FireMarshal: Making HW/SW Co-Design Reproducible and Reliable

Nathan Pemberton, Alon Amid
University of California, Berkeley

***Abstract*—Reproducibility in the sciences is critical to reliable inquiry, but is often easier said than done. In the computer architecture community, research may require modifying systems from low-level circuits to operating systems and high-level applications. All of these moving parts make reproducible experiments on full-stack systems challenging to design. Furthermore, the computing ecosystem evolves quickly, leading to rapidly obsolete artifacts. This is especially true in the realm of software where applications are often updated on a monthly, or even daily, cadence. In this paper we introduce FireMarshal, a software workload management tool for RISC-V based full-stack hardware development and research. FireMarshal automates workload generation (constructing boot binaries and filesystem images), development (with functional simulation), and evaluation (with cycle-exact RTL simulation). It also ensures, to the extent possible, that the exact same software runs deterministically across all phases of development, providing confidence in correctness and accuracy while minimizing time spent on slow and expensive RTL-level simulation. To ease workload specification, FireMarshal provides sane defaults for common components like firmware and operating systems, freeing users to focus only on project-specific components. Beyond reproducibility, FireMarshal enables continued development of workloads through the use of inheritance, where new workloads can be derived from established and continually updated base workloads. Users communicate their designs through the use of simple JSON configuration files that can be easily version controlled, reused, and shared. In this paper, we describe the design of FireMarshal along with the associated software management methodology for architectural research and development.**

I. Introduction

The computer science community, like other scientific domains, is facing a reproducibility crisis [1]–[5]. To quote Krishnamurthi and Vitek: "Science advances faster when we can build on existing results, and when new ideas can easily be measured against the state of the art" [3]. However, reproducibility can be a challenging goal to achieve. Research artifacts are often difficult to (re)produce and use, requiring specialized knowledge only possessed by the authors [1]. In this paper, we focus on one such class of artifacts: software workloads for full-stack system-on-chip (SoC) research. That is, the software artifacts intended to be run on an experimental hardware system. RISC-V, coupled with open-source SoC development frameworks like OpenPiton, BlackParrot, ESP, and Chipyard, has enabled increasingly complex and capable designs [6]–[9]. Likewise, our ability to simulate complex RISC-V based SoCs has grown rapidly in recent years with new simulators like FireSim, Sniper, and gem5/RISC-V [10]–[12]. Together, these advances have greatly increased the complexity of software that can be reasonably used for evaluation. A working software stack needs to track the exact version of various hardware interfaces (serial ports, reset logic, peripherals, etc.), with software functionality from the firmware up to user-level applications. This increased complexity and velocity presents challenges to the management of software workloads for experimentation and research. Firstly, we must be able to rebuild and re-run our own experiments in a consistent way (repeatability). Second, we must communicate our experiments in a way that allows the community to evaluate and compare them (reproducibility). Furthermore, we would like to avoid duplication of effort within the community by reusing workloads, even as software and hardware evolve (benefaction)[1]. To achieve these goals, we present FireMarshal, a software workload management system to wrangle this complexity by allowing users to describe and share workloads in an unambiguous human and machine readable form that can be stored, version controlled, and shared.

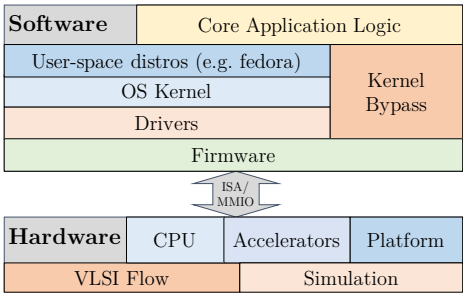

Fig. 1: Full-stack hardware development components. Software typically interacts with the hardware through an ISA for the CPU core and MMIO or ISA extensions for accelerators and platform devices. A hardware research project may need to change a few components of this stack, but is unlikely to change *all* components.

A. Background

1) SoC Development Frameworks: There are now a number of open-source designs for full-stack RISC-V based SoCs [6]–[9]. By "full-stack", we mean that these designs include RTL implementations of processors which support the RISC-V privileged specification [14] and can boot full operating system kernels such as Linux with support for a broad range

[1]The terms "repeatability" and "reproducibility" are used as defined by the ACM [13] while the term "benefaction" is derived from the work of Collberg and Proebsting [1].

of applications. Together with additional RTL platform-level components, these frameworks enables the design of complete RTL SoC implementations at fabrication quality. Figure 1 depicts the typical components included in such a system. Open-source SoC development frameworks often provide a baseline hardware implementation and allow users to modify or add components in order to customize the SoC for a particular use-case. Evaluating such a system often requires a fully functioning software stack from firmware all the way up to user-space applications.

2) Software Stacks: Generating and maintaining a working software stack is challenging. The firmware and kernel require careful configuration to support the underlying hardware platform, and specific versions are often required. In addition to the initial boot binary, a full-featured OS will also require a disk image to provide the user-space environment. This is usually provided by an operating system distribution like Debian [15], Fedora [16], or custom tailored distributions which can be generated using frameworks such as Buildroot [17]. We refer to the combination of boot-binary and disk image as the "software workload". A change in any part of this stack may require changes to others. For example, an updated platform device (e.g. a network interface) may require a new software driver, or a change to the CPU boot configuration may require updates to the firmware. Likewise, updated software components may expose bugs in the hardware implementation or require new features.

3) Simulation: Once a software workload is generated, it can be executed on a spectrum of simulators at different levels of detail and performance. On one end of the spectrum we find functional simulators such as QEMU [18] and riscvOVPSim [19] which aim to faithfully implement the system specification without particular concern for timing modeling, and can often be used as golden models of system behavior for verification. On the other end of the spectrum we have cycle-exact RTL simulators such as VCS, NCSim, ModelSim and Verilator [20]–[23], as well as RTL hardware emulation tools such as Palladium, Zebu, and FireSim [10], [24], [25]. In between, we find functional ISA simulators such as Spike [26], as well as cycle-approximate modeling simulators such as gem5 and Sniper [11], [12]. The general trade-off across the spectrum of simulators is between modeling-detail and performance. While functional simulators are very fast and flexible, RTL simulation is much slower and requires complete hardware designs but provides a higher fidelity of performance results and feature correctness. Ideally, initial software development can be done on functional simulation while slow and expensive cycle-exact simulation is only used for hardware verification and final performance evaluation. However, "toggling" between simulators is not a trivial task, and software setup is often tightly intertwined with some simulator assumptions.

