Crafting a Million Instructions/Sec RISCV-DV
HPC Techniques to Boost UVM Testbench Performance by Over 100x

Puneet Goel <puneet.goel@incoresemi.com>, Incore Semiconductors
Ritu Goel <ritu@coverify.com>, Coverify Systems Technology
Jyoti Dahiya <jyoti@coverify.com>, Coverify Systems Technology

Abstract

The advent of opensource hardware, epitomized by the RISC-V processor, is opening up the avenues for opensource hardware verification tooling and flows. RISCV-DV [1] is an opensource random instruction generator widely used for functional verification of RISC-V cores [2]. This paper expounds on a parallelized RISCV-DV port [3] coded in Embedded UVM (eUVM). Novel techniques for enhancing testbench performance are also explored to affect a multicore UVM testbench implementation that generates millions of constrained-randomized RISC-V instructions in a second (using multicore parallelism), a speedup of over 100x when compared to the original RISCV-DV implementation coded in SystemVerilog (SV) UVM.

I. Introduction

A. The Curious Case of Processor Verification

High-end RISC CPU cores encompass complex hardware architectures comprising intricate maneuvers like instruction re-ordering, pipelines, branch prediction, and hyperthreading. Functional verification of such processors requires a significant effort involving generation of 10^{15} (thousands of millions of millions) constrained-random instructions [4]. The RISCV-DV project, coded in SV/UVM, generates only about 10,000 instructions in a second. At this rate, it takes several thousand machine years just to generate the required number of sequences of randomized RISC-V instructions.

B. The Era of High Performance Computing

Thanks to Dennard's Scaling [5], CPU frequency and performance grew exponentially until the year 2005. For verification engineers, Dennard's Scaling proved to be a boon. During the period of its impact, any increase in complexity of hardware designs (owing to Moore's Law) was offset by a corresponding increase in CPU frequency and performance, thus affecting proportionately faster simulation runs.

Dennard's Scaling continued to hold its sway until the year 2005. Post that, its impact started to wane. That very year, Herb Sutter wrote a seminal paper titled *"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"* [6]. Henceforth, software developers could not rely on an increase in processor frequency as a means to improve software efficacy. Programmers were now required to code multicore-enabled concurrent programs in order to boost software performance, marking the start of the era of High Performance Computing (HPC).

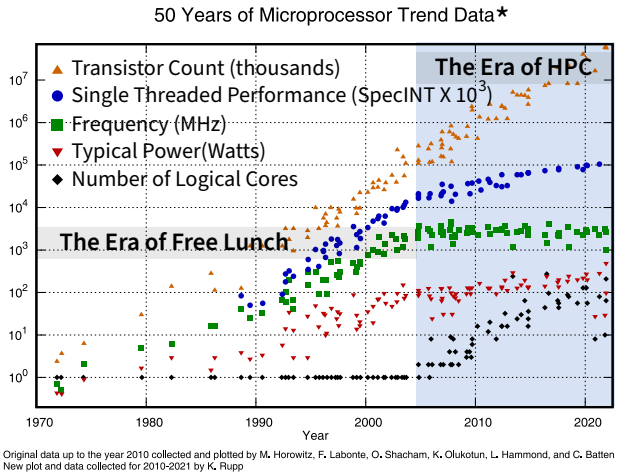

Fig. 1: CPU Performance Over the Last 50 Years

C. The Elephant in the Room

Testbench performance is proving to be the proverbial elephant in the room. Even as modern HDL Simulators enable parallelized multicore simulation of RTL designs [7], little has been done to enable multicore parallelization of testbenches (refer section II-A), forcing verification engineers to adopt pragmatic techniques to enhance testbench efficacy. A distributed stimulus generator architecture that invokes Inter-Process Communication (IPC) for the exchange of generated transactions between multiple simulator instances (running as separate Linux processes) is explored in [8]. A multicore predictor architecture that utilizes a C++ thread pool via Direct Programming Interface (DPI) is realized in [9].

Contemporarily, the emergence of hybrid CPU/FPGA architectures and SoCFPGAs is enabling co-emulation platforms [10], wherein the Design under Test (DuT) gets mapped to an FPGA, but the testbench still continues to run on an HDL simulator. A "transaction-based acceleration" approach that proposes "an untimed hardware verification language domain" for accelerating the testbench for co-emulation has been broached in both [11] and [12].

Incidentally, SV lacks support for Transaction Level Modeling (TLM) [13] at the very fundamental level. An essential tenet of TLM requires temporal decoupling of data from simulation time and events. All integral variables in SV (including the two-state types such as `byte`, `shortint`, `int`, and `longint`), and the expressions thereof, implicitly hold a value-change event that enables the end-user to write *wait* expressions (such as `wait(a > b)`). As a result, any computational algorithm coded in SV executes an order of magnitude slower than the corresponding code in C/C++ and any other native programming languages. Section IX-I reflects further on the impact of non-native datatypes on the algorithmic performance of SV testbenches.

II. The State of the Art

A. SystemVerilog/UVM

The golden reference RISCV-DV instruction generator implementation is coded in SV. But complex constraint solving and sub-optimal algorithmic implementation (refer section VI for details) limits the speed of its execution to only about ten thousand instructions in a second. Additionally, SV lacks native data-types. Consequently, any interface to an emulation platform requires interfacing with C/C++ via the DPI layer. Reference [14] offers a detailed exposition on how the runtime overhead of data exchange via DPI can have an adverse impact on testbench performance.

From the HPC perspective, tool-level support for multicore parallelism is limited to RTL and Gate-Level simulations and requires Design Level Partitioning (DLP) [15]. Figure 2 depicts a typical verification setup with a partitioned RTL design as DuT. Since RTL coding semantics support only static variable scoping, multicore-enabled SV simulators are able to identify static RTL segments that can be simulated concurrently on multiple threads. Note that the exchange of signal data across the partition boundaries happens only by value (not by reference) and only via pins and ports that can be formally identified by the simulation tool. Therefore, it becomes possible for the simulator to introduce appropriate synchronization locks when passing data across the threads.

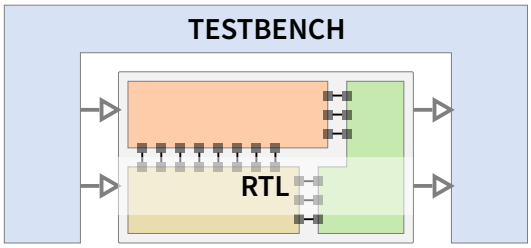

Fig. 2: Multicore RTL Simulators

On the other hand, a testbench is a completely different beast. A testbench is essentially behavioral, enables automatic data scoping, and allows sharing of data by reference across its various components. To this end, enabling parallelism in a testbench requires user-level programming language constructs [16] that facilitate synchronized access to shared data, which the current SV language standard lacks.

Note that, though a testbench is always encapsulated in a separate module (or a program block), events corresponding to the testbench are temporally staggered from those pertaining to the DuT in order to avoid race conditions [17]. Temporal staggering of the DuT signal from those of the testbench is generally accomplished by use of the clocking block construct and/or the program block construct. Consequently, tasks related to a testbench are never executed in parallel to the simulation tasks associated with the DuT. As a result, a sequentially executed single threaded testbench becomes a bottleneck in a UVM/RTL simulation.

B. Python - CocoTB [18] and PyUVM

Despite being proposed as a futuristic language for Verification [19], Python has a huge drawback from the testbench performance perspective. Being an interpreted scripting language, Python is inherently inefficient and has been benchmarked to be fifty-seven times slower in comparison to native programming languages such as C/C++ [20].

1) Pygen – The Python Port of RISCV-DV : Pygen is a python-based opensource port of RISCV-DV and is hosted as a sub-folder of the main RISCV-DV GitHub repository [1]. Pygen uses PyVSC [21] for solving constraints and is currently marred by poor runtime performance. Additionally, PyVSC does not deploy a Binary Decision Diagram (BDD) solver and relies completely on Satisfiability (SAT) solvers for solving multi-variable constraints. Consequently, the solutions produced by PyVSC tend to have a non-uniform spread compared to SV and eUVM. Pygen currently performs more than a hundred times slower when compared to the SV version of RISCV-DV and generates less than a hundred random RISC-V instructions in a second.

C. SystemC and C++

The SystemC port of the UVM library [22] currently relies on the CRAVE library [23] for constrained randomization. CRAVE provides an integrated interface to a set of BDD and SAT solvers. But as of now, it can not handle complex RISCV-DV constraints. Additionally, SystemC and CRAVE do not offer support for multicore parallelism.

The CRAVE library uses wrapper template types (*crv_variable* on line 2, in listing 1) to represent random variables. On account of that, random variables in CRAVE do not retain their native memory footprint, thus severely hindering its efficacy and prowess.

