UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DEDUPLICATING REPEATED LOGIC TO ACCELERATE SIMULATION

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Thomas J. Nijssen

June 2021

The Thesis of Thomas J. Nijssen is approved:

Professor Scott Beamer, chair

Professor Tyler Sorensen

Professor Jose Renau

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Deduplicating Repeated Logic to Accelerate Simulation

by

Thomas J. Nijssen

As time goes on, hardware designs get ever more complicated, and producing designs that work properly on time is crucial. An essential part of the design process is debugging by way of a digital logic simulation. Frustratingly, simulation feels very slow, taking hours of real time to simulate just a few seconds of a sufficiently complex design. Several solutions exist in this space, most of them commercial, but also some free and open source options. ESSENT is a relatively novel simulator that improves simulation performance by cleverly partitioning the design and then re-computing only the changed parts of the design. However, an important shortcoming is that all repeated hardware components are inlined, leading to bloated code sizes since there is effectively no longer any reuse. This work describes an improvement to the tool that deduplicates the largest and most frequently reused components to achieve a speedup of up to 4.7× compared to the previous version of ESSENT, and up to 15× compared to Verilator, greatly reducing the time needed to perform a simulation and improving designer productivity.

Acknowledgements

Behind every successful engineer stand numerous people that mentor, inspire, guide, and shape. I am most grateful for their impact and encouragement throughout this exciting and most transformative chapter of my life.

First, I would like to express my gratitude to Dr. Scott Beamer. I joined Scott's lab without an exact idea of what I wanted to do, but I knew that it would lie at the intersection of some of our mutual interests: hardware design, computer architecture, and the never-ending quest to make everything go faster. With open arms he welcomed me, and we quickly identified this project as something that excites us both. Scott, I thank you for the patience and direction you showed me, even when the path ahead was not clear. I admire your passion for the field and selfless dedication to your students' well-being and progress.

I would also like to extend my gratefulness to my committee members, Dr. Jose Renau and Dr. Tyler Sorensen. Their quick feedback on my thesis as I approached the deadline was invaluable to my success, and their diverse backgrounds gave me unique perspectives on what I had written. Without your support and encouragement, I would not have been able to do this.

Finally, I would like to give thanks to the FIRRTL and Scala communities, especially to the kind people in the various chat rooms who were happy to answer my questions and point me in the right direction when it came to using a library or understanding some corner of their code. Also, my thanks go out to the Free Chips Project, and the open source hardware communities. This entire project was made

possible by the openness and freedom granted to everyone to push the boundaries of what we know to be possible with hardware design.

1 Introduction

When designing digital hardware circuits, an indispensable part of the development process is testing to ensure that the design works as intended. Simulation has long been used to test hardware designs, although for a long time the paradigms and approaches have remained stagnant. Traditionally mired in expensive commercial software packages, logic design and simulation has not been easily accessible to hobbyists and academics as long for as software development has. Only in 1998 did the now-popular Verilator and Icarus Verilog tools become freely available to simulate synthesizable Verilog.

Verilog originally began its life as a language to write simulations only [6] (hence the "Veri-" in the name"), and was later co-opted for use as a language to also express the design as well. Along with VHDL, most commercial digital logic design is expressed in these venerable yet ancient languages; unfortunately, due to the complexity of these languages, and many organizations being slow to upgrade their tooling to newer versions supporting the latest versions of the Verilog and VHDL standards, logic design has developed a reputation of being tedious and arcane, since those languages do not offer the same flexibility that popular modern programming languages afford. However, an exciting trend in the open-source hardware community is the rising emergence of alternative and agile hardware languages that allow a designer to design and iterate quickly and comfortably. One example of such a language is Chisel [3], a Scala Domain-Specific Language providing an agile development environment allowing a designer to express their design in an object-oriented way while not falling into the trap of making the abstractions too high-level. The language is easily extensible, allowing for many developer tools to be easily created and integrated with minimal effort.

The Flexible Intermediate Representation for RTL (FIRRTL) [8] is an intermediate language to express an elaborated design, which can be output from many other higher-level tools and design languages. Coupled with the powerful eponymous Scala library, it becomes easy to write programs to apply transformations to a hardware design. However, the main benefit in the context of this work is that it provides a simple, unambiguous representation of the circuit; a Verilog representation of the same circuit (commonly used by other tools even as an intermediate representation) could potentially require the use of a preprocessor, or a parser supporting some non-standard syntax, in addition to being much more complex. The AST representation of FIRRTL is convenient to operate on, and the Scala library conveniently exposes all details of the design in an object-oriented fashion. Chisel, as well as several other emerging HDLs use FIRRTL as their intermediate representation.

ESSENT [5] is a novel hybrid simulator which combines the full-cycle approach with clever ahead-of-time partitioning and scheduling to allow parts of the design whose state has not been mutated to be skipped from evaluation. The design is first flattened, removing all hierarchy from the design, and simply inlines any reused modules. Then the design is partitioned into many partitions that are conditionally evaluated on any given clock cycle. Finally, each statement in the design is converted into C++, which is combined with the user-written testbench and then compiled into an executable.

Unfortunately, in a large design with many instantiations of the same module, this approach is sub-optimal as it leads to huge bloating of the final executable, increasing compile times, and causing inefficient cache utilization. Prior work [2] has shown that a major cause of slowdown in large programs is instruction cache misses – in particular, those caused by fetching [4] – and so it is worthwhile to pursue any improvements that would shrink the code size to increase the chance that the code of interest is already in the cache. **Figure 1** confirms that for large designs (these are introduced and discussed later on) there are many instruction cache misses while running the simulators, bottlenecking the performance as the host processor must constantly stall to fetch the appropriate code. The goal of this work, therefore, is to reduce the simulation time by way of reducing the number of instruction cache misses suffered while running the simulator binary.