B. Software Workload Management Pitfalls

An ad-hoc approach to software workload management can require a significant up-front time investment with ongoing maintenance costs as designs evolve and workloads are added. There are a number of common pitfalls to this approach. The first is *simulator compatibility*. Each simulation platform may require a slightly different configuration and care must be taken to ensure that software remains correct and faithful to the experiment when switching simulators. Another common pitfall is the generation of *"magic" images*: software workload artifacts that were built ad-hoc and are hard to reproduce. System configuration is challenging and error-prone, if multiple manual steps are needed there is significant room for forgotten steps or inconsistencies. Furthermore, a poorly documented build process can make experiment reproduction difficult or impossible. Finally, without additional system support, experiments may require *manual interventions*. Users may need to wait for the system to boot completely before logging in and running a benchmark, and results need to be manually extracted from the serial output or disk image after a run. These interventions can introduce non-determinism in the experiment and, again, are time consuming and error prone.

C. Requirements

In contrast to the ad-hoc approach, we advocate for the use of an automated *workload management system* where the software workload life-cycle is managed automatically through standardized workload descriptions. We now identify several key requirements that a more general workload management tool should provide:

1) **Flexible Design:** *Users should be able to change any part of the system, but provide only what is needed for their specific project. Reasonable and up-to-date defaults must be available for all system components.*
2) **Maximal Reuse:** *Workloads must be described in a way that can be shared and built upon without inside knowledge.*
3) **Flexible Simulation:** *It must be easy and reliable to switch between different levels of simulation while minimizing software differences.*

II. The FireMarshal Tool

In order to address the requirements of section I-C, we developed FireMarshal. FireMarshal is an open source software workload management tool for RISC-V-based hardware systems development[2]. The system currently supports the Chipyard framework [9] (based on the RocketChip SoC generator [27]), and by extension the FireSim FPGA-accelerated cycle-exact simulator [10]. However, FireMarshal is designed to be extended to other platforms.

FireMarshal generates workloads from machine-readable configuration files (in JSON or YAML). Under FireMarshal, workloads can be tracked in a version-controlled repository and reproduced as-needed. Configuration files typically specify a base workload to serve as a starting point, and any workload-specific changes that must be made to that base (see section III-A for details). FireMarshal comes with several standard

[2]https://github.com/firesim/FireMarshal

workloads that are configured to work on the target platform and are updated regularly to keep in sync with the evolving ecosystem. Complex projects may create hierarchical workloads, where common options are defined once and inherited by many workloads (e.g. unit tests and benchmarks likely share startup code). Most software development can occur in functional simulation on any development machine, with slow and expensive RTL simulation utilized only to drive the final performance evaluation.

FireMarshal is designed around five major phases of the workload lifecycle: **specify, build, launch, test,** and **install**. Users begin by creating a FireMarshal **specification** for their workload (a JSON or YAML file), they can then **build** the software artifacts (i.e. a boot binary and disk image). After building the workload, users can **launch** it in fast functional simulation for **testing** and software development. Once users are satisfied with their workload, they can **install** it to a cycle-exact RTL simulator for performance evaluation. The same tests can be run on both functional and RTL simulation to ensure consistent behavior. Figure 2 depicts a typical workflow.

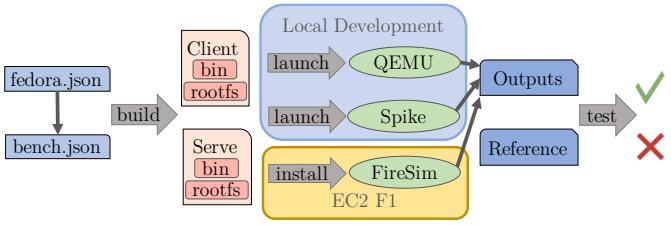

Fig. 2: Typical FireMarshal flow. Configuration files are built into a boot binary and rootfs for each job in the workload. The jobs are then launched in either functional or cycle-exact simulation (FireSim in this example). Finally, run outputs are collected and compared against known-good outputs.

III. Design and Implementation Details

FireMarshal is implemented as an open source command line application along with a set of preconfigured software components. Table I summarizes the commands that FireMarshal supports. In the following sections we will describe how FireMarshal supports each phase of the software workload lifecycle.

Command	Description
build	Construct the filesystem image and boot-binary
launch	Launch this workload in functional simulation
test	build and launch the workload and compare its outputs against a reference
install	Set up a cycle-exact RTL simulator to launch this workload.

TABLE I: Commands supported by FireMarshal.

A. Specify

The workload lifecycle begins with users specifying their workload through a JSON configuration file and any artifacts that should be included (e.g. benchmark sources). FireMarshal provides options for workload inputs and outputs, component customization (e.g. custom Linux source), and hooks for user scripts to run at different points in the workload lifecycle. Table II describes several common options. All options except `base` (described next) are optional.

Option	Description
base	Start from a pre-existing workload - inherit all options unless explicitly overridden
overlay/files	Files to include in the image (e.g. utilities, benchmarks, config files, etc...)
host-init	Script to run before building (e.g. cross-compile)
guest-init	Script to run once on guest (e.g. install packages)
run/command	Script to run every time the image boots (e.g. default experiment)
outputs	Files to copy out of the image after an experiment
post-run-hook	Script to run on the output of the experiment (parse or format results)
linux	Linux customization options including Linux source directory, kernel configuration options to modify, as well as any needed kernel module sources
firmware	Firmware-related options including choice of firmware, and build options.
spike	custom Spike binary to use
spike/qemu-args	Additional arguments to pass to functional simulators
jobs	Additional, related images to build (e.g. each node of a networked workload)

TABLE II: Common FireMarshal configuration options.