Listing 1: Example Randomizable Class in CRAVE

```
1 class myrand: public crv_sequence_item {
2   crv_variable<int> x;
3   crv_constraint constraint{x() < 32};
4   // ....
```

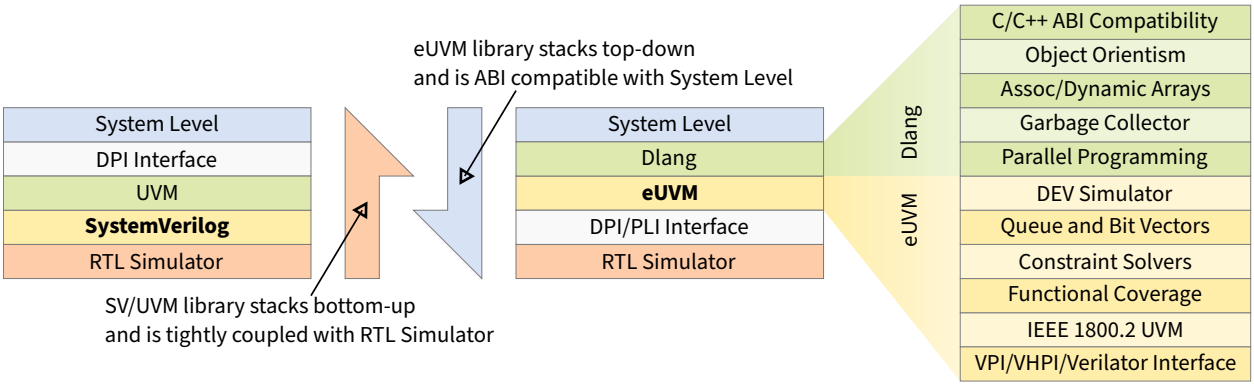

Fig. 3: eUVM and its Systems-Centric Testbench Approach

III. An Introduction to eUVM

eUVM is an opensource multicore-enabled implementation of Universal Verification Methodology (UVM) (IEEE 1800.2 1.0-2020 Standard) coded in the D Programming Language (Dlang) [24]. Like SV, eUVM features sophisticated constraint solvers, functional coverage constructs, and an automatic garbage collector (which it inherits from the D Programming Language).

Unlike the SV implementation of UVM which is tightly integrated with RTL simulation, eUVM takes a systems-centric top-down approach, as illustrated in figure 3. A testbench coded in SV can interface to the system-level only through the DPI layer, which may offer a significant performance penalty [14]. On the other hand, eUVM, having been implemented in the D programming language, enables Application Binary Interface (ABI) level compatibility with C/C++. Consequently, C/C++ functions can be directly called from eUVM testbenches, and vice-versa, without any runtime overhead.

Being opensource, another advantage of eUVM is its portability across the various operating systems and machine architectures. eUVM testbenches run on Windows, Android, Mac OS, and most versions of Linux. Additionally, eUVM testbenches can be cross-compiled for embedded system platforms (running either Linux or Android), thus making it feasible to execute eUVM testbenches on embedded boards, including SoCFPGA boards. Reference [25] contains a detailed exposition on how to render an SoCFPGA board into a mini co-emulation platform, that enables a completely portable stimulus across the co-emulation and simulation environments. Note that eUVM integrates with any RTL simulator via PLI/VHPI/FLI or DPI interface, and interfaces natively with Verilator.

eUVM uses a native constraint solver to handle elementary single-domain constraints. BDD-based solvers are used to handle multi-domain constraints of medium complexity. Convoluted constraints that can not be handled by BDD, are delegated to SMT/SAT solvers including Z3 [26], Boolector [27], and CMSGen [28]. eUVM exploits advanced meta-programming capabilities of the D language, such as User-Defined Attributes (UDA), Compile-Time Function Evaluation (CTFE), Code-Introspection, and Generative Programming, to realize a high-performance constraint processor. While Code-Introspection and CTFE enable eUVM to parse and analyze constraints at compile time, UDAs (syntactically akin to Python Decorators) allow eUVM to maintain a native memory footprint of the randomized data elements, thus enabling ease of data interoperability with C and C++. Note that all the constraint solvers integrated into eUVM are multicore-enabled.

A. Grokking Testbench Parallelism in eUVM

The UVM base class library needs a discrete event simulator in order to synchronize testbench tasks and processes. While PyUVM does not implement a simulator of its own, it piggybacks on an interfaced RTL simulator to schedule its python co-routines. On the other hand, UVM-SystemC [22] uses the underlying SystemC simulation engine to schedule tasks. Likewise, eUVM too implements its own discrete-event simulator to schedule testbench related tasks and events. As illustrated in figure 4, the eUVM simulator can be broadly partitioned into a *Task Executor* and a *Scheduler*.

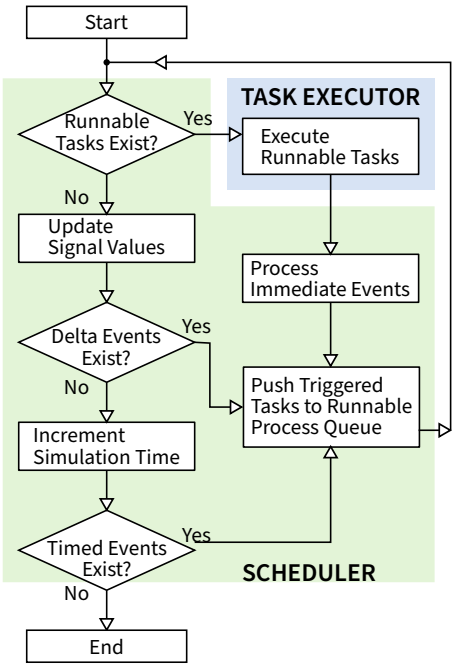

Fig. 4: eUVM Discrete-Event Simulator

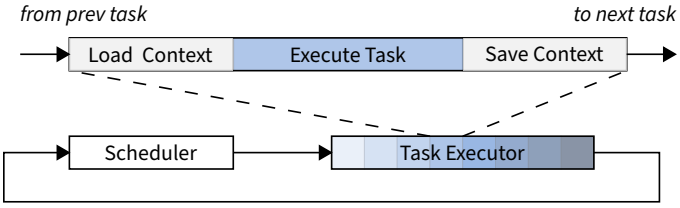

Fig. 5: A Discrete Event Simulator

Listing 2: UVM Sequence Body

```
override void body() {
  trans = new seq_item("trans");
  trans.addr = addr;
  // ..
  start_item(trans);
  finish_item(trans);
}
```

For the limited purpose of understanding the feasibility of parallelism in testbenches, figure 5 outlines a high-level abstraction of an event simulator. The scheduler manages event queues and schedules the triggered tasks for execution. Note that in a single-threaded simulator, the *task executor* executes runnable tasks sequentially on a single CPU thread on the host machine.

Before delving into a multicore-enabled parallelized testbench simulator, a little perspective into the sequential execution of tasks is in order. In the context of a UVM testbench, these tasks are nothing but the *run_phases* of the various UVM components, the *body* methods of the various sequences, and the *forks* that may have been spawned. Most simulators implement these tasks as user-threads (or cooperative threading).

Listing 2 outlines the *body* task of a typical UVM sequence. As the simulator encounters a call to any blocking function such as *start_item* or *finish_item*, the simulation control moves over to another testbench component such as the driver that transfers the *uvm_sequence_item* 'trans' to the interface of the DuT. At this point, the sequence *body* task goes to sleep (yields) only to be woken again once the driver has processed the sequence item and has sent back a notification to that effect by calling the *item_done* method.

An essential constituent of cooperative threading is *context switching* (as illustrated in figure 5). Whenever a task yields (waits for an event), it must relinquish the control over the CPU thread so that another scheduled task may execute. But before yielding, the task must save its state (comprising the call-stack and the CPU registers) in the host machine's memory so that it can recover from where it left, at a later stage when it wakes up again.

Note that *context switching* is a simulator's runtime overhead and does not accomplish any useful functionality related to the testbench. In order to optimize the performance, a testbench must avoid frequent context switching, by way of reducing simulation events, as explained in [11], [12].

B. VIP Level Parallelism – A Multicore Testbench Architecture

Figure 6 abstracts the functionality of a multicore simulator. A parallel simulator implements a stack of *Task Executors*, each of which gets its own CPU thread (posix thread) to execute its share of tasks. Synchronization Barriers are required to make sure that the task executors (now executing tasks on multiple threads) remain synchronized with the scheduler.

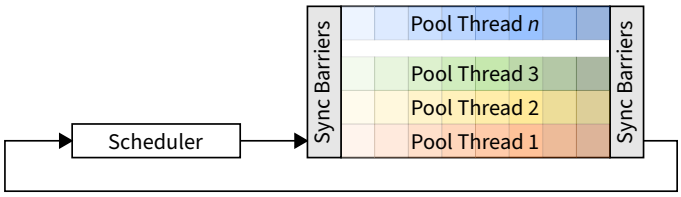

Fig. 6: eUVM's Multicore Simulator

For the performance aspect of multicore testbenches, it is imperative that we turn to Amdahl's Law (figure 7) [29], which states that *"the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"* [30]. Since, at a given simulation time, a testbench simulator would only deal with a small number of active events and processes, not much can be achieved by parallelizing the scheduler. In addition to the scheduler being a sequential component, there is yet another overhead in multicore testbenches in the form of synchronization barriers. Yet, if the tasks are comparatively more compute intensive, a multi-threaded testbench may bring about good performance gains.

Figure 8 depicts a typical testbench scenario that potentially gains from multi-threaded parallelism. A (sub)system-level testbench often has multiple *uvm_agents* or Verification IP (VIP)s [9]. Since it involves solving complex constraints, often the most compute intensive process in a testbench is the randomization of the sequences. To distribute sequence randomization across multiple threads, eUVM maps each *uvm_agent* on a separate CPU thread.