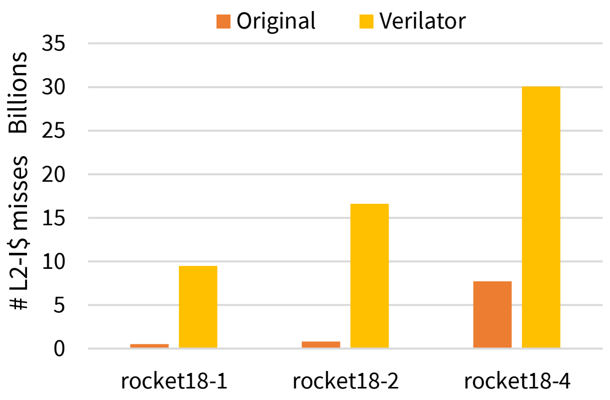


Figure 1: Number of L2 Instruction Cache misses in several representative designs' simulators while running a QuickSort benchmark.

This thesis makes the following major contributions:

1. A method of identifying the "greatest common shared module" (GCSM), which if deduplicated has a high probability of leading to a simulation speedup; and

2. Deduplication of that GCSM by partitioning it separately and adding an efficient layer of indirection to enable just one copy of the code needed for the GCSM to be reused many times; and
3. Evaluating the impact of the aforementioned efforts, to explore the feasibility of future investigation in this area; and finally
4. Contributing this work as free and open source software to benefit everyone in the hardware design community, available in the main ESSENT repository.[1]

The rest of this work is organized as follows. Section 2 provides background information. Section 3 presents details of the technical work and implementation that enables the results. Section 4 describes the metrics used to evaluate the work, and the results compared to the previous state of the art. Section 5 discusses future work and directions for the project. Section 6 presents and discusses related work, and finally, Section 7 concludes the paper.

[1] https://github.com/ucsc-vama/essent

2 Background

2.1 Simulation Paradigms

There are two dominant paradigms when it comes to logic simulator design: full-cycle, and event-driven simulation. Their key difference is their scheduling methods.

Event-driven simulators are the traditional design paradigm. They dynamically evaluate which signals to evaluate based on analyzing changes to all signals from one cycle to the next. Changes propagate through the design and result from a simulation testbench changing inputs or from a previous cycle's effects on other components of the design [7]. This approach has a heavy scheduling overhead, especially in large designs with complex dataflow graphs. The programming model of Verilog is designed with this execution model in mind. Icarus Verilog is an example of a tool that leverages this paradigm; traditionally the commercial tools do as well.

Full-cycle simulators evaluate the entire design on every clock cycle, so they avoid the overhead resulting from conditionally evaluating only some signals. Of course, this is inefficient when there is a low activity factor; that is to say, only a small fraction of the signals toggle over a given period of time. Verilator is an example of this kind of simulator design.

ESSENT's **hybrid approach** enables the scheduling to be done at compile time, performing full-cycle simulation only on the parts of the design that actually changed. More details about how this works are given in Section 2.3 below.

2.2 Anatomy of a FIRRTL design

Due to its increasing ubiquity and existing tooling support, FIRRTL [8] is the input format chosen by ESSENT. Some discussion of the advantages of FIRRTL and the structure of a design is in order, and this section will provide a brief overview. A complete design is referred to as the *circuit* and has at least one *module*. There must also be a module with the same name as the circuit itself, which is designated the *main module*. Each module defines a set of inputs and outputs, which are connected to *statements* inside the module's body. Modules cannot be parameterized,[2] nor can they instantiate themselves. The statements inside the module body can define a signal, register, or memory; connect one signal or expression to another signal; print a message to the simulation console; halt simulation; instantiate another module; or evaluate a conditional. Expressions are literal numeric values, references to other signals (including sub-accesses of fields of other signals), or some computation involving the preceding types.

An important advantage of FIRRTL is the formally specified *forms* that a design can be in: from *Chirrtl* being the highest-level form with many aspects of the design allowed to be unspecified so that they may be inferred later, to *LoFIRRTL* in which all widths are explicitly defined and a restricted subset of the statements are allowed to be used. The same approach is also used in the LLVM compiler framework. This allows tool authors to write simpler routines operating on the design: instead of being required

[2] In Verilog, one can add parameters to a module declaration, not unlike a template in C++. FIRRTL does not have this feature, so tools like the Chisel compiler will expand these into separate modules before FIRRTL emission.

to support the complexity of the higher forms of the design, the author can simply require that the design be transformed appropriately beforehand. ESSENT, for its part, requires input designs to be in the lowest FIRRTL form; the Scala FIRRTL library includes transformations to perform this lowering operation.

Some deduplication already happens in the FIRRTL Scala library. Tools like Chisel tend to emit duplicate modules which are identical internally, for example, the same queue module will be emit as *Queue_1*, *Queue_2*, and so on, despite having the exact same functionality, inputs, and outputs. As touched upon earlier, however, sometimes those duplications are in fact necessary when there are different parameterizations of the module. To address this, the Scala FIRRTL library applies the *DedupModules* transformation, which runs before ESSENT processes the design.

2.3 Current State of affairs in ESSENT

As a simulator-generator, ESSENT takes a given FIRRTL input design and compiles it to a C++ behavioral model. First, several custom FIRRTL passes transform the design to add and remove information to simplify later transformations. Next, the design is flattened, removing all the hierarchy and inlining any reused modules. Each signal reference is rewritten to be a fully-qualified name to reflect the original hierarchy (for example, the signal `a` in the module which was instantiated with the name `foo` at the top level would be rewritten as `foo.a`). Each statement produces one signal and can consume one or more other signals. The flattened list of statements is then converted to a directed graph by analyzing the dependencies of each statement and creating an edge from each producer to each consumer, called the *statement graph*. In particular, registers are split into two separate signals: the current value that can be used by other circuit elements, and the next value of the register, which will be placed into the register on the next clock cycle if it is enabled. Next, the statement graph is partitioned[3] to exploit low activity factors by combining several statements into one supernode[4]. Finally, the graph representation is emitted as the C++ behavioral model. The user, in turn, writes a testbench that interfaces with the model, like in Verilator; the top level module is a C++ class with an evaluation function to be called once every clock cycle of the simulation. The testbench author provides stimuli to the design by setting members of the class in between each clock cycle. The signals and circuit state are expressed as classes that the testbench program can inspect.

Even before this work started, ESSENT was in quite a good state: it already outperformed Verilator by 1.5-11.5× [5]. However, the major disadvantage is that repeated modules are simply inlined and any information about reuse is discarded; although simpler to handle, this leads to a massive bloating of the emitted code. Obviously, if the repeated modules have a lot of statements, then the replication factor is quite high. As with any program, all of the instructions will at some point be stored in the host processor's instruction cache, but if the program is too large, then there will be many capacity misses. It is therefore desirable to re-use code where possible. This

[3] This does depend on which optimization level is selected by the user; there is generally no reason to select anything other than the maximum.

