Using a modern software build system to speed up complex hardware design

Varun Koyyalagunta
Tenstorrent
Austin, TX

***Abstract*-Bazel [1] is a modern open source build system that has been deployed by companies and open source projects [2] for its correctness in build output, scalability to parallelize tasks across a multiple machines, and times savings by using a shared remote cache of build and tests products across users and machines. Some projects have had 6x speed up in test time [4]. We explore how these same features can speed up the hardware design cycle, including the building of simulation models and running of qualification checks (smoke).**

I. Introduction

The RTL verification workflow usually involves

1) Checking out a design and external dependencies
2) Making a change
3) Building simulation models
4) Running smoke tests
5) Pushing that change to other users

Bazel's external dependency management, remote caching, checking of build correctness, and remote execution features speed up almost all these steps.

II. External dependency management

Larger RTL projects generally have external dependencies, such as 3rd party IP or tools. These IPs may be used by some or all of the units in a project. Bazel's external dependency management system will fetch those dependencies automatically on-demand when needed, without users having to take any additional steps. A top-level repo specifies a URL and git commit for each IP dependency, and then Bazel will fetch the dependencies on-demand when instantiated. For example, say there is a top-level repository with testbenches 'tb0' and 'tb1', which depend on external IP 'ip0' and 'ip1', respectively. If you checkout the top-level repository and build and run 'tb0', Bazel recognizes that it depends on 'ip0' and will fetch it into an output area. 'ip1' will not be fetched until 'tb1' is built.

Some similar tools such as git's submodules force users to learn new steps and more manually manage the updating of dependencies, resulting in a possibly on-going maintenance and developer burden. Git submodules, without extra tooling, also require you to fetch all dependencies, even when not required. At Tenstorrent on a complex CPU design, Bazel's external dependency management system allows IP such as caches, cores, and memories, and VIP such as test generators and checkers to be maintained in separate git repos and seamlessly stitched together.

III. Remote caching

After checking out a design, a user may build and run some tests. Usually those simulations models have already been built by another user or automated continuous integration (CI) tool. And those same tests may have already been run as well. Bazel's remote caching allows those previous results to be downloaded instead of doing redundant work. No costly simulation license is even checked out if there is a 100% remote cache hit. At Tenstorrent, redundant smoke testing time was reduced from over 1 hour down to as little as 2 minutes with remote caching.

Remote caching requires no extra code from the user, they simply passes a `--remote_cache` [14] switch to bazel pointing to where the remote cache is. It can even be a directory on the file system. When a Bazel build is executed, Bazel stores a checksum of each input of every action. Bazel first looks up in the local build output area (the local cache) to see if all the outputs are up-to-date. If yes, then do nothing. If no, then Bazel looks up in a remote cache whether the action, with all the same inputs, has already been done. If yes, then it will download the outputs from the remote cache. If no, then it will run the action.

IV. Build correctness (sandboxed execution)

When an RTL or testbench change is made, the change is usually isolated to a single unit. However, basic qualifications usually involve running tests in all unit and system testbenches. With Bazel's dependency checking, only simulation models that are affected by that change are rebuilt and rerun. Tools like Make [7] have similar dependency checking and work optimization, however they are error-prone because the dependencies are not verified. This results in users pessimistically doing `make cleans` to make sure their results are correct, thereby negating the advantages of the dependency checking. Bazel by default uses a sandboxed execution strategy where every build step can only access the files that are explicitly listed as inputs, allowing the user to trust the dependency checking. The default implementation of sandboxed execution does this by executing every build action in a temporary directory, and populating that directory with symlinks to the explicitly listed inputs. This way if an action tries to access an unspecified, it will not be able to find it.

V. Remote execution

When simulation models are built and run, users may have access to multiple machines on which that work could be parallelized. If the compute machines are setup correctly, Bazel's remote execution allows that work to be executed on other machines with the addition of a single switch and no changes to the user's codebase. At Tenstorrent, the smoke testing time was reduced from over 1 hour down to under 20 minutes when using 100 remote threads. (When there are remote cache hits, this is reduced even further).

Like remote caching, remote execution simply requires an additional `--remote_execution` [15] switch on the client side to point to the remote execution server. There are commercial and open source remote execution servers that can be deployed [9].

A. Interfacing with existing job dispatchers

Bazel uses an open remote execution protocol to dispatch remote execution request job requests [8]. The remote execution protocol has the added advantage of managing all file dependency transfers, so that no shared filesystem, such as NFS, is required between machines. All input files are transferred to the remote job, and all output files are downloaded from the remote job.