C. Sequence Level Parallelism Using Asynchronous Worker Threads

The simple multicore testbench architecture discussed in the previous section only applies to testbenches with multiple VIPs and therefore limits testbench parallelism to (sub)system-level verification. Moreover, the performance gain from such an architecture

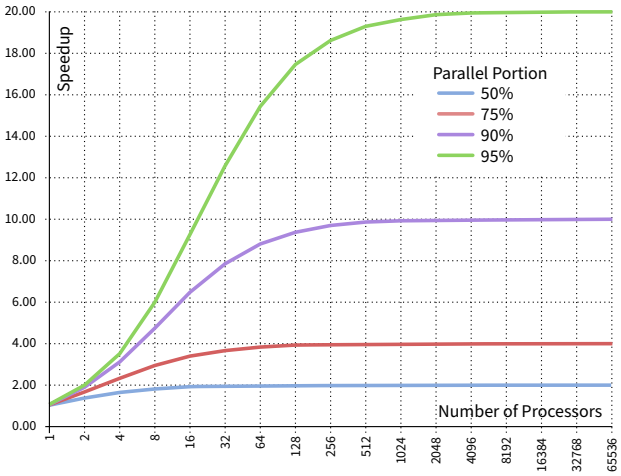

Fig. 7: Amdahl's Law

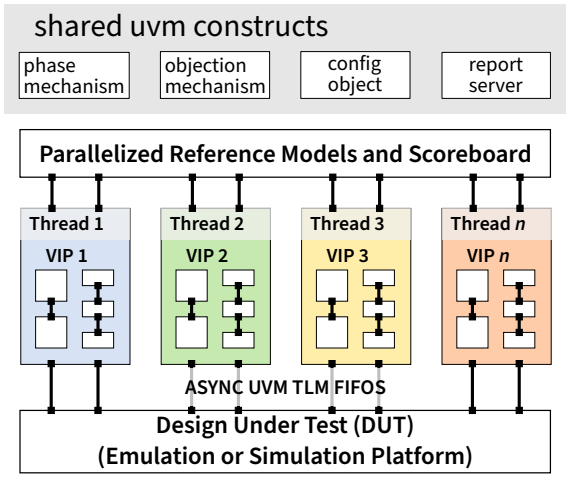

Fig. 8: VIP Level Parallelism in eUVM

may not be optimal. One issue is that when the scheduler gets active, all the task executors deactivate. Another issue is that for a high gain, the task load on each executor has to be balanced. That may not be the case when different types of VIPs get mapped to the various executors.

In this section, we take a look at *Sequence-Level Parallelization* technique that enables us to take advantage of multicore concurrency in simple module level testbenches that may constitute of only a limited number of UVM components.

Figure 9 exhibits a simple improvisation of the multicore simulator architecture discussed in the previous section. In addition to multiple task executors, the new simulator architecture allows free-running threads that are termed *worker threads* in eUVM.

Worker threads are free-running asynchronous threads, but are hierarchically owned by the simulator. Though owned by the simulator, a worker thread is completely decoupled from the scheduler and keeps running even when the scheduler activates. Meaning thereby, a worker thread can not wait for an event, though they can trigger events.

But how would a worker thread exchange data with synchronous simulator tasks? Though the worker threads too share memory with the tasks, to maintain synchronization across the threads, UVM stipulates that data exchanges happen via *tlm_fifos*.

Figure 10a outlines the basic architecture of a UVM TLM Fifo. When the fifo is empty, a read operation gets blocked by an explicit event encapsulated in the read port of the Fifo. A write operation into the TLM Fifo triggers the read event, thereby re-activating the blocked task. Similarly, when the TLM fifo is full, it is the explicit event associated with the write port, that blocks a write operation.

An asynchronous worker thread, being completely decoupled from the scheduler, does not have the ability to wait for a simulator event. We need to formulate a synchronization mechanism to make it possible for a worker thread to share data with other threads. To this end, eUVM implements three types of asynchronous TLM fifos as illustrated in figures 10b, 10c, and 10d. There are three use-case scenarios covered by the three async fifo architectures.

An *async write* (figure 10b) type of asynchronous TLM fifo is required when a worker thread is used for generating UVM transactions and the generated transactions are required to be transferred to a regular UVM task. When an *async write* TLM Fifo is full, the worker thread gets blocked by a software semaphore instead of a UVM event. When the receiving task pulls a transaction from the fifo, the pull method determines that a slot has become available. It then frees up the semaphore thus unblocking the waiting worker thread. On the other hand, when the fifo is empty, the receiver gets blocked by a regular UVM event associated with the read port. When the worker thread puts a transaction on the fifo, it also triggers the read event, thus unblocking the synchronous task on the receiving end. An *async read* TLM fifo handles the reverse scenario, where a worker thread is supposed to receive data from a regular UVM task.

Note that functionally asynchronous fifo constructs play the same role as played by IPC channels used in the *distributed sequencer* [8] discussed in section I-C. To begin with, the eUVM testbench architecture uses shared-memory parallelization semantics. Meaning thereby, a transaction (or its handle) generated on one CPU thread can simply be shared with other threads without any IPC, packing/unpacking, or DPI overhead that the *distributed model* stipulates. As a result, a verification engineer does not have to put extra effort for coding IPC and DPI related constructs. Moreover, since the communication between the threads follows a shared-memory model, the execution time required for handing over data from one thread to another is miniscule compared to a testbench architecture that deploys Inter-Process Communication.

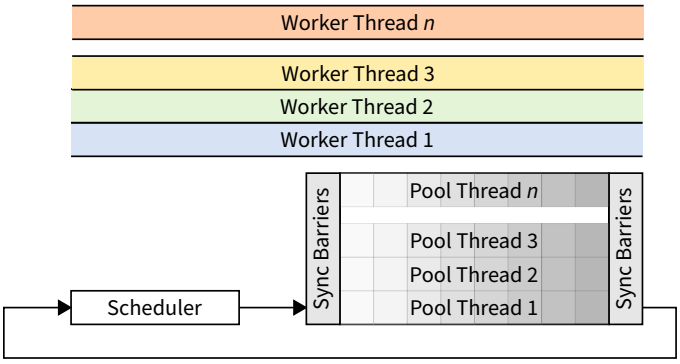

Fig. 9: eUVM's Multicore Simulator

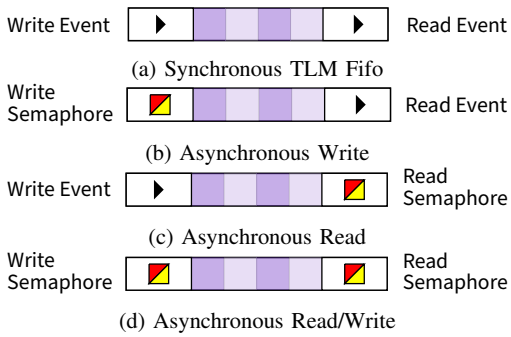

Fig. 10: eUVM TLM Fifos

eUVM also implements a completely asynchronous TLM fifo (as depicted in figure 10d) with a pair of semaphores replacing the guard events at both the read and the write ports. This kind of TLM fifos are meant for data exchanges between two asynchronous worker threads.

Worker thread constructs in eUVM make it possible to parallelize a UVM sequencer by creating a number of free-running worker threads. In the scheme illustrated in figure 11, these threads feed a virtual sequencer in a round robin fashion. This may be useful in a scenario where transaction generation and randomization is overly complex and time consuming and is choking an RTL simulation.

D. Fine-Grained Parallelism – A Multicore Parallelized Fork

The parallelization scenarios covered so far are useful only when the generation and randomization of each transaction takes substantial time. The overhead introduced by the exchange of data over asynchronous *tlm_fifo* reads and writes would offset any gains in case of simple transactions that have naive constraints and therefore a characteristic fast randomization.

eUVM offers yet another parallelization construct useful in such scenarios. Normally (as in SV), a thread created by the fork construct creates a task and the created task executes on the same CPU thread as the parent thread that spawned the fork. But eUVM also lets the user distribute the forked tasks on the CPU threads associated with the multiple task executors.

This can be useful when a sequence constitutes thousands of transactions, that are stored in a container (a queue or an array, for instance). To accelerate the generation of such a sequence, eUVM allows forking multiple tasks with the sequence carrying container sliced and split between the different tasks. Each spawned fork is tasked to process a slice of the original container.

The code snippet in listing 3 illustrates a typical parallelized form in eUVM. A call to the *fork* construct in eUVM returns a *Fork* object that gets appended to the array (*forks*) of the Fork objects (line 3) for further processing. The *body* of the fork is enveloped in a lambda function in order to capture the scoped variable *i*. This is slightly more intriguing compared to SV, which allows the user to simply declare the scoped variables in the header part of the fork. On line 10, the method *set_thread_affinity* is used to bind the forks to the respective parallel task executors. Finally, we join the forks, and akin to SV, the forks start executing only after a call to *join* (or any other blocking construct for that matter).

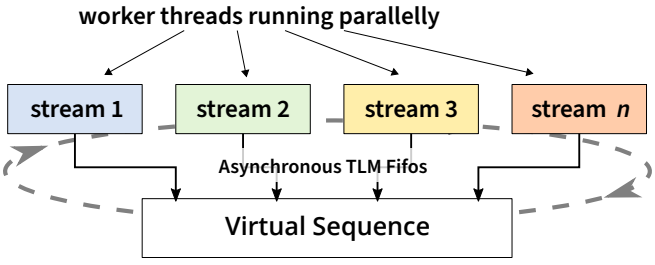

Fig. 11: Sequence Level Parallelism

Listing 3: A Parallelized eUVM Fork Semantics

```
1  Fork[] forks;
2  for (int i=0; i!=threadCnt; ++i) {
3    forks ~= (int i) {
4      return fork (() {
5          randomizeSome(i);
6        });
7    } (i);
8  }
9  foreach (i, f; forks)
10   f.set_thread_affinity(i);
11 foreach (f; forks) f.join();
```

IV. RISCV-DV Testbench Optimization Techniques

A comprehensive survey of generic testbench optimization techniques can be found in [31] and [32]. This paper focusses primarily on HPC techniques as applied to the RISCV-DV project.

