LiveSim: A Fast Hot Reload Simulator for HDLs

Haven Skinner*, Rafael Trapani Possignolo†, Sheng-Hong Wang‡, Jose Renau§
Dept. of Computer Science and Engineering
University of California, Santa Cruz
Email: *hskinner@ucsc.edu, †rafael@ucsc.edu, ‡swang203@ucsc.edu, §renau@ucsc.edu

***Abstract*—With the increased complexity of digital architectures and aggregation of specialized hardware, functional simulation has become a major bottleneck in digital design. During functional and performance verification of a design, engineers make several iterations to determine the impact of code changes into the simulation result. These iterations are time-consuming both because the compilation time of hardware description to binary is slow and because simulation can take several hours until the point of interest is reached. In contrast, live programming environments allow developers to manipulate the system under development as it is being run. They have become increasingly popular as they provide rapid feedback, yet there is no available live environment for hardware development. In this paper, we propose a live programming and simulation environment that targets hardware design. Our approach is language-independent and leverages incremental compilation, hot binary reloading, and checkpointing to provide fast feedback to the user. We take special care to not replicate code for multiple instances of the same module and thus prevent code bloat, for instance, for multi- and many-core architectures. Our framework also is careful in verifying the consistency across checkpoints, to leverage parallel execution and reduce the amount of code that requires compilation. Our results show that this approach can provide simulation feedback in under 2 seconds, even when simulating a 256 RISC-V multicore architecture. As a reference, Verilator did not finish compiling this architecture after 24 hours of runtime.**

I. Introduction

Digital designers spend much time waiting for the compilation and simulation of hardware designs during the functional and performance verification phases of a project. While there have been many attempts to accelerate simulation [1], [2], [3], the underlying issues remain: 1) designs are complex, in particular for large multi-core systems, and 2) to guarantee code coverage, simulations need to be long to account for different scenarios. These issues lead to long compilation and simulation times. Modern languages like Chisel [4] or PyMTL [5] aim to improve design productivity by raising the abstraction level but typically further the compilation and runtime issues. This adds a real cost to developer productivity, as it is not uncommon that after fixing a single bug, the engineer relaunches the simulation only to find another close by.

Moreover, studies [6] show that debugging accounts for over 40% of the verification time and that its importance has been increasing in the last few years. The process of debugging involves observing a problem with the system under development, reasoning about the root cause, and correcting the issue. For large-scale digital design, the primary bottlenecks in debugging is the time required to observe the behavior of the system (due to the speed of simulation/synthesis tools), which incurs in latency between making a change and observing the effects of the change. Reducing the feedback time is expected to improve designer productivity [7].

Live programming stands in stark contrast to traditional hardware design flows. With live programming, developers write the program as it is running and see the effects of the code change immediately reflected in the system's behavior. Prior work has shown that faster feedback leads to a faster and less frustrating development experience [8], [9], [10]. In practice, live programming is accomplished by a combination of incremental compilation techniques. Hot binary reload (Hot Reload) means swapping the binary without stopping execution, which allows the current state of the program to be maintained. Checkpointing is another technique used by live compilation flows where a state is saved throughout execution. A checkpoint can then be loaded, and execution continues from that point, avoiding the need to execute the code from the beginning. In hardware design, this is especially useful for very long input sets for booting architectural simulation to an initial state, which is often repeated across many different simulations and could be skipped.

Recently, incremental techniques [7], [11] have been proposed to address lengthy synthesis times for hardware design. However, hardware simulation is notoriously slow and typically run much longer than synthesis, during functional and performance validation. Typical industrial simulation flows require tens of minutes just to do a minimal code change; for large-scale designs like modern processors, it is not uncommon for a simulation to take hours to reach a failure point.

In this paper, we propose LiveSim, a framework for Live Simulation of hardware designs. The main goal of LiveSim is to reduce the edit-run-debug (ERD) loop, *i.e.*, the latency between a code change and the availability of updated simulation results (Figure 1). LiveSim can significantly speed up the ERD loop by reducing compilation and simulation times. First, LiveSim compiles only the parts of the design that have changed and patch them into the executable. Then, LiveSim leverages simulation checkpoints to prevent executing the whole simulation from the beginning. We set a goal of under two seconds, which is the threshold considered *responsive user* interaction [9].

LiveSim includes the capacity to: **(1)** hot reload combinational logic and state, and **(2)** using the lightweight checkpointer, quickly verify in parallel that the checkpoint's initial state is consistent. Item (2) describes a simple algorithm leveraged by LiveSim's simulator, allowing it to provide a quick estimate on the updated state of the system after a hot patch, but continue to refine that estimate on the backend, hiding as much of the computation as possible from the user.

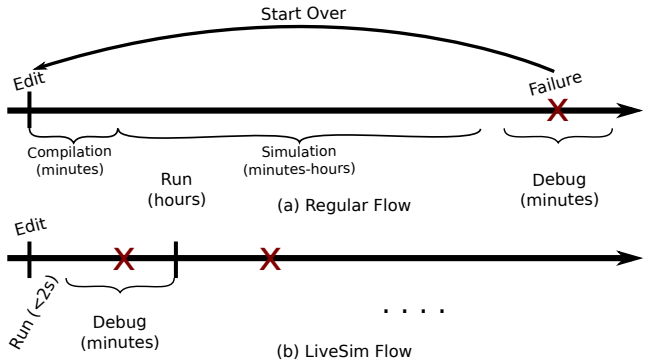

Fig. 1. The proposed workflow for LiveSim is to allow iterations within a few seconds. This contrasts with the traditional flow where after a change, there is a lot of latency in each iteration. With the much faster ERD loop, enabled by LiveSim, productivity is expected to improve.

To support the two-second goal, LiveSim implements a fast compilation technique, which includes **(3)** incremental HDL parsing and hierarchical partial compilation. LiveSim uses incremental parsing and compilation to compile changes in the architecture into hot loadable shared libraries, for later patching into the simulation.

Finally, **(4)** we develop and prototype a language-neutral Hot Reload for hardware. Users of hardware simulations would greatly benefit from being able to hot load source code changes rather than having to recompile and rerun the entire program. This paper presents the first hot reload architecture for digital hardware simulation. We show that the LiveSim infrastructure is not only able to perform Hot Reload in under 2 seconds, but it is also more scalable than currently available simulators.

