Enabling Coverage-Based Verification in Chisel

Andrew Dobis, Hans Jakob Damsgaard, Enrico Tolotto,
Kasper Hesse, Tjark Petersen, Martin Schoeberl

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Lyngby, Denmark

andrew.dobis@inf.ethz.ch, hans.damsgaard@tuni.fi, {s190057, s183735, s186083}@student.dtu.dk, masca@dtu.dk

2022 IEEE European Test Symposium (ETS) | 978-1-6654-6706-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ETS54262.2022.9810435

***Abstract*—Ever-increasing performance demands are pushing hardware designers towards designing domain-specific accelerators. This has created a demand for improving the overall efficiency of the hardware design and verification cycles. The design efficiency was improved with the introduction of Chisel. However, verification efficiency has yet to be tackled. One method that can increase verification efficiency is the use of various types of coverage measures. In this paper, we present our open-source, coverage-related verification tools targeting digital designs described in Chisel. Specifically, we have created a new method allowing for statement coverage at an intermediate representation of Chisel, and several methods for gathering functional coverage directly on a Chisel description.**

***Index Terms*—Hardware Verification, Statement Coverage, Functional Coverage, Chisel, Scala.**

I. Introduction

As time passes, contemporary hardware design is met with tighter development and verification time-constraints. Additionally, hardware designers are turning to domain-specific accelerators in order to keep up with the ever-increasing performance demands [1]. This means that more and more hardware must be designed from scratch in ever shortening time periods [2]. To help meet these demands, researchers at the University of California in Berkeley proposed Chisel [3], a Scala embedded high-level hardware construction language.

This solution is powerful, but is lacking verification functionalities. One of the main tools needed for the verification of digital systems is coverage. Coverage allows verification engineers to measure their progress throughout the testing process and have an idea of how effective their tests actually are. Coverage can be separated into multiple distinct categories, but we will focus on the following two: statement and functional coverage. Statement coverage defines a measure for *“how many code statements have been tested?”*, whereas functional coverage gives a measure for *“which functionalities have been tested?”* [4].

In this paper, we explore how to use existing tools, both for Scala and for Verilog, in order to obtain statement coverage in Chisel. We will start by presenting code coverage at the Scala level. Second, we will briefly discuss how to use Verilator [5] in order to enable statement coverage of the generated Verilog description. Third, we will show our solution for getting statement coverage of the so-called Flexible Internal Representation of Register Transfer Level (FIRRTL) intermediate representation [6]. In total, these measures represent test coverage of different language levels of a Chisel design.

Next, will present our solution for gathering functional coverage of a Chisel design directly in Scala. And finally, we evaluate our coverage solutions with example use cases to illustrate its efficiency. The solutions are shown to reduce the amount of code (measured in lines of code) needed to gather functional coverage on a Chisel design in comparison to using SystemVerilog (SV) extended by Universal Verification Methodology (UVM) [7].

The contributions of this paper are a method for extending FIRRTL execution engines for gathering statement coverage, as well as novel tools and methods for defining and gathering functional coverage in a high-level hardware construction language such as Chisel.

The paper is organized as follows. The following section presents related work. Section III presents background on Chisel and various coverage methodologies. Section IV discusses how existing tools can be used to gather statement coverage of a Chisel design. Section V presents our work on adding statement coverage at the FIRRTL level. Section VI then presents our Scala-based functional coverage framework. In section VII, we evaluate our solution by comparing to similar tests written in SV with UVM. Section VIII concludes.

II. Related Work

SV, a mostly non-synthesizable extension of the Verilog Hardware Description Language (HDL), contains certain constructs capable of gathering coverage information [4]. These can be used when working with the Verilog description generated by Chisel. Specifically, Chisel 3.4.0’s experimental `cover` statement maps directly to its SV equivalent [8], but is yet to be supported by FIRRTL execution engines like Treadle[1]. Our solution differs from this approach, since we use pre-existing language features from Chisel/Scala in order to enable coverage in the simplest possible manner. Additionally, our functional coverage solution allows for greater customizability in terms of the behavior captured by the different cover

[1]Available at https://github.com/chipsalliance/treadle

978-1-6654-6706-3/22/$31.00 ©2022 IEEE

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.

constructs, while Chisel's `cover` statement is limited by a simple predicate evaluated on sampling.