1) Inheritance and Jobs: A key concept in FireMarshal is *inheritance*. There are many available options for each workload, some fairly complex. To minimize repeated work, FireMarshal allows users to specify only the options that have changed relative to a `base` workload. For example, many workloads change only the `run` option to create workloads for different benchmarks while the base may include a filesystem overlay or a script for installing benchmark prerequisites.

While individual workloads are intended to run independently, some simulators (like FireSim) support multi-node simulations. In this case, several workloads are expected to run simultaneously. The `jobs` option allows users to specify multiple related workloads. Jobs are implicitly based on the top level workload description and follow all inheritance rules.

2) Boards and Bases: The most basic workloads that users can inherit from are provided by FireMarshal and target a specific hardware platform (called a "board"). This means supporting SoC details, peripherals, and any associated logic or quirks. Users will rarely need to define or modify a board, they should be provided by the SoC generation framework. To define a board, the framework authors must provide a number of key components:

- **Linux Source:** A version of Linux known to work with the board or a link to the default version included with FireMarshal (a recent release of the official Linux kernel).
- **Firmware:** RISC-V systems require a supervisor binary interface (SBI) to perform low-level functions. Users may provide their own implementations of either OpenSBI [28] or the Berkeley Boot Loader (bbl) [29] (or use the included defaults).
- **Drivers:** If the board includes any additional devices such as a network or disk interface, the user must include the

needed Linux drivers. Drivers will be automatically built and loaded by FireMarshal.

- **Base Workloads:** A board must include base workloads for supported distributions (FireMarshal currently supports Buildroot [17] and Fedora [16]).

B. Build

The next step in the workload lifecycle is to build the workload. A FireMarshal build produces a bootable binary, including firmware, Linux kernel, and initramfs (containing platform drivers and other early-boot code), as well as a disk image (figure 3). In some cases, users may wish to produce a workload that does not involve a disk device. In this case, they can specify the `--no-disk` command line option, which causes the disk image to be embedded in the initramfs. This process happens transparently and does not require further user intervention.

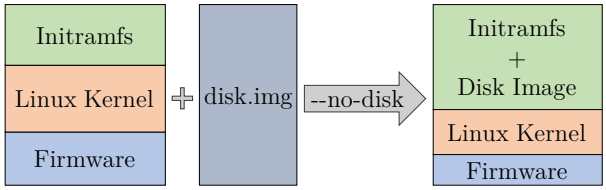

Fig. 3: The outputs of the build command. By default, a complete bootable binary and a disk image are produced. For diskless builds, users provide the `--no-disk` option, in which case the disk image is embedded in the Linux initramfs.

1) Build Phases: When performing a build, FireMarshal goes through a number of steps, although not every step is required for every workload:

1) **Configuration:** The first step is to read the workload configuration file and any potentially related configurations. FireMarshal employs a search order similar to the `$PATH` variable in a Unix shell to locate workloads. Parent workloads are parsed recursively, with children inheriting options from their parents (and overwriting as needed).
2) **Build Parents:** The build process from this step forward is performed recursively to produce filesystem images for all parents. This will be needed later in step 5a.
3) **host-init:** If the workload includes a host-init script, this is run before proceeding to ensure that any generated artifacts are available in future steps.
4) **Boot Binary:** If the user has hard-coded a boot binary (generally a bare-metal workload generated in host-init), the following steps are skipped. If the child workload would not generate a different binary than its parent, FireMarshal simply makes a copy of the parent's binary and skips this step.
    a) **Final Linux Configuration:** To form the final Linux configuration, FireMarshal begins with the RISC-V default configuration. If needed, users can provide Linux kernel configuration "fragments" that contain a list of options to change in the default configuration. These are merged in order, with more recently defined options overwriting earlier duplicates. The use of configuration fragments makes workloads more portable between kernel versions.
    b) **Kernel Module Generation:** With a valid kernel configuration, any needed kernel modules defined in the workload can now be built. This includes system-provided device drivers, as well as user-provided kernel modules.
    c) **Generate Initramfs:** In order to load drivers as early as possible, and to provide a mostly workload-independent boot phase, FireMarshal generates an initramfs as the first-stage init. This initramfs loads both system and user-provided kernel modules.
    d) **Linux Compilation:** The full Linux kernel can now be compiled with a reference to the initramfs to embed.
    e) **Firmware:** The desired firmware is compiled and linked with the Linux binary. At this stage, the boot binary is complete.
5) **Disk Image:** As with the boot binary, users may provide a hard-coded disk image (or no image at all in the case of bare-metal workloads), in which case the following steps are skipped.
    a) **Copy Parent Image and Add Files:** FireMarshal makes a copy of the parent's disk image and then copies over any files from the `file` or `overlay` options.
    b) **guest-init:** At this stage, we have a bootable (albeit incomplete) workload. FireMarshal now configures the workload to run the guest-init script (if provided) and boots it in QEMU. This script is run exactly once.
    c) **Boot Command:** The final step in filesystem generation is to configure the workload to run user-provided code on every startup (from the `command` or `run` options). This is done by inserting a new step in the Linux distribution's init system (init for Buildroot, systemd for Fedora).
6) **Initramfs-Embedded FileSystem:** As shown in figure 3, users may provide the `--no-disk` option to FireMarshal to eliminate the need for a disk device. To do this, FireMarshal runs the build process as described above, but recompiles the kernel with the generated disk image as its initramfs payload.

As this process can be quite time consuming, especially for workloads with deep inheritance hierarchies, FireMarshal uses a dependency tracking system (similar to GNU make) to avoid unnecessary rebuilding. This is done with the doit python package [30].