LiveSim is faster in raw simulation speed than Verilator [12], the state-of-the-art open-source simulator for hardware. For instance, when evaluating a 16 node large Partitioned Global Address Space (PGAS) RISC-V multi-core, Verilator in single-thread mode has a global speed of 51.4KHz. LiveSim, with or without checkpoint overheads, has a global speed of 93KHz. Verilator is slightly faster for small designs, but as the design complexity increases, the generated code footprint does as well, and it suffers from significant instruction cache misses in the host machine. LiveSim does not have such scalability issues. As shown in our evaluation, when using checkpoints, LiveSim can get to any simulation cycle in under 2 seconds for a 256 RISC-V core design.

We plan to open-source LiveSim in the near future, once the paper is accepted. In summary, the main contributions of this paper are:

- Proposes LiveSim, a live simulation flow for hardware designs
- Integrates Incremental Compilation, Hot Reload, Checkpointing, and Improved Binary Replication in a hardware simulation flow
- Present a methodology to fast verify the consistency of checkpoints
- Evaluates LiveSim against state-of-the-art open-source simulators and shows that instruction cache misses is an issue in current flows

II. Related Work

A. Live Programming and Hot Reloading

There is a variety of research and related work in *Live programming* techniques for software development. Tanimoto [13] defines a hierarchical principle for measuring the liveness of the live programming loop, from source code editing toward output evaluation, which is similar to the evaluation paradigm that we adopt here. DS.js [14] embeds a JavaScript programming environment in webpages; a live execution model is proposed to runs the entire code block when detecting the code statement writing activity and thus triggering compilation. Pharo [15] is a object-oriented language with the same focus as LiveSim on getting live ERD. Since Pharo IDE updates all its applications at run-time, including the debugger, the developer can edit the source and observe the effect on the debugger immediately. LiveSim primarily draws inspiration from debugger-based live programming environments like Pharo, as the simulation of the system can be seen as analogous to a debugger.

In hardware, there are no approaches for live programming. However, some existing approaches could be adapted to provide a live environment. Jupyter Notebook [16] is an open-source web application that allows users to create interactive documents and code. Chisel [4] is a hardware description language that allows for integration in Jupyter as an educational aid. Chisel also has a simulator called Treadle [17]. This integration has similar motivations to our project, the main goal being to manipulate a system and get fast feedback, which aids the user in understanding it. Chisel's integration with Jupyter is not optimized for large designs and therefore limits its application severely. Updating even trivial designs that involve only a single stage can take many seconds, as the entire simulation must be rerun from the beginning. Moreover, Chisel is not intended for incremental compilation, and the toolchain is currently not targeting turnaround time in the magnitude proposed here.

Schkufza et. al. proposed a JIT compilation method for Verilog FPGA designs [18], which runs the design in a software simulation while it waits for the FPGA compilation process to finish. This is aimed at addressing the same problems that motivated LiveSim, but is designed for FPGAs, not ASICs, and does not support live updates of the simulation after a code change.

Hot reloading is a common technique used in Live programming environments. It is typically implemented as a specific library for existing languages, like for React [19] and Java [20]. The primary motivation behind implementing this feature was to improve the development experience, as it would allow developers to make changes to the code without losing the state of the app they are debugging.

B. Checkpointing and Parallel Replay

LiveSim leverages checkpoints in order to provide a live development experience. There are a variety of projects [21], [22], [23], [24] which use multi-tiered simulations, a fast

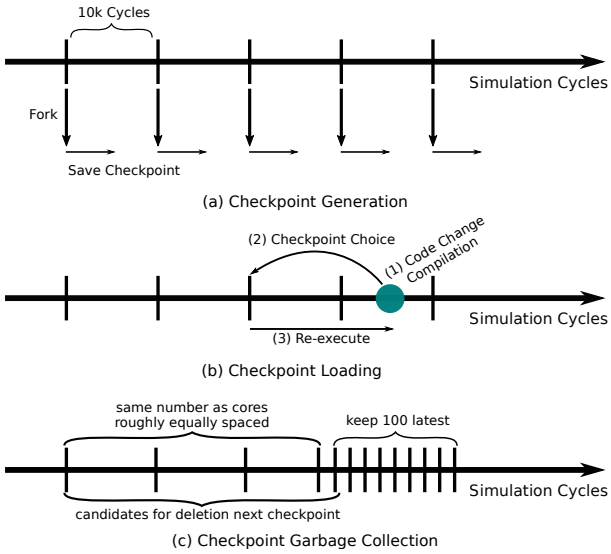

Fig. 2. During the baseline execution, checkpoints are regularly created. Then, during the live mode, the code changes are recompiled, reloaded, a checkpoint is selected, loaded and the execution continues until the cycle before the code change was inserted. Checkpoints are regularly deleted to avoid too much overhead.

simulator to create checkpoints and a slow simulator which can evaluate them in parallel, to speed up HDL simulation. One key difference between these projects and LiveSim is that LiveSim uses only one simulator to generate the checkpoints.

A similar technique has been proposed for approximate computing [25], where a fast approximation of the application runs and takes checkpoints, while detailed implementations validate them in parallel. This approach does have similarities to the item (3), though like the fast HDL simulators listed above, would have to do considerably more work to validate the checkpoint deltas.

III. LiveSim

The goal behind LiveSim is to improve the productivity of hardware developers by providing fast feedback as changes are made to the codebase. When LiveSim begins executing a simulation from the beginning, it takes checkpoints at regular intervals. When a change is made to the codebase, LiveSim incrementally compiles and hot reloads the new logic, and moves the simulation along the checkpoint delta using the method shown in Figure 2. The key features of LiveSim which enable this are listed below.

- Incremental Compilation, triggered after a code change
- Hot reload of recompiled objects
- Checkpoint selection and reload
- Execution from the checkpoint to the current cycle
- Checkpoint consistency check (done in parallel in the background)

In this section, we first discuss how we envision that LiveSim will be used, then we discuss the high-level architecture and main components of the simulator. We follow by giving more details on the compilation infra-structure and reloading both of the object and of the simulator state. We also discuss how the consistency of the checkpoints is verified after a change.

A. Use Cases

We do not expect that LiveSim will replace a full functional verification flow, with long simulation hours that target extensive coverage of the system. However, we believe that our technique is extremely valuable during the early phases of development and bug-fixing, where a large number of very specific scenarios need to be tested and fixed. LiveSim may be particularly useful during performance validation, where designers typically want to make sure a specific number of cycles is obtained far in the simulation.