A possible solution to obtaining statement coverage for Chisel, is to rely on the generated Verilog code. As part of the Chisel development tools, `ChiselTest` allows a Chisel design to be simulated with the open-source Verilator [5] back-end by using the generated Verilog description. Some coverage metrics can be enabled using a few simulation parameters in Verilator [9]. The simulator then generates a coverage report in a SystemPerl coverage report file [10].

SV is widely used in the scope of UVM [7]. In UVM, verification engineers can define verification plans using `Bins` describing ranges of values to test for, `CoverPoints` defining which ports to sample, and `CoverGroups` comprising `CoverPoints` that are to be sampled simultaneously. UVM also supports cross coverage, a special kind of `CoverPoint` where a hit is only considered when two or more `CoverPoints` have specific values at the same time. Coverage is sampled whenever the `sample` method is called on a `CoverGroup`. In contrast to this, our solution offers tools which allow for a more customizable verification plan, enabling a definition that closely models the target specification.

In relation to timed coverage, Property Specification Language (PSL) and the similar SystemVerilog Assertions (SVA) [11] are two current solutions allowing for the use of temporal logic in relation to both coverage and assertions. PSL for example offers a wide variety of Sequential-Extended Regular Expressions (SEREs), which define temporal relations, taken from Linear-Temporal Logic (LTL), between different boolean expressions. This language also has extensions adding other functionalities from LTL, such as the `past` [12] operator. These solutions are both quite complex and require the use of many operators to describe potentially simple temporal relations. Rather than relying solely on temporal operators to express a form of LTL, our solution aims to provide a simplified set of temporal constructs that encompass a similar range of relations when used in conjunction with different types of bins.

We defend the idea of creating verification methods in Scala (in open-source), rather than relying on an external language, since it improves the overall cohesion of the Chisel/Scala ecosystem. Following that same idea, we briefly mention the work conducted in parallel to the research presented in this paper on constructing a hardware verification library for Chisel entirely in Scala, namely `ChiselVerify` [13].

III. Background

We begin by presenting a brief overview of Chisel and FIRRTL. After that, we will give an introduction to statement and functional coverage.

A. Chisel

Chisel is a hardware construction language embedded in the functional and object-oriented programming language Scala [3], [14], [15]. A Chisel design generates a Verilog description that can then be synthesized using existing tooling. Chisel syntax is rooted in Scala. As a result, it enables the description of hardware in a high-level manner. Scala also allows for both functional and object-oriented programming constructs, which makes it possible to organize a design very intuitively using Scala classes and objects, and to use the power of functions as first-order objects to simplify descriptions thanks to constructs like *mappings* or *reductions*.

B. FIRRTL

In a Chisel design, the source code is first compiled into an intermediate representation named FIRRTL [6]. FIRRTL is used as a sort of "optimization layer" before being converted into the final Verilog form. During this optimization process the original Chisel description goes through three different intermediate representation layers:

- High-FIRRTL, which is a form that maps perfectly back to Chisel, but with the FIRRTL structure.
- Mid-FIRRTL, which is a form where abstract constructs are simplified, i.e., loops are unrolled and arrays are flattened.
- Low-FIRRTL, which maps to RTL code with high-level conditional statements turned into multiplexers.

Throughout this optimization process, custom FIRRTL compiler passes, known as *Transforms*, can be used to modify the design. This is often done when trying to apply simplifications to the design to make the generated hardware more optimal [6].

Figure 1 shows an overview of the Chisel compilation pipeline and how our coverage methods incorporate into it. More specifically, it shows the different steps taken before the coverage FIRRTL pass can be run, and shows where the functional coverage tools fit in. For the sake of clarity, the synthesis part of the pipeline has been omitted.

C. Test Coverage

In software development, test coverage is used as a metric to measure the completeness of a testing suite. In recent years, these techniques, originally used for software, have been brought into the hardware verification universe. In this paper, we will mostly focus on two key approaches to test coverage: statement coverage and functional coverage.

a) Statement Coverage: Statement coverage measures which percentage of statements in our code has been executed during the test. It is widely used in the software world and may be ported directly to the hardware world. However, despite being simple to use, statement coverage alone does not suffice as it only provides information about which code statements were tested, but not how well they were tested [16].

b) Functional Coverage: If we instead are looking to measure the completeness of a test suite, we need to ask the following question: *"are we even testing the right thing?"*. In the hardware world, engineers usually implement designs based on pre-determined specifications, so when testing, we should also have a metric for how well we are implementing the specification. Functional coverage is that metric. The idea is to first define what is called a *verification plan* [4], which represents the specification we are implementing. A good verification plan tests only the required functionality of

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.