[4] For a full explanation, refer to [5] as the details of the partitioning algorithm are too complicated to restate here.

also has the added benefit of reducing the compile time of the C++ model. Section 4 shows how this impacts program runtime.

The key invariant of the partitioned graph as described above is that the supernodes (partitions) must be acyclic. If this holds, then each partition need only be evaluated one time per cycle. Otherwise, there is a chance that a change in a “downstream” signal causes a change to the output of an already-evaluated signal, and the entire partition (at least that signal) would need to be re-evaluated.

3 Elements of Effective De-Duplication

This work currently only deduplicates one module per design, due to development time constraints as well as to provide a first order estimate as to how useful deduplication is for performance. Of course, in a given design, there might be many modules which are each reused many times, but here only one will be chosen to be deduplicated. We posit that that the largest, topmost module that is instantiated the most often is going to make the most impact when deduplicated. We call this module the Greatest Common Shared Module (GCSM).

3.1 Finding the GCSM

The essence of the heuristic used to determine the GCSM is quite simple: sort the multiply-instantiated modules from largest to smallest (by statement count, including any modules instantiated within), and then pick the module which is instantiated the most times. For example, in a multi-core CPU design, this is likely to be the individual cores. The algorithm to compute the score for a module is shown in **Figure 2**.

```
ComputeModuleScore(module):
    accumulator = 0
    For each statement in module:
        If statement is instantiating another module module2:
            accumulator += ComputeModuleScore(module2)
        Else accumulator += 1
    Return accumulator
```

Figure 2: GCSM Scoring Algorithm. This is computed for each reused module in the design.

This naturally selects the topmost module since the scoring recursively counts all statements used in a given module, so if there is a high-scoring module, but that module is instantiated by another module (which itself must be multiply-instantiated as well), then the latter will include the score of the former and thereby win. Of course, it is possible that some module obtains the largest score by virtue of being instantiated at many different points in the design hierarchy, but then it seems likely that deduplicating this module would be beneficial anyways, so this is an acceptable outcome.

3.2 Determining module instance compatibility

The goal of this entire exercise is to partition the GCSM separately from all of the other logic in the design. Of course, depending on the connectivity and size of the module, without considering the rest of the design it could be possible to incorrectly partition the design in a way that would cause a cycle in the statement graph. For example, it is tempting to simply consider the entire module as a single partition, but in most cases that will not work when taking the rest of the design into account. Therefore, a more careful approach is required, and it may be the case that not all instances of the GCSM can be partitioned in a compatible way.

Although each instance of the GCSM is identical internally, the external connectivity to each instance can be very different. Therefore, the combinational paths through the rest of the design – from each output of each GCSM instance to any input of another instance of the GCSM – must be found and taken into consideration in order to keep the design acyclic. Each of these output-input pairs we call a constraint. Consider two instances of the GCSM, X and Y, and their constraint sets C_X and C_Y respectively. If $C_X \subseteq C_Y$ then since X has fewer constraints than Y, any partitioning of

Y will be compatible with X. The instance which is compatible with the largest number of other instances is chosen as the master instance. The master instance is partitioned first, and its partitioning is copied onto the other compatible GCSM instances (hereafter referred to as cGCSM). The incompatible instances are treated as if they are part of the rest of the design and partitioned normally.

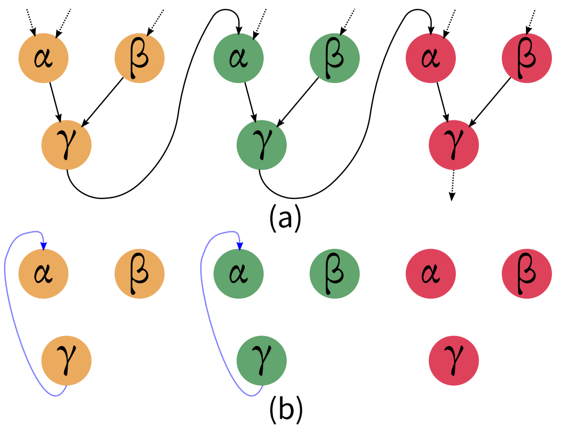


Figure 3: (a) Three instances of the same module, in which an output is connected to the input of the next module combinationally. **(b)** The dashed blue line indicates the realized constraints.

Figure 3 shows an example to demonstrate the importance of ensuring the compatibility of the GCSM partitionings. There are three instances of the same module which is the GCSM. Since eventually only one instance of this module will truly be emitted, the goal is to alias each node of one instance with the corresponding node of each other instance. Effectively, this means that there exists a path $\gamma \rightarrow \alpha$ (back to itself) for the first two instances. If the third module were selected as the master GCSM, it could happen that the partitioning $(\alpha\gamma, \beta)$ is chosen, but then there is a cycle in the effective graph, violating the acyclic invariant.

3.3 Partitioning the modules

Once the constraints have been identified and the master GCSM instance selected, the latter can be partitioned. To do this, the same partitioner from ESSENT is instructed to only partition the nodes belonging to that master instance. Then, the nodes from the master GCSM instance are mapped to their contemporaries in each cGCSM instance (this mapping is bijective since each instance must be exactly the same). Now the master partitioning can be applied to each cGCSM[5]. Finally, the remaining nodes not part of the cGCSM (including any instances of the GCSM that were not compatible) are partitioned to produce the final graph.

Up to now, all statements in the graph have their inputs and outputs statically defined as fully-qualified signal names. In order to be able to reuse the same code from the cGCSM, the inputs and outputs of each partition must be able to accept several possible inputs, so we add a layer of indirection to the generated code. In our implementation, we use *placeholder signals* for those indirected signals. First, each partition of the master instance is inspected, and every reference to a signal external to that partition is replaced with a placeholder signal; the same is done for every cGCSM instance with their corresponding signals. In the top level of the simulation model, a structure is instantiated for each of the compatible GCSM instances connecting each placeholder to its actual signal. That structure is referenced by each GCSM partition evaluation function to access the placeholders. At compile time, this structure is filled with pointers to the corresponding signals. The code to evaluate a partition belonging to the GCSM dereferences the placeholder signal to get to the actual signal; just as in

[5] This is mainly done for convenience within the program; strictly speaking the other cGCSM nodes could be left unpartitioned since they will be ignored later on.

original ESSENT it is a global variable named according to the hierarchy of the original FIRRTL design (Section 2.3).