However a hardware company with many compute resources may already have an existing job dispatching system that they are using and want to share with both Bazel workloads and non-Bazel workloads. Because Bazel uses an open protocol to send remote execution requests, a server can be written to translate bazel remote execution job requests into calls to an existing job dispatching system. Tenstorrent has put in place such a server to convert bazel remote execution requests to IBM Spectrum LSF [12] requests.

VI. Comparison to other Build Systems

Make is one of the most widespread build tools. But modern compute farms have much more resources then when it was first invented, and Bazel takes advantage of these.

Feature Comparison

Feature	Make	Bazel
Local parallel execution	Yes	Yes
Local caching	Yes[1]	Yes
Remote caching	No[2]	Yes
Remote execution	No[2]	Yes
Sandboxed execution	No	Yes
External dependency management	No[3]	Yes

[1] Make's caching is timestamp-based, so simply touching a file would cause a recompile. Bazel's caching is contents-based, meaning the file contents would need to change to trigger a recompile.
[2] Certain workloads can manually be cached or remotely executed by using tools like ccache [11] or distcc [10], but the process is manual and those tools only applies to certain workloads, eg C compiles and not Verilog compiles.
[3] This is sometimes managed clunkily by having Make initialize git submodules.

While the rest of the points are performance improvements, sandboxed execution resulting in correct builds is the greatest functional improvement over Make provided by Bazel.

VII. Setting up Bazel

While Bazel has extensive support for a wide variety of software lanuages and tools [3], hardware development flows are not as widely supported, barring some nascent open source projects [5] [6]. However, Bazel allows for adding custom rules to support new workflows. This requires users to learn the Bazel rule writing process and the Starlark language, but enables the user to self-support any workflow, including ones using commercial and open source Verilog tools.

In the following example we use the open source rules_verilator project to setup a small example testbench in Bazel. This testbench has SystemVerilog and C++ components.

The SystemVerilog component includes a simple dut and a test harness which calls a C++ reference model.

top.sv

```
/* verilator lint_off DECLFILENAME */

module dut(

   input logic[7:0] i,
   output logic[7:0] o,
   input clk

);

   always_ff @(posedge clk) o <= i + 1;

endmodule

module top(
   input clk
);

   logic [7:0] o, i;

   initial begin
      if (!$value$plusargs("i=%d", i)) i = 0;
   end

   dut dut(.*);

   int count = 0;

   import "DPI-C" function byte dut_model(byte i);

   always @(posedge clk) begin
      count <= count + 1;
      if (count == 1) begin
         automatic logic [7:0] expected = dut_model(i);
         assert(o == expected) else
            $error("Mismatch: Expected %0d, Actual %0d", expected, o);
         $finish;
      end
   end

endmodule
```

The C++ component includes a simple reference model.