C. Launch

Once a workload is built, users can run it in functional simulation. The `launch` command takes a workload and calls the specified simulator (QEMU by default) to run it. Serial inputs and outputs are presented to the user interactively and logged to a file for later analysis. For workloads with a `command` or `run` option, the user does not need to interact with the simulation. It is common to omit these options in a parent workload for interactive testing and development. When the simulation completes, FireMarshal copies any output files and the serial port log to an output directory. The `post-run-hook` script (if any) is run against this output to produce final results.

D. Test

While the `launch` command is primarily used for interactive debugging and development, FireMarshal supports hands-off testing with the `test` command. This is useful for running automated regression tests. The test command builds and launches the workload, and then compares the outputs against any provided reference outputs. A complete comparison of outputs is not typically appropriate as there may be irrelevant or non-deterministic output (e.g. time stamps). Instead, FireMarshal is able to clean outputs and allows the reference to contain only a subset of the expected output. A test that produces that subset somewhere in its output is considered a success. Workloads with more complex success criteria can use the `post-run-hook` option to perform custom analysis of outputs.

E. Install

Once a workload passes functional simulation, users may wish to run it against a cycle-exact RTL-level simulator. Unlike functional simulation, RTL-level simulators require hardware-specific configuration and build processes that are out of scope for a workload management tool like FireMarshal. Instead, FireMarshal provides the `install` command to convert the workload specification into a valid configuration for the RTL-level simulator. From there, users interact with the simulator normally to launch the workload. After a simulation, users can verify the outputs using the `test` command with the `--manual` option to compare outputs as if FireMarshal had run the workload. It is important to note that the workload outputs are not modified in any way between the `launch` and `install` commands; the exact same artifacts are run on both simulators. FireMarshal currently supports FireSim, though integration with VCS and Verilator is planned.

IV. Case Studies

A. Page Fault Accelerator

We now describe how FireMarshal was used to develop and evaluate a new accelerator called the Page Fault Accelerator (PFA) [31]. While a complete description of the PFA itself is out of scope for this paper, we quickly summarize its relevant features (see figure 4). The PFA was designed to improve the performance of systems that use remote memory as a swap device (e.g. infiniswap [32]) by handling the basic remote memory lookup and fetch in a new hardware module embedded in the MMU. The complex paging logic in the OS (e.g. LRU schemes, reverse lookups, etc) could be deferred to an asynchronous background thread. The OS interacted with the PFA through several memory-mapped queues and special page table entry values. Similar to regular RDMA, local memory regions were registered with the PFA for fetched pages. The PFA directly interacted with the network interface through its exposed queues (much the same way the OS driver would).

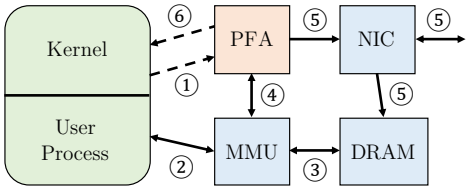

Fig. 4: Block diagram of the Page Fault Accelerator. The kernel asynchronously provides free physical pages to the PFA①. On a page fault②, the MMU consults the page table③ and requests any remote pages from the PFA④. The PFA initiates an RDMA operation from the network adapter⑤. The kernel can now asynchronously request a list of fetched pages for bookeeping purposes⑥. The critical path for a remote page fault (steps ②-⑥) is handled synchronously in hardware while slow kernel interactions (steps ① and ⑥) are moved off the critical path.

The accelerator itself was implemented in a few hundred lines of Chisel [33] and required relatively little engineering time (made possible by leveraging an existing RDMA-capable network interface). However, the kernel modifications to support the accelerator were extensive and complex. Beyond the kernel, user level services like systemd and cgroups required careful configuration and integration with experimental procedures. Once the system was configured, we needed to perform a number of software tasks:

1) Bare Metal Unit Tests: To verify software drivers and the hardware implementation, we implemented a golden model in the Spike functional simulator. The golden model exposed all software-visible interfaces and emulated remote memory. Low-level tests were implemented either completely bare metal or in the RISC-V proxy kernel [29]. These tests were critical for debugging hardware implementation issues and served as a reference for the specification. In our FireMarshal workload configuration, we included a reference to our modified simulator (using the `spike` option) and a script to cross-compile the benchmarks (using the `host-init` option). This test was run using the `launch` command and debugged interactively until we were satisfied that it worked correctly. The serial port output was saved as a reference output (using the `testing/refDir` option) and used for regular automated tests (with the `test` command). This same workload could then be run on FireSim to verify the hardware implementation using the `install` command. This test was

revisited periodically as the hardware specification evolved, or as new corner-cases were identified that required additional unit-tests.

```
{ "name" : "pfa-base"
  "base" : "buildroot",
  "host-init" : "cross-compile.sh",
  "linux" : {
    "source" : "pfa-linux/",
    "config" : "pfa-linux.kfrag",
  },
  "overlay" : "pfa-test-root/",
  "spike" : "pfa-spike"
}

{ "name" : "latency-microbenchmark",
  "base" : "pfa-base",
  "post-run-hook" : "extract_csv.py",
  "jobs" : [
    {"name" : "client",
     "linux" : {"config" : "pfa.kfrag"}
     "command" : "latencyTest.sh"},
    {"name" : "server",
     "base" : "bare-metal",
     "bin" : "serve"} ]
}
```

Listing 1: Base workload for PFA Linux unit tests (upper) and an example microbenchmark workload (lower). The microbenchmark has one Linux-based job for the client benchmark, and a bare-metal job to serve remote memory. The jobs will be instantiated as network nodes in FireSim simulation.

2) Linux Unit Tests: The most significant engineering challenge in the PFA project was in modifying the Linux kernel to asynchronously process page faults, and configuring the experimental environment in the OS (e.g. configuring an appropriate swap device, setting cgroup limits, etc.). We also needed to modify the default Linux kernel build configuration to enable certain swapping-related features. To test these changes, we required a simple Linux environment for unit tests. We implemented these tests on Buildroot, a bare-bones Linux distribution designed for embedded workloads. Buildroot could boot quickly and minimized the amount of extra code running in the system. We began with `pfa-base`, a base workload that would handle the common setup tasks. Individual tests and benchmarks inherited from `pfa-base`, typically adding only a Linux configuration fragment and a `command` option to run a particular benchmark. Listing 1 shows an example for one particular benchmark, a microbenchmark that measured the latency of each step in a remote page fault (see figure 5 for example results). Note that there were many workloads similar to `latency-microbenchmark`, but only one `pfa-base`.

