2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up

Hans van der Schoot, Emulation Technologist, Mentor Graphics Corp., Ottawa, Canada

Ahmed Yehia, Verification Technologist, Mentor Graphics Corp., Cairo, Egypt

{hans_vanderschoot, ahmed_yehia}@mentor.com

***Abstract* — This paper aims to demystify the performance of SystemVerilog and UVM testbenches when using an emulator for the purpose of hardware-assisted testbench acceleration. Architectural and modeling requirements are described, followed by a recommended systematic approach for maximizing overall testbench acceleration speed-up and achieving your ultimate performance expectations.**

Keywords* — *testbench acceleration; transaction-based acceleration; emulation; systemVerilog; UVM; performance.

I. INTRODUCTION

Testbench acceleration, also known as simulation acceleration or co-emulation, is a targeted use case of emulation, where the RTL design-under-test (DUT) running in a hardware emulator can interact with a testbench running on a workstation. Interest in this use case has grown rapidly in recent years, as it plays a significant role in making hardware emulation easier to use and adopt. Project teams building complex designs that require millions of clock cycles to attain full coverage closure or measure performance metrics are deploying emulation at chip and system levels to boost performance, and often, in the spirit of scalable verification, right at the block level. Clearly, as these teams move from block to chip to system level, it is imperative that SystemVerilog and UVM (as the industry standard verification language and reuse methodology, respectively) and other advanced verification techniques like assertions and coverage continue to be effective in the context of emulation [1, 2].

Using a hardware emulator to accelerate logic simulation can be very effective if done correctly. You can easily estimate raw performance if you know the speed of the design in simulation as measured in design clocks per second of wall clock time. As an example, assume your simulator runs the design at 1,000 design clocks/second (yes, perhaps be a tad optimistic). An emulator will run at +/- 1M clocks/second, delivering a raw performance improvement of 1,000X (raw performance equates to the emulator running unencumbered by the need to wait for a testbench or external event).

A modern testbench can have a huge impact on hardware emulation performance, however, particularly with the use of advanced verification methods, such as functional coverage, assertions, constrained-random stimulus, and UVM, that make today's testbench increasingly heavy-weight. Consider a simulation where the workstation spends 80% of its time processing the design and 20% on the testbench. With the emulator running the design infinitely fast, you get a 5X speed-up over simulation — a far cry from the 1,000X raw performance speed-up the emulator is capable of. To achieve the minimum acceleration of two orders of magnitude desired by most companies considering emulation, we must shift much of the testbench load into the emulator, while ensuring the behavioral portion running on the workstation is efficient enough to keep pace with the emulator. But how?

Using accelerated (i.e. emulation-ready) transactors lets us shift testbench load to the emulator. Bus cycle state machines of an accelerated transactor accept high-level data from the behavioral testbench and spin it out into multiple design clocks at emulation speed. The higher the ratio of design clocks per chunk of data (i.e. per transaction) the more likely the emulator is free to run at its raw speed of +/- 1 MHz. A simple AMBA bus transfer hovers around the 10X ratio while Ethernet packets can reach thousands of design clocks each.

With accelerated transactors in place, the focus changes to the ability of the testbench to generate and process high-level data fast enough to keep up with the emulator, and to the efficiency of communication between the testbench executing on the workstation and the transactors running in the emulator. Both are inherently tied to the modeling abstraction utilized, which for all practical purposes must be at the transaction-level. In particular, as

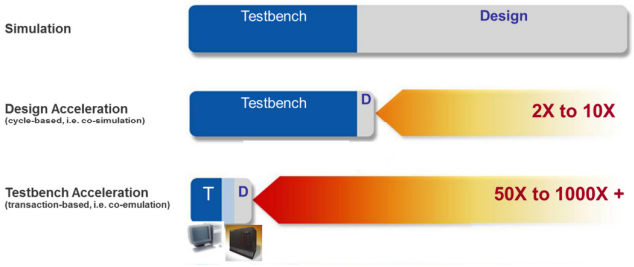

Figure 1. Transaction-based communication is key to acceleration performance

suggested by Figure 1, the communication between simulator and emulator must be transaction-based, not cycle-based, to minimize communication overhead and enable adequate acceleration. This enables the required hardware-software synchronization and data exchange to be both infrequent and information-rich, while the high-frequency cycle-based pin activity is confined to run in hardware at full emulator clock rates. The foundation of this transaction-based simulator-emulator communication is the Accellera Standard Co-Emulation Modeling Interface version 2 (SCEMI 2) [3], which defines a set of standard function call-based modeling interfaces for multi-channel communication between software models describing system behavior (e.g. the testbench or HVL domain) and structural models describing the implementation of a hardware design (e.g. the HDL domain).

In this paper we further demystify SystemVerilog and UVM testbench acceleration using hardware emulation, describing the architectural and modeling requirements, and the steps needed to achieve your ultimate acceleration speed-up expectations. Section II provides an overview of our recommended dual-domain framework for transaction-based testbench acceleration drawn from earlier work [4]. Readers already familiar with the associated modeling concepts can move straight to Section III. Sections III and IV comprise the focal point of this paper—optimizing overall SystemVerilog and UVM testbench acceleration.

II. Unified Dual Domain Framework for Testbench Acceleration

We previously introduced a comprehensive dual-domain architecture and methodology for creating emulation-ready SystemVerilog and UVM testbenches [4]. Our recommended practices adhere to the principles of emulation to instill a sound and unified testbench framework with the following traits:

Interoperability: Users can run the same testbench in both pure simulation and emulation. In general, there is one codebase for simulation and emulation platforms. Conventional simulation efforts are not hindered.

Flexibility: The main benefits of SystemVerilog for verification are its advanced features for transaction-level and behavioral modeling. The object-oriented paradigm supported offers the ability to create dynamic, reusable, hierarchical verification components and testbenches, which is at the heart of UVM. These modeling benefits of SystemVerilog and UVM continue to apply in the context of emulation.

Performance: To justify adoption, users must get optimal performance in this emulation-ready framework. Typically, they should see a few orders of magnitude speed-up using emulation without slowing down (and possibly improving) pure simulation.

Fundamentally, the convention of a single top-level module established for simulation to encapsulate all elements of an overall testbench falls short for emulation. Transaction-based testbench acceleration using emulation demands two separate hierarchies—one synthesizable and the other untimed—that transact together without direct cross-hierarchy signal accesses. Our unified dual-domain emulation-ready testbench framework with interoperability, flexibility and performance traits can be described in three high-level steps:

1. Employ two distinct HVL and HDL domains.
2. Model the timed testbench portions for synthesis in the HDL domain.
3. Devise a transaction-based HVL-HDL intra-domain API.