Section V delves into the RISCV-DV testbench architecture. Section VI lists the profiling tools and the methodology to find the bottlenecks in the RISCV-DV generator. In section VII we explore an algorithmic bottleneck and refactor the code to ensure a scalable (wrt instruction count) RISCV-DV architecture.

Multicore eUVM parallelization techniques as applicable to RISCV-DV generator, are expounded upon in section VIII. A minor OOP architecture refactoring of RISCV-DV required to enable multicore-parallelism is discussed in section VIII-A.

Section IX lists some generic optimization techniques deployed to help optimize RISCV-DV performance. These relate to constraint solving, memory allocation, and certain UVM related bottlenecks.

Section X establishes how the eUVM port of the RISCV-DV directly generates a binary dump that can straightway be loaded in the simulation/emulation instruction memory, thus skipping assembler and linker altogether. Finally, section XI explains why shared-memory parallelism is important for the future evolution of multicore testbenches.

V. RISCV-DV Architecture and Implementation

The output from the RISCV-DV generator is a bare-metal RISC-V assembly language program that can be compiled and executed on a RISC-V processor core or its model. Alternately, the generator can also directly generate an executable binary dump that may directly be loaded into a memory instance for the purpose of simulation or emulation. However, since SystemVerilog does not provide a way to process native binary data, the SV version of RISCV-DV outputs an ASCII string instead of a binary dump.

The RISC-DV instruction generator is highly customizable. A number of command-line options such as the required count of sub-programs, the ratio of various kinds of instruction streams to generate, and many others, enable the user to fine-tune the test program generation. In its simplest form, the generator can be used to generate millions of disparate random instructions that are dumped into a single main function. When a count of sub-programs is specified from the command-line, the generator randomly divides the specified total number of instructions into the specified number of sub-programs and stitches the test program's call stack with a suitable function call control-flow.

However, some categories of instructions do require sequencing and can not be executed or tested entirely in isolation. Take the jump (*jalr*) instruction for instance. Injecting an unconstrained jump instruction in the generated test code can make the test program jump directly to a location just before the test program's exit, thus making the simulation skip most of the executable test code in the test program. An unconstrained jump instruction can also lead to some entirely unwarranted scenarios such as an infinite loop, or jumping to a location outside the test program's scope. RISCV-DV therefore categorizes execution use-cases such as loop sequences, load/store hazards, numeric computation exceptions etc, to name a few – into *directed streams*. The instructions in these streams need to be constrained together as a sequence so that a desired and meaningful execution scenario can be created. These streams are generated, randomized, and then inserted into an initially generated dump of non-directed randomized instructions in a random manner. While injecting these streams into the dump, care is taken so that a *directed stream* does not get inserted inside another *directed stream* that has already been injected in the initial dump.

Towards the end of the generation process, RISCV-DV revisits the merged stream and looks at the jump instructions embedded therein in order to annotate the target locations with appropriate labels.

There are other finer details of the generation process that are not in the scope of this paper. References [33], [34] may be useful for a more detailed exposition of some subtle aspects of the generator's internal working. In this paper, however, we limit ourselves to the macro-level details that are important to understand, analyze, and optimize the generator's performance.

VI. Profiling the RISCV-DV Generator

A comprehensive profiling strategy is essential to analyze algorithmic bottlenecks in the execution of an application before trying to optimize its runtime. To this end, eUVM implements a *uvm_trace* method that works just like the standard *uvm_info* method, but also lists the machine's run-time clocked since the start of the simulation. As illustrated in listing 4, in order to measure the time taken by a particular code fraction, a user needs to put a *uvm_trace* call right before the code block, and another just after it.

Listing 5 provides representative output from the uvm_trace invocations coded in listing 4. The wall-clock time, listed in square brackets just after the tag UVM_TRACE, represents the snapshot of time taken at the uvm_trace invocation.

Note that the RISCV-DV is highly parameterized and the execution time taken by its various components varies significantly with the set of parameters chosen by the end-user. For the purpose of profiling, a comprehensive test, namely *riscv_instr_base_test*, was chosen with a mix of seven *directed streams* covering the whole spectrum of possible RISC-V instruction categories.

Listing 4: Example uvm_trace Invocation

```
uvm_trace("GEN INSTR", "START", UVM_DEBUG);
foreach (ref instr; instr_list)
  randomize_instr(instr, is_debug_program);
uvm_trace("GEN INSTR", "DONE", UVM_DEBUG);
```

Listing 5: Example uvm_trace Log Message

```
UVM_TRACE [1.645711] ../riscv/gen/riscv_instr
_stream.d(1) @ 0: reporter [GEN INSTR] START
UVM_TRACE [4.502635] ../riscv/gen/riscv_instr
_stream.d(4) @ 0: reporter [GEN INSTR] END
```

Four major bottlenecks were identified based on the results of profiling and analysis of the algorithmic complexity of the various components of RISCV-DV. These bottlenecks are enumerated below in the decreasing order of their impact on the overall execution time of the RISCV-DV generator.

1) The generator creates and randomizes *directed instruction streams* in a ratio specified by the user from the command-line. These streams are created and randomized for each of the sub programs and the main function. Most of the time taken in the generation of the streams can be traced to the time taken for execution of the constraint solvers.
2) RISCV-DV dumps a big chunk of the *non-directed instruction stream* into the respective *instr_list*s meant for the main program and the sub-programs. Here too, most of the server effort is spent on randomization and solving constraints.
3) The third bottleneck is formed when the RISCV-DV generator inserts the *directed instruction streams* into the *non-directed instruction stream*. This bottleneck was found to become more severe with the increase in the number of instructions to be generated, with an algorithmic complexity of $\mathcal{O}(n^2)$.
4) The generator creates a formatted string to be dumped as an assembly language program. As it dumps thousands of instructions, *$sformatf* is invoked repeatedly, resulting in as many memory allocations.

The first two in the above list of bottlenecks involve constraint solving and are linear in terms of algorithmic complexity. Since linear complexity algorithms scale well with multicore parallelization, these bottlenecks were dealt with accordingly. Section VIII explores the intricacies involved in parallelization, as well as the ways to achieve optimal performance gains. While the fourth bottleneck too has a linear complexity, the possibility of its parallelization is marred by frequent memory allocation. In section IX-D, we discover ways to reduce memory allocation (calls to `malloc`). A reduced count of memory allocation calls helps reduce execution time and also paves the way to more scalable parallelization.

On the other hand, the third bottleneck is due to a sub-optimal algorithmic implementation. Since it requires a significant architectural change, we deal with it first in the section that follows.

VII. RISCV-DV Algorithmic Optimizations

A. Hammer a Nail, but Drive a Screw

Figure 12 helps in understanding the process of merging the *Directed Streams* in the *Initial Random Dump* of undirected RISCV instructions. The RISCV instructions in a directed sequence are all tagged as *atomic* in order to identify and keep them intact. The original RISCV-DV code implements the merge process in a greedy fashion.

First, a location to inject a directed stream sequence in the initial dump is picked randomly. If the randomly picked location happens to be inside another directed sequence, another random number is picked to find another location that does not violate the inserted sequence. The process of picking a random location is repeated up to ten times and if it still can not find a location outside of already inserted directed streams, the last picked location is incremented until the boundary of the directed stream it falls into is found and the new directed stream is inserted right thereafter.

Note that this process involves inserting a slice of elements in a SystemVerilog queue. It is a costly operation since all the elements (before or) after the point of insertion need to be moved. If the number of instructions being generated is large, the number of directed streams to be inserted also rises correspondingly, and the average size of the block of memory to be shifted increases proportionately, resulting in an $\mathcal{O}(n^2)$ complexity. The generator therefore does not scale well for an instruction count

