riscv64-linux-gnu` to cross-compile most Guix packages for RISC-V. While QEMU supports both dynamic and static linking, Spike, gem5, and Ariane only support static linking. So we cross-compile the derived Smith-Waterman package to produce a statically linked executable. Guix automatically takes care of installing and configuring an appropriate compiler to enable reproducible cross compilation.

```
 % cd $HOME/tmp/misc/test-guix
 % DIR=$(guix build \
        --target=riscv64-linux-gnu smithwaterman-static)
 % ln -sf $DIR/bin/smithwaterman sw-riscv64
```

5.5 Build RISC-V proxy kernel

The RISC-V proxy kernel is needed to handle system calls for both Spike and Ariane. So we cross-compile the proxy kernel which has also been packaged and is available in the main Guix package repository.

```
 % cd $HOME/tmp/misc/test-guix
 % DIR=$(guix build \
        --target=riscv64-linux-gnu riscv-pk)
 % ln -sf $DIR/bin/pk pk
```

5.6 Run Smith-Waterman on QEMU, Spike, gem5, and Ariane

We can now run the Smith-Waterman workload on QEMU, Spike, gem5, and Ariane. Obviously RTL simulation is quite slow, so the Ariane simulation can take over an hour.

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

 % qemu-riscv64 ./sw-riscv64 -p TGATTGTACCAAA TGATCATGTACCA

 % spike ./pk ./sw-riscv64 -p TGATTGTACCAAA TGATCATGTACCA

 % gem5.opt \
     $GUIX_PROFILE/share/gem5/configs/example/se.py \
     --cmd=./sw-riscv64 \
     --options="-p TGATTGTACCAAA TGATCATGTACCA"