A. *Dual HVL and HDL domains*

The foremost architectural requirement is to create dual HVL and HDL top-level module hierarchies. This is conceptually easy to understand. The HDL domain must be synthesizable and contain essentially all clock synchronous code to be mapped and run on the emulator, namely the RTL DUT, clock and reset generators, and the various bus cycle state machines for driving and sampling DUT interface signals. Basically, everything that interacts directly with the DUT or otherwise operates at the signal-level must be included in the HDL domain.

In sharp contrast, the HVL domain need not adhere to synthesis restrictions but instead must be strictly untimed. This means no use of any explicit time advance statements, like clock synchronizations, # delays, and other unit-delay statements. Time advance may occur only in the HDL domain. Abstract event handshakes are permitted in the untimed HVL domain, and it is time-aware in the sense that the current time, as communicated with every context switch from HDL to HVL domain, can be read. As a consequence, the HVL domain should contain the non-synthesizable behavioral testbench code, such as the various transaction-level stimulus and analysis components (generators, scoreboards, coverage collectors, etc.). Note that this does not preclude selectively migrating some of this code into the HDL domain as well, at the expense of modeling flexibility, but potentially improving performance.

This required domain partitioning is, in large part, already facilitated by UVM through its abstraction and associated layering principles, which should continue to apply flexibly, independent of the execution platform. UVM already advocates absence of timing and cross-scoped references for upper testbench layer components, and delegation of timing control to the lower transactor layer components (i.e. UVM agents containing the drivers and monitors) (Figure 2). These recommendations merely become mandatory when using emulation. As such UVM usage can continue mostly in accord with established best practices, especially for the upper testbench layer components, with the exception of a couple of small, but notable, advanced usage considerations detailed in [4] (involving the UVM factory and configuration database for emulation-ready agents).

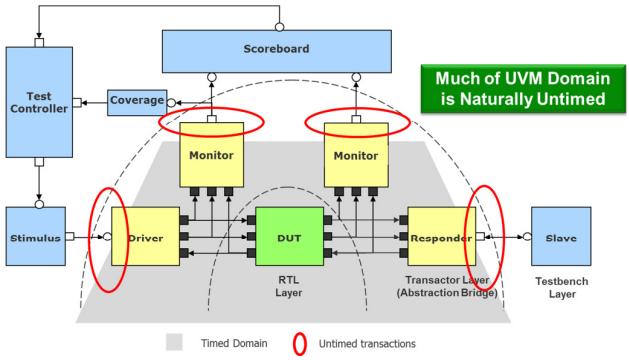

Figure 2. UVM layered testbench

With transaction-based HDL-HVL domain partitioning, performance can be maximized because testbench and communication overhead is reduced, and all intensive pin wiggling takes place in the dedicated timed domain (grey area in Figure 2) targeted to run at emulation speeds. This modeling paradigm is facilitated by advancements in modern synthesis technology across multiple tools. For example, the Mentor Graphics' Veloce™ technology can synthesize not only SystemVerilog RTL, but also implicit FSMs, initial and final blocks, named events and wait statements, import and export DPI-C functions and tasks, system tasks, memory arrays, behavioral clock and reset specification along with variable clock delays, assertions, covergroups, and more. All supported constructs can be mapped on the emulator, and all models synthesized with Veloce run at full emulator clock rate for high performance. Moreover, they can be simulated natively on any IEEE 1800 SystemVerilog-compliant simulator. These synthesis advancements were a precursor to the Accellera SCEMI 2 standard that enables effective development of emulation-ready transactors [3].

It is pertinent to note that, beyond hardware-assisted testbench acceleration, there are other good reasons to adopt a dual-domain testbench architecture. For instance, it can facilitate the use of multi-processor platforms for

simulation, the use of compile and run-time optimization techniques, or the application of good software engineering practices for the creation of highly portable, configurable VIP.

B. Time advance modeling

Forming the bridge between the timed signal level and untimed transaction level of abstraction, transactor layer testbench components convert "what is being transferred" into "how it must be transferred", or vice versa, in accordance with a given interface protocol. Specifically, per Figure 2, these transactors are responsible for transforming untimed transactions into a series of cycle-accurate clocked events applied to a given signal interface, and/or conversely, for "transacting" observed cycle-accurate signal activity into higher-level transactions. The timed portion of a transactor is reminiscent of a conventional bus functional model (BFM), a collection of threads and associated tasks and functions for the (sole) purpose of translating to and from timed interface activity on the DUT. With UVM, this is commonly modeled inside classes (e.g. classes derived from the `uvm_driver` or `uvm_monitor` base classes). The DUT pins are bundled inside SystemVerilog interfaces and accessed directly from within these classes using the Virtual Interface (VIF) construct. Virtual interfaces thus act as the link between the dynamic object-oriented testbench and the static SystemVerilog module hierarchy.

With regard to the dual-domain emulation-ready testbench framework, BFMs are naturally timed and must be part of the HDL domain, while dynamic class objects are generally not synthesizable and must be part of the HVL domain. In addition, a transactor layer component usually has some high-level code next to its BFM portion that is not synthesizable, for example, a transaction-level interface to upstream components in the testbench layer. All BFMs must therefore be surgically extracted and modeled as synthesizable SystemVerilog HDL modules or interfaces.

With aforementioned advanced IEEE 1800 SystemVerilog standard HDL modeling constructs supported by the Veloce acceleration solution, tone can write powerful state machines to implement synthesizable BFMs without much difficulty. Furthermore, when modeling these BFMs as SystemVerilog interfaces, one can continue to use virtual interfaces to bind the dynamic HVL and static HDL domains — as a complement to the original Accellera SCEMI 2 module and DPI-C function-based communication model. The key difference with conventional SystemVerilog and UVM object-oriented testbenches is that the BFMs have moved from the testbench to the HDL domain, and the link between the dynamic testbench and the static HDL hierarchy must now be in the form of transaction-based APIs. That means testbench objects may now only access interface signals indirectly, by calling interface functions and tasks. This yields the testbench architecture illustrated in Figure 3, which works natively in simulation, and has also been demonstrated to work in emulation with the Veloce acceleration solution [4].

C. Function-based HVL-HDL communication interface

With the timed synthesizable and untimed behavioral portions of a testbench fully partitioned, what remains is devising a (transaction-level) function-based communication interface between the two domains. As suggested, the use of virtual interfaces in the HVL domain bound to concrete interface instances in the HDL domain enables a flexible intra-domain communication model provided that the synthesizable BFMs are implemented as SystemVerilog interfaces, not as modules. The flexibility stems from the fact that user-defined tasks and functions in these interfaces form the API.