Debugging a single simulation: This is the primary use-case for LiveSim. The developer can benefit by using fast checkpoint reloading to identify an error, test multiple possible fixes, and continue debugging without having to fully rebuild and reload the simulation. For example, as an observed error usually originates from the prior cycle in most cases, the designer can use a previous checkpoint to identify where the condition originated.

Debugging "what if" conditions: Fast reloading can also be leveraged to check "what if" situations, such as test what would happen should a branch misprediction be detected on an arbitrary cycle in a simulation. Some of those conditions may be hard to trigger from a long simulation due to the complexity of the state space of a large design, and a lighter framework such as the proposed LiveSim may help create the "what if" scenario. Once created, the scenario could be incorporated into the full set of tests of the system for longer simulations.

Regression System: Instead of viewing the session history as a linear list of individual checkpoints, a regression system could be built on top of LiveSim, which could run a set of testbenches on the system and report their result as a batch. Regression is particularly useful to test if the system state progresses as expected, starting from an arbitrary state, not necessarily from the initial state.

Allow simulations to skip initialization: Restarting a simulation is universally slow. As an example, starting the BOOM [26] core from reset is particularly slow because the processor initializes a debug monitor. Some companies take great pains to modify their flows to skip some of the initialization work for their testbenches. With hot reload, parallel checkpoint history verification, and deterministic register transformations (discussed in Section III-E), this behavior can come for free.

B. The LiveSim Architecture

The LiveSim environment is composed of LiveParser to parse code changes, LiveCompiler that creates object files for the code, and LiveSim itself for running the simulation (Figure 3). Users could interact with the system both by manipulating the source code of the active project and by sending commands to the simulator.

As the user manipulates the design, LiveSim tracks where the changes were made, and sends commands to LiveParser

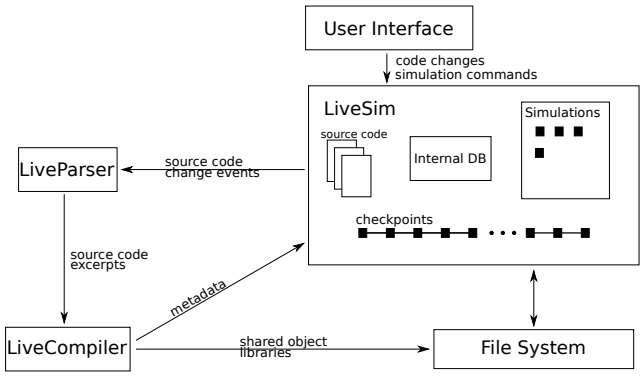

Fig. 3. The LiveSim development environment.

(Section III-C). LiveParser determines if the code changes impact behavior. In cases it does, only the modified sections are sent to LiveCompiler (Section III-C), which will compile those blocks into hot-loadable shared object files. LiveCompiler will also update LiveSim with metadata that describes the updated parts of the simulation. LiveSim can then retrieve those newly hot-loadable objects.

Internally, LiveSim maintains a session to track the history of target testbenches, to record the simulated cycles, and to creates checkpoints regularly. To prevent checkpoint creation from being in the simulation critical path, the process forks, the child process creates the checkpoint and halts. Thus we keep the very minimal interference within the baseline process (Figure 2). Should the simulation be patched, the history will be used to update the results more quickly (Section III-D).

Prior art, like Verilator [12], tries to inline small blocks to allow for better optimizations across modules. Even when inlining is not done, each module is replicated for each instance and set of parameters. The main goal is to allow for better quality in the generated code, which could, at least in theory, increase the simulation speed (Figure 4). On the other hand, LiveSim leverages the modularity of hardware designs to break up the work that needs to be done. Each module is only compiled once, which drastically reduces the amount of code that needs to be compiled, in particular for large multicore systems. On top of that, LiveSim also uses multiple shared object libraries to allow for swapping out parts of the binary in flight.

1) Internal Objects: "Live" refers to the fact that the system is running as it is being developed. For this reason we don't view a simulation as a procedure with an entry and exit point, but as an interaction between objects. At a high-level, LiveSim works with two types of objects, which are all hot reloadable at runtime from shared libraries:

Stage: A block of logic. A stage has external IO, and could also have internal registers and memory depending on the source code. A stage can also create and connect to other stages.

Testbench: An operation that can be performed on a stage.

The Unit Under Test (UUT) is the simulation entry-level entity of the design and is composed of multiple stages. In this paper, we use pipe, pipeline or UUT interchangeably. Each stage is transformed from RTL into a runtime-linkable shared object library and may instantiate additional stages from the same or other shared object libraries. The testbench is also loaded from a shared object library, which contains procedures that can be run on the UUT for any given number of cycles. Doing so changes the design's internal state. Such changes are viewed by LiveSim as operations on the UUT, whose history is tracked and checkpointed as part of the simulation session. This allows those same operations to be applied again, should the design be updated due to a change in source code.

When the user makes changes to the source code, the changed code are compiled into new shared object libraries. These new shared libraries can be hot reloaded into the simulation and instantiate new parts of the UUT, which can be swapped piecemeal into the executable. This mechanism allows the simulation to be quickly patched without needing to be fully recompiled or even needing to fully copy the simulation state.

2) Internal Tables: The *Object Library Table* (Table II) are internal tables to keep track of stages and testbenches objects being simulated. To manage stage and testbench objects, LiveSim keeps track of the path where object files are stored. At the beginning of a session, LiveSim will use a standard shared libray interface to retrieve the stages and testbenches in the tables.

The *Object Library Table* lists all of the objects that the current session of LiveSim is aware of. Each stage and testbench object found is given an arbitrary name. The name is mapped to its **source-path**: its location in the source, and its **object-path**. Using the API described in Table I, which is bound to a globally available LiveSim context, threads within the LiveSim environment can create and manipulate these objects.

The *Pipeline Table* (Table III) lists the instantiated pipeline objects and the corresponding name in the session. Objects get added to this table through the $instPipe$ and $copyPipe$ commands. When a pipe is created, it will also call $instStage$ to create stages. The newly created stages are then saved in the *Stage Table* (Table IV). Function calls to $swapStage$ cause changes to the *Pipeline Table* as well; $swapStage$ usually happens when changes to the source code have prompted LiveCompiler to create new shared object files.