3.4 Leveraging Activity Tracking

The key advantage of ESSENT is its activity tracking that allows it to only evaluate the necessary parts of the design; it is imperative that this behavior is also applied to the GCSM logic to keep the speed advantage. Of course, since now the code is generalized, the partition triggers can no longer be statically defined in the evaluation functions directly, necessitating an indirection through a mapping table. The insight here is that since each GCSM instance is partitioned identically, there exists a one-to-one mapping between any partition in one instance and each other instance. To facilitate the mapping of the aliased partitions, there is a lookup table to map the master instance partition's view of the other partitions it can trigger, to the view that the other repeated partitions have. **Figure 4** shows an example of this visually. It is possible that an output of one of the instances is left unconnected: in that case, a dummy output signal is created.

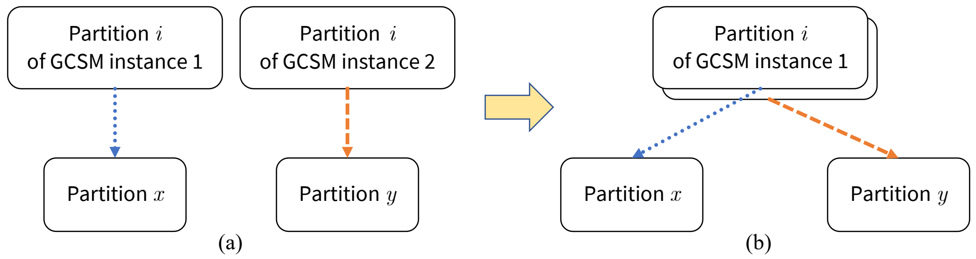

Partition i of GCSM instance 1
Partition i of GCSM instance 2
Partition x
Partition y
(a)
Partition i of GCSM instance 1
Partition x
Partition y
(b)

Figure 4: Example of partition aliasing. **(a)** Each partition of a given instance of the GCSM activates one or more external partitions, and the different instances of the GCSM may activate different ones. **(b)** To reconcile this, the repeated partitions are merged, and their external dependencies tracked.

Furthermore, one cGCSM instance partition may trigger multiple other partitions, whereas another cGCSM instance triggers only one other partition. In that case, the unmapped output trigger slots are mapped to an invalid partition which has no body and will never be evaluated. **Figure 5** shows an example of this.

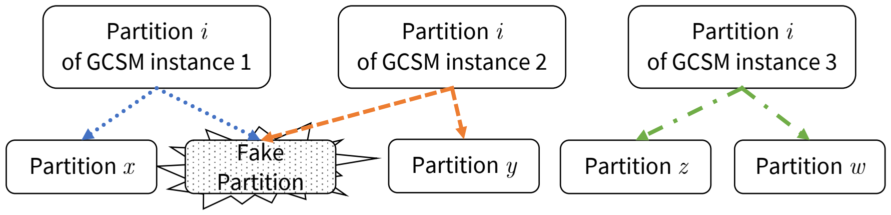

Partition i of GCSM instance 1
Partition i of GCSM instance 2
Partition i of GCSM instance 3
Partition x
Fake Partition
Partition y
Partition z
Partition w

Figure 5: When the master instance's partition (1) activates fewer other partitions than the corresponding partition of another instance (3), then a fake partition is inserted and activated to "balance" the outgoing activations.

4 Evaluation and Results

4.1 Measurement Methodology and Workload Selection

The key metric by which this work is measured is the speedup compared to an unmodified version of ESSENT and against Verilator 4.024. We also analyze instruction cache performance. Our initial hypothesis is that deduplication will shrink application binaries and thus improve instruction cache performance. Each simulator is run 10 times, and the average is reported here.

All experiments were performed on a dual-socket Intel Xeon Platinum 8260 Cascade Lake processor (2.4 GHz base), with 386 GB of DDR4 memory. The emitted C++ model and testbench are compiled with GCC 9.3.0 with `-O3`. The L1 instruction cache is 32KB per core (with another 32KB for the data cache, also per core), and has a 24 MB L2 cache shared among all cores. The host operating system is running Linux 5.4.0.

Design	FIRRTL Nodes	FIRRTL Edges	Total replication factor	GCSM replication factor
rocket18-1	62967	123145	5.5%	1.4%
rocket18-2	91190	173595	54.7%	50.7%
rocket18-4	146614	273665	65.7%	63.1%
50 DinoCPU	27848	55998	99.3%	99.3%

Table 1: Designs used for evaluation.

To validate the hypothesis, we select several hardware design workloads that form a representative cross-section of typical digital logic designs (**Table 1**) and have varying replication factors (that is, the percentage of the design that is replicated and also how much of the overall is identified as the GCSM). Rocket Chip [1] generates a

RISC-V SoC and is written in Chisel; variations with one, two, and four cores are tested here. In the single-core version the GCSM is one parameterization of the Chisel Queue module,[6] and for the multi-core versions it is one of the RocketChip Tiles (comprising the core and associated circuitry). The version from 2018 was chosen to best compare with the results obtained in the previous paper. DinoCPU [9] is a simple RISC-V processor designed for teaching, with the provided open source configuration being a single-cycle configuration. Since by itself it is only a small design, to enlarge it we make 50 instances that are each executing a program at the same time. In this case, the GCSM is one instance of a complete core, comprising a CPU and the memory.

Furthermore, to measure the impact of the number of repeated instances of the GCSM, we repeat the DinoCPU experiment described above, sweeping the core count from just one core (no reuse) to 256 cores. Of interest here are the compilation runtimes and the simulation runtime, as well as the simulation binary size.

To measure the performance of the processor design simulations, we need a program for them to run. We select a QuickSort program (taken from the riscv-tests project) due to the large number of instructions in it, allowing for the simulator code to be run for an adequate amount of time.

Although there are of course other simulators available (Section 6), we only discuss Verilator and ESSENT to better compare aspects about their emitted code.