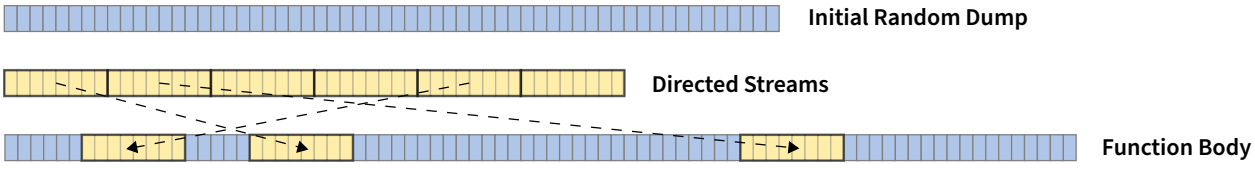

Fig. 12: Merging of Directed Streams in to the Initial Random Instruction Dump

Listing 6: Lazy merging of directed streams to make RISCV-DV scalable

```
void mixin_dir_instr_list(riscv_instr_stream[] dir_list) {                    1
  riscv_instr[]        mixed_list;                                            2
  uint instr_count = cast(uint) instr_list.length;                            3
  uint mixed_count = instr_count;                                             4
  dir_n.length = instr_count;                                                 5
  this.dir_instr_list = dir_list;                                             6
  foreach (size_t dir_idx, dir_instr; dir_instr_list) {                       7
    mixed_count += dir_instr.instr_list.length;                               8
    uint rnd_idx = urandom(0, instr_count);                                   9
    uint next_dir = dir_n[rnd_idx];                                           10
    if (next_dir == 0) // no directed stream at this location                 11
      dir_n[rnd_idx] = cast(uint) (dir_idx + 1);                              12
    else { // directed stream already at this location -- store as insert_idx 13
      uint insert_idx = cast(uint) (dir_idx + 1);                             14
      riscv_instr_stream instr = dir_instr_list[next_dir-1];                  15
      while (instr.next_stream != 0) {                                        16
        if (rnd_idx % 2 == 0) { // insert at the end or in the beginning      17
          uint next_idx = instr.next_stream;                                  18
          instr.next_stream = insert_idx;                                     19
          insert_idx = next_idx;                                              20
        }                                                                     21
        instr = dir_instr_list[instr.next_stream-1];                          22
      }                                                                       23
      instr.next_stream = cast(uint) (insert_idx);                            24
    }                                                                         25
  }                                                                           26
  mixed_list.length = mixed_count;                                            27
  uint n = 0;                                                                 28
  foreach (i, instr; instr_list) { // create the merged function body         29
    uint next_dir = dir_n[i];                                                 30
    while (next_dir != 0) {                                                   31
      uint next = dir_instr_list[next_dir-1].next_stream;                     32
      foreach (dinstr; dir_instr_list[next_dir-1].instr_list)                 33
        mixed_list[n++] = dinstr;                                             34
      next_dir = next;                                                        35
    }                                                                         36
    mixed_list[n++] = instr;                                                  37
  }                                                                           38
  instr_list.assimilate(mixed_list);                                          39
}                                                                             40
```

larger than a hundred thousand. Interestingly, as noted in [35], larger test programs help generate increasingly complex program flows that are more likely to uncover processor bugs.

An algorithmic refactoring of the code, as outlined in listing 6, brings the algorithmic complexity down to $O(n)$, thus making the insertion process scale linearly. The improvised merging process does not involve queue insertions. *Directed Streams* are now merged with the *Initial Randomized Dump* in a lazy manner. First, a random number (rnd_idx on line 9) is used to mark the locations where the directed sequences are sought to be injected. No insertion is actualized for now, and the locations marking the points of injection of the respective directed streams just get enumerated in an array. In case another directed stream has already been marked for injection at the same location, the information is stored in a linked list structure actualized by the variable `insert_idx` embedded in the stream object itself. When the locations have been tabulated for all the injectable directed streams, a merged instruction dump is created in one go (line 29). On re-profiling after this refactoring, it was found that the merge process now scaled well and no longer posed a bottleneck for the generation of large test programs.

VIII. A Parallelized Instruction Generator

A. Changes to Accommodate Parallelization of the RISCV-DV Architecture

Adding parallelism to UVM testbenches does not require too many changes at the architecture level since the object-oriented paradigm provides the right kind of framework to enable concurrency [16]. Yet, a few changes were required to the RISCV-DV

architecture to make it compliant with the concurrency semantics of the D Programming Language.

In general global or statically scoped variables, when accessed by multiple threads, prove to be detrimental to the run-time efficacy of a concurrent software. This happens because of the shared nature of globally/statically scoped variables, whereby (to avoid race conditions) any access to such variables has to be synchronized with a synchronization lock. It was noticed that the RISCV-DV source code has a few statically scoped variables in the file *riscv_instr.sv*. These variables serve the purpose of registration of various RISC-V instructions with the instruction generator.

eUVM RISCV-DV port refactors these variables along with the related functions to a separate class named *riscv_instr_registry* specifically created for the purpose of instruction registry. These changes are summarized in listing 8 which exhibits the eUVM code for instruction registry. The corresponding code for the instruction registry in SV is outlined in listing 7. An instance of the *riscv_instr_registry* class is then created inside the singleton *riscv_instr_gen_config* class, thus preserving the singleton nature of the variables, just as the SV version which declares these variables as static.

The RISCV-DV generator involves large instruction sequences. Consequently, the best approach towards its parallelization works out to be a parallelized *fork*. Just like SV, in eUVM too the most fundamental unit of testbench execution is a process. A process may be declared as a task or may get forked from an already executing task by making a call to the eUVM construct *fork*. This is where eUVM starts to differ from SV, in as much as eUVM is capable of executing threads on multiple cores. When a process is forked out in eUVM, it is feasible to delegate the newly forked process to execute on a specified processor thread.

A parallelized fork (listing 9) is used to mitigate the first bottleneck that pertains to the initial dumping of non-directed instruction streams that form the backbone of the main function and the sub-programs. Note that the parallelization process is not effective when the number of transactions (or instructions in the case of RISC-V) to be randomized is small. This is so because the constraint solvers take more time when invoked for the first time for a given constraint. Subsequently, when more transactions are constraint-randomized, the solvers take advantage of the already solved constraints and just pick a random solution from the already solved set. Note that a parallelized testbench instantiates a separate solver for each thread of execution. Sharing the same solver instance across the different threads would defeat the very purpose of parallelization.

Based on the number of instructions in the dump, a decision is first made whether to invoke parallelization or not. The default threshold value of 4000 is kept in the *par_instr_threshold* configuration object parameter. When the parallelized RISCV-DV generator encounters a larger number of instructions to randomize, it splits the process into *par_num_threads* (defaults to 8) slices and delegates the randomization of the instructions in each slice to a separate thread.

An avid reader may have noticed the finer details of the fork construct in eUVM. Unlike the SV fork, an eUVM fork returns a Fork object that can be collected into a list, thus enabling a lot of flexibility in configuring the forks and joining them later in the code. In particular, the *set_thread_affinity* method configures the thread that a particular fork is assigned to execute on. The **~ =** operator (on line 28) is a shorthand for appending an object to a dynamic array in the D language.

A slightly different approach (listing 10) is taken for parallelization of directed instruction streams. Since there are multiple groups of directed streams, a separate thread is designated for the randomization of the streams in each of the groups. This

Listing 7: Static variables in SV riscv_instr class enable instruction registry

```
class riscv_instr extends uvm_object;
  // All derived instructions
  static bit instr_registry[riscv_instr_name_t];
  // Instruction list
  static riscv_instr_name_t  instr_names[$];
  // Categorized instruction list
  static riscv_instr_name_t
   instr_group[riscv_instr_group_t][$];
  static riscv_instr_name_t
   instr_category[riscv_instr_category_t][$];
  static riscv_instr_name_t   basic_instr[$];
  static riscv_instr
   instr_template[riscv_instr_name_t];
  // Privileged CSR filter
  static privileged_reg_t    exclude_reg[];
  static privileged_reg_t    include_reg[];
```

Listing 8: eUVM implements the instruction registry as a separate class

```
class riscv_instr_registry: uvm_object {
  // All derived instructions
  bool[riscv_instr_name_t] instr_registry;
  // Instruction list
  riscv_instr_name_t[]      instr_names;
  // Categorized instruction list
  riscv_instr_name_t[]
    [riscv_instr_group_t]   instr_group;
  riscv_instr_name_t[]
   [riscv_instr_category_t] instr_category;
  riscv_instr_name_t[]       basic_instr;
  riscv_instr[riscv_instr_name_t]
                            instr_template;
  // Privileged CSR filter
  privileged_reg_t[]        exclude_reg;
  privileged_reg_t[]        include_reg;
```

Listing 9: Parallelizing Randomization of the Initial Instruction Dump

```
void gen_instr(bool no_branch = false, bool no_load_store = true,
               bool is_debug_program = false) {
  GC.disable();
  setup_allowed_instr(no_branch, no_load_store);
  assert (instr_list.length != 0);
  if (instr_list.length <= cfg.par_instr_threshold ||
      cfg.par_num_threads == 1) {
    foreach (ref instr; instr_list) {
      randomize_instr(instr, is_debug_program);
    }
  }
  else {        // parallelise with cfg.par_num_threads
    Fork[] forks;
    size_t instr_count = instr_list.length;
    size_t instr_grp_length = instr_count / cfg.par_num_threads;
    for (size_t i=0; i!=cfg.par_num_threads; ++i) {
      size_t start_idx = i * instr_grp_length;
      size_t end_idx = (i + 1) * instr_grp_length;
      // last group
      if (i == cfg.par_num_threads - 1) end_idx = instr_count;
      // capture start_idx and end_idx and fork
      Fork new_fork = (size_t start, size_t end) {
        return fork({
            for (size_t i=start; i!=end; ++i) {
              randomize_instr(instr_list[i], is_debug_program);
            }
          });
      } (start_idx, end_idx);
      new_fork.set_thread_affinity(forks.length);
      forks ~= new_fork;
    }
    foreach (f; forks) f.join();
  }
  // Do not allow branch instruction as the last instruction because
  // there's no forward branch target
  while (instr_list[$-1].category == riscv_instr_category_t.BRANCH) {
    instr_list.length = instr_list.length - 1;
    if (instr_list.length == 0) break;
  }
  GC.enable();
}
```

choice is made for sake of code simplicity. Furthermore, taking note of the fact that for a given group of directed streams, the constraints pertaining to the streams therein would be identical, this strategy puts less stress on the thread specific constraint solvers.