Following a remote proxy design pattern, components in the HVL domain acting as proxies to BFM interfaces in the HDL domain can call relevant tasks and functions declared inside the BFMs, using virtual interfaces to drive and sample DUT signals, initiate BFM threads, configure BFM parameters or retrieve BFM status. Specifically retaining the original transactor layer components (like driver and monitor classes) as the BFM proxies, minus the extracted BFMs themselves (i.e. the bus cycle state machines), minimizes impact on the original UVM testbench. The proxies form a thin layer in place of the original transactor layer, which allows all other testbench layer components and their interconnections to remain intact, as shown in Figure 3. Figure 4 similarly shows that the testbench view of a UVM agent remains the same; only the structure underneath has changed, but that is (and should be) transparent to the UVM testbench layers. herefore, this revised emulation-ready testbench architecture offers maximum leverage of existing verification techniques and capabilities.

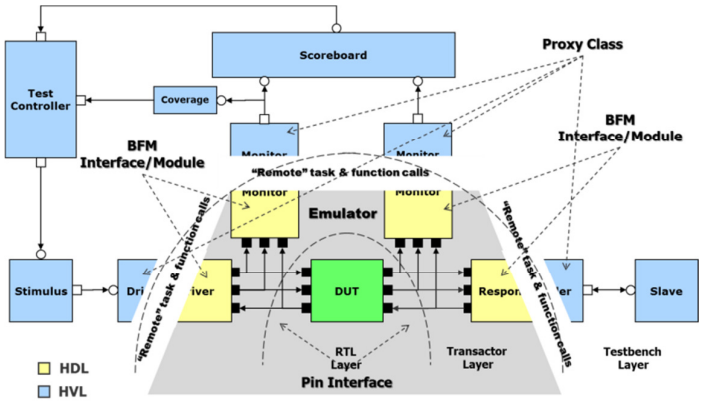

Figure 3. Unified UVM testbench for simulation and acceleration

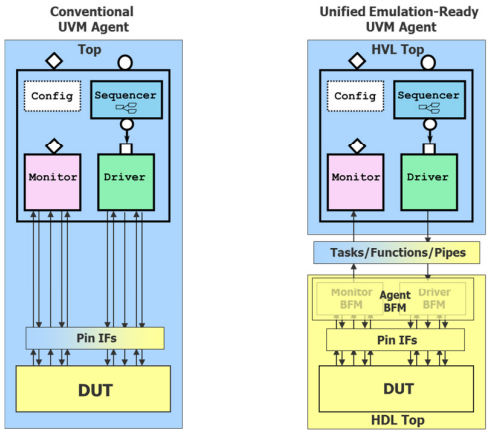

Figure 4. Conventional vs. emulation-ready UVM agent

Figure 5 provides a snippet of an emulation-ready UVM driver to illustrate the code structure of an accelerated transactor following the prescribed modeling paradigm. Time-consuming tasks and non-blocking functions in the interface can be called by the driver proxy via the virtual interface to execute bus cycles, set parameters, or get status information. The remote task/function call mechanism is founded on the known Accellera SCEMI 2 DPI-C function model to gain the same SCEMI 2 performance benefits. In the original SCEMI 2 function-based model, it is the SystemVerilog DPI-C interface that is the natural boundary for partitioning workstation and emulator models [3], whereas our emulation-ready testbench framework uses the SystemVerilog class-object-to-SystemVerilog-interface instance boundary as the natural boundary for the same partitioning. Extensions specifically designed for SystemVerilog and UVM testbench modeling have been added, most notably, interface task calls in the workstation to emulator direction (analogous to SCEMI 2 DPI export calls), in which the use of time-consuming/multi-cycle processing elements is allowed. This is essential for modeling BFMs in the HDL domain that are callable from a class-based HVL domain.

For increased modeling flexibility and completeness, function calls can also be made in the opposite direction which enables transaction-based intra-domain communication initiated from the HDL domain. Specifically, a BFM interface may call dedicated class member functions of a class object in the HVL domain (analogous to DPI SCEMI 2 import calls), for instance, to provide sampled transactions for analysis or notify other status information. The use of such *back-pointers* in BFM interfaces for what we refer to as *outbound* (i.e. HDL to HVL) communication can be significant for performance, particularly for components like monitors. While it is natural to have a monitor BFM push transactions, instead of its proxy pulling transactions out, the more important benefit of the former is that it can be done using one-way outbound non-blocking calls (i.e. function calls with no return value and no output arguments) that can be dispatched in the untimed testbench domain and executed by

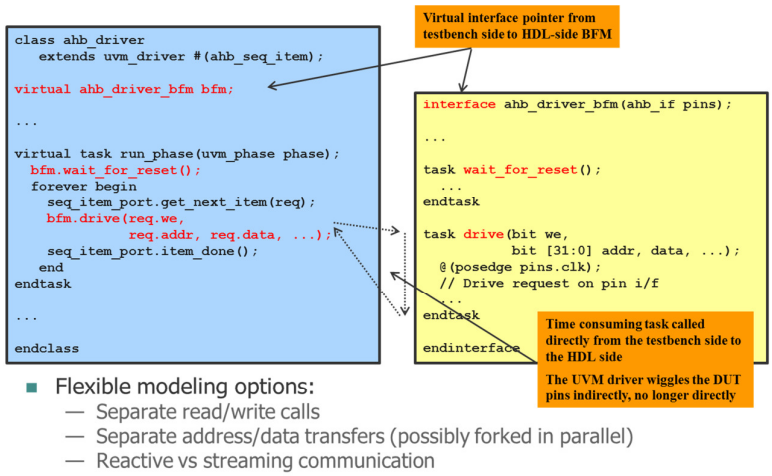

Figure 5. Emulation-ready UVM driver

the simulator, all concurrent with advancing the design in the emulator, thus reducing communication and testbench overhead. We return to this "emulation throughput booster" in upcoming sections. For now, Figure 6 depicts the structure of a UVM monitor in our unified emulation-ready framework. As shown, the handle of a BFM interface to the BFM proxy can be assigned inside the proxy itself, via its virtual BFM interface. Access to any data members in the BFM proxy are not permitted, just as cross-scoped signal references into the BFM are not allowed. A more comprehensive account of the modeling aspects and details of the dual domain framework for SystemVerilog and UVM testbench acceleration is provided in our earlier work [4].