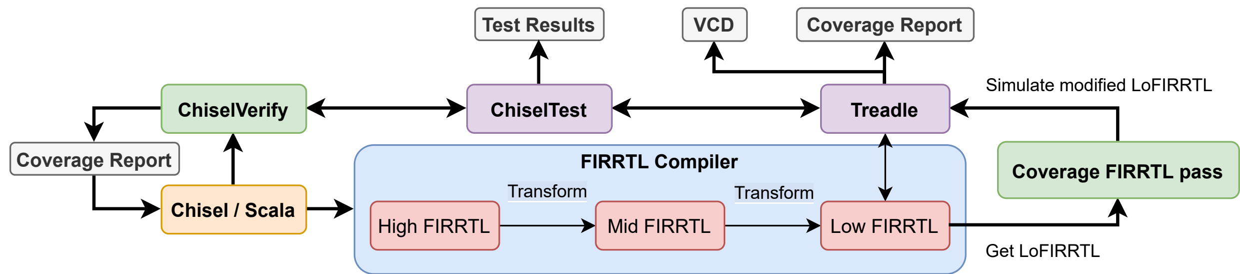

Fig. 1. Overview of the Chisel pipeline. Chisel code is first given to the FIRRTL compiler, where it is transformed by a series of "lowering passes". It is then output by the compiler in a low-level form, where only hardware primitives are used without loops. Our Functional Coverage library works as an intermediate between the Chisel source and ChiselTest. The optional custom FIRRTL pass, modifies the loFIRRTL code being fed into Treadle.

a device under test (DUT) by applying only realistic input patterns to it. Once a plan is defined, we sample the different points defined in the specification during the testing process to obtain results in the form of: *"test suite T covered a total of x% of the values specified by point P in specification A"*.

D. What to Aim For

When verifying a design, the greatest pitfall is the idea of needing to achieve 100 % statement coverage. Indeed, reaching 100 % statement coverage is often either unreasonable due to computational requirements or impossible due to statements not meant to be executed. One should rather aim for fully exercising the most important features of one's design; thus, targeting high functional coverage. Statement coverage can still be useful, however, as it indicates whether or not sections of code are executed.

Additionally, it is crucial to remember that coverage metrics only indicate the thoroughness of a verification suite, and not its correctness or completeness. Designing a good verification plan is therefore of the utmost importance [16].

E. Coverage in Chisel

The most common way to test a design implemented in Chisel is by using `ChiselTest` [17]. It is a testing framework for Chisel that gives access to a simple `peek`, `poke` and `expect` testing interface. `ChiselTest` also helps the user create golden models, to test their design against, entirely in Scala or by using the Java Native Interface (JNI) with already defined models written in, e.g., C. This is further supported by a simple `fork` functionality for concurrency. This framework, however, is lacking in verification functionalities and does not give the user any way to gather coverage on a Chisel design. Hence, if one wants to gather test coverage on a current Chisel design, they need to rely on basic Scala software coverage tools (which we explore in more detail in the next section). We are, therefore, introducing coverage solutions that are specifically tailored for Chisel.

IV. Applying an Existing Tool

We will start with measuring coverage at a high level (i.e., coverage of the Scala/Chisel code itself). Scala and its corresponding software testing framework ScalaTest have some coverage functionalities built-in, specifically, we apply the Scoverage plugin [18]. Scoverage supports statement coverage and should work with Chisel [19].

To enable coverage, the Scoverage plugin must be added to the Scala project, including optional arguments, defined in the *plugins.sbt* file. Subsequently running the test and coverage reporter tool will create a graphical coverage report in a set of HTML files. The report may optionally include the original source files with statements that were executed during the test marked in green.

While this sounds promising, experimenting with Scoverage in combination with Chisel revealed some limitations. In particular, some of the basic constructs introduced by Chisel, e.g. `switch` statements, are simply not considered by the coverage tool and thus, incomplete results are reported. In practice, this makes Scoverage insufficient for hardware verification. However, it does provide coverage results for the hardware generators, which can also be important to consider for verification or optimization purposes by pointing the designer's attention toward unused generator code.