dpi.cpp

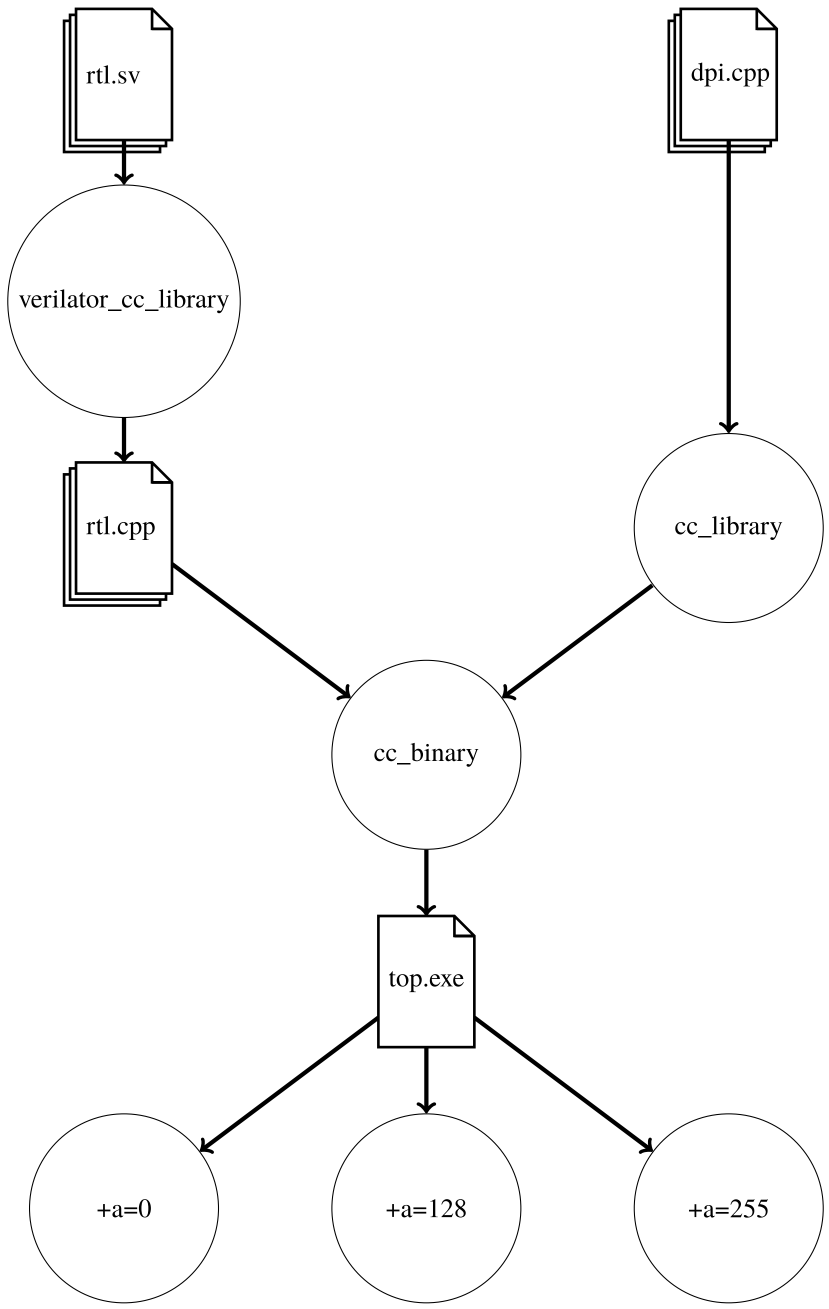
```
#include <cinttypes>

extern "C" std::uint8_t dut_model(std::uint8_t i) {
    return i + 1;
}
```

BUILD.bazel is the Bazel component tying everything together. It is written in a Pythonic language called Starlark. The file is mostly composed of ”rules.” Rules generally describe a set of inputs, a command to run, and a set of outputs. Rules come together to describe a build graph that Bazel executes.

BUILD.bazel

```
# These loads import external dependencies from
# https://github.com/kkiningh/rules_verilator
# These imported rules create the whole toolflow needed to build and run
# verilog, including fetching and building verilator from
# https://github.com/verilator/verilator
load("@rules_verilator//verilator:defs.bzl", "sv_library",
    "verilator_cc_library")

# Th verilator_cc_library rule elaborates top.sv through verilator and outputs
# a logical grouping of C++ files that are generated by Verilator
# The internals of the verilator_cc_library rule, including the call to verilator, are
    ↪ implemented in https://github.com/kkiningh/rules_verilator
verilator_cc_library(
    name = "top_vlt",
    vopts = ["--assert"],
    srcs = ["top.sv"],
    mtop = "top",
)

# The cc_library rule creates a logical grouping of C++ files, here just one
# dpi.cpp file
# The implementation of the cc_library rule are packaged with Bazel, and can be
    ↪ configured with more Bazel options to specify which C++ compiler to use if a non-
    ↪ default one is required.
cc_library(
    name = "dpi",
    srcs = ["dpi.cpp"],
)

# The cc_binary rule takes its inputs, top.cpp, dpi.cpp, and generated C++ from
# Verilator, and compiles it into an executable
cc_binary(
    name = "top",
    srcs = ["top.cpp"],
    deps = [":top_vlt", ":dpi"],
)

# The following code specifies three tests to run on the top executable
tests = [
    0,
    128,
    255,
]

[
    sh_test(
        name = "test_{a}".format(a = a),
        srcs = [":test.sh"],
        data = [":top"],
        args = ["$(location :top)", "+a={}".format(a)],
    )
    for a in tests
]

exports_files(["top.sv", "dpi.cpp", "BUILD.bazel"])
```

Bazel reads the above BUILD.bazel and constructs a build action graph like the following. Each circular node describes an action, and each rectangle represents input and/or output files. Bazel can then check to see if the action has already been done in the local and remote action cache. If not, it will execute the actions, parallelizing the actions that don't have dependencies. The execution can take place locally, or remotely on a machine to where the inputs are uploaded and the outputs are downloaded.