The first step in developing the kernel modifications was to create a non-accelerated baseline by emulating the PFA's behavior in the regular page fault handler. These modifications were non-trivial and introduced complex, non-deterministic, bugs. QEMU allowed us to run long-running tests in a reasonable time frame, as well as providing an integrated GDB server for interactive debugging. Once we were satisfied with the emulated behavior, we introduced the real hardware driver and ran the tests against our Spike golden model. The only change required in the workload was a one-line Linux configuration fragment (to enable the PFA driver). This meant that the experimental setup and test parameters were identical between the two simulators, giving us confidence that any errors were due only to the driver change. When everything worked in both Spike and QEMU, we could run the unmodified workload in RTL simulation to verify the hardware implementation (using `install`). Since the software had been verified against the golden model in the Linux workloads, and the golden model had been verified with the bare-metal unit tests, we could narrow down any errors quickly. Since the process of targeting different simulators was automated, there was minimal room for human error.

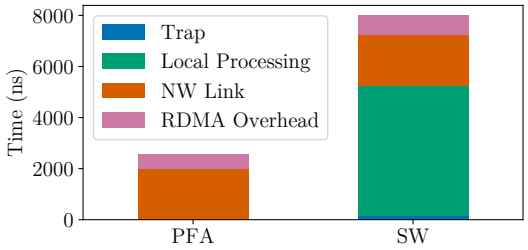

Fig. 5: Example result from the PFA fault latency microbenchmark. The figure shows the time contribution of different phases of a remote page fault between the accelerated PFA case and a pure software implementation.

3) End-To-End Benchmarks: Once we had confidence that the system operated correctly, the PFA was evaluated against end-to-end macro-benchmarks. Some of these real-world applications had many dependencies that would be difficult to fulfil manually as required by Buildroot. Instead, we leveraged the package management system of a full-featured OS (Fedora) to install dependencies at build time (using a `guest-init` script). While more full-featured, Fedora took significantly longer to boot and introduced hard-to-debug features like asynchronous systemd services. Therefore, it was critical to ensure the kernel implementation was correct before running the end-to-end benchmarks.

The workload description process was similar to that of the Buildroot unit tests, but we additionally included a `post-run-hook` option to automatically process experimental results (from the serial output) into CSV files for analysis. This ensured that experiments could be re-run by ourselves or external users and processed in a consistent way. In other words, the FireMarshal workload served as unambiguous documentation of our experimental procedure for reproducibility and comparison.

In all, FireMarshal allowed us to run unit tests and track experimental results with minimal human interaction, enabling a tight feedback loop as the project evolved.

B. Benchmarking: SPEC2017

Not every research project requires custom software for evaluation. For example, changes to a branch predictor or cache design are best evaluated using standard benchmarks. In this section, we describe how FireMarshal was used to provide one common benchmark used in the architecture community: SPEC2017 [34]. SPEC provides a number of scripts for interacting with the benchmark, while projects like Speckle [35] simplify the process of cross compiling for new architectures. However, having the binaries alone is not sufficient for a benchmark. Users still need to invoke and measure the benchmarks in a consistent way as well as compile and format results. Listing 2 shows one example workload for the intspeed benchmark suite.

In this section we describe an experiment to compare two different branch predictors on the Berkeley Out-Of-Order Machine (BOOM [36]) using the intspeed benchmark suite from SPEC2017 (on the reference dataset). In one case, we use an older branch predictor from BOOM v2 (based on Gshare [37]), in the other we use the more recent TAGE-based predictor [38], [39].

```
{ "name" : "intspeed",
  "base" : "buildroot",
  "host-init" : "speckle-build.sh
     intspeed ref",
  "overlay" : "overlay/intspeed/ref",
  "rootfs-size" : "3GiB",
  "outputs" : ["/output"],
  "post-run-hook" : "handle-results.py",
  "jobs" : [
    { "name": "600.perlbench_s",
      "command": "./intspeed.sh 600.
         perlbench_s --threads 1"},
    ...
    { "name": "657.xz_s",
      "command": "./intspeed.sh 657.xz_s
         --threads 1"}
}
```

Listing 2: Workload for the intspeed benchmark suite from SPEC2017. In total, there are 10 jobs, one for each benchmark in the suite. Jobs are able to run in parallel in FireSim.

In the general case, SPEC does not require changes to system software such as the Linux kernel, and simply inherits from the default Buildroot environment. This means that the SPEC workload will transparently receive any updates to the built-in workloads and will be portable across many boards and versions. Furthermore, users are free to copy this workload description and change the base if their particular example requires additional configuration (no changes were needed for our branch-predictor experiment). Cross-compilation of the benchmark is provided by Speckle (in the `host-init` option), while the FireMarshal workload marks the Speckle outputs as an overlay. For each benchmark, the run script will place results in `/output`, so FireMarshal is instructed to retrieve these after the workload finishes running (using the `outputs` option).

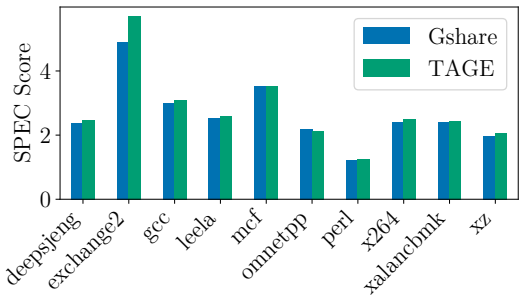

Fig. 6: Combined graph output of the spec2017_intspeed workload. A similar graph is generated automatically for each experiment while the combined graph can be generated manually using an included script.