V. Statement Coverage at the FIRRTL Level

Statement coverage at the FIRRTL level is interesting for two reasons: first, FIRRTL is intuitively mapped back to the Chisel code, and second, it contains only expanded hardware code meaning that we obtain coverage information on the generated hardware rather than the hardware generators.

To enable it, we developed a method allowing for coverage measurements in Treadle, a FIRRTL execution engine used to simulate Chisel designs with the `ChiselTest` framework. We do so by running tests through an extended version of Treadle (available as a SNAPSHOT at the time of writing).

We base this on Baxter's method for collecting statement coverage in arbitrary languages [20], which we adapt for HDLs. The idea is to extend each multiplexer of a design with additional outputs, known as *coverage validators*, which are set depending on the paths taken by each multiplexer using *coverage expressions*. During testing, the ports are sampled and used for checking whether a certain multiplexer path was taken. This results in a particular branch coverage percentage which can be mapped to a statement coverage percentage.

We implement this by simply extending each multiplexer in the expanded FIRRTL code by adding the required extra ports. As such, our implementation is based on a custom pass of the FIRRTL compiler that traverses the source's abstract syntax tree and adds the additional outputs and coverage expressions

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.

as needed. During a test, the outputs are sampled on each call to the `expect` method. Finally, the results are compiled to a Scala `case class` containing the multiplexer path coverage percentage, the coverage validator lines that were covered by a test, and the modified LoFIRRTL code as a `List[String]`. The case class may be serialized into a coverage report. The report contains annotated LoFIRRTL code lines marked by "+" if they have been executed at least once during the tests and "-" otherwise. A component's input/output ports are, by default, considered executed. Hence, "-" can only appear on lines containing a coverage validator as it represents a non-explored multiplexer path. Consider for example the following report of a simple circuit in which only the path for which io_a is 1 has been tested:

```
COVERAGE: 50.0% of multiplexer paths tested
COVERAGE REPORT:
+ circuit Test_1 :
+   module Test_1 :
+     input io_a : UInt<1>
+     input io_b_0 : UInt<2>
+     input io_b_1 : UInt<2>
+     input clock : Clock
+     output io_cov_valid_0 : UInt<1>
+     output io_cov_valid_1 : UInt<1>
+     output out : UInt<2>
+     io_cov_valid_0 <= io_a
-     io_cov_valid_1 <= not(io_a)
+     out <= mux(io_a, io_b_0, io_b_1)
```

We see that a single coverage validator line isn't covered. Thus, our simple test did not cover the path where `out` took the value io_b_1.

VI. Functional Coverage in Chisel

As described previously, statement coverage is not sufficient for verifying complex hardware. As such, a tool for hardware verification would not be complete without constructs allowing one to define a verification plan and retrieve a functional coverage report from it.

A verification plan describes which ports should be sampled and kept track of in the coverage report. In SV and UVM, verification plans are based on three elements: `Bins`, `CoverPoints` and `CoverGroups`. We take a similar approach but extend upon their functionality. As such, we need to be able to define a verification plan, sample DUT ports, keep track of the number of hits in each bin, and compile the results into a comprehensible report.

We implemented this on top of the existing `ChiselTest` [17] framework with a top-level `CoverageReporter` class whose `register` method is used to define verification plans. Internally, the coverage reporter stores `CoverPoint` to `Bin` mappings inside a `CoverageDB`. Once defined, the `sample` method (based on `ChiselTest`'s `peek` method) is used to update bins. After a test, a coverage report can be generated using the `printReport` method. Our solution extends existing tools with special coverage constructs that allow the user to define relations between ports.

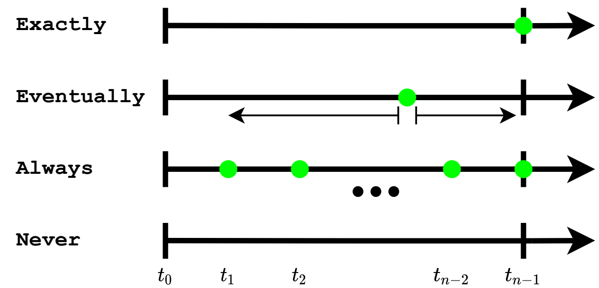

Fig. 2. Illustration of the four delay types supported by our coverage tools.