Bazel example action graph

The specifying of tests in Bazel can be tedious, however specifying tests within Bazel allows remote caching and execution to work seamlessly on them. For larger scale testing, users could use other scripting languages to generate the BUILD.bazel file and get the flexibility of a full programming language and the power of Bazel.

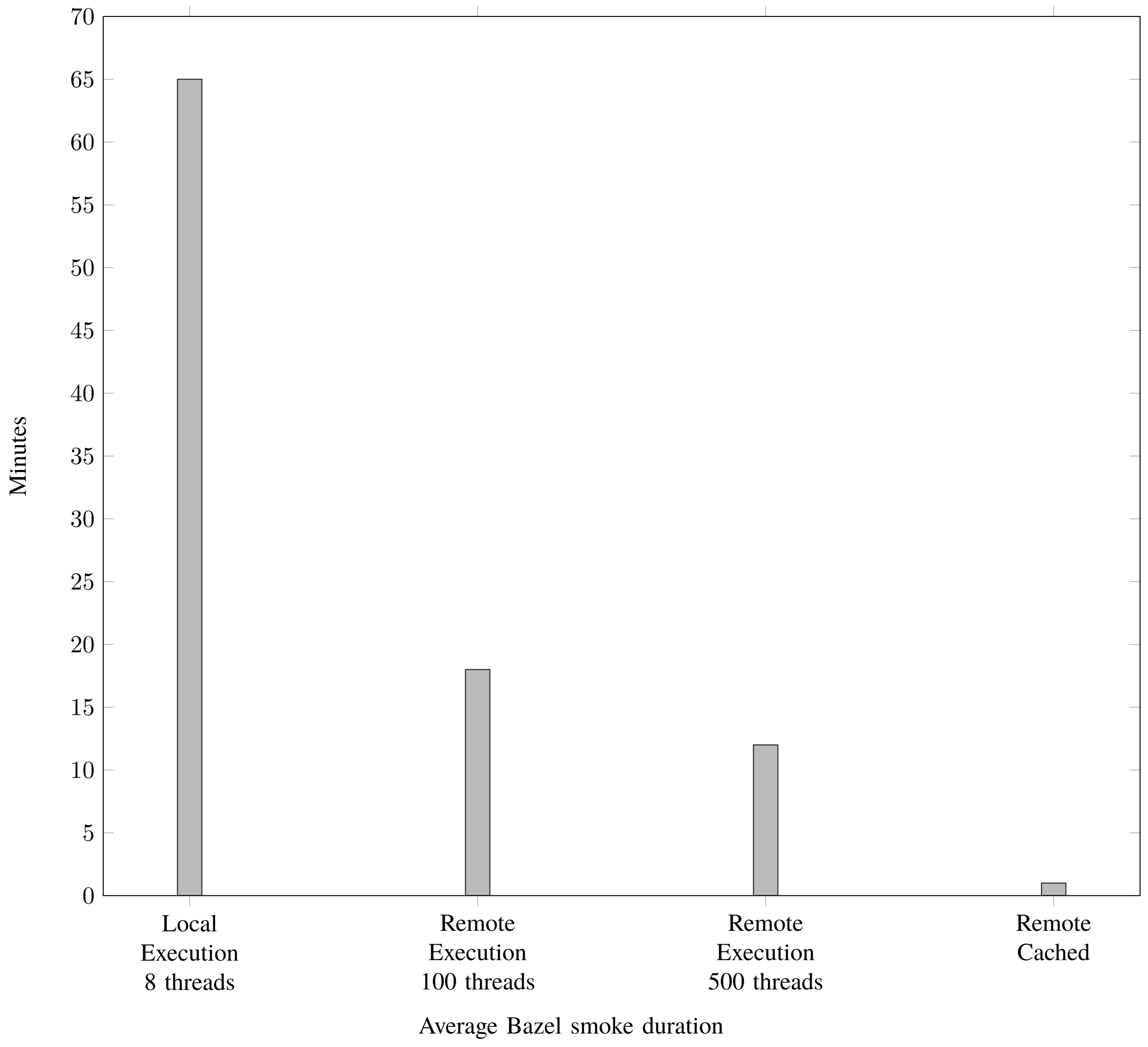
The above example shows a simple monolithic compile with verilator and run. However the individual actions of the build, such as `verilator_cc_library`, can be completely user defined. For example, users could define rules for partition compile, multi-step compile, synthesis, emulation, and more. Additional rules would simply become more nodes in the build graph above.