 % ariane +max-cycles=100000000 +time_out=100000000 \
     ./pk ./sw-riscv64 -p TGATTGTACCAAA TGATCATGTACCA
```

6 RELATED WORK

A detailed survey of software package managers is beyond the scope of this paper. However creatively applying software package managers in the context of hardware is more novel. So in this section, we discuss other hardware package managers and frameworks, including Bender [9], FuseSoC [7], LiteX [6], and HDLMake [5]. Table 1 summarizes the comparison among hardware package managers and Guix. The key detail that separates Guix from other package managers is that *guix enables bit-exact reproducible builds*. Most managers surveyed offer some form of source file management and dependency resolution (using appropriate versions of dependencies). The surveyed hardware package managers also incorporate some form of EDA tool interfacing (such as to Verilator) while Guix does not. The problem with the approach taken by other managers is that they have no mechanism to ensure that the tool versions are appropriate for use because tools are not considered dependencies (nor any way to ensure cross-compiler tool-chains will reliably produce software to run on programmable processors). In addition, the managers do not have isolation to ensure that environment does not leak into the build and alter results. Because of these shortcomings, we find that Guix is the only manager among those surveyed that can provide cross-platform, reproducible builds. In addition, Guix offers build caching due to its ability to hash inputs and recognize if a build has been completed previously.

Bender: Bender is a dependency management tool for hardware design projects [9]. The stated goals for Bender are: (1) compatible with multiple electronic design automation (EDA) tool vendors and flows, (2) allow for reproducible builds, (3) manage source files for a project and its intellectual property (IP), (4) manage source dependencies, and (5) generate tool scripts for supported tools. Bender uses a lock file (`Bender.yml`) to list git hashes of dependencies. This ensures that Bender will use the correct versions of IP, however it does not ensure that the tool versions nor the environment are consistent. Thus, the output can vary greatly or not build at all. Bender manages build order of IPs to ensure proper compilation by tools and uses dependency resolution based on semantic versioning to ensure that the dependencies are compatible. Bender can also generate scripts for supported EDA tools.

FuseSoC: FuseSoC is a package manager and a set of build tools for HDL (Hardware Description Language) code [7]. Its main purpose is to increase reuse of IP cores and be an aid for creating, building and simulating SoC solutions. FuseSoC claims to enable: (1) reuse of existing cores, (2) creating compile-time or run-time configurations, (3) running regression tests against multiple simulators, (4) porting designs to new targets, (5) encouraging other projects to use your code, and (6) setting up continuous integration. The main selling point of FuseSoC is that it enables IP reuse across different tool vendors and easy management of source code. FuseSoC uses Edalize as an interface library to several different EDA tools and vendors. However, FuseSoC does not have management over tool versions or environment.

LiteX: LiteX provides an infrastructure to create FPGA cores & SoCs, explore digital architectures, and create full FPGA-based systems [6]. LiteX provides many common components required for SoCs, including interconnects, memory interfaces, and CPU cores. It also provides direct interfacing with EDA tools and support for debug infrastructure. LiteX is heavily focused on FPGA development and includes easy interfaces to FPGA debug and software unlike other hardware package managers. However, LiteX is a small ecosystem which is not intended to be as extensible as the other platforms.

	Guix	Bender	Fuse Soc	LiteX	HDL Make
Source file management	✓	✓	✓	✓	✓
Dependency resolution	✓	✓	✓		✓
EDA tool interface		✓	✓	✓	✓
EDA tool packaging	✓				
Environment management	✓				
Bit-exact reproducible builds	✓				
Build caching	✓				

Table 1: Comparison of package manager features

Hdlmake: Hdlmake is a tool designed to help FPGA designers to manage and share their HDL code by automatically finding file dependencies, writing synthesis & simulation Makefiles, and fetching IP-Core libraries from remote repositories [5]. Hdlmake generates GNU Makefiles for (1) fetching IP from repositories, (2) simulating and synthesizing HDL projects, and (3) generating multi-vendor project files. Hdlmake is also targeted mainly towards FPGA EDA flow management. Hdlmake offers some abstraction for IP versions, but mainly relies on git or other version control.

Among the surveyed hardware package managers, Guix is the only one which can ensure a consistent environment and tool versions for reproducible builds. Other package managers aim to provide ease of use by interfacing with multiple EDA tools, but they forgo management of the environment and tool versions, which can lead to different results or build failures without effort (and perhaps guesswork) from the user to replicate the environment. An interesting direction for future work would be to integrate Guix into one of these hardware package managers to combine the benefits of truly reproducible builds with simpler EDA tool interfacing.

7 CONCLUSION

This paper describes our on-going efforts to bring RISC-V into the Guix ecosystem. We have described our work on using Guix both for packaging RISC-V software stacks and RISC-V hardware models. The goal of this paper is to introduce the RISC-V computer architecture community to the Guix package manager and hopefully spark interest in contributing to our efforts to enable reproducible RISC-V software and hardware.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)* 39, 2 (Aug 2011), 1–7.

[3] Bruce R. Childers, Alex K. Jones, and Daniel Mossé. 2015. A Roadmap and Plan of Action for Community-Supported Empirical Evaluation in Computer Architecture. *ACM SIGOPS Operating Systems Review* 49, 1 (Jan 2015), 108–117.

[4] Grigori Fursin and Anton Lokhmotov. 2019. Artifact Evaluation for Reproducible Quantitative Research. *SIGARCH Computer Architecture Today* (Feb 2019).

[5] Javier D. Garcia-Lasheras. 2014. Introducing hdlmake Version 2.0. *8th White Rabbit Workshop* (Oct 2014).

[6] Florent Kermarrec, Sébastien Bourdeauducq, Hannah Badier, and Jean-Christophe Le Lann. 2019. LiteX: an Open-Source SoC Builder and Library Based on Migen Python DSL. *Workshop on Open Source Design Automation* (Mar 2019).

[7] Olof Kindgren. 2019. A Scalable Approach to IP Management with FuseSoC. *Workshop on Open Source Design Automation* (Mar 2019).

[8] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (Jul/Aug 2020), 93–102.

[9] Fabian Schuiki. 2019. BENDER: A Dependency Management Tool for Hardware Design Projects. *Week of Open Source Hardware* (Jun 2019).

[10] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)* 27, 11 (Nov 2019), 2629–2640.