IX. Generic Testbench Optimization Techniques

A. Use Appropriate Tools to Identify Performance Bottlenecks

This is the most important step in the process of performance optimization. The RISCV-DV constitutes more than fifteen thousand lines of code. Profiling helped reduce focus to about two hundred lines of code that get executed repeatedly, and are key to the performance of the generator. In section VI, we already explored *uvm_trace* (a construct in eUVM) that helps in the formal identification of testbench bottlenecks.

Note that the `uvm_trace` method enables only a macro level profiling. It is prudent to keep in mind that every invocation of `uvm_trace` involves an operating system call to fetch the current clock time. Reckless use of the command may result in an inordinate blowup of the testbench runtime.

For micro-level profiling the opensource tool *gprof* comes in handy. Reference [36] contains a detailed exposition of the tool.

Listing 10: Parallelizing Randomization of the Directed Instruction Streams

```
void generate_directed_instr_stream(in int hart, in string label,
    in uint original_instr_cnt, in uint min_insert_cnt,
    in bool kernel_mode, out riscv_instr_stream[] instr_stream) {
  if (cfg.no_directed_instr) return;
  else {
    Fork[] forks;
    uint instr_stream_length = 0;
    uint stream_idx = 0;
    foreach (stream_name, ratio; directed_instr_stream_ratio) {
      uint  insert_cnt = original_instr_cnt * ratio / 1000;
      if (insert_cnt <= min_insert_cnt) insert_cnt = min_insert_cnt;
      instr_stream_length += insert_cnt;
      uvm_info(get_full_name(), format("Insert directed instr stream %0s %0d/%0d times",
                                       stream_name, insert_cnt, original_instr_cnt), UVM_LOW);
      // capture stream_name, ratio, stream_idx and insert_cnt and fork
      Fork new_fork = (string name, uint ratio, uint idx, uint cnt) {
        return fork ({
          generate_directed_instr_stream_idx(hart, label, original_instr_cnt, kernel_mode,
                                             name, ratio, instr_stream, idx, cnt);
          });
      } (stream_name, ratio, stream_idx, insert_cnt);
      new_fork.set_thread_affinity(forks.length);
      forks ~= new_fork;
      stream_idx += insert_cnt;
    }
    instr_stream.length = instr_stream_length;
    foreach (f; forks) f.join();
    assert (stream_idx == instr_stream.length);
    instr_stream.shuffle();
  }
}
void generate_directed_instr_stream_idx(in int hart, in string label, ....
    // Like the non-parallelized instr stream generator
    // snipped for sake of brevity here ....
}
```

B. Skip the UVM Factory

The UVM Factory implements many advanced object creation features like the ability to define *type* and *instance* overrides. Under the hood the UVM Factory construct maintains a complex data structure to implement these features. A seemingly simple invocation of `<class_name>::type_id::create` method requires matching the given *class_name* against the names of the classes that have been registered with the factory. Consequently, the UVM factory becomes significantly slow when a large number of types are registered with the factory.

RISCV-DV implements each instruction in RISC-V ISA as a separate class. On this account, hundreds of classes are required to get registered with the factory and that would make the UVM factory extremely slow. To accelerate instruction generation, RISCV-DV implements its own custom factory as outlined in listing 11.

Listing 11: RISCV-DV Custom Factory

```
riscv_instr[riscv_instr_name_t] instr_template;
riscv_instr_name_t[]            instr_names;

bool register(riscv_instr_name_t instr_name,
              string qualified_name) {
  instr_registry[instr_name] = qualified_name;
  return true;
}

void create_instr_list() {
  foreach (instr_name, instr_class_name;
           instr_registry) {
    riscv_instr instr_inst;
    if (canFind(unsupported_instr, instr_name))
      continue;
    instr_inst = create_instr(instr_name,
                              instr_class_name);
    instr_template[instr_name] = instr_inst;
  }
}
```

Note that the RISCV-DV uses an enumeration `riscv_instr_name_t` to tag each instruction as a numeral. As a result, the custom factory hashes the factory objects using the enumeration, thus resulting in much faster matching when an instruction object is created.

C. Prefer Shallow Copy Over UVM Clone Method

UVM provides a clone method as part of the *uvm_object_utils* macros. The UVM clone method enables some useful features like *deep copy*. In many scenarios, we do not need a *deep copy* and a *shallow copy* suffices. RISCV-DV uses the native *shallow copy* construct. Listing 12 outlines the low level implementation of the shallow copy construction in eUVM. On lines 5-7, eUVM uses object introspection to determine the underlying memory footprint of the object to be copied. On line 8, the underlying memory slice is simply copied from the source object to the destination. Note that such a slice copy operation in the D language results in a single call to `memcopy`, which is much more efficient compared to copying individual elements of the class object that would have resulted from a call to the *uvm_utils copy* construct.

Listing 12: Shallow Copy Implementation in eUVM

```
T shallowCopy(T)(T obj) {
  if (obj is null) return null;
  ClassInfo ci = obj.classinfo;
  Object clone = _d_newclass(ci);
  size_t start =
      Object.classinfo.m_init.length;
  size_t end   = ci.m_init.length;
  (cast(void*) clone)[start..end] =
       (cast(void*) obj)[start..end];
  return staticCast!T(clone);
}
```

D. Minimize Memory Allocation

Every time a memory chunk (big or small) is allocated, the runtime memory manager possibly takes several micro-seconds to process the request. Note that memory allocation takes place even in trivial operations like a call to SV *$sprintf* construct that returns a formatted string. The RISCV-DV generator invokes *$sprintf* as part of the *post_randomize* method when randomizing a RISC-V instruction:

At line 13 (listing 13), a system call to *$sformatf* is made to transform a 32 bit number *imm* to its string form. While numbers are stored directly in memory, strings are arrays of variable size. The function *$sformatf* has to therefore allocate some memory in order to return the string to the caller.

While it is not obvious how to avoid such memory allocations in SV, eUVM offers a simple solution. We know that we are formatting a 32 bit number to a decimal string form. It would take a maximum of 10 characters. On line 12, we declare a fixed size *char* array by name *_imm_str_buffer*. Since the array size is fixed it gets allocated as a part of the *riscv_instr* class object. Note that performance gets degraded by the number of times memory gets allocated, not as much by the size of the allocation. With this improvisation, the total amount of allocated memory eventually remains the same since the formatted string has to be stored at some place, but the count of calls to `malloc` reduces by half.

The underlying D Language (on top of which eUVM is built) provides two flavors of *format* function. While the Dlang function `format` has the same functionality as `$sformatf` in SV, an alternate Dlang function `sformat` allows the user to specify a scratch memory where the formatted string gets stored and returns the resulting pointer to the character array. A cast operation is required to convert it to a string type before storing it in the *imm_str* variable.

Listing 13: Trivial Memory Allocation in SV

```
class riscv_instr extends uvm_object;
  // snipped
  rand bit [31:0]              imm;
  string                       imm_str;
  // snipped
  // ..
  function void post_randomize();
    extend_imm();
    update_imm_str();
  endfunction : post_randomize
  function void update_imm_str();
    imm_str =
      $sformatf("%0d", $signed(imm));
  endfunction
endclass
```

Listing 14: Avoiding Trivial Memory Allocation in eUVM

```
class riscv_instr: uvm_object {
  // snipped
  @rand ubvec!32               imm;
  string                       imm_str;
  // snipped  ....
  void post_randomize() {
    extend_imm();
    update_imm_str();
  }
  char[12] _imm_str_buff;
  void update_imm_str() {
    imm_str = cast(string) sformat!("%0d")
      (_imm_str_buff, cast(int) imm);
  }
}
```

E. Reuse Allocated Memory

When coding a non-reentrant function, dynamic arrays can be scoped *static* so that the same dynamic array can be reused the next time a function is called. However, it is important to reset the length of the array to *zero* at the start of the function. Also, remember to *reserve* some length for the array so that it does not free up the memory when its size is reset.

Listing 15 illustrates the technique as applied to the array `inter_set`, which is intentionally declared as static on line 1, though its contents get reset on line 3.

Listing 15: Reusing static Allocated Array

```
1 static riscv_instr_name_t[] inter_set;
2 inter_set.reserve(inter_set.length);
3 inter_set.length = 0;
4 inter_set~=setDifference(setIntersection
5   (instr_set, include_set, allowed_set),
6             disallowed_instr[].sort());
7 idx = urandom(0, inter_set.length);
8 name = inter_set[idx];
```

Initially, as the first RISC-V instruction is generated, the *inter_set* array variable gets populated by the list of allowed (and not disallowed) instructions. As the array grows, it asks for memory to be allocated in order to hold the set of possible instructions. In the next cycle, however, the array would already have the requisite memory since we have declared it *static*. Had we declared the array as automatically scoped (without the *static* tag), every time we randomize an instruction fresh memory will need to be allocated again, leading to an avoidable adverse impact on performance.

Note that the keyword *static* has a different connotation in Dlang as compared to SV and C++. The *static* keyword in Dlang makes a variable get scoped as *thread local* (similar to the keyword *thread_local in C++*). It does not get shared across threads.

F. Minimize Constraint Processing

Whenever feasible, prefer functional/algorithmic processing over constraint solving. Constraint solving is inherently a complex process. In case a variable can be randomized using a sequential method, that approach should be preferred. In listing 16 we take a look at how RISCV-DV picks an instruction in a constrained random way.