[6] Actually, the GCSM algorithm discussed in Section 3.1 identifies a tiny (yet frequently used) debugging module, but for the purpose of obtaining useful results this was overridden to be the Chisel Queue which is the next-most-used.

4.2 Results

In general, performing deduplication is a worthy optimization: when the amount of replication is sufficiently high, it is both faster to run the simulation itself as well as to compile the C++ model. As seen in **Table 2**, a deduplicated dual-core RocketChip (rocket18-2) obtains a 4.7× speedup compared to the original ESSENT. The instruction cache miss rate is 1.5× lower and the host IPC improves by 1.8×, demonstrating that more of the simulator remains in the instruction cache. For the replicated DinoCPU, the IPC is 2.1× better and there is an order of magnitude reduction of the instruction cache miss rate. For a single-core RocketChip, the speedup is more modest (1.6×) but certainly enough to make this a worthy optimization. Unfortunately, a bug in the activity tracking in deduplicated ESSENT prevents us from being able to run rocket18-4, and more investigation is required than time permits.

As noted earlier, the numbers reported here the averages of running each simulator 10 times. The variance from these averages reported is very low, since all experiments were performed on a quiet system with nobody else logged in. Therefore, we are optimistic that these results are useful to validate the idea of deduplication in digital logic simulation.

Design	Tool	Bin size	Instructions executed ($\times 10^9$)	Host IPC	Runtime (seconds)
rocket18-1	Original	2.8MB	50.8	1.62	8.09
	Verilator	3.4MB	151	0.78	50.4
	Dedup	1.6MB	32.3	1.67	4.99
rocket18-2	Original	3.5MB	82.1	1.17	22.4
	Verilator	4.2MB	194	0.69	72.7
	Dedup	1.7MB	32.2	1.75	4.76
rocket18-4	Original	4.9MB	108	0.83	33.8
	Verilator	5.7MB	297	0.64	119
	Dedup	-	-	-	-
50 DinoCPU	Original	636KB	12	0.82	3.80
	Verilator	608KB	6.33	0.60	2.75
	Dedup	312KB	17.1	1.97	2.25

Table 2: Results of running the RISC-V qsort benchmark on each of the RISC-V processor designs. Each design is run under the original ESSENT, with Verilator, and with this work's deduplicating ESSENT. Note that the number of instructions executed refers to the simulator itself, not the program on the design.

In all cases, the Verilator simulation performance is much worse, which is not surprising given its scheduling model of re-evaluating the entire design's state on every cycle. Furthermore, its generated binary size is a lot larger than either ESSENT implementation; although here the difference is not large in an absolute sense, it is large enough to produce an extreme difference in the instruction cache miss rate, as much more of the code must be executed per simulated clock cycle than is necessary in ESSENT. One can imagine that in a much larger design with sufficient replication, this difference would become quite stark. **Figure 6a** shows the normalized L2 instruction cache miss rate for each simulation. As the design gets larger, the simulator binaries also grow, and logically the number of instruction cache misses will grow as well. By deduplicating, a lot of code is eliminated and therefore the number of misses drops precipitously.[7] **Figure 6b** shows the average number of misses per simulated cycle, showing that since each cycle is responsible for nearly an order of magnitude fewer stalls in most cases. Verilator does already apply some deduplication, so some parts of the code can indeed be reused, and therefore the number of instruction cache misses do not rise as rapidly as the design becomes larger, compared to original ESSENT. Verilator and deduplicated ESSENT are essentially evenly matched on the 50 DinoCPU design since the binaries are so small that they easily fit in the L2 instruction cache.[8]

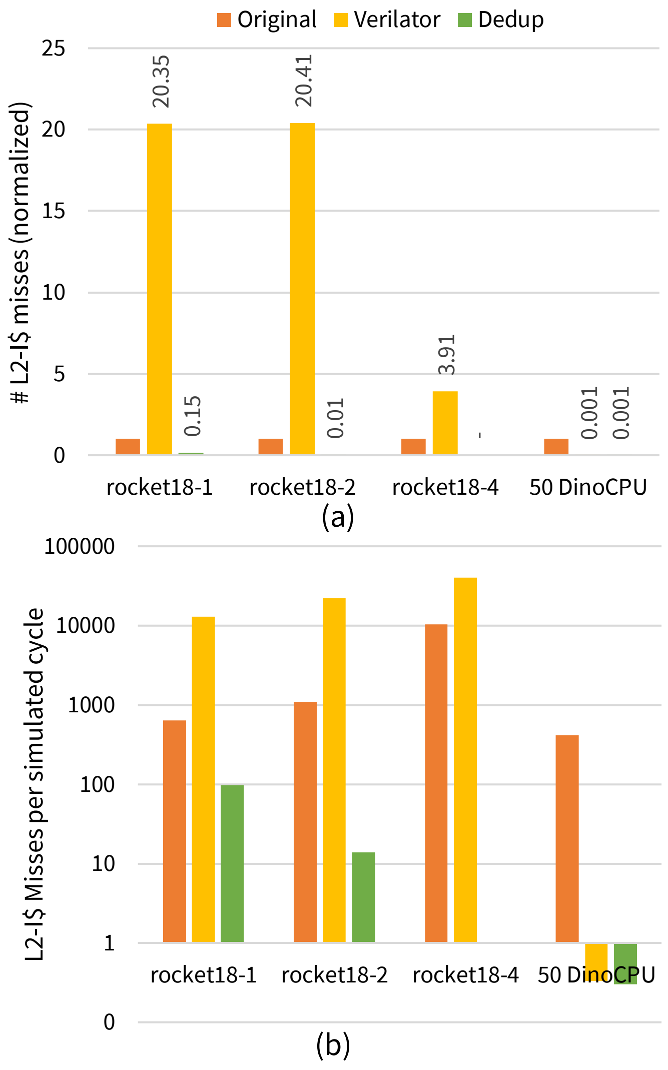


Figure 6: (a) Host L2 instruction cache misses normalized to that of the original ESSENT; **(b)** per simulated cycle.

[7] In the case of rocket18-4, although it may seem like Verilator is performing closer to the original ESSENT, the former in fact experiences an order of magnitude more misses.

[8] In fact, the reused parts probably even fit into the L1 instruction cache, although we have not measured this precisely.