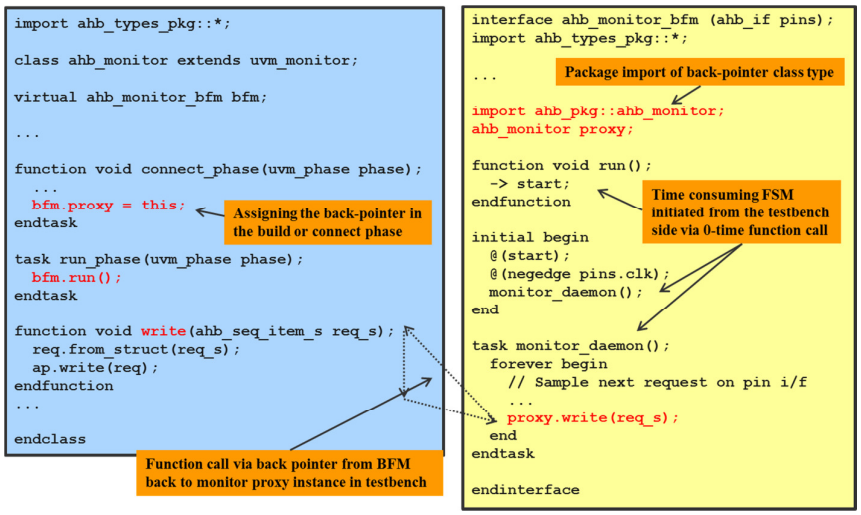

Figure 6. Emulation-ready UVM monitor

III. TESTBENCH ACCELERATION PERFORMANCE DEMYSTIFIED

With the dual-domain testbench modeling framework in place for both normal simulation and acceleration, let's re-examine run-time execution performance. For simulation nothing material changes. Performance should be largely dictated by the size of the RTL DUT, albeit that an advanced testbench may have overhead as well which often intensifies with increased verification complexity. Favorably, we have found that the firm partitioning between the timed cycle-level and untimed transaction-level abstraction does not appear to hamper conventional simulation performance. In fact, performance can improve by some useful margin, due to the ability of today's simulator optimizations working more effectively with the dual top separation (as opposed to the traditional single top setup).

As we hinted at the beginning, performance has different dynamics for testbench acceleration. With the RTL DUT, clocks, and BFMs now all running several orders of magnitude faster in the emulator than in the simulator, the overhead of the transaction-based (untimed) testbench is accentuated, and the simulator-emulator communication via accelerated transactors (i.e. BFM-proxy pairs) is a new source of extra overhead to consider. The efficiency of the testbench plus the function calls and data transfer to and from the HDL domain becomes pivotal for overall acceleration performance.

To better understand the performance interplay between simulator and emulator—between the HVL and HDL domain—consider simulation time as the time output from the simulator, versus run-time as the time it takes a test to actually run in terms of real time or wall clock time. Since simulation time can only be advanced in the HDL domain, where the design clocks reside (derived from the free-running emulator clock), the design clocks are stopped when control is passed from emulator to simulator to execute testbench code in the HVL domain. Simulation time advance halts, and the design remains frozen until testbench processing in the current simulation time step is done and control is passed back to the emulator to resume design execution. At this simulation time step, the HVL domain may use a little run-time or a lot, but it always uses zero simulation time. As long as control is with the HVL domain, it may even interact repeatedly with the HDL domain, but only via non-blocking (i.e. 0-time) function calls. Thus, for the sake of the current discussion, both domains execute somewhat in lock-step, with the HVL domain not able to advance simulation time but always having knowledge of it, which is when the HDL domain last stopped. That is, the HVL domain is untimed, but time-aware.

Total run-time consumption can hence be broken down into three regions: real time consumed by HDL domain execution ($t_{[HDL]}$), by HVL domain execution ($t_{[HVL]}$), and by context switching between the two domains including data exchange between simulator and emulator ($t_{[HVL\text{-}HDL]}$):

$$t_{[total]} = t_{[HDL]} + t_{[HVL]} + t_{[HVL\text{-}HDL]} \quad (1)$$

The time $t_{[HDL]}$ spent in the emulator and the time $t_{[HVL]}$ spent in the simulator (or in general on the workstation) are also called hardware time and software time, respectively. Acceleration performance can be assessed by analyzing hardware time relative to total run-time, i.e. $t_{[HDL]} / t_{[total]}$. The larger this fraction, the better, indicating that the design running at fast emulation speed is active a high percentage of the time and, accordingly, the software (i.e. testbench) and communication overhead is low. In this case, we say the environment is *hardware-bound*, versus *software-bound*, or that it exhibits a "healthy" *throughput*:

$$t_{[HDL]} / t_{[total]} >> (t_{[HVL]} + t_{[HVL\text{-}HDL]}) / t_{[total]} \quad (2)$$

While we briefly assumed above that the design clocks stop during testbench and communication activity, this may not strictly be the case. There may be opportunities for both simulator and emulator to operate concurrently, i.e. HDL domain execution in the emulator may continue in parallel with HVL domain execution (testbench and communication activity). Clearly, this would further enhance emulator throughput because $t_{[HDL]}$ stays the same while $t_{[HVL]} + t_{[HVL\text{-}HDL]}$ decreases (partially absorbed by $t_{[HDL]}$), and thus $t_{[total]}$ and $t_{[HDL]} / t_{[total]}$ both improve. Simulator-emulator concurrency (without harm of nondeterministic behavior) and its bearing on testbench acceleration performance is briefly revisited in Section IV, and an excellent, more thorough elaboration can be found in [6].

Following are three characteristic scenarios associated with the above three run-time regions as reference to help frame and evaluate acceleration performance potential and set corresponding realistic expectations:

- Scenario 1 – Nearly free-running hardware
  This is the ideal case of the emulator running very efficiently, executing the HDL domain almost without interruption. Typical examples are OS boot-up, or an internalized testbench where the HVL domain is responsible only for starting and ending a test (negligible software overhead or none at all for a fully synthesizable testbench).

- Scenario 2 – Highly interactive testbench
  This is a testbench where switching between HDL and HVL domain activity is frequent, yet without a lot of data being sent back and forth. Examples comprise bus transactors, like AXI and PCIe, with each bus operation interactive and generally rather small. Performance varies for this case dependent on the

complexity of the HVL domain; the testbench can be simple, or it can be intricate (with file I/O or advanced stimulus generation, response checking, and functional coverage alike).

- Scenario 3 – Intensive data streaming
  This scenario refers to a testbench distinguished by massive amounts of data exchanged across the HVL–HDL intra-domain boundary. This causes the time for message passing and context switching (i.e. communication overhead) to be very significant. Pertinent examples involve traffic for applications such as video or networking, which can be particularly intensive when traffic happens in parallel (or otherwise uses high bandwidth).

Clearly, scenario 1 characterizes the case of very good throughput (near maximum, actually). Since the HVL domain overhead is insignificant, performance optimization efforts should focus on gaining the best possible emulator speed. Hardware aspects such as critical paths, clock utilization (inactive edges, edge alignment), and hardware parallelism are all optimization options to achieve higher emulator frequency and level out capacity. Some optimizations can even be attained semi-automatically with state-of-the-art technology like the Veloce performance and capacity advisor. Intriguingly, increasing emulator speed not only decreases hardware time, but also hardware time as a fraction of total run-time, i.e. emulator throughput. It is important to understand that optimizing acceleration performance is, in fact, a sort of balancing act between improving emulator speed and managing the resultant intensifying testbench and communication overhead.