3) A Note on Implementation: LiveSim support could also be added to newer HDLs like Chisel, PyMTL, and PyRTL [27], though it would require tooling in their compilation and simulation infrastructure. The JVM in Chisel and Python interpreter in PyMTL could be leveraged to provide hot reloading, though that would require the simulations to be run with non-native code, which the developers of the respective projects do not recommend. In another perspective, Chisel and PyMTL do compile their source code to Verilog, and could leverage the Verilog output to integrate into the LiveSim system. PyRTL uses its own simulator, so we leave that discussion to future work.

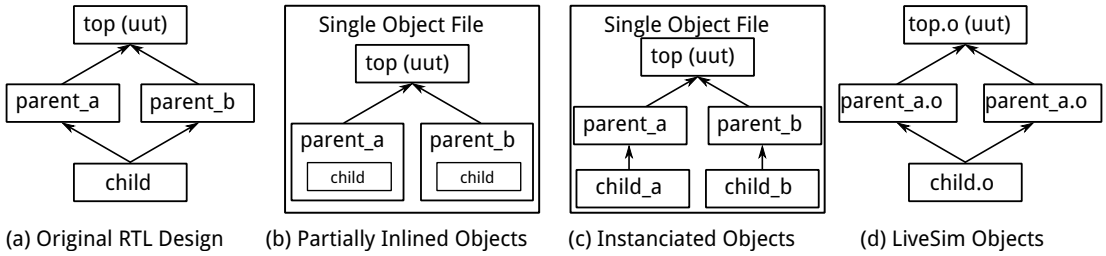

Fig. 4. LiveSim reduces the amount of code that needs to be compiled by reusing compiled modules across all the instances. Also, the use of shared library enables the ability to partially HotReload functions when changes are inserted in the code.

TABLE I
LIVESIM COMMANDS. (STAGE/TB)-HANDLE VARIABLES REFER TO THE HANDLE COLUMN ON TABLE II. STAGE-NAME VARIABLES REFER TO THE CORRESPONDING COLUMNS IN TABLES III AND IV.

Syntax	**Description**
ldLib *name*, *path*	Load shared library. Add entries to Table II for all object-paths found
instPipe *name*, *pipe-handle*	Instantiate a pipe and bind it to "name"
instStage *pipe-name*, *stage-name*, *stage-handle*	Instantiate a stage from *stage-handle*, bind it to *pipe-name* and *stage-name*
copyPipe *new-name*, *old-name*	Copy a pipeline, including its state
run *tb-handle*, *pipe-name*, *cycles*	Run a testbench on a pipe for a given number of cycles
chkp *pipe-name*, *path*	Take a checkpoint of *pipe-name* and save the state to *path*
ldch *pipe-name*, *path*	Load the state from a checkpoint into a pipeline
swapStage *pipe-name*, *stage-name*, *stage-handle*	Replace a stage in a given pipeline with a new instance

TABLE II
THE OBJECT LIBRARY TABLE CONTAINS A LISTING OF ALL THE OBJECTS LIVESIM COULD FIND IN THE SHARED OBJECT LIBRARIES ON ITS PATH.

Handle	Type	Code-Path	Object-Path
pipe0	Pipe	`/projects/src/toplevel.v#core`	`/livesim/objs/.../libc0.so#core`
stage0	Stage	`/projects/src/adder.v#adder`	`/livesim/objs/.../libc0.so#adder`
stage1	Stage	`/projects/src/sadder.v#sadder`	`/livesim/objs/.../libc0.so#sadder`
tb0	Testbench	`/projects/tests/tb1.cpp`	`/livesim/objs/.../libtb0.so#tb1`

TABLE III
THE PIPELINE TABLE. CONTAINS A NAME TO POINTER REFERENCE FOR EACH PIPELINE OBJECT ACTIVE IN THE SESSION.

Name	Handle	Pointer
p0	pipe0	0x...
p1	pipe0	0x...

TABLE IV
THE STAGE TABLE. CONTAINS A POINTER REFERENCE FOR EACH STAGE OF EACH PIPELINE.

Pipe Name	Stage Name	Handle	Pointer
p0	s0	stage0	0x...
p0	s1	stage1	0x...
p1	s1	stage1	0x...

C. LiveParser and LiveCompiler

Being able to quickly translate changes in code to updated simulation results is the primary function of a live programming backend. The first step in this flow for LiveSim is the LiveParser. The LiveParser identifies which stage the change in code took place in, and confirm that actual behavior was changed, not just comments or spacing. LiveParser then extracts those sections of the codebase and sends *only those* to LiveCompiler for re-compilation.

Since in Verilog, parameters are decided for each instance at compile time, to fully determine whether a module changed, it is necessary to check whether each instance of that module has a different set of parameters. Pre-processor directives also need to be checked. Other newer HDLs like CHISEL and Pyrope, which do type resolution, also allow programmers to make local changes which affect the global state.

LiveParser divides the code into regions based on the module structure, and the locations of pre-processor directives. Once it has confirmed a behavioral change in a code region, it must identify every region of the codebase which *may* have been affected by those changes. For Verilog, if it is just a change within a module, the affected parts of the system are all the module instantiations. For pre-processor directives, this could affect any code "below" the affected lines in the source code, thus much more will have to be recompiled.

When deployed LiveSim, compilation should happen on a different execution thread than the simulation and editor interface, or even in different servers. If not doing so, the compiler may end up re-compiling code only to determine that many of those modules were ultimately not changed. Whenever it compiles a new version of a stage, pipeline, or testbench, it compares the output against a cached copy to determine if this new version needs to be swapped into the simulation. These decisions must take place off of the editor and simulation execution threads. The process of detecting a

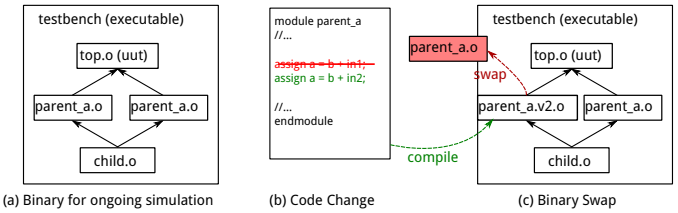

Fig. 5. During the live, or interactive phase, the compiler detects code changes and recompiles only the modules that were affected by the change. This is made easier by the use of modular shared libraries. Finally, the shared library can be swapped on the fly and simulation continues from a checkpoint.

code change, recompiling modules that need re-compilation, and swapping the binaries is illustrated in Figure 5.