VIII. Deploying Bazel

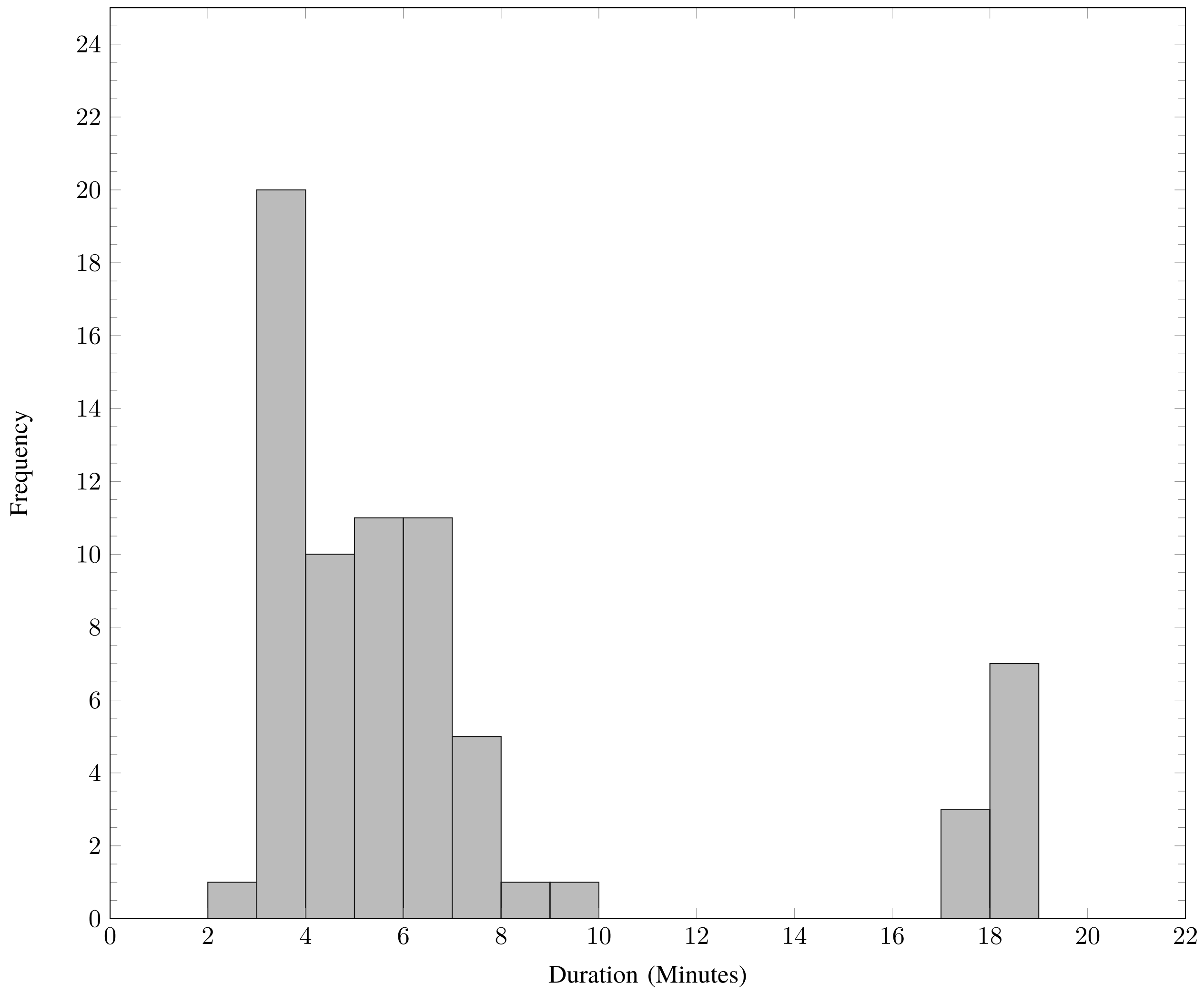
At Tenstorrent we use Bazel for our RTL generation, RTL compilation, test generation, and simulation regressions. All of these actions benefit from remote caching and remote execution.

Tenstorrent has a CI system that runs smoke regularly, so this can be used to measure the effectiveness of remote caching and remote execution.