Scenario 2—emulator throughput dictated primarily by the testbench—calls for managing testbench complexity and reducing testbench overhead where practicable to optimize simulation performance. Pertinent considerations are the optimal use of simulator compile and run-time switches, as well as addressing frequent causes of run-time issues, such as file I/O, constraint solving, coverage sampling, messaging and verbosity control, usage of UVM macros, configuration, phasing, objections, etc.

Scenario 3—emulator throughput impeded by communication overhead between testbench and HDL domain—generally demands that cross-domain transaction utilization is improved. The rationale: since bigger transactions or data chunks are bound to stay inside the emulator longer (i.e. be processed uninterrupted), it is beneficial to aggregate smaller ones, even artificially and agnostic to implied protocol transaction boundaries, and subsequently stream these “meta-transactions” to and from the emulator using streaming (versus reactive) communication modeling mechanisms—for instance using SCEMI pipes [3] or specialized transaction aggregation functions. The aforementioned use of concurrency modeling techniques can also prove effective to absorb in part both testbench and communication overhead during emulator execution [6]

Hardware emulator optimization particulars tend to be fairly platform-dependent, and are beyond the scope of this paper. In the remainder of this paper we focus on tackling testbench and communication overhead as the definite hurdles to gaining adequate acceleration performance.

IV. Tackling Testbench Acceleration Performance

For modern-day RTL testbenches with demanding project deadlines, it is not uncommon for engineers to write testbench code with an emphasis on functionality over performance. This often leads to unpleasant surprises with simulation run-times, when seemingly harmless code regions add significant run-time overhead. The ensuing effort to remodel or even restructure the testbench for better performance is generally not straightforward.

For hardware-assisted testbench acceleration, both the motivation and challenge to pursue performance optimizations are further amplified. As a result of the hardware (HDL) and software (testbench) domain separation, optimizing testbench code solely for performance may make little, if any, difference amidst other pertinent factors like hardware-software communication overhead and emulator throughput. With the dynamics of testbench acceleration performance just explained, this section provides the blueprint for tackling performance overhead to enable substantial acceleration speed-up.

A. *Profiling simulation runs*

Early assessment of pure simulation performance through simulation profiling is essential for identifying code areas that contribute significantly to total simulation time (a.k.a. code hotspots) and estimating acceleration speed-up potential. Mentor Graphics’ Questa™ simulator contains profiler technology that provides interactive

graphical representations of both memory and CPU usage on a per-instance basis. It shows what part of the design is consuming resources (CPU cycles or memory), enabling you to identify the run-time-intensive areas in the design code.

Testbench areas of tangible overhead in simulation profiles must be analyzed and enhanced to improve run-time performance. The time attributed to RTL DUT execution can be discounted, as it will be allotted to the emulator. However, areas of small run-time overhead should not be underestimated, because tiny overhead in pure simulation may be amplified in testbench acceleration when the simulator is relieved from executing the HDL domain. Figure 7 depicts simulation profile statistics, where user foreign code (C code interacting with HVL via PLI/DPI interfaces), assertions, and constraint-solving consume around 10% of the simulation run-time. Unless reduced, this overhead prevents overall testbench acceleration speed-up from reaching even a factor of 10.

```
# Profile Summary:
#                 Verilog          30345    29.78%
#                 VHDL             46180    45.32%
#                 Kernel           15488    15.20%
#                 Foreign           2272     2.23%
#                 Assertions        3423     3.36%
#                 Solver            4188     4.11%
#                 ==========      =======  =======
#                 Total           101899   100.00%
```

Figure 7. Questa pure simulation profile summary

In general, designs targeted for emulation are large and suffer from long simulation run-times (e.g. hours, or even days). However, there is no need to run a full simulation for simulation profiling. Some practical amount of time can be run to collect enough profile samples to exercise the various simulation phases, although profiling should never be halted right after the initialization phase of a test, as it could yield inaccurate profiling statistics for the entire simulation, leading to bad decisions.

B. Tackling testbench hot-spots

The most common causes of simulation performance degradation in modern testbenches are: inefficient SystemVerilog/UVM native code [7, 8, 9], constrained randomization, assertions/checkers, and foreign applications/models. Pertinent performance aspects with coding guidelines for these areas follow.

SystemVerilog/UVM

SystemVerilog packages

With packages, code is defined and compiled once inside a package, then referenced in other components. This is much more efficient than using `` `include `` for file inclusion, where code can be dispersed across the compilation cluster. Also, referencing parameters, functions and tasks from packages is generally more efficient than referencing them hierarchically from a module.

SystemVerilog dynamic arrays

Heed the use of queues and associative arrays; dynamic arrays are a more efficient alternative when suitable.

UVM macros

As recommended in[7], a good rule of thumb is to use the UVM factory, messaging, and TLM IMP macros, and avoid using `` `uvm_do* `` or `` `uvm_field* `` macros.

UVM configuration database

Each time a `uvm_config_db::set()` or `uvm_config_db::get()` method is used, a full scan of the UVM configuration database is performed using regular expressions to match strings. The larger the database and the less specific the search, the longer the time to find a match. Minimize the usage of `uvm_config_db`, especially in dynamic UVM objects, avoid wildcard matching, and by all means, avoid the `wait_modified()` method. On a similar note, avoid `super.build_phase()`unless inheriting from a user-defined class that implements the

`build_phase()`. The `super.build_phase()` call enables UVM auto-configuration, which can be very wasteful in querying visible configurations in the resource database for each component.

UVM objections

Each time a UVM objection is referenced, it must be propagated throughout the entire UVM hierarchy. Hence, it is wise to use UVM objections primarily in pseudo-static UVM components, like test classes, instead of dynamic UVM objects like sequences. This is not to say that UVM objections should not be used in sequences at all. For instance, an interrupt routine sequence that raises an objection prohibiting test termination is an appropriate use.

UVM messaging

UVM messaging macros can significantly improve performance over their actual function call counterparts [7]. On the other hand, decreasing UVM messaging verbosity levels where possible (e.g. UVM_NONE) can help filter out messages of high verbosity, reducing I/O overhead. Use of the SystemVerilog `$display` task is not recommended, since it cannot be filtered out. Nevertheless, extra overhead may well persist when checking the verbosity level of each `` `uvm_info `` call, especially for testbench acceleration. Questa simulator functionality can optimize away all `` `uvm_info `` calls with verbosity lower than a specified threshold, completely eliminating all underlying checks.