A. Cross Coverage

Basic coverage relations between multiple ports can be defined using a `CrossPoint`. This allows one to associate a set of ports to a set of ranges of equal size. A sample of the ports is considered a hit if all of the port values are contained in their associated ranges. This is particularly useful for checking whether all interesting combinations of values on a set of ports were tested [16]. An example being the injection of valid and corrupt packets on all ports of a NoC router.

B. Timed Coverage

Timed coverage relations in our solution aim to provide a simple and accessible form of temporal logic related to functional coverage. Adding a temporal argument to our `cover` construct allows us to define relations between ports sampled in different cycles. The user specifies the expected delay and temporal construct to use. For this, we provide the following four basic operators, which we call delay types:

- `Exactly`, a hit will only be considered if the second point is sampled in its range at exactly the given number of cycles after the first point was.
- `Eventually`, a hit will be considered if the second point is sampled in its range at any point within the given number of cycles after the first point was.
- `Always`, a hit will be considered if the second point is sampled in its range during every cycle for a given number of cycles after the first point was.
- `Never`, a hit will be considered if the second point is never sampled within its range during every cycle for a given number of cycles after the first point was.

The delay types are illustrated in Figure 2. Green dots indicate when the second point must be sampled in its range for it to be considered a hit.

We find that using these four simple operators applied on different bins, one can express complex timing relations. For example, an event happening at some point between 5 and 10 cycles after a given point can be expressed using a conditional bin with a `Never(5)` delay, combined with a second bin using the same condition and a `Eventually(10)` delay.

C. Conditional Coverage

Working in Scala allows us to use functions as higher-order objects. We exploit this and provide a `CoverCondition` construct which may be used to gather coverage information on an arbitrary number of ports with a user-defined predicate.

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.

A sample is considered a hit if the predicate evaluates to `true`. Since we are working with an arbitrary number of ports, computing the set of possible value combinations in order to obtain a coverage percentage is computationally expensive. To alleviate said problem, we added an optional `expectedHits` parameter that lets the user define a number of expected hits to generate a coverage percentage. If passed, the coverage report will include a hit percentage for the particular `CoverPoint`. This coverage information can be very useful when wanting to create points that cover sparse ranges. We also provide this predication functionality for regular bins to support non-trivial ranges, e.g., covering only odd numbers in the range $[0, 100)$ with condition { `case Seq(x) => x % 2 != 0` }.

D. Example Verification Plan and Coverage Report

Using the aforementioned elements, users can define accurate representations of their specifications in the form of verification plans. The following listing shows an example verification plan utilizing some of the tools to verify a DUT denoted `dut`:

```
1 val cr = new CoverageReporter(dut)
2 cr.register(
3   cover("accu", dut.io.accu)(
4     bin("lo10", 0 until 10),
5     bin("First100odd", 0 until 100,
6       { case Seq(x) => x % 2 != 0 })),
7   cover("aAndB", dut.io.outA, dut.io.outB)(
8     bin("asuptobAtLeast100",
9       condition = { case Seq(a, b) => a > b },
10      expectedHits = 100),),
11  cover("accuAndTest", dut.io.accu, dut.io.test)(
12    cross("both1", Seq(1 to 1, 1 to 1))),
13  cover("timedAB", dut.io.outA,
        dut.io.count)(Exactly(3))(
14    cross("ExactlyBoth3", Seq(3 to 3, 3 to 3))))
```

During the subsequent tests, we must call `cr.sample()` whenever we wish to sample the defined `CoverPoints`. Once our tests are done, we can ask for a coverage report by calling `cr.printReport()`, which results in the following:

```
============== COVERAGE REPORT ==============
COVER_POINT PORT NAME: accu
BIN lo10 COVERING 0 until 10 HAS 8 HIT(S) = 80%
BIN First100odd COVERING 0 until 100
WITH CONDITION onlyOdd HAS 1 HIT(S) = 1,00%
============================================
COVER_CONDITION NAME: aAndB
CONDITION asuptobAtLeast100 HAS 5 HITS
EXPECTED 100 = 5.0%
============================================
CROSS_POINT accuAndTest FOR POINTS accu AND test
BIN both1 COVERING 1 to 1 CROSS 1 to 1
HAS 1 HIT(S) = 100%
============================================
CROSS_POINT timedAB WITH AN EXACT DELAY OF 3
BIN ExactlyBoth3 COVERING 3 to 3 CROSS 3 to 3
HAS 1 HIT(S) = 100,00%
============================================
```

The above report shows each point with its user-defined name and the bins it contains. These are associated to a number of hits and, whenever possible (as described in section VI-C), a coverage percentage. Another option would be, for example if we want to do automated constraint modifications depending on coverage results, to generate the coverage report as a Scala `case class` and then to use its `binNcases` method to get numerical and reusable coverage results.