The following figure shows the durations of running smoke under different configurations in our CI. With no remote caching or execution, and 8 local threads of execution, a smoke run from scratch takes more than an hour. Using 100 remote workers brings the time down to under 20 minutes. Our Verilator compilation results in hundreds of generated C++ files that need to be compiled, and massively benefit from parallelization. If the remote cache is 100% hit, the CI job only takes 3 minutes, with most of that overhead coming from setting up the directory and cloning the repo and dependencies.



The following figure shows the distribution of our CI runs during one typical day. There were 71 total runs. With no remote caching, each run would on average take 18 minutes, for a total of 1278 minutes of CI run duration. However because of remote caching, the total CI time is 472 minutes. That's more than 13 hours of CI time saved in just one day.



Bazel smoke durations in one day's worth of CI runs

Builds that hit 100% in the remote cache are usually because the exact code was already run either locally by a user or in some other branch on the CI and uploaded to the remote cache. Or the change may have been to documentation or other sources that do not affect smoke. Builds that partially hit in the remote cache are usually from changes to a particular unit or testbench. Because of Bazel's dependency checking, a change to one unit does not require parallel units to be recompiled or rerun.

IX. Conclusion

We showed how a modern software build system like Bazel can be beneficial for a complex hardware design project. Going back to the initial typical RTL verification workflow, we annotate how the steps are augmented by Bazel -

1) Checking out a design and external dependencies
   - Only dependencies that are needed are downloaded on demand, saving download time and space for a large design that has many IP and tool dependencies.
2) Making a change
3) Building simulation models
   - Builds are done in a sandbox, allowing immediate feedback if a build dependency is misspecified.
   - Compile times are reduced by only doing compiles that need to be done, and leveraging a remote cache of previously finished builds by others.

    - Compiles can be paralellized on an existing compure farm, down to the granularity of a single C++ file compile.
4) Running a suite of tests
    - Simulation runs can also leverage the remote cache and download the full simulation results, including logs and any other requirements.
5) Pushing that change to other users
    - We found CI test times go from about 20 minutes down to as little as 3 minutes using the above benefits.

References

[1] ”Bazel,” Bazel. https://bazel.build/ (accessed Oct. 13, 2022).
[2] ”Who’s Using Bazel,” Bazel. https://bazel.build/community/users (accessed Oct. 13, 2022).
[3] ”Rules,” Bazel. https://bazel.build/rules (accessed Oct. 13, 2022).
[4] ”JavaScript rules for Bazel,” GitHub, Oct. 12, 2022. https://github.com/bazelbuild/rules_nodejs#user-testimonials (accessed Oct. 13, 2022).
[5] ”bazel_rules_hdl,” GitHub, Sep. 27, 2022. https://github.com/hdl/bazel_rules_hdl (accessed Oct. 13, 2022).
[6] K. Kiningham, “Bazel build rules for Verilator,” GitHub, Aug. 31, 2022. https://github.com/kkiningh/rules_verilator (accessed Oct. 13, 2022).
[7] ”Make - GNU Project - Free Software Foundation,” www.gnu.org. https://www.gnu.org/software/make (accessed Oct. 13, 2022).
[8] ”remote-apis/build at main · bazelbuild/remote-apis,” GitHub. https://github.com/bazelbuild/remote-apis/blob/main/build/ (accessed Oct. 13, 2022).
[9] ”Remote Execution Services,” Bazel. https://bazel.build/community/remote-execution-services/ (accessed Oct. 13, 2022).
[10] ”distcc: a fast, free distributed C/C++ compiler,” www.distcc.org. https://www.distcc.org/ (accessed Oct. 13, 2022).
[11] ”cache — Compiler cache,” ccache.dev. https://ccache.dev/ (accessed Oct. 13, 2022).
[12] ”IBM Spectrum LSF Suites - Overview,” www.ibm.com. https://www.ibm.com/products/hpc-workload-management (accessed Oct. 13, 2022).
[13] ”Bazel Buildfarm,” GitHub, Oct. 26, 2022. https://github.com/bazelbuild/bazel-buildfarm (accessed Oct. 28, 2022).
[14] ”Command-Line Reference,” Bazel. https://bazel.build/reference/command-line-reference#flag–remote_cache (accessed Oct. 28, 2022).
[15] ”Command-Line Reference,” Bazel. https://bazel.build/reference/command-line-reference#flag–remote_execution (accessed Oct. 28, 2022).