UVM factory

Not all classes need to be registered with the UVM factory and subsequently constructed with the `create()` method. For example, TLM ports, exports, and fifos for UVM component interconnect should all be created with the normal SystemVerilog class constructor `new()`, simply because they do not generally need to be subjected to factory overrides. The `create()` method adds a run-time penalty that is clearly wasteful when factory semantics are not required. On a related note, when constructing and de-referencing objects dynamically (e.g. sequences and sequence items), there is no need to use the `create()` method on the same object handle indefinitely in a loop. Instead, construct the object once outside the loop, and then clone it inside. Further details are provided in [8].

UVM registers

Use of `get_*_by_name()` methods requires attention, as these are expensive when traversing entire register blocks for matching names. Additionally, register coverage is probably not needed in every simulation run. Generally speaking, enabling functional coverage on large register blocks results in poor run-time performance, so enable UVM register coverage only when required.

UVM recording

Enabling UVM or simulator-specific transaction recording APIs should be limited to debug runs. Refrain from using the `` `uvm_field* `` automation macros—instead, provide an efficient custom implementation via the `do_record` method.

Constrained randomization

In many cases, architecting constraints and solution spaces with performance and efficiency in mind can result in significant performance boosts. The question often raised is: *why doesn't the tool do optimizations implicitly/automatically when executing my code?* The answer: simply, it is generally very hard to deduce applicable optimizations from extensive user code. Considerations for SystemVerilog constrained randomization performance are well-detailed in [9], including the following:

- Reduce the problem's solution space as much as possible. When possible, free up the constraint solver from unnecessary redundant activity. For example, if a running test does not inject errors in packets or generate invalid instructions, the constraint solver need not deal with the error injection or invalid instruction constraints. Safely disable these activities using the SystemVerilog `constraint_mode()` and `rand_mode()` built-in methods.
- Be prudent with the use of rand variables. Class data members modeling transaction attributes generally do not all need to be generated randomly. Declare rand variables judiciously with respect to their

datatypes. For example, do not use an integer if a few bits or a byte will do, and do not use a signed data type if all you need is unsigned.

- Replace simple constraints with procedural assignments in the built-in `pre_randomize()` or `post_randomize()` methods where practical, using system functions like `$urandom()` and `$urandom_range()` as needed for plain random value generation. These system functions would tap into the same random number generator as the `randomize()` method itself.
- Replace complex operators with equivalent simple operators. For example, use the shift operator to implement multiplication, division, or power operations, and use a simple part-select or range (in)equalities to implement relational operations.
- Avoid using `randc` modifiers and `foreach` constraints unless really needed.

Assertions and functional coverage

For the purpose of assertion-based and coverage-driven verification, the Veloce emulator can synthesize a generous subset of SVA assert and cover directives, and SystemVerilog covergroups [11]. Just as for procedural testbench code, implement assertions and coverage models with synthesis for emulation in mind, so they can be readily mapped onto the emulator when needed. For testbench coverage and assertions in the HVL domain, follow these performance guidelines to lower testbench overhead:

- Disable assertions and functional coverage when possible (understanding the visibility given up to gain performance).
- Avoid assertion pass and fail messages, since these increase I/O time. All simulators (should) have methods for querying statistics of passing and failing assertions.
- Prevent functional coverage over-sampling, i.e. triggering sampling of a SystemVerilog covergroup too frequently for the coverage objective at hand. Instead, sample covergroups only on relatively infrequent abstract transaction boundaries and events. Also, do not create covergroups inside dynamic transaction objects, since these may be constructed manifold. Associate covergroups with the fixed testbench components instead.
- Use guard conditions to disable assertions and covergroup sampling when possible.

Foreign-language applications

Foreign applications are applications written in another language (usually C/C++ or SystemC) that run with your simulation, typically acting as reference models or performing fairly complex analysis tasks via runtime queries. Foreign application run-time overhead is frequently overlooked, so it is important to profile foreign application code as well. The Questa simulator includes technology that profiles the source code of foreign apps as a result of their run-time activity. Note: the SystemVerilog Direct Programming Interface (DPI) is an efficient solution for integrating foreign language models with Verilog designs, avoiding the complexity and overhead of the traditional Verilog PLI. The latter often requires a level of design visibility that may well impact aggressive optimizations performed by the simulator.

In general, prevent too many context switches between the running simulator and the foreign app (e.g. calls on every clock cycle). Instead, aim to aggregate and buffer calls as much as possible. In regard to the SystemVerilog DPI, bias high-frequency communication in the import (SystemVerilog-to-C) direction over the export (C-to-SystemVerilog) direction, and avoid complex expressions to boundary method arguments (e.g. concatenation, casting, non-matching datatypes, open array argument of non-top-most array dimension, etc.).

C. Profiling acceleration runs

As explained in Section III, testbench acceleration run-time consists of three major constituents: hardware time, software (or testbench) time, and communication time. The Veloce emulation solution provides the means to profile testbench acceleration runs accordingly, giving a detailed account of communication activity and context switching across the hardware-software boundary, and of related aspects of software activity. Figure 8

shows an excerpt of an acceleration profile report. Additional aspects and specifics of software execution activity can be obtained from the Questa simulation profiler, as previously discussed.

```
================================================================================
TIME STATISTICS
================================================================================
Total run-time        : 3191.99
   Hardware Time      : 2179.59
   Software Time      :  974.86
   Communication Time :   37.54
--------------------------------------------------------------------------------
   Software Time : 974.86
      Two-Way Callers         :   3.20 (See Two-Way Callers TABLE for more info)
      Callees                 :   1.20 (See Callees TABLE for more info)
      System Tasks            : 723.61 (See System Tasks TABLE for more info)
      Input Pipe              :   2.31 (See Input Pipes TABLE for more info)
      Software Execution Time : 244.53 (See Software Execution Time TABLE for more info)
--------------------------------------------------------------------------------
   Communication Time : 37.54
      Total number of context switches : 393368
      One-Way Caller Execution Time    : 890.00
================================================================================

...
```

The hardware time to execute the HDL domain with RTL design on the emulator depends on the emulator clock frequency—the higher the clock frequency, the smaller the hardware time. Albeit for the right reason (faster emulation), recall that improving the clock frequency actually lowers emulator throughput (as defined earlier) because the relative overhead from testbench execution and communication is amplified. Clock frequency is a function of the combinational logic critical path, and even though critical path and other emulator platform optimization specifics are outside the scope of this paper, it should be considered that this emulator critical path may stem from the RTL design itself, or from the hardware-software glue logic forming part of the hardware-software communication boundary.