Each benchmark in the suite is independent and can run in parallel. We exploit this in the workload by specifying 10 jobs (one for each benchmark). Each job differs only in the `command` option, specifying which benchmark to run. When installed to FireSim, each job is instantiated as a node in the simulated cluster and run in parallel. This optimization reduced the runtime for our experiment from about two weeks to roughly two days.

Once the workload has finished, we pass the results through a `post-run-hook` script that combines all results into a CSV file (listing 3), as well as plotting a simple diagram for quick reference (omitted for brevity). Figure 6 shows the combined output of our two experiments from the result CSVs.

```
name,RealTime,UserTime,KernelTime,score
600.perlbench_s,1428.54,1428.0,0.43,1.24
...
657.xz_s,3034.63,2999.81,34.63,2.04
```

Listing 3: Example CSV output of the spec2017_intspeed workload for the TAGE configuration.

This workload was developed entirely on a cheap local machine using QEMU and without regard for the eventual branch-prediction experiment. It was run for the first (and only) time on cycle-exact simulation to gather the final performance numbers on the real hardware designs. Since FireMarshal ensures that identical inputs are run on both functional and cycle-exact simulations, we had confidence that the workload would run correctly the first time.

1) User Experience: In practice, most users do not need to look at the workload description. It was written once and can be reused by anyone without modification. A typical user would run the SPEC workload with the following steps:

1) **Install SPEC:** Since SPEC is closed-source software, we are unable to automate installation. Users must first acquire and install the SPEC benchmark suite sources and a license to use them.
2) **Download the FireMarshal Workload:** The FireMarshal workload can be cloned from a public git repository[3].
3) **Build the Workload:** Once SPEC is installed, FireMarshal can build the entire workload suite with one command: `marshal build intspeed.json`.
4) **Install the Workload:** Once built, the workload could be run in functional simulation with the `launch` command, but this is not typically needed since users do not need to do any software development. Instead, users will typically have FireMarshal create RTL simulator-compatible configuration files: `marshal install intspeed.json`.
5) **Run the Simulation:** Users now interact with their RTL simulator as usual, providing their hardware configuration and any other simulation parameters they wish. When the simulation completes, the simulator will provide an output directory containing the benchmark results as generated by the post-run-hook script (see figure 6 and listing 3).

Other than acquiring the licensed SPEC suite itself, we did not need to interact with any target software in order to run the branch-prediction experiment. All that was required was to generate our desired hardware configurations (the feature we actually cared about); the software "just worked". Furthermore, results were recorded in a standard and reproducible way. If we were to add a new branch predictor in the future, we could have confidence in our experimental setup to compare against previous results without needing to re-run them. Most importantly, now that the workload has been implemented, it is freely available for anyone to use or improve without repeated effort.

While we describe SPEC here, there are other similar benchmark workloads already available including CoreMark [40] and the ONNX-runtime deep learning framework [41], [42]. As new benchmarks are ported or developed, they too can be shared with the community in a similar fashion.

C. Education: Computer Architecture and Engineering

Educational settings are notoriously sensitive to consistency and reproducibility of results. As computer science classes scale to a large numbers of students, mass assignments and automated grading are becoming necessities in many university courses. However, reproducibility is often extremely sensitive to software versioning and simulator compatibility. A simple change in the Linux kernel version can dramatically change performance characterization results, which would be reflected in various student assignment submissions.

Furthermore, we would like students to invest their time in the educational objectives of characterization and measurement, rather than spending the majority of their time on setting up environments and building boiler-plate setup procedures.

While system environment platforms such as Docker or Vagrant provide a solid platform for systems-oriented classes, they are insufficient for hardware-simulation classes that require support for a broader set of configurations and cross-compilation. We used FireMarshal in an advanced graduate and undergraduate class on the subject of hardware for machine learning [43]. As part of this class, students had to optimize tiled convolution and matrix multiplication implementations for an RTL implementation of a machine learning accelerator integrated into a RISC-V SoC. The optimizated software implementations were to be used as a library within DNN inference applications utilizing ResNet-50 and MobileNet DNN models. Figure 7 depicts the student workflow for this assignment.

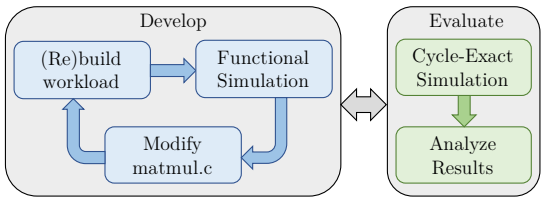

Fig. 7: Student workflow. Students were asked to tune a matrix multiplication routine for a particular deep learning accelerator. The course staff provided a FireMarshal workload as a starting point. Students used fast and inexpensive functional simulation when developing their code, while slow and expensive cycle-exact simulation was only used to evaluate performance. Due to FireMarshal's deterministic builds, results were portable between simulation environments and *repeatable* between iterations. Results could be *reproduced* accurately by the course staff for grading.

In order to enable students to integrate their optimized libraries with the DNN inference applications, we used a FireMarshal workload definition. This enabled students to focus their time on the development of their library implementations rather than spending it on setting up their testing environment on various iterations and platforms.

Furthermore, as part of their development process, students initially developed their implementations using the Spike functional simulator, and then performed measurements using FireSim. By using the same FireMarshal workload definition, students were able to take advantage of the portability of FireMarshal workloads across different RISC-V simulation platforms.

Thanks to the determinism of FireSim simulations, and the reproducibility of FireMarshal workloads, students were able to obtain repeatable results down to an exact cycle-count of each executing application and course staff could reproduce these results for grading purposes.

D. Other Use Cases

We now briefly summarize additional real-world use cases.

[3]https://github.com/ucb-bar/spec2017-workload

Centrifuge is a tool for design-space exploration of accelerators using high level synthesis [44]. Typical Centrifuge benchmark variants differ only in the accelerator wrapper code, inheriting from a common base that provides experimental procedures. This base, in turn, inherits from a generic centrifuge workload that provides kernel and user-space modifications to support the accelerator interfaces.