Some newer HDLs, like Chisel and Pyrope, employ type resolution. Compilation can be sped up by keeping the old type state in memory, but allowing it to be updated based on new code. This can significantly reduce the number of resolution cycles needed on subsequent compilations.

D. Fast Reloading

Once the simulation has been patched with the latest changes, the results to the user must be updated. The algorithm LiveSim uses is designed to provide a fast estimate of the updated simulation, and check/refine that estimate in the background, updating if necessary. The reloading mechanism is also tuned for the ERD loop, where the developer edits a line of code, runs the simulation, and debugs the result. In other words, the reload algorithm is tuned to the use case of fixing an error observed in a simulation.

LiveSim first reloads the checkpoint that is closest to 10K cycles before the stopping point (this parameter is tunable). From the literature we have found on the topic of hardware debugging [28] and talking with experts in the area, this distance in cycles between bug and detection is often not that large, rarely more than a few thousand cycles. LiveSim reloads the state from that checkpoint and reruns the simulation. This result is reported immediately to the user as an estimate of the correct, updated simulation result.

The estimate may be inaccurate because the changes to the pipeline or testbench behavior which caused the update may also cause the system state to diverge at an earlier point in the simulation. To check for this, we duplicate the newly patched pipeline and/or testbench and rerun the earlier checkpoints in parallel, to see if the states diverge at an earlier point in the session history. If so, update the final results as necessary. This allows LiveSim to provide fast updates to the user and hide as much of the computational effort on the backend as possible.

E. Reloading Rules

A checkpoint consists of the entire state of the pipeline object. In some cases, when the user makes changes to the system, they may add, remove, or rename registers, making it impossible to blindly transfer checkpoints between iterations of the same system.

LiveSim uses standardized rules to enable patched pipelines to deterministically load checkpoints from previous versions of the system, shown in Table V. Since updates should happen frequently, ideally once per second as the user types, these cases can handle a large percent of the update batches without user intervention. Take register renaming for example, if there have been too many changes for LiveSim to unambiguously determine the register mapping between two checkpoint versions, then it will make its best guess based on the similarities of names and types. The user can manually edit the *Register Transform History* if the mapping is incorrect.

The *Register Transform History* maps the transformations applied to the architecture's registers in order to translate the checkpoints of one version into those of another. An example of such a history is shown in Table VI. The table is designed to support branching so that developers are not limited to a linear sequence of changes when exploring the design space.

F. Checkpoint Consistency Verification

One thing to note is that after a code change, it is possible that the checkpoints generated with the old code may be invalid. That could be caused by a divergence in the execution, *i.e.*, a different condition was triggered somewhere throughout the simulation, or simply because the logic of the architecture was reworked. In any case, since LiveSim starts the simulation from a saved checkpoint, it is necessary to make sure that the checkpoint is a valid state for the given input set of the simulation. This requires re-running the simulation from the beginning, which is the opposite of what we are trying to achieve.

Instead, this consistency verification can be done from checkpoint to checkpoint, seeing if the change in state between two checkpoints with simulation post code change matches the change in state with the prior version. LiveSim can restart the simulation once we find the earliest point of divergence between the checkpoint deltas.

Since each checkpoint is now independent of other results, this operation can be easily made parallel and can scale to a large number of cores (as many as checkpoints before the current cycle). In general, if n cores will be used, the best strategy would be to divide the whole simulation into up to $n-1$ parts with roughly the same number of checkpoints in each. The last core is assigned the reload from the last checkpoint and executes up to the current simulation cycle. For instance, for 1B cycles in a 4 core, 3 cores will execute roughly 333M cycles, while the remaining core will execute the cycles from the last checkpoint up to the last cycle, feeding those results back into the debugging process. Figure 6 illustrates how the consistency verification works. In cases where the checkpoints are not consistent, this approach allows for identifying at which checkpoint the divergence occurred, which may also be useful for debugging.

G. A Final Note on Usage

This section describes the internal design of LiveSim, specifically how it internally optimizes the structure of the simulation, and allows for fast checkpointing and hot reloading. This requires significant infrastructure, and a mechanism to

TABLE V
IF THE REGISTER NAMES DO NOT COMPLETELY MATCH FROM VERSION TO VERSION, TRANSFORMATION RULES ARE APPLIED TO PROVIDE A DETERMINISTIC WAY TO LOAD CHECKPOINTS FROM THE OLD VERSION TO THE NEW ONE.

Scenario	Action
Register created	Initialize to 0 (or other value)
Register deleted	Ignore data from checkpoint
Single Register renamed	Map old-name to new-name in the checkpoint

TABLE VI
THE *Register Transform History Table* LISTS THE CHANGES TO AN ARCHITECTURE'S REGISTER TOPOLOGY REQUIRE TO TRANSLATE THE SYSTEM'S STATE FROM ONE VERSION TO THE NEXT. IT IS STRUCTURED TO ALLOW FOR BRANCHING.

Version	Operations	Parent
1.1	`create newR`	null
1.2	`create newR1` `rename someR, newR`	1.1
1.3	`delete otherR` `rename newR1, myR1`	1.2
1.3a	`delete newR`	1.2

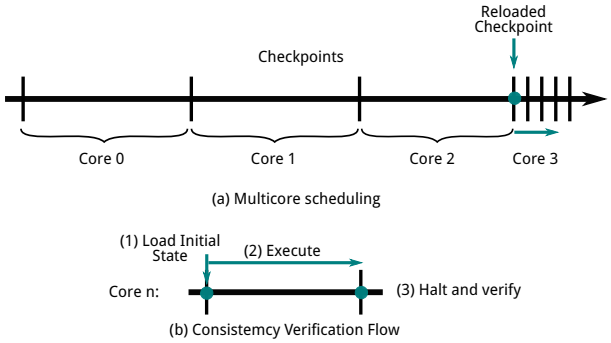

Fig. 6. After a code change, it is important to verify the consistency across the checkpoints. This can be done in parallel and is easily scalable to a large number of cores.

load both changes in code and changes in state in a completely modular way.