To achieve the fastest execution times, as much as possible of the binary should be able to fit into the L1 instruction cache. To measure how much the code must be fetched and invalidated, we take a first-order measurement of the amount of instruction data flowing to the L1 cache. We measure the number of L1 instruction cache misses per simulated cycle, and multiply this by the cache line size of our host processor (64 bytes). As seen in **Figure 7**, the deduplicated ESSENT causes less data transfer from L2 to L1, largely by virtue of its smaller binaries. This also suggests that the deduplicated code enjoys temporal locality by staying in the smaller L1 cache (32 KB per core). To be sure, this is not a perfect analysis, and future work could include a more accurate analysis of what parts of the simulator binary are repeatedly transferred.

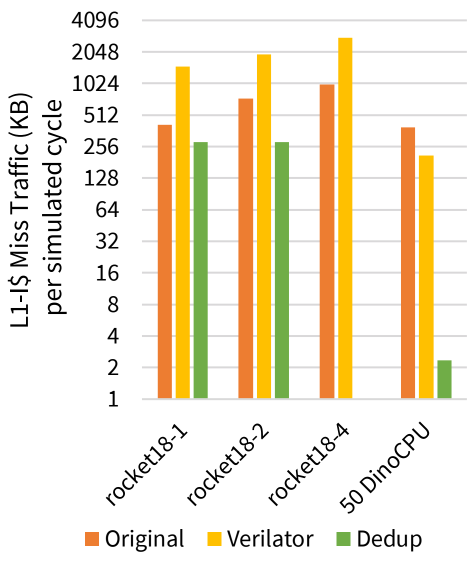


Figure 7: L1 Instruction cache miss traffic volume shows how well a given design fits in the L1 cache.

Another point in Verilator's favor is that it includes some features not yet found in ESSENT, such as the ability to show simulation progress in the form of Value Change Dumps (VCD) and perform deeper inspection of all the signals in the design (many signals in ESSENT are scoped only to the partition they are used in, if they are not used anywhere else, whereas in Verilator any signal can be addressed with a deterministic name).

Contrary to our expectations, the branch prediction accuracy was not significantly impacted for any of the designs on any of the simulator generators. The miss rate is around 1.5% in all cases, which is quite low. This suggests that the indirection from the deduplication does not negatively affect performance in this regard.

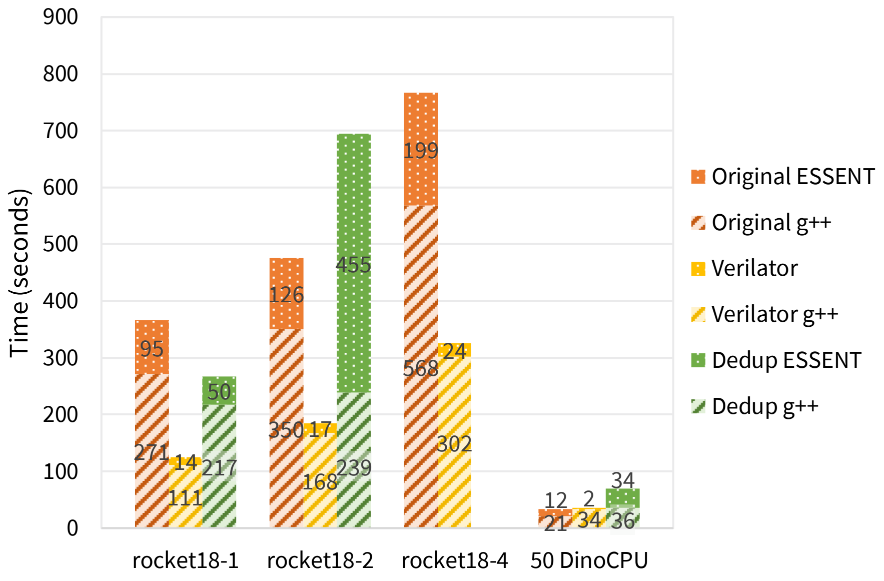


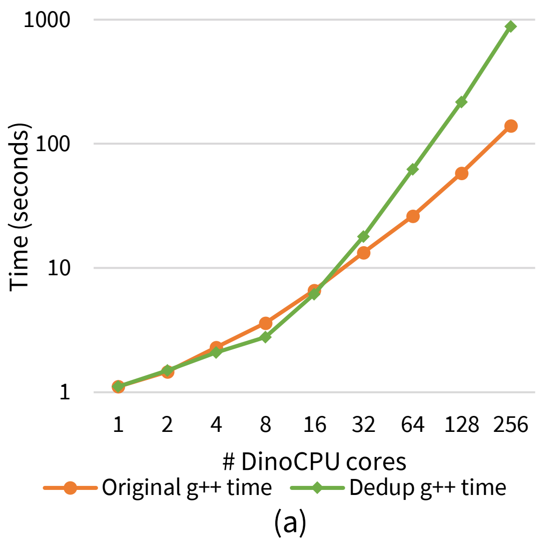
Figure 8: Time taken for C++ generation and compilation for each tool and its emitted code.

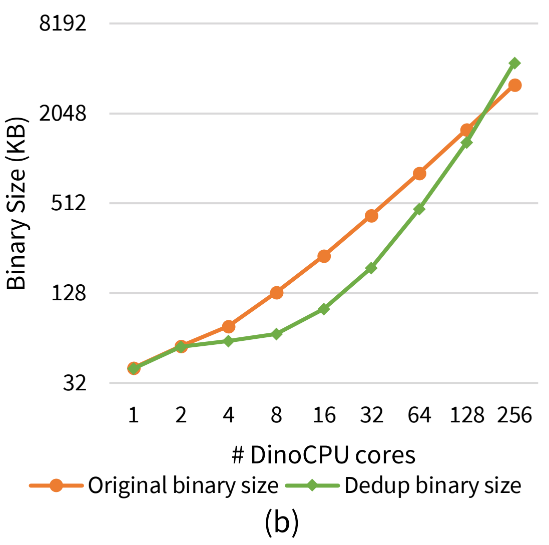
Not only does the simulation performance benefit from the deduplication, so does the C++ compilation speed, since there is less code to compile. As shown in **Figure 8**, a modest speedup is obtained over the original ESSENT C++ compilation time. On the other hand, Verilator's compilation is still faster for RocketChip, perhaps due to the fact that the emitted code is split across multiple files.[9] Only in the case of the 50 DinoCPU is Verilator's C++ compilation slightly slower, likely thanks to optimization. For the 50 DinoCPU design, the deduplicated C++ compilation is slightly