VII. Evaluation

To evaluate our new coverage tools, that we include in our `ChiselVerify` package, we will verify both an ALU accumulator from the Leros processor [21] and an arbiter circuit, both implemented using Chisel. Note that we will only focus on the proposed functional coverage tools, since statement coverage was added directly into Treadle and requires additional work from the verification engineer. This will be done by comparing the verbosity of our Scala-based coverage solutions to that of the same verification done with UVM. The verbosity is measured in "verification lines per source lines of code", a metric used in other works to partially evaluate verification languages [22], [23]. We consider a verification line to be any explicit declaration of a cover or bin construct, as well as any function call or standard statement, and have formatted our listings accordingly. In order to verify our DUT with UVM, we work with the verilog description generated by Chisel, since our goal is to enable verification of Chisel designs, in order to have it be compatible with SV. The ALU accumulator supports operations such as `add`, `load`, `shiftRight` and logic operations. In order to test this entirely, we need to try all operations on the `op` input and all input value corner cases on the `din` input. A good verification plan would thus capture what we just described.

```
1 val cr = new CoverageReporter(dut)
2 cr.register(
3   cover("ops", dut.io.op)(DefaultBin(dut.io.op)),
4   cover("dataInputs", dut.io.din)(
5     bin("minValue", Integer.MIN_VALUE),
6     bin("neg1", -1),
7     bin("zero", 0),
8     bin("one", 1),
9     bin("maxValue", Integer.MAX_VALUE)))
```

In the above code snippet, we registered a `CoverGroup` that ensures that the edge cases of all operations are tested. Here, the implicit `DefaultRange` and `DefaultBin` functions are used to cover all possible values for the `dut.io.op` port. All that is left now is to call `cr.sample()` once per cycle in our `ChiselTest` test class.

In order to do the same with UVM, we need to create a UVM-subscriber based coverage class, instantiate the current DUT, declare the verification plan, define `build_phase` and `write` functions, and define the coverage class constructor. The UVM subscriber class must then be used inside of a whole UVM testbench, which itself contains many other classes and constructs.

Our second DUT has the interesting aspect of taking a Vector of `DecoupledIO` elements as an input. These expand to 3 separate flattened arrays in the generated verilog, which will make the declaration of the verification plan more complex. In order to conduct coherent comparisons, which do not depend

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.

on the size of the input `Vector`, we have mapped these back to 3 separate arrays. To verify this, we register cover constructs for both the output and each input's ready, valid, and bits signals. With our solution, this can be done using a `foreach` call on the input vector in which we simply register the cover constructs we want. However, in SV, this is done by creating three generic `covergroups`, each containing a single `coverpoint`, which are then initialized in three separate foreach loops.

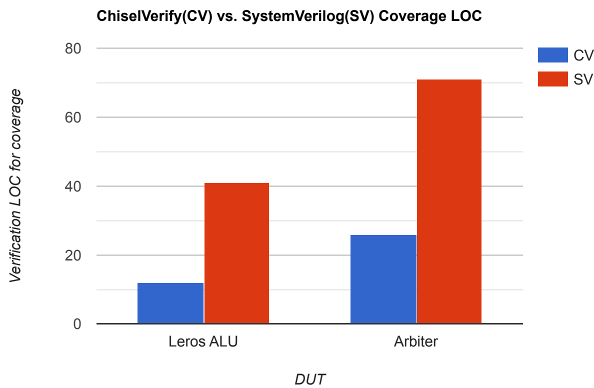

Fig. 3. A comparison between the lines of coverage-specific verification code needed to gather the same coverage data using `ChiselVerify` and SV with UVM. This considered code is publicly available in the example tests in the chiselverify public repository.