At the beginning of the randomization process, the RISCV-DV finds out whether it needs to exclude some instructions (the *if* condition on line 3). When the disallowed set is empty, a branch is taken that completely avoids constraint processing. The

Listing 16: Picking a Random RISC-V Instruction

```
1  disallowed_instr = {disallowed_instr,
2                      exclude_instr};
3  if (disallowed_instr.size() == 0) begin
4    if (include_instr.size() > 0) begin
5      idx = $urandom_range(0,
6                 include_instr.size()-1);
7      name = include_instr[idx];
8    end
9     else if (allowed_instr.size() > 0) begin
10      idx = $urandom_range(0,
11                    allowed_instr.size()-1);
12     name = allowed_instr[idx];
13   end else begin
14     idx = $urandom_range(0,
15                  instr_names.size()-1);
16     name = instr_names[idx];
17   end
18 end else begin
19   std::randomize(name) with {
20     name inside {instr_names};
21     if (include_instr.size() > 0)
22       name inside {include_instr};
23     if (allowed_instr.size() > 0)
24       name inside {allowed_instr};
25     if (disallowed_instr.size() > 0)
26       !(name inside {disallowed_instr});
27   }
28 end
```

Listing 17: Picking a Random RISC-V Instruction (eUVM)

```
1  else {
2    import std.algorithm.sorting: sort;
3    import std.algorithm.setops:
4      setIntersection, setDifference;
5    import std.array: array;
6    riscv_instr_name_t[] instr_set =
7                      instr_names_sorted;
8    riscv_instr_name_t[] include_set =
9                      instr_set;
10   riscv_instr_name_t[] allowed_set =
11                     instr_set;
12   if (include_instr.length > 0) {
13     include_set = include_instr;
14     include_set.sort();
15   }
16   if (allowed_instr.length > 0) {
17     allowed_set = allowed_instr[];
18     allowed_set.sort();
19   }
20   static riscv_instr_name_t[] inter_set;
21   inter_set.reserve(inter_set.length);
22   inter_set.length = 0;
23   inter_set~=setDifference(setIntersection
24     (instr_set, include_set, allowed_set),
25               disallowed_instr[].sort());
26   idx = urandom(0, inter_set.length);
27   name = inter_set[idx];
28 }
```

else branch (line 18 in the listing) uses a *randomize with* constraint block to deal with the more complex scenario that involves excluding the disallowed instructions.

The eUVM RISCV-DV port avoids constraint processing in the *else* branch as well (listing 17). For the sake of brevity only the *else* branch is outlined here.

The eUVM port takes advantage of set operation algorithms available in the standard D language library. Note that the D language implements these set processing operations as lazy algorithms, thus avoiding memory allocations altogether.

G. Avoid Array (foreach) Constraints

Avoid randomizing elements of an array in one go. When possible, call the *randomize* method on each element of the array separately. Each call to the *randomize* method processes all the constraints listed in the class. When dealing with a larger set of constraints, the solver generally does not scale well and needs to put in extra effort.

H. Prefer Structs Over Classes

Prefer *struct* instances over *class* objects. This is not always possible, but there are scenarios where sub-objects are required inside *uvm_object* instances. While a memory for a structure instance gets allocated along with the enclosing class, using a class instance as a sub-object results in additional runtime overhead at the time of its allocation (call to *new*).

I. Use a Native (Systems Programming) Language

Because of the proprietary nature of SV simulators, not much has been reported on the performance of SV compared to native systems programming languages (C/C++/Rust/Go/D). A simple algorithm that calculates the number of primes in a given range can be used for a basic comparison. Note that performance benchmarking is a complex domain and simple experiments like the one that follows are only indicative (at best).

The code in listing 18 implements a naive algorithm that counts the number of prime numbers smaller than ten million in SV. The same algorithm coded in the D language is listed in 19.

On compiling and running both the implementations (D version compiled with '`-O3`' optimization flag), you may find the SystemVerilog version to be slower by an order of magnitude.

It is not difficult to figure out why algorithms coded in SV execute much slower compared to the code implemented in native languages. Despite the C-like nomenclature used by SV for its 2-state data types, *int*, *byte* etc are not comparable to the similarly named data types in the native programming languages. Unlike the C *int*, the SV version of *int* allows both blocking

Listing 18: Counting Primes in SV

```
program main();
  automatic function
  int count_primes(const int count);
    int primes[$];
    for (int n = 2; n <= count; ++n) begin
      bit is_prime = 1;
      foreach (primes[i]) begin
        if (primes[i] * primes[i] > n) break;
        if (n % primes[i] == 0) begin
          is_prime = 0;
          break;
        end
      end
      if (is_prime) primes.push_back(n);
    end
    return primes.size();
  endfunction: count_primes
  initial begin
    $display ("Found %d Primes between 0 \
and 10000000", count_primes(10000000));
  end
endprogram
```

Listing 19: Counting Primes in D

```
import std.stdio;
size_t count_primes(const int count) {
  int[] primes;
  for (int n = 2; n <= count; ++n) {
    bool is_prime = true;
    foreach (prime; primes) {
      if (prime * prime > n) break;
      if (n % prime == 0) {
        is_prime = false;
        break;
      }
    }
    if (is_prime) primes ~= n;
  }
  return primes.length;
}

void main() {
  writefln("Found %s Primes between 0" ~
           " and 10000000",
           count_primes(10000000));
}
```

Listing 20: An Example RISCV-DV Constraint

```
constraint no_hint_illegal_instr_c {
  if (instr_name == C_JR) {
    rs1 != ZERO;
  }
}
```

Listing 21: Compile Time Constraint Selection in eUVM

```
static if (RISCV_INSTR_NAME == C_JR) {
  constraint! q{
    rs1 != ZERO;
  } no_hint_illegal_instr_c;
}
```

and non-blocking assignments. An *int* variable in SV also allows another process to wait for a change in its value. On this account, SV needs to do extra processing when executing integral operations, making it noticeably slower when compared to systems programming languages.

If we consider functional semantics, the closest to SV *int* type is the SystemC *sc_signal<int>* type.

A similar reasoning applies to Python [20] which, given its interpreted nature, actually turns out to be much slower than even SV. The RISCV-DV project has been implemented in eUVM, SV, and Python. While the eUVM version runs the fastest, the Python version trails the eUVM version by a factor of more than 1000x.

J. When Possible, Avoid Dynamic Casting

Under the hood, a dynamic cast involves a string comparison that has an adverse impact on software performance. When down-casting an object, there are scenarios where there is a certainty that the cast operation will succeed. For example, every time an object is cloned, or a call to the uvm_utils' create construct is made, an underlying cast operation comes into effect. Since the down-casting operator involved in create or the clone operations is never going to fail, these casts can better be replaced with static casts.

K. Use Compile Time Functional Evaluation to Expedite Constraint Processing

The RISC-V processor has a modular ISA with a vast instruction set of over 1400 instructions (including the unratified extensions) [37]. The RISCV-DV project tries to group these instructions using various criteria for the sake of adding constraints to the relevant instructions. For example, a constraint that applies only to the `C_JR` instruction is shown in listing 20. This code snippet is taken from the `riscv_compressed_instr` base class in the RISCV-DV code base.

The problem is that though this constraint is valid for only one instruction, it gets applied to all the compressed instructions in the RISCV ISA. The constraint solver will have to process the constraint guard for every compressed instruction only to know that it does not apply to most of the instructions.

Listing 21 outlines an approach taken by eUVM, wherein using D language construct `static if`, eUVM individualizes each instruction at compile time so that the solvers get to process only the constraints that are valid for the given instruction.

X. Native Generation of Binary Executable

As illustrated in figure 13, the RISCV-DV generator outputs an assembly program that needs to be compiled and linked before it can be used for simulation or emulation. The eUVM port, having the benefit of low-level programming constructs, combines the linker functionality in the generator and directly generates the binary file.

XI. The Memory Wall

A recent trend in the server-end processors is to integrate a large on-chip RAM along with the CPU cores [38]. In addition to ensuring low-latency access to memory, this technology also reduces power dissipation significantly. As a side-effect of this technique, the amount of RAM available to the multi-core processors reduces significantly, leading to a phenomenon popularly known as the *Memory Wall* [39]. Note that Memory Wall also applies to server-end processors that use external memory due to limited shared level-3/4 caches [40].

As can be seen in the benchmarking results in section XII, the parallelized eUVM version of RISCV-DV mitigates the Memory Wall well enough. When we scale the RISCV-DV to multicore parallelism, the incremental usage of RAM is insignificant.