The software time taken by the workstation to execute tasks while simulation time is halted (design clocks in the emulator are stopped) includes, but is not limited to, time spent by the simulator to execute the testbench untimed processes, the elapsed time of system tasks called from the HDL domain (e.g. $display, $fwrite/$fread, $readmemh, $writememh), import DPI functions (a.k.a. callers or HDL-to-C calls) and export DPI functions (a.k.a. callees, or C-to-HDL calls), waveform dumping, etc. The communication time comes from context switching between emulator and workstation and is directly proportional to the number of context switches and associated data transfer sizes.

D. Minimizing acceleration overhead

While the adoption of testbench acceleration using emulation does not generally immediately ensure solid performance improvements, due to the comparatively much slower simulator and the significant added cost of simulator-emulator communication, the following guidelines promote best practices for minimizing acceleration overhead.

Reducing software overhead

In addition to the testbench optimization recommendations from Section IV.B., the following considerations aid in reducing software overhead:

- When suitable, and hardware capacity is not an issue, migrate testbench segments of significant overhead to the HDL domain (e.g. CRC generators, partial processing/analysis of results, etc.).
- Minimize SystemVerilog system tasks with large overhead. Each time a system task is called, the emulator must be stopped to yield to the host machine to execute the system task, and then resume after the system task is finished. System tasks are typically used for dumping log files or loading memories. While booting memories may be critical for functionality, dumping log files generally is not. The

acceleration run-time profile in Figure 8 demonstrates that the overhead from system tasks can be significant (23% of total runtime), so reconsidering their use is essential for boosting performance.

Selecting the most appropriate hardware-software communication scheme

Transaction-based communication for testbench acceleration can be classified into two main categories, streaming and reactive. Streaming communication is characterized by producer and consumer (i.e. sender and receiver) being largely decoupled, with little or no dependence on state. Examples of streaming applications are audio, video, and Ethernet traffic. In reactive communication, transaction transport from producer to consumer happens instantaneously, in one delta-cycle. Typical examples are configuration or control transactions.

The aforementioned *SCEMI 2 transaction pipe* is a good candidate for streaming communication modeling. It constitutes a buffer, accessed via function calls, that provides a means for streaming transactions to and from the HDL domain [3]. Input pipes pass inbound transactions to the HDL domain, while output pipes pass outbound transactions from the HDL domain. Data sent by the producer, though guaranteed to arrive in order at the consumer, is not guaranteed to be available to the consumer immediately after it is sent, depending on how buffering is used.

Function-based HVL-HDL (i.e. VIF) or DPI-C communication interfaces are well-suited for reactive communication modeling. These interfaces are pertinent when data needs to be consumed immediately, and is dependent on the current design state (e.g. interrupt responses). Reactive transactions can increase context switching, since transactions are often small and passed back and forth frequently, without fully utilizing co-model channel maximum capacity per transfer.

Optimizing hardware-software data transfers

Optimizing data transfer efficiency between simulator and emulator is predominantly about maximizing physical bandwidth utilization. Extra efficiency can come from hardware-software concurrency opportunities. As a general guideline, use pipelined transfers as much as possible (e.g. pure import DPI functions, pipelined export functions, SCEMI pipes), and use round-trip calls (i.e. those with output arguments) sparingly.

Maximizing bandwidth utilization

- Widen transactions by raising the abstraction level to “meta-transactions” to better utilize the available co-model channel bandwidth per transfer. Typically, a co-model channel can exchange a few kilobytes of data per transfer. Also, in concept, wider transactions remain longer inside the RTL DUT, and hence, the emulator.
- Minimize context switching by buffering/aggregating small transactions to send them together at once. This can be accomplished with both reactive and streaming channels of communication. For streaming pipes, as long as the interface permits, maximize the pipe buffer depth. Also, do not flush a pipe unnecessarily every time a transaction is being sent; it is automatically flushed when the buffer is filled. Occasionally, a user may want to manually flush a pipe for functional reasons (e.g. control or barrier transactions). For reactive interfaces, there are often cases when specific transactions do not need to be consumed instantaneously. For example, when performing boot or register initialization sequences, write transactions from HVL to HDL domain can be aggregated and transferred together. High performance protocols such as AMBA AXI or ACE support streaming options for these kind of scenarios. Figure 9 demonstrates how AXI4 successive register write transactions can be aggregated and consumed together at one time, typically when encountering a register read transaction, or otherwise when the completion of all outstanding write transactions is required.

Maximizing hardware-software concurrency

As pointed out earlier, hardware and software domains need not permanently proceed in lock-step, but can sometimes be executed concurrently. For instance, when the hardware initiates a *one-way* outbound non-blocking transfer to be consumed by the untimed testbench (by way of a void function with no output arguments initiated from the HDL domain; see Figure 6), the Veloce emulation solution generally need not stop the hardware design clocks running on the emulator while communicating with the software, and both the emulator

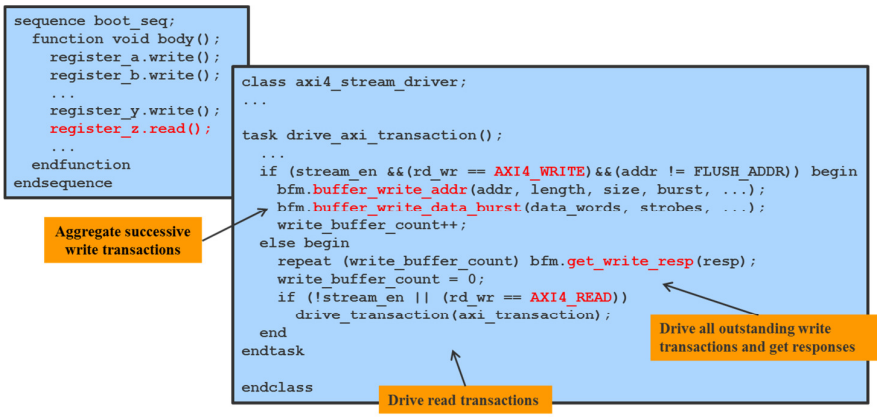

Figure 9. Aggregating AXI4 transactions into wider data transfers

and the simulator can proceed in parallel, boosting emulator throughput and thus testbench acceleration speed-up. Common testbench applications for this so-called One-Way-Caller (OWC) *outbound concurrency* feature include:

- Analysis components (like scoreboards, coverage collectors, register predictors, etc.) waiting for transactions from the accelerated monitor transactors (see Figure 6).
- Control components waiting for specific design states at runtime. For such scenarios, the software does not need to poll the hardware for specific design conditions; rather, the polling can be executed in the HDL domain, which can then notify the HVL domain when the poll condition is satisfied, using a one-way outbound function call. Figure 10 shows a scenario where a stimulus generator is waiting for the