Figure 3 plots the results of our verifications using our solution and UVM. We only consider coverage-specific code, meaning that only the `uvm_subscriber` class is taken into account in our UVM verification. These results clearly show that our solution requires, on average, 30% of the amount of verification code required by UVM, in order to describe the same verification plan and obtain the same coverage results. This is largely due to our tool automating many of the structural code that must be handwritten in UVM. On top of that, our solution enables the verification engineer to work directly with the Chisel description, instead of the generated Verilog, allowing for the benefits of Chisel to be fully reaped. This shows that our work allows for the efficient definition of coverage structures in a manner that is more concise, and more adapted to Chisel, than existing coverage solutions.

VIII. Conclusion

In this paper, we described the lack of coverage-oriented verification tools for Chisel designs and showed that existing Scala tools do not have acceptable coverage features. To address this, we implemented extensions to the Treadle simulator and the `ChiselTest` framework to enable statement and functional coverage measurements. Furthermore, we showed that the implemented framework enables less verbose and more advanced verification suites than what is possible using existing tools. These features include `Bins` defined by an arbitrary predicate, and temporal logic enabled `cover` constructs.

Acknowledgements

We would like to thank the researchers at Berkeley who are constantly working on making Chisel a more ubiquitous hardware construction language and for the feedback related to the work done on adding coverage to Treadle.

Source Access

The source code for the coverage package is available at https://github.com/chiselverify/chiselverify.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
[2] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020.
[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *The 49th Annual Design Automation Conference (DAC 2012)*. San Francisco, CA, USA: ACM, June 2012, pp. 1216–1225.
[4] C. Spear, *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.
[5] Veripool, "Verilator," https://www.veripool.org/wiki/verilator.
[6] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 209–216.
[7] A. S. I. (Accellera), "Universal Verification Methodology (UVM) 1.2 user's guide," https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf, 2015.
[8] T. Alcorn, "Basic model checking api for chisel," https://github.com/chipsalliance/chisel3/pull/1499, 2020.
[9] E. Tolotto, "Verification of Digital Designs with Chisel," https://github.com/chiselverify/documentation/tree/master/enrico-thesis, 2020.
[10] Veripool, "Systemperl," https://www.veripool.org/projects/systemperl/wiki/Manual-systemperl, 2020.
[11] *PSL and SVA Assertion Languages*. Dordrecht: Springer Netherlands, 2008, pp. 55–82. [Online]. Available: https://doi.org/10.1007/978-1-4020-8586-4_4
[12] C. Dax, F. Klaedtke, and M. Lange, "On regular temporal logics with past," *Acta Informatica*, vol. 47, no. 4, pp. 251–277, 2010. [Online]. Available: https://doi.org/10.1007/s00236-010-0118-3
[13] A. Dobis, T. Petersen, H. J. Damsgaard, K. J. Hesse Rasmussen, E. Tolotto, S. T. Andersen, R. Lin, and M. Schoeberl, "Chiselverify: An open-source hardware verification library for chisel and scala," in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2021, pp. 1–7.
[14] M. Schoeberl, *Digital Design with Chisel*. Kindle Direct Publishing, 2019, available at https://github.com/schoeberl/chisel-book.
[15] B. Venners, L. Spoon, and M. Odersky, *Programming in Scala, 3rd Edition*. Artima Inc, 2016.
[16] J. Bergeron, *Writing Testbenches - Functional Verification of HDL Models*, 2nd ed. Springer, 2003.
[17] R. Lin. ChiselTest. https://github.com/ucb-bar/chisel-testers2.
[18] S. S. et al., "sbt-scoverage," https://github.com/scoverage/sbt-scoverage, 2020.
[19] M. Wachs, "Test coverage," https://www.chisel-lang.org/chisel3/docs/developers/test-coverage.html, 2021.
[20] I. D. Baxter, "Branch coverage for arbitrary languages made easy," http://www.semdesigns.com/Company/Publications/TestCoverage.pdf, Austin, Texas, 78579 USA, 2002.
[21] M. Schoeberl and M. Petersen, "Leros: The return of the accumulator machine," in *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, M. Schoeberl, T. Pionteck, S. Uhrig, J. Brehm, and C. Hochberger, Eds. Springer, May 2019, pp. 115–127.
[22] M. Eilers and P. Müller, "Nagini: A static verifier for python," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 596–603.
[23] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

Authorized licensed use limited to: Princeton University. Downloaded on May 09,2024 at 23:51:48 UTC from IEEE Xplore. Restrictions apply.