The primary goal of this project is to provide developers with more flexibility and faster feedback to allow them to explore and evaluate different design choices, there is no way around the fact that developing with LiveSim will require some consideration from the developer as well. Like all tool developers, we can only hope that the advantages of using our system outweigh the challenges.

The register renaming mechanic discussed in Section III-E is necessary to allow fully-modular checkpoint reloading, a key component of how LiveSim can reflect code changes in seconds rather than having to rerun the entire simulation from the beginning. It means changes in register names and sizes need to be more deliberate, and that it is possible that the developer may need to manually manipulate the history table to reflect the changes in state properly. We are aware of this downside, but still hope that the advantages of LiveSim outweigh it.

When LiveSim is fully deployed, it runs as a multi-process distributed system with packets moving back and forth containing code changes, compiled modules, and metadata (Figure 3). Depending on the network and computing resources of the underlying system, introducing a huge amount of code changes all at once has the potential to saturate the system and significantly degrade performance. We imagine that developers using LiveSim will adjust to making more incremental changes to the design, this is reflective of many developers workflow, especially when hunting for bugs.

As of this writing, we have not had the opportunity to run extensive user tests with full deployments of LiveSim. When we are able to, we will have more insights on the user experience, and what sort of interface will best allow a developer to leverage LiveSim.

IV. SETUP

In this section, we discuss the evaluation setup for LiveSim, the benchmarks used in the evaluation, and the infrastructure used for both LiveSim and Verilator [29]. We also discuss optimization options and how they were selected, since this ended up having an important impact on the results observed.

All evaluations were performed in an Archlinux 4.3.3-3 server, with a Intel i7-6700K, 32GB of ram. We used Verilator version 3.9 [29], and g++ 7.3 for all the simulations.

As benchmark, we use a partitioned global address space (PGAS) of RISC-V cores, in sizes: 1x1, 2x2, 4x4, 8x8, and 16x16. Each PGAS node is a 5-stage RISC-V RV64i core with 32K internal memory. The nodes are arranged in a mesh, with each core having a memory address mapped section of the global address space. The address decides whether the memory operation is local or remote. This architecture resembles Parallella [30] and Celerity [31] which are also a multicores that don't employ cache coherence. We looked for code changes in the core GitHub repository to replicate changes actually made in the core and apply them to the code.

The way this architecture was designed, each of the five stages of the RISC-V cores is placed in a module, which is instantiated by a single top-level parent, which is also its own module. LiveSim places each of these modules in its own shared library. Thus, all of the PGASs simulations are made up of seven shared libraries: 5 for the stages, 1 of the top-level which instantiates them, and 1 for the testbench. These libraries are used to create instances of the objects they contain, so for the 16x16 PGAS, there will be 6 instances for each of the 256 nodes in the mesh.

Both LiveSim and Verilator generate C++, all testbenches were compiled with g++ using the same O2 optimization flag.

Verilator was unable to produce a C++ implementation for the 16x16 PGAS even after 24 hours of compilation time. We tried many Verilator options like trying to reduce optimization,

"output-split" and "inline-mult" manual selection, and even clang instead of g++. In all the cases, we could not finish the compilation in less than 24 hours. With regards to the "–inline-mult" option, we found there was no difference in the generated code regardless of the value we set it to. The documentation states that this option is ignored for "very small modules", though these modules are approximately 500 lines of Verilog, and we clearly show in Table VII that the I-cache is being thrashed by this simulation, indicating that the inline is being performed.

V. Evaluation

We start our evaluation presenting the overall results, *i.e.*, the simulation speed when all the LiveSim features are active. Then, we dissect how the speed-ups were obtained and look into the overheads of Hot Reload and checkpointing to the overall simulation.

A. Simulation Efficiency

Figure 7 shows the overall compilation and simulation speed (ERD loop) of LiveSim and Verilator. The plot shows a few different scenarios, first complete simulation (from cycle 0 to x) for Verilator and LiveSim for different PGAS sizes (1x1 to 16x16), we normalized the cycle count by the number of cores in the simulation. The y-axis shows the time to reach that point in the simulation. Verilator and LiveSim start at different y coordinates depending on the compilation time (Table VIII). The slope of the lines comes from the simulation speed. The faster the simulation, the lower the slope. In the best case for LiveSim, Hot reload is used and the simulation starts from a checkpoint. In this case, the ERD loop is of $\approx 2s$ from code change to reach virtually any simulation cycle count.

As an example, LiveSim has a faster starting point for the 4x4 PGAS, Verilator needs 63 seconds to start (vs 2 seconds for LiveSim). LiveSim also has a smaller slope because it is faster. While Verilator achieves 718 KHz, LiveSim runs at 1223 KHz. This is over 1.7x speedup. As a result, to run something 10 million cycles, Verilator needs 257 seconds, and LiveSim needs 109 seconds. The Verilator simulation seems faster on small designs but at larger compile time. For the 1x1 PGAS, Verilator only passes LiveSim after running 76 million cycles. As the design gets more complex, LiveSim gets faster even when no checkpoints are available.

To understand the simulation speed difference between Verilator and LiveSim we gathered performance counter statistics. Table VII shows a breakdown of performance statistics for each PGAS size. Here we can see the source of the performance divergence observed in Figure 7. Verilator's cache performance, particularly for the I-cache, degrades significantly for the PGAS 4x4 and larger, causing a drop in IPC. For small designs, the Verilator design fits in the host instruction cache, but as more cores are added, the I$ pressure makes the simulator very slow. The difference can be explained by the fact that LiveSim does not replicate code for multiple instances of the same block, reducing code bloat. Inlining the entire system is more efficient for small designs because of superior data locality and better cross module optimization, but once the simulation is large enough to saturate the caches, then that quickly becomes a bottleneck.

LiveSim also has a better data cache. The LiveSim C++ code generator employs a simple algorithm which groups muxes with the same condition into $if - else$ block wherever possible. The result is an increase in branches (reflected in the "BR MPKI" row) and a decrease on cache conflicts, the latter only being present on large designs. This optimization does not yield benefits for designs that fit in the data cache. As the design gets larger, however, the branch miss prediction rate increases, but the rate of increase of data cache Misses Per Thousand Instructions (MPKI) is slower than in Verilator, which ultimately has a greater impact. This optimization does seem to have a limited impact because with a large enough design under simulation the data set will not fit in the cache regardless.