[9] This allows those files to be compiled in parallel, although that was not done in this evaluation for the purposes of fairly comparing compilation times.

slower than original ESSENT's, which is likely thanks to the optimizer exploring some (futile) optimizations. Also, the time needed for ESSENT to generate the C++ model is much higher than Verilator, and our deduplication enhancements did not reduce this factor any further. The dominant component of the tool runtime is the partitioning; Verilator does not do this and so it handily beats either ESSENT implementation in that regard. This work partitions the design twice, which of course contributes to the runtime even further. Finally, most of Verilator is written in C++, while ESSENT runs on the Java Virtual Machine, so the former has a big advantage in that way as well.

As the reuse factor goes up, it is expected that the deduplicated binary size grows slower than without deduplication. To measure the impact of the amount of replication in a design, we perform an experiment in which we sweep the number of DinoCPU instances and measure the impact on C++ compilation time, simulator binary size, and simulator runtime. **Figure 9** shows the results. Our hypothesis is true, to an extent; however, when the total design is large enough (32 cores or more), the C++ compiler has difficulty keeping the binary size to a reasonable level. In addition, since the repeated portion grows without significantly growing the rest of the design, it is logical to predict that the C++ compilation time would not grow too much. However, as before, after 32 cores, the compilation times start to surpass those of the original ESSENT output. However, simulator runtime benefits tremendously, taking far less time to run. There are a few other noteworthy points. First, in **Figure 9b** we observe that the deduplicated binary size does not grow rapidly from 2 to 8 cores, but then shoots up quadratically after that. This is likely a result of the compiler trying to inline





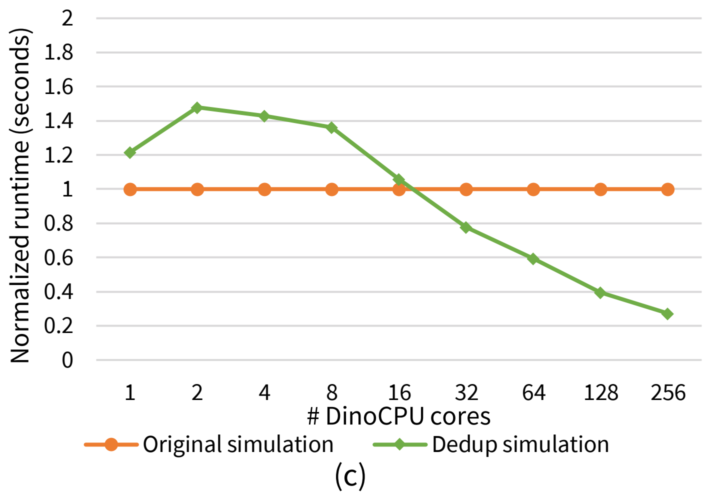


Figure 9: For each configuration of DinoCPU cores: **(a)** The C++ compilation time, **(b)** the simulation binary size, **(c)** the simulation runtime (normalized to the original ESSENT simulation runtime).

some functions in the name of performance; we have not yet investigated if this delivers any benefit to the performance. Next, in **Figure 9c** we see that for smaller amounts of replication, the deduplication is actually slightly slower than before. According to our performance data, there is a considerably higher L1 instruction cache miss rate, which may be the result of poor spatial locality of the deduplicated code segments.

Metric	Original ESSENT	Dedup Singleton Instance
# Instructions Executed (B)	50.8	41.6
IPC	1.62	1.67
L2-I$ misses (M)	468±2.2%	9.3±13%
Binary size (MB)	2.8	1.58
Runtime (s)	8.09	6.43

Table 3: Simulation runtime overhead of "deduplicating" (applying the indirection treatment to) the singleton instance of *RocketTile* in rocket18-1. Note that the original ESSENT results are duplicated from **Table 2**.

It is worth considering the cost of the indirection (as described in Section 3.3) to gauge its impact on simulation performance. To that end, we return to the rocket18-1 design, and treat the top-level *RocketTile* (the main module holding all of the components of the CPU) as the GCSM, even though of course it is not actually reused at all. We run the same QuickSort benchmark as in the other evaluations. **Table 3** summarizes the results; whereas there are many more instructions executed as a result of all the extra pointer lookups, and a slightly larger binary in the latter case, ultimately the simulation completes faster and enjoys a slightly higher IPC and a lower instruction cache miss rate.

5 Future Work

As with any emerging project in this space, there is a long laundry list of desired features and nice-to-haves. This list covers but a few of them that are relevant to this line of research.

Multiple GCSMs. The code written for this thesis only identifies one module to be the GCSM, although it may be the case that there are several modules which are repeated such that they make up a significant percentage of the overall logic in the design. Even better gains could be realized if those are also deduplicated. For example, another good candidate for deduplication in RocketChip is the Floating-Point Unit, which is also instantiated in each tile, each one consisting of 5600 statement nodes.

Better performance counting and more intelligence in choosing whether or not to deduplicate. To enhance the results obtained in this work, the performance of the simulator as a whole is measured. A good next step is to investigate the performance of the deduplicated code in greater detail. For example, how much of the deduplicated design is active every simulated cycle? If the module that was chosen to be deduplicated has low activity, perhaps there is another module which if deduplicated would provide a greater benefit overall. Ideally, this would be done by ESSENT so that the designer does not need to try out many different configurations (which, due to the dynamic nature of the field, may change over time).

Compiler optimization improvements. As the designs get larger, so do the compile times. However, for some reason, in the case of the deduplicated code the compile times increase far more rapidly than with the original ESSENT. We hypothesize that this is thanks to the compiler attempting to optimize the code in a way that is ultimately futile. It is worth looking into various compiler flags to control that behavior and temper the exploding compile times.

Testers interface. A somewhat recent development in the Chisel space is the Testers interface which allows unit tests to be defined along with the modules in the design itself. Currently, this is able to target Treadle, a simulator written in Scala intended to be used specifically with Chisel designs, and Verilator. There have been efforts in the past to integrate ESSENT into this interface but have unfortunately stagnated.