Keystone is a secure enclave for RISC-V based systems [45]. Unlike other hardware enclaves, most of Keystone is implemented in the firmware and operating system. Unfortunately, this makes the system difficult to set up correctly. To avoid these issues, Keystone provides a FireMarshal workload that can be used as the base for any existing experiments. Enabling Keystone is as simple as switching the `base` option in a workload from the board default to `keystone-base.json`.

V. Related Work

Containers have become a standard mechanism for building and distributing workloads in the cloud and other server-side environments. Docker is a popular tool for describing and distributing these workloads [46]. Docker was influential in the design of FireMarshal and they share many features and design principles. In terms of scope, Docker provides mechanisms for workload inheritance (like FireMarshal's `base` option), mechanisms for setting up the disk image (e.g. `guest-init` or `files`), and startup software. However, Docker works only on Linux userspace setup and cannot modify the full software stack as needed for architectural research. It is also not designed for automated experiments with testing, output parsing, or simulator integration.

There are also numerous tools for Linux distribution setup. For example, Buildroot allows users to describe a minimal Linux environment in a configuration file [17]. Indeed, FireMarshal uses Buildroot internally to construct the lowest base workload. There are other similar tools such as Kickstart [47], AutoYast [48], and others. Of particular note is Yocto which includes a flexible and composable build system (called bitbake) similar to FireMarshal's inheritance model for disk-related options [49]. However, these systems are primarily designed for system administration and deployment rather than experimentation and do not provide full-stack setup, experiment management, or simulator integration. As such, they serve as good starting points for FireMarshal base workloads.

Collective Knowledge Workflow (CK) is a general purpose workflow framework, similar to bitbake [50]. FireMarshal focuses specifically on workload management for architecture research, including portability between simulators. CK does not prescribe a particular schema or base components while these are central contributions of FireMarshal.

Finally, it is common for hardware development frameworks to include some form of SDK to jump-start software development. For example, Raspberry Pi, Nvidia and Xilinx all provide SDKs for some of their products [51]–[53]. For RISC-V SoCs, examples include the Ariane SDK and the SiFive freedom-u-sdk [54], [55]. The SDKs integrate an embedded distribution generator (Buildroot [17] and OpenEmbedded [56], respectively), along with a default Linux kernel configuration and firmware tuned for their platforms. These SDKs are similar to FireMarshal in their ability to produce working images, but are primarily targeted at producing a production software platform rather than a suite of experiments over the rapidly changing and non-standard hardware used by architecture researchers.

VI. Future Work

FireMarshal is designed to be extensible, especially in its support for new platforms. In the future, we hope to extend the available boards to include other SoC development frameworks like OpenPiton. We are currently working on support for a post-tapeout bring up and evaluation effort where the existing suite of FireMarshal-based benchmarks are run in an identical manner in both function simulation and during bringup allowing researchers to triage issues with potentially faulty hardware. While the board and base distribution systems are well modularized, simulator integration is not. In the future, we hope to enable pluggable simulator connectors to expand the scope and capability of the `install` command, including support for other RTL simulation platforms.

Another major limitation is the lack of a network model in functional simulation. QEMU-based simulations can access the host's internet, but we cannot currently model inter-job networking. This means that we cannot fully test and develop networked applications without using FireSim.

Finally, FireMarshal was designed to promote re-use and sharing among the broader community. To succeed in this, more workloads and benchmarks need to be ported and shared. We also need to continue to maximize re-use by providing more tools and base workloads for common tasks like integrating accelerators, or running bare-metal applications. All of which currently require poorly documented and complex techniques.

VII. Conclusion

In this paper, we have presented FireMarshal, a tool to create and share software artifacts for reproducible and repeatable experiments on full-stack SoCs. FireMarshal is available under a BSD-style open source license and already includes a number of useful workloads. Reproducibility and repeatability are critical to academic inquiry, but the complexity of full-stack hardware systems makes these properties challenging to achieve in practice. FireMarshal manages software complexity in a concrete and shareable form while automating the tedious and error prone tasks in the development life cycle. With FireMarshal, researchers can develop repeatable workloads that can be reproduced by the academic community and built upon by anyone.

Acknowledgements

This work was funded in part by the NSF CCRI Award 2016662, in part by the Advanced Research Projects Agency-Energy, U.S. Department of Energy, under Award Number DE-AR0000849, and in part by ADEPT Lab industrial sponsors and affiliates.

References

[1] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Commun. ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016, ISSN: 0001-0782. DOI: 10.1145/2812803. [Online]. Available: https://doi.org/10.1145/2812803.

[2] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Comput. Surv.*, vol. 51, no. 3, Jul. 2018, ISSN: 0360-0300. DOI: 10.1145/3186266. [Online]. Available: https://doi.org/10.1145/3186266.

[3] S. Krishnamurthi and J. Vitek, "The real software crisis: Repeatability as a core value," *Commun. ACM*, vol. 58, no. 3, pp. 34–36, Feb. 2015, ISSN: 0001-0782. DOI: 10.1145/2658987. [Online]. Available: https://doi.org/10.1145/2658987.

[4] J. Vitek and T. Kalibera, "Repeatability, reproducibility, and rigor in systems research," Oct. 2011, pp. 33–38. DOI: 10.1145/2038642.2038650.

[5] B. Rous, *The acm task force on data, software, and reproducibility in publication*. [Online]. Available: https://www.acm.org/publications/task-force-on-data-software-and-reproducibility.

[6] J. Balkind, M. McKeown, Y. Fu, *et al.*, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, Atlanta, Georgia, USA: ACM, 2016, pp. 217–232, ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872414.

[7] D. Petrisko, F. Gilani, M. Wyse, *et al.*, "Blackparrot: An agile open source RISC-V multicore for accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020. DOI: 10.1109/MM.2020.2996145.

[8] P. Mantovani, D. Giri, G. Di Guglielmo, *et al.*, "Agile SoC development with open ESP," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, Virtual Event, USA: Association for Computing Machinery, 2020, ISBN: 9781450380263. DOI: 10.1145/3400302.3415753. [Online]. Available: https://doi.org/10.1145/3400302.3415753.