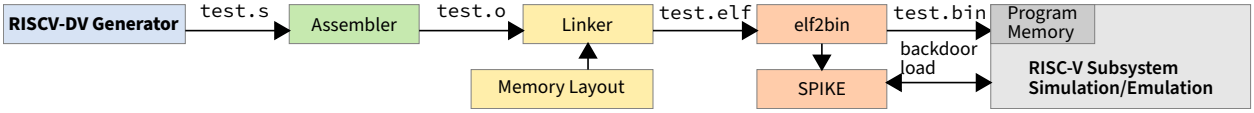

Fig. 13: RISC-V Co-Simulation/Emulation Flow

Listing 22: Benchmarking Instruction Randomization in RISCV-DV

```
void gen_instr(bool no_branch = false, bool no_load_store = true,
               bool is_debug_program = false) {
  GC.disable();
  setup_allowed_instr(no_branch, no_load_store);
  assert (instr_list.length != 0);
  uvm_trace("GEN INSTR", "START", UVM_NONE);
  if (instr_list.length <= cfg.par_instr_threshold ||
      cfg.par_num_threads == 1) {
    foreach (ref instr; instr_list) {
      randomize_instr(instr, is_debug_program);
    }
  }
  else {        // parallelise with cfg.par_num_threads
    Fork[] forks;
    size_t instr_count = instr_list.length;
    size_t instr_grp_length = instr_count / cfg.par_num_threads;
    for (size_t i=0; i!=cfg.par_num_threads; ++i) {
      size_t start_idx = i * instr_grp_length;
      size_t end_idx = (i + 1) * instr_grp_length;
      // last group
      if (i == cfg.par_num_threads - 1) end_idx = instr_count;
      // capture start_idx and end_idx and fork
      Fork new_fork = (size_t start, size_t end) {
        return fork({
            for (size_t i=start; i!=end; ++i) {
              randomize_instr(instr_list[i], is_debug_program);
            }
          });
      } (start_idx, end_idx);
      new_fork.set_thread_affinity(forks.length);
      forks ~= new_fork;
    }
    foreach (f; forks) f.join();
  }
  uvm_trace("GEN INSTR", "END", UVM_NONE);
  // Do not allow branch instruction as the last instruction because
  // there's no forward branch target
  while (instr_list[$-1].category == riscv_instr_category_t.BRANCH) {
    instr_list.length = instr_list.length - 1;
    if (instr_list.length == 0) break;
  }
  GC.enable();
}
```

XII. Benchmarking Results

In order to measure multicore efficiency, calls to `uvm_trace` were introduces around the block of code that randomizes RISC-V instructions (refer to line number 6 and 35 in the lising 22). The *arithmetic only* RISCV-DV test was run for an instruction count of 10 million. The test was compiled with compile and link time optimizations on a 64 core server with an AMD Epyc 7713 processor. Note that the timing values listed in the following table account only for the time taken for generation and constrained-randomization of the instructions. The time to format the instructions into ASM mnemonics is not included since the eUVM version of RISCV-DV can directly create a binary dump.

Table I lists the execution time and memory usage for generating 10 million instructions on a multicore RISCV-DV. Table II lists the approximate performance gains achieved by the various contributing factors.

XIII. Conclusion

With the help of the HPC techniques listed in this paper, we are able to achieve a throughput of around 2.5 million instructions in a second. While this throughput has been achieved for the simplest RISCV-DV test generation, we are able to achieve more

TABLE I: Performance Improvements for a 10 million instruction RISCV-DV test (All timing values in seconds)

Instr Count	Thread Count	Execution Time	Performance	RAM Usage
10000000	1	57.86	1.00x	4.9 GB
10000000	2	31.22	1.85x	4.9 GB
10000000	4	18.03	3.21x	5.0 GB
10000000	8	10.35	5.59x	5.0 GB
10000000	16	5.53	10.46x	5.0 GB
10000000	32	4.23	13.68x	5.0 GB

TABLE II: Factors Contributing to RISCV-DV Performance Gain

Performance Contributing Factor	Approximate Gain
Multicore Sequence Generation (32 threads)	14x
Constraint Reduction and Optimization	2.5x
Native Data types and Testbench Code (Systems Programming)	2x
Thread Affinity (Sticking OS threads to Physical Cores)	2x
Optimizing Memory Allocation and Reuse	1.5x
Binary Instruction Generation (Skipping Assembler and Linker)	$>$ 2x

than a million instr/sec throughput for more complex tests as well. This throughput is in the range of the clock frequency of any DuT that runs on an emulation platform.

Most of the techniques discussed in this paper can only be implemented in a language that allows low level memory access and enables multicore parallelism. Currently, only eUVM fits the criteria and is the only library that has a parallelized implementation of the IEEE 1800.2 UVM standard.

A. It is Opensource, Folks!

The source code and the instructions to compile and test the parallelized eUVM version of RISCV-DV are hosted on GitHub [41]. The source code of the opensource project eUVM can also be accessed from GitHub [42]. An enterprising reader is invited to explore it, fork it, and contribute to the fledging opensource verification tooling.

REFERENCES

[1] Random Instruction Generator for RISCV. Accessed: 2022-08-14. [Online]. Available: https://github.com/chipsalliance/riscv-dv

[2] Introduction to Ibex Verification. [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/verification.html

[3] Random Instruction Generator for RISCV (eUVM Port). Accessed: 2023-08-14. [Online]. Available: https://github.com/chipsalliance/riscv-dv/tree/master/euvm

[4] System Validation at ARM. Accessed: 2023-08-16. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/System%20IP/System_Validation_at_ARM_Enabling_our_partners_to_build_better_systems.pdf

[5] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[6] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[7] S. Sudhakar and R. K. Jain, "The Need for Speed: Understanding Design Factors that Make Multi-core Parallel Simulations Efficient," *Design and Verification Conference, San Jose*, 2013.

[8] T. Yang, "Distributed Simulation of UVM Testbench," *Design and Verification Conference, San Jose*, 2016.

[9] B. Applequist, V. Rousseau, A. Tan, and J. Sesham, "Multithreading a UVM Testbench for Faster Simulation," *Design and Verification Conference, San Jose*, 2020.

[10] I. C. Cristea and D. Dospinescu, "Open-Source Framework for Co-Emulation Using PYNQ," *Design and Verification Conference, San Jose*, 2021.

[11] V. Billa and S. Haran, "Challenges and Mitigations of Porting a UVM Testbench from Simulation to Transaction-Based Acceleration (Co-Emulation)," in *Design and Verification Conference*, 2018.

[12] H. V. D. Schoot and A. Yehia, "UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up," *Design and Verification Conference, San Jose*, 2015.

[13] L. Maillet-Contoz and F. Ghenassia, *Transaction Level Modeling*. Boston, MA: Springer US, 2005, pp. 23–55. [Online]. Available: https://doi.org/10.1007/0-387-26233-4_2

[14] A. Sharma, P. Goel, and H. V. Balisetty, ""C" you on the faster side: Accelerating SV DPI based co-simulation," *Proceedings of Design and Verification Conference, San Jose*, 2014.

[15] G. Kumar, S. Pagey, M. Sinha, and M. Chopra, "Automatic Partitioning for Multi-core HDL Simulation," 2015.

[16] B. Goetz, *Java Concurrency in Practice*. Addison Wesley, 2006.

[17] Clifford E. Cummings, "Applying Stimulus & Sampling Outputs - UVM Verification Testing Techniques," *Synopsys Users Group, US*, 2016.

[18] S. Kumari and D. N. Gadde. (2023) Effective Design Verification – Constrained Random with Python and Cocotb.

[19] R. Salemi and T. Fitzpatrick, "Verification Learns a New Language: – An IEEE 1800.2 Implementation," *Design and Verification Conference, San Jose*.

[20] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[21] M. Cieplucha and W. Pleskacz. (2017) New Constrained Random and MetricDriven Verification Methodology Using Python.

[22] T. Vörtler1, T. Klotz2, K. Einwich3, and F. Assmann, "Simplifying UVM in SystemC," in *Design and Verification Conference, Munich*, 2015.

[23] "UVM-SystemC Randomization - Updates From The SystemC Verification Working Group," 2021.

[24] W. Bright, A. Alexandrescu, and M. Parker, "Origins of the D Programming Language," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386323

[25] M. Gupta, S. Kadyan, and P. Goel, "Transform your SoCFPGA into an Emulator," 2021.

[26] L. De Moura and N. Bjorner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[27] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," 03 2009, pp. 174–177.

[28] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing Samplers is Easy: The Boon of Testers," Oct. 2021.

[29] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[30] M. Reddy, *API Design for C++*. Elsevier Science, 2011. [Online]. Available: https://books.google.co.in/books?id=IY29LylT85wC

[31] F. Kampf, J. Sprague, and A. Sherer, "Yikes! Why is My SystemVerilog Testbench So Slooooow?" *Design and Verification Conference, San Jose*, 2012.

[32] C. Cummings, J. Rose, and A. Sherer, "Yikes! Why is My SystemVerilog Still So Slooooow?" *Design and Verification Conference, San Jose*, 2019.

[33] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "Constrained random verification for risc-v: Overview, evaluation and discussion," in *MBMV 2021; 24th Workshop*, 2021, pp. 1–8.

[34] T. Liu, R. Ho, and U. Jonnalagadda, "Open Source RISC-V Processor Verification Platform," *RISC-V Summit*, 2019.

[35] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU Fuzzing via Intricate Program Generation," in 33^{rd} *USENIX Security Symposium*, 2024.

[36] G. Tumbush and M. Hupp, "Dramatically increase the performance of SystemC simulations," in *Design and Verification Conference*, 2007, pp. 21–23.

[37] riscv-opcodes. Accessed: 2023-08-14. [Online]. Available: https://github.com/riscv/riscv-opcodes

[38] Z. Zhang, "Analysis of the Advantages of the M1 CPU and Its Impact on the Future Development of Apple," in *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*, 2021, pp. 732–735.

[39] S. A. McKee and R. W. Wisniewski, *Memory Wall*. Boston, MA: Springer US, 2011, pp. 1110–1116. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_234

[40] A. F. A. Furtunato, K. Georgiou, K. Eder, and S. Xavier-De-Souza, "When Parallel Speedups Hit the Memory Wall," *IEEE Access*, vol. 8, pp. 79 225–79 238, 2020.

[41] Random Instruction Generator for RISCV (eUVM Port). Accessed: 2023-08-14. [Online]. Available: https://github.com/coverify/riscv-dv/tree/master/euvm

[42] Embedded UVM Github Repository. Accessed: 2022-08-14. [Online]. Available: https://github.com/coverify/euvm