Though LiveSim underperforms Verilator on smaller designs, its internals provide better scalability. The result is that going from 1 core to 256 (16x16) LiveSim has just a 40% performance degradation. We were unable to compile the 16x16 PGAS at all with Verilator, but going from 1 to 64 PGAs has 70% performance degration. As a reference, LiveSim has just a 25% degradation for this data point.

B. Hot Reloading and Checkpointing Overhead

Figure 8 shows the main latency of the ERD loop for each size of the PGAS. 1x1 is a mesh with a single PGAS core, the 16x16 is a mesh with 256 PGAS. Each runtime includes compilation after a code change and the time it took for the simulation to reach a specific number of cycles denoted in the x-axis. The code changes were taken from actual bug fixes made during the development of the PGAS CPU cores, taken from the git logs.

LiveSim provides an ERD loop of less than 2 seconds in all the cases, even for the 256 core case (256 PGAS). This is because the PGASs are made by replicating the same RISC-V core, making a change to one core necessitates doing a swap on every core in the mesh. Moreover, replacing one or more pipeline stages has a small impact in the ERD loop, despite the fact that the number of stages being swapped is growing exponentially. This is due to the fact that the latency of the ERD loop is dominated by other parts of the flow, specifically LiveParser and LiveCompiler, which only happen once regardless of the size of the mesh.

Also, there is very little variation independent of the bug or pipeline stage being handled. All these bugs affected a single pipeline stage. Not shown in the figure, but if two pipeline stages are affected, the time increase is linear. Since LiveParser only has to do a minimal amount of work, its execution time was negligible for all benchmarks, less than 20ms. This is not so surprising considering the process of hot patching a stage in detail. Once we have the shared library compiled and loaded into memory, a process which only has to be done once, LiveSim calls a method from the library which creates the new stage object, and copies the register values from the old one to the new one (taking into account any which have been added, removed, or renamed). For a single core, the size

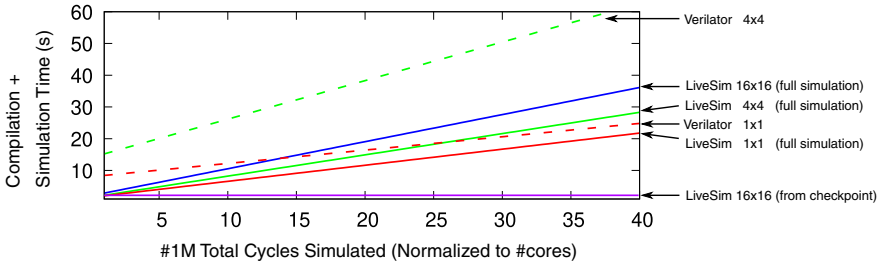

Fig. 7. LiveSim has a faster compile time and, for large designs, simulation time than Verilator, reducing the ERD loop. When restarting from checkpoints, the advantages is even greater.

TABLE VII
THE SPEEDUP OF LIVESIM COMPARED TO VERILATOR COMES MOSTLY FROM THE INSTRUCTION CACHE EFFICIENCY. THE SMALLER CODE FOOTPRINT ACHIEVED DUE TO REUSE OF CODE GIVES ORDERS OF MAGNITUDE REDUCTION IN CACHE MISSES FOR DESIGNS WITH A LARGE NUMBER OF INSTANCES.

	PGAS 1x1		PGAS 2x2		PGAS 4x4		PGAS 8x8		PGAS 16x16	
	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator
KHz	1974	2378	1957	2351	1492	823	1223	718	1172	NA
IPC	2.50	2.86	2.47	2.71	1.94	0.96	1.62	0.80	1.24	NA
I$ MPKI	0.25	0.01	0.02	8.45	0.01	20.57	0.01	24.37	0.01	NA
D$ MPKI	0.96	0.01	10.78	0.01	36.42	84.04	39.09	42.99	48.08	NA
BR MPKI	1.54	0.01	1.33	0.01	2.83	0.01	3.62	0.29	4.27	NA

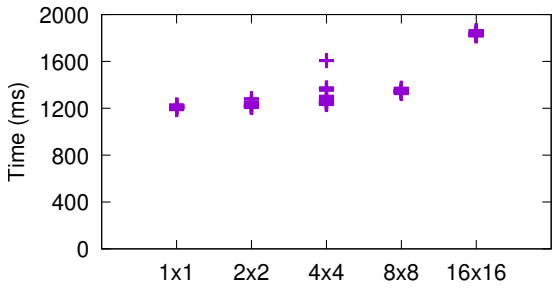

Fig. 8. The time required to hot reload and update the simulation remains under 2 seconds, even for a large mesh with 256 PGAS multicore (16x16). Swapping latency is dominated by the parsing and compilation costs which remain constant with respect to the number of instances of the modified module.

of all their internal memories (2x32 KB) and the register file (32x64 bits) comes to under 100 KB. The cost of copying that, even 256 times, is still eclipsed by other parts of the incremental compilation process.

TABLE VIII
COMPILATION TIME FOR LIVESIM AND VERILATOR. HOT RELOAD CAN SWAP A MODULE IN UNDER 2 SECONDS. EVEN FULL COMPILATION WITH LIVESIM IS FASTER THAN WITH VERILATOR.

	1x1	2x2	4x4	8x8	16x16
LiveSim Hot Reload	1.5	1.5	1.5	1.6	2
LiveSim Full	4.9	4.7	4.8	15.6	176
Verilator	8	14	63	327	NA

We compare the incremental compilation with Hot Reload in LiveSim with a full compilation flow in Verilator, since that would be the standard mode of work in the later simulator. Table VIII shows the compilation times for LiveSim and Verilator. As previously stated, LiveSim Hot Reload achieves a ERD of less than 2 seconds. A clean full compilation with LiveSim is slower. It goes from 4.9 seconds in a trivial single core to 176 seconds for a 256 PGAS. Verilator is always slower compiling, and we were not able to compile the 256 PGAS in Verilator in under 24 hours even though we tried clang, different verilator options, and even removed compile optimizations. The summary is that for full compilation LiveSim has a higher starting overhead to partition the design, but it scales better because it shares code between pipeline stages.

To evaluate the overhead of checkpointing, we measured the simulation speed of LiveSim with and without checkpointing enabled. The process of taking checkpoints, as discussed in Section III-D, requires stopping the simulation, copying the internal state, which for the 256-core PGAS is ¡3MB, and continuing. The overhead depends on the design under test and in our experiments varied from 10 to 20%. Although high, checkpoints allow LiveSim to quickly get back to a specific point of the simulation without the need to go back from cycle 0. Disabling checkpoints is possible for long runs in full system verification, *i.e.*, outside of the ERD loop.