Better identifiers. Internally, many of the data structures and routines do their bookkeeping by way of strings that refer to other objects in the design. Of course, that means that when one of these identifies changes, all of the places that it is referenced must also be modified. In addition, there is some possibility for a conflict if a truncated name ends up aliasing to multiple other objects. We suspect that this could also help fix the problem with rocket18-4, in which our debugging done so far points to a problem concerning a conflicting name internally; the generated C++ code is syntactically correct but the behavior is incorrect.

Multiple clocks and asynchronous resets. Many practical designs require multiple clocks and asynchronous resets, as found in many ASIC designs. Currently, ESSENT only supports one clock and one synchronous reset which is implicitly connected to all state elements. This will be a complicated task involving much more bookkeeping in the tool and will undoubtedly be the subject of further research within our group.

Multi-threading and parallelization. As described in [4], parallelization is a worthwhile future research avenue. With regards to the work performed in this thesis, it could be possible to evaluate each GCSM instance independently, especially if each instance is not connected to the others. For example, if the GCSM is one core of a multicore processor design, then each of the cores could be evaluated in their own thread.

6 Related Work

Verilator [11] is a similar tool to ESSENT in that it generates a behavioral C++ model from RTL sources, and has already been discussed in this paper. Unlike ESSENT, however, its input format is synthesizable Verilog, which has the benefit of being able to leverage myriad existing designs, both open source and commercial. While elaborating the design, Verilator does handle deduplication: any module which is instantiated more than once is placed into its own C++ class (all others are inlined). As a full-cycle simulator, Verilator evaluates the entire design on every clock cycle, leading to extra work being performed even in designs with a low activity factor.

Icarus Verilog (*iverilog*) [12] is a free and open source event-driven Verilog simulator. It takes a design and a testbench written in Verilog. This tool implements more of the Verilog standard, enabling much greater compatibility with the broader ecosystem. As an event-driven simulator, it carries the burden of runtime design re-evaluation scheduling.

LiveSim [10] aims to reduce the *edit-run-debug* loop's latency – that is, shorten the amount of time needed before updated simulation results are ready after editing the HDL. To accomplish this, only the parts of the design that changed are compiled, and then patched into the final executable. Furthermore, it implements checkpointing and hot reloading to make it faster for the user to get results from their updated HDL. Importantly, it also demonstrates that instruction cache misses due to a large code size are a major bottleneck for simulation performance.

Treadle [13] is an event-driven simulator integrated into the Chisel language, and benefits from being tightly integrated with the rest of the Chisel ecosystem; both the design and the tests are written in Scala, and can even be placed next to each other in the same file.

Commercial Simulation Tools are generally also event-driven simulators. Like Icarus, they support most of the Verilog standard, but have the same scheduling burden. Of course, it is not known what optimizations, if any, are made internally to speed up execution. These tools were not studied in this work; however, as shown in [5], even the original ESSENT handily beats their performance.

Formal Verification Methods are not typical simulation tools, and do not simulate the design cycle by cycle. Instead of writing a testbench, the designer writes a set of assertions about the valid state(s) of the circuit, given a set of constraints on the inputs. Then an SMT solver is used to mathematically prove that the system cannot under any circumstance enter an invalid state; if it is possible to get to an invalid state then the solver will produce an example. While this provides the strongest guarantee of proper circuit operation, in practice most systems are too complex to precisely model their behavior in this way; typically, only smaller components of the design will be formally verified. Although out of scope for this paper, it is known that solvers, too, are typically slow.

7 Conclusion

This paper presents an improvement to the state of the art in ESSENT by inspecting the input design, identifying reused hardware components, partitioning them in a way that is compatible with the maximal set of other instances of the GCSM, and emitting more efficient code. As demonstrated by the speedups we achieve of up to 4.7× over the previous version of ESSENT and up to 15× compared to Verilator, by deduplicating just one module of the design, this research shows that deduplication is a worthwhile research direction. Clearly, there is much more to explore, but it is certainly clear that deduplication is an important optimization to apply in digital logic simulators. Finally, by contributing this work back to the open source community, all may benefit from faster simulation times.

8 References

[1] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[2] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, Phoenix Arizona, Jun. 2019, pp. 462–473, doi: 10.1145/3307650.3322234.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, San Francisco, California, 2012, p. 1216, doi: 10.1145/2228360.2228584.

[4] S. Beamer, "A Case for Accelerating Software RTL Simulation," *IEEE Micro*, vol. 40, no. 4, pp. 112–119, Jul. 2020, doi: 10.1109/MM.2020.2997639.

[5] S. Beamer and D. Donofrio, "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jul. 2020, pp. 1–6, doi: 10.1109/DAC18072.2020.9218632.

[6] International Computer Symposium, "Proceedings of International Computer Symposium 1980 : December 16-18, 1980, Taipei, Republic of China.," 1980. http://books.google.com/books?id=QHQ_AQAAIAAJ.

[7] D. M. Lewis, "A hierarchical compiled code event-driven logic simulator," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 10, no. 6, pp. 726–737, Jun. 1991, doi: 10.1109/43.137501.

[8] P. S. Li, A. Izraelevitz, and J. Bachrach, "Specification for the FIRRTL Language," EECS Department, University of California, Berkeley, UCB/EECS-2016-9, Feb. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.

[9] J. Lowe-Power and C. Nitta, "The Davis In-Order (DINO) CPU: A Teaching-focused RISC-V CPU Design," in *Proceedings of the Workshop on Computer Architecture Education - WCAE'19*, Phoenix, AZ, USA, 2019, pp. 1–8, doi: 10.1145/3338698.3338892.

[10] H. Skinner, R. Trapani Possignolo, S.-H. Wang, and J. Renau, "LiveSim: A Fast Hot Reload Simulator for HDLs," in *2020 IEEE International Symposium on Performance*

Analysis of Systems and Software (ISPASS), Boston, MA, USA, Aug. 2020, pp. 126–135, doi: 10.1109/ISPASS48437.2020.00028.

[11] W. Snyder, *Verilator*. .

[12] S. Williams, *Icarus Verilog*. .

[13] *Treadle - a chisel/firrtl execution engine*. .