[9] A. Amid, D. Biancolin, A. Gonzalez, *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020. DOI: 10.1109/MM.2020.2996616.

[10] S. Karandikar, H. Mao, D. Kim, *et al.*, "Firesim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, Los Angeles, California: IEEE Press, 2018, ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014.

[11] N. B. Mallya, C. Gonzalez-Alvarez, and T. E. Carlson, "Flexible timing simulation of RISC-V processors with sniper," in *Second Workshop on Computer Architecture Research with RISC-V*, ser. CARRV '18, Los Angeles, California, USA.

[12] T. Ta, L. Cheng, and C. Batten, "Simulating multi-core RISC-V systems in gem5," in *Second Workshop on Computer Architecture Research with RISC-V*, ser. CARRV '18, Los Angeles, California, USA.

[13] *Artifact review and badging*, version 1.1, Association of Computer Machinery, Aug. 2020. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[14] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, volume ii: Privileged architecture*, version 20190608-Priv-MSU-Ratified, RISC-V Foundation, Jun. 2019.

[15] *Debian – the universal operating system*, Software in the Public Interest, Inc. [Online]. Available: https://www.debian.org/.

[16] *Fedora*, Red Hat Inc. [Online]. Available: https://start.fedoraproject.org/.

[17] *Buildroot*, Buildroot Association. [Online]. Available: https://buildroot.org/.

[18] *Qemu*, version 5.0.0, Software Freedom Conservancy, Apr. 28, 2020. [Online]. Available: https://github.com/qemu/qemu/tree/v5.0.0.

[19] *Riscvovpsim*, Imperas Software, Ltd., Oct. 22, 2020. [Online]. Available: https://github.com/riscv/riscv-ovpsim.

[20] W. Snyder, *Verilator*, Veripool, 2021. [Online]. Available: https://www.veripool.org/ftp/verilator_doc.pdf.

[21] G. Singh, "Gate-level simulation methodology: Improving gate-level simulation performance," Cadence Design Systems, Inc., Tech. Rep., 2015.

[22] *Modelsim*, Mentor, a Siemens Business. [Online]. Available: https://www.mentor.com/products/fv/modelsim/.

[23] *Vcs*, Synopsys, Inc. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html.

[24] "Cadence palladium xp ii verification computing platform," Cadence Design Systems, Inc., Tech. Rep., 2013.

[25] *Zebu server asic emulator*, Synopsys, Inc. [Online]. Available: https://www.synopsys.com/verification/emulation/zebu-server.html.

[26] *Spike risc-v isa simulator*, https://github.com/riscv/riscv-isa-sim, The Regents of the University of California, 2019.

[27] K. Asanović, R. Avižienis, J. Bachrach, *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[28] *Opensbi*, Western Digital Corproration. [Online]. Available: https://github.com/riscv/opensbi.

[29] *The RISC-V proxy kernel*, https://github.com/riscv/riscv-pk, The Regents of the University of California, 2019.

[30] E. Schettino, *Pydoit*. [Online]. Available: https://pydoit.org/.

[31] N. Pemberton, "Enabling efficient and transparent remote memory access in disaggregated datacenters," Master's thesis, EECS Department, University of California, Berkeley, Dec. 2019. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-154.html.

[32] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 649–667, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[33] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[34] *Spec2017*, Standard Performance Evaluation Corporation. [Online]. Available: http://spec.org/cpu2017/.

[35] C. Celio, A. Waterman, D. Palmer, and D. Biancolin, *Speckle*. [Online]. Available: https://github.com/ccelio/Speckle/tree/firesim-2017.

[36] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[37] S. McFarling, *Combining branch predictors*, 1993.

[38] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020.

[39] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-level Parallelism - JILP*, vol. 8, Jan. 2006.

[40] S. Gal-On and M. Levy, *Exploring coremark™ – a benchmark maximizing simplicity and efficacy*, EEMBC.

[41] H. Genc, S. Kim, A. Amid, *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021, To be published.

[42] *Onnx runtime*, Microsoft Corporation. [Online]. Available: https://microsoft.github.io/onnxruntime/.

[43] A. Amid, A. Ou, K. Asanović, Y. S. Shao, and B. Nikolić, "Vertically integrated computing labs using open-source hardware generators and cloud-hosted FPGAs," in *IEEE International Symposium on Circuits and Systems*, May 2021, To be published.

[44] Q. Huang, J. Wawrzynek, C. Yarp, *et al.*, "Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-acceleration," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942048.

[45] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020.

[46] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[47] C. Lumens, *Kickstart*. [Online]. Available: https://pykickstart.readthedocs.io/en/latest/index.html.

[48] *Autoyast*, SUSE. [Online]. Available: https://doc.opensuse.org/projects/autoyast/.

[49] *Yocto project*. [Online]. Available: https://www.yoctoproject.org/.

[50] G. Fursin, A. Lokhmotov, and E. Plowman, "Collective knowledge: Towards R&D sustainability," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16, Dresden, Germany: EDA Consortium, 2016, pp. 864–869, ISBN: 9783981537062.

[51] *Noobs (new out of box software)*, version 3.0, Raspberry Pi. [Online]. Available: https://github.com/raspberrypi/noobs/releases/tag/v3.0.

[52] *Nvidia jetson linux developer guide*, version 32.4.3, NVIDIA Corporation, Jul. 2020. [Online]. Available: https://docs.nvidia.com/jetson/l4t/index.html.

[53] *Zynq ultrascale+ MPSoC software developer guide*, version v12.0, Xilinx, Inc., Jul. 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1137-zynq-ultrascale-mpsoc-swdev.pdf.

[54] *Ariane sdk*. [Online]. Available: https://github.com/pulp-platform/ariane-sdk.

[55] *Freedom-u-sdk*, Sifive, Inc. [Online]. Available: https://github.com/sifive/freedom-u-sdk/tree/2020.09.

[56] *Openembedded*. [Online]. Available: http://www.openembedded.org/wiki/Main_Page.