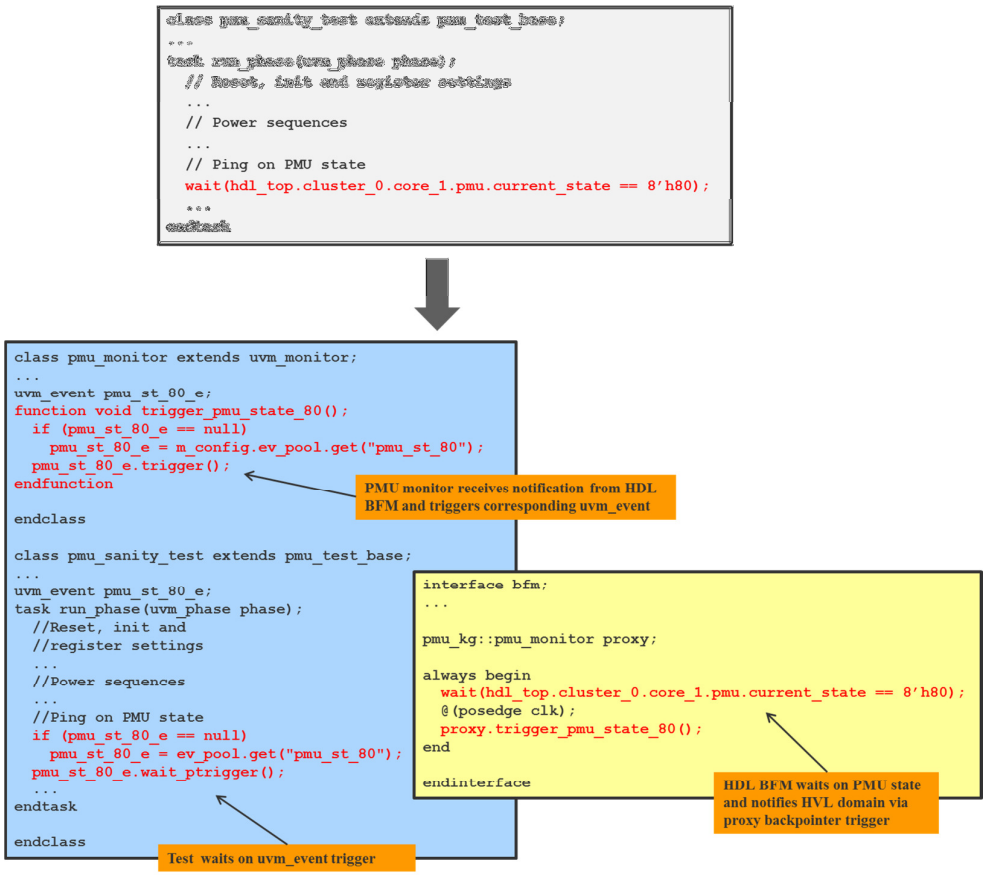

Figure 10. Maximizing hardware-software concurrency via OWC

occurrence of a specific DUT state. Instead of coding the generator to wait on hardware events, it can wait instead on pure testbench events triggered, or notified, by the hardware. The code shown prevents lock-step behavior between the hardware and testbench, making them work concurrently to maximize performance.

The Veloce emulation solution also supports concurrent data transfer in the inbound direction (i.e. from HVL to HDL domain, though again initiated from the HDL domain). *Inbound concurrency* can be of benefit to both reactive and streaming communication scenarios [6].

E. Optimizing the overall testbench acceleration flow

Environment components and solutions must be chosen carefully to ensure a unified interoperable environment between pure simulation and testbench acceleration. As for pure simulation, testbench acceleration should always be run in the best condition; i.e. do not enable full visibility and simulation debug flags by default, but only when rerunning failing tests. Likewise, enable functional and code coverage only when relevant. Additionally, eliminate significant initialization phases that are common across tests (e.g. initial memory upload, PCIe link training, etc.) using the Questa and Veloce checkpoint-restore technology [12].

F. Optimizing regression and test scenarios

Tests must be chosen wisely—select only effective tests for testbench acceleration runs. Unlike pure simulation, the privilege to run many tests in parallel should not be taken for granted when using an emulator, due to limited hardware resources or unavailable emulator slots. Grading tests for efficiency and metrics closure is crucial, as is selecting effective long-running tests of significant contribution to the verification plan. Consolidation of related or overlapping tests/scenarios may also be useful. Test grading against metrics contribution is further elaborated in [13].

V. CASE STUDIES

Table I lists several diverse case studies of SystemVerilog and UVM testbenches that benefited from recommendations discussed in this paper. A detailed exposé of one of these success stories can be found in [15].

Table I. SystemVerilog and UVM testbench acceleration results

Design Description	Design Size (gates)	Pure Simulation Time (hrs)	Testbench Acceleration Time (sec)	Speed-up
Application Processor	+200M	151	3060	177X
Network Switch	34M	16 ½	240	245X
Graphics Sub-System	8M	86 ½	635	491X
Mobile Display Processor	1.2M	5	46	399X
Memory Controller	1.1M	5	308	60X
Face Recognition Engine	1M	½	6.58	128X
Wireless Multi-Media Sub-System	1M	53	658	288X
Raid Engine Controller I	25M	13	174	268X
Raid Engine Controller II	25M	15.5	327	171X

VI. REFERENCES

[1] IEEE Standard for SystemVerilog, Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2012, 2012.

[2] UVM User Manual, uvmworld.org.

[3] Standard Co-emulation Modeling Interface (SCEMI) Version 2.2, Accellera, January 2014.

[4] Off To The Races with Your Accelerated SystemVerilog Testbench, Hans van der Schoot et al., DVCon 2011.

[5] UVM Cookbook - Emulation, Mentor Verification Academy.

[6] Concurrent Operation of Emulator and Testbench, Russell Vreeland, Mentor Graphics Tech Note, 2012.

[7] SystemVerilog Performance Guidelines, Verification Academy.

[8] UVM Performance Guidelines, Verification Academy.

[9] Are Macros in OVM & UVM Evil? – A Cost-Benefit Analysis, Adam Ericsson, DVCon 2011.

[10] The SystemVerilog and UVM Constrained Random Handbook, Ahmed Yehia, Mentor Graphics Tech Note, 2015.

[11] Mentor Graphics Veloce Reference Manual.

[12] Want a Boost in Your Regression Throughput? Simulate Common Setup Phase Only Once, Rohit K Jain & Shobana Sudhakar, DVCon 2015.

[13] Are you really confident that you are getting the very best from your verification resources, Darron May, DVCon 2014.

[14] UCIS Applications: Improving Verification Productivity, Simulation Throughput, and Coverage Closure Process, Ahmed Yehia, DVCon 2013.

[15] STMicroelectronics: Simulation + Emulation = Verification Success.