VI. CONCLUSION

Hardware development is plagued by notoriously long compile and simulation times which forms a bottleneck to productivity as developers spend much of their time waiting for feedback. There are a variety of software design tools and libraries which have embraced the idea that faster feedback is better for developers and have implemented a variety of features aimed at providing it. We believe its time for hardware design tools to do so as well.

LiveSim is the first work that tries to address the gap in productivity in hardware simulation. LiveSim pushes the state of the art to allow under two seconds turnaround time for compilation and simulation of complex designs are a large number of cycles. On compilation, LiveSim relies on

efficiently partitioning the design to allow for incremental compilation. Whereas during simulation, LiveSim relies on reuse of code across instances to reduce code bloat and checkpointing to reduce simulation time.

The evaluation shows that LiveSim compiles faster than Verilator, and it also has a faster simulation time for larger designs. Perhaps even more significant, we demonstrate a 2 second edit-run-debug (ERD) loop for even large scale SoCs and a large number of cycles.

LiveSim hot reload is a new way to interact with hardware simulators. While current flows are slow and require the user to be very careful about the code changes when debugging, hot reload opens the opportunity to create new ways of thinking. For example, since hot reload is fast, the designer can insert "printfs" and replay from any given point with very low overhead. Similarly, the designer can explore very fast "what ifs". Another interesting feature for traditional flows, hot reload allows for hiding costly initialization. If there is a slow reset and program load in a design, hot reload can start afterwards.

ACKNOWLEDGMENTS

We like to thank the reviewers for their feedback on the paper. This work has been made possible by the Center for Research in Open Source Software at UC Santa Cruz (cross.ucsc.edu), which is funded by a donation from Sage Weil and industry memberships This work was also supported in part by the National Science Foundation under grant CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] H. Qian and Y. Deng, "Accelerating rtl simulation with gpus," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2011.

[2] S. Banerjee and T. Gupta, "Optimized simulation acceleration with partial testbench evaluation," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014.

[3] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, and Y. Liu, "Functional verification of large asics," in *Proceedings of the 35th Annual Design Automation Conference*, ser. DAC '98. New York, NY, USA: ACM, 1998. [Online]. Available: http://doi.acm.org/10.1145/277044.277210

[4] 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 *DAC Design Automation Conference 2012*. IEEE, 2012.

[5] D. Lockhart and C. Zibrat, Garyd Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *Microarchitecture, Proceedings of the 47th Annual IEEE/ACM International Symposium on*, ser. MICRO'14. Washington, DC, USA: IEEE Computer Society, Dec. 2014.

[6] (2016) The 2016 Wilson Research Group Functional Verification Study. Mentor Graphics. Santa Clara, CA. [Online]. Available: https://blogs.mentor.com/verificationhorizons/blog/2016/08/29/part-3-the-2016-wilson-research-group-functional-verification-study

[7] R. T. Possignolo and J. Renau, "LiveSynth: Towards an interactive synthesis flow," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017.

[8] R. E. Barber and H. C. L. Jr., "System response time operator productivity, and job satisfaction," *Communications of the ACM*, vol. 26, 1983.

[9] J. Dabrowski and E. V. Munson, "40 years of searching for the best computer system response time," *Interacting with Computers*, vol. 23, 2011.

[10] G. N. Lambert, "A comparative study of system response time on program developer productivity." *IBM Systems Journal*, vol. 23, 1984.

[11] R. T. Possignolo and J. Renau, "SMatch: Structural matching for fast resynthesis in fpgas," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019.

[12] Verilator 4.018, Open-source tool for Verilog HDL simulation. https://www.veripool.org/wiki/verilator. Online; accessed on 5 September 2019.

[13] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 2013.

[14] X. Zhang and P. J. Guo, "Ds. js: Turn any webpage into an example-centric live programming environment for learning data science," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017.

[15] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, "Pharo by example. square bracket associates, 2009," *URL http://pharobyexample. org*.

[16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016.

[17] "Treadle – a chisel/firrtl execution engine," https://github.com/freechipsproject/treadle, accessed 2020-02-02.

[18] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog," *Architectural Support for Programming Languages and Operating Systems*, 2019.

[19] "Introducing hot reloading - react native," https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html, accessed: 2018-04-06.

[20] S. Alhazbi and A. Jantan, "Multi-level mediator-based technique for classes hot swapping in java applications," in *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, vol. 2. IEEE, 2006.

[21] D. Kim, M. Ciesielski, K. Shim, and S. Yang, "Temporal parallel simulation: A fast gate-level hdl simulation using higher level models," in *2011 Design, Automation Test in Europe*, 2011.

[22] D. Kim, M. Ciesielski, and S. Yang, "Multes: Multilevel temporal-parallel event-driven simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2013.

[23] K. Shim, Y. Cho, N. Kim, H. Baik, K. Kim, D. Kim, J. Kim, B. Min, K. Choi, M. Ciesielski, and S. Yang, "A fast two-pass hdl simulation with on-demand dump," in *2008 Asia and South Pacific Design Automation Conference*, 2008.

[24] T. B. Ahmad and M. J. Ciesielski, "Fast sta prediction-based gate-level timing simulation," *2014 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[25] C. Zilles and G. Sohi, "Master/slave speculative parallelization," *35th International Symposium on Micro-Architecture (MICRO)*, 2002.

[26] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "Boom v2: an open-source out-of-order risc-v core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[27] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017.

[28] S. Sudhakrishnan, L. Su, and J. Renau, "Processor verification with hw-bughunt," in *ISQED '08: Proceedings of the 9th international symposium on Quality Electronic Design*. Washington, DC, USA: IEEE Computer Society, Mar. 2008.

[29] Verilator 3.9, open source tool for verilog hdl simulation. Online Webpage. https://www.veripool.org/wiki/verilator.

[30] Adapteva Inc., "Parallella-1.x reference manual," 2014.

[31] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Lotfi, J. Puscar, A. Rao *et al.*, "Celerity: An open-source RISC-V tiered accelerator fabric," in *A Symposium on High Performance Chips (Hot Chips 29)*, Aug. 2017.