Conscious of Streams

Managing Parallel Stimulus

Jeffrey Wilcox
Paradigm Works, Inc.
300 Brickstone Sq. Suite 903
Andover, MA, USA
jeff.wilcox@paradigm-works.com

Stephen D'Onofrio
Paradigm Works, Inc.
300 Brickstone Sq. Suite 903
Andover MA, USA
stephen.donofrio@paradigm-works.com

Abstract— **Applying parallel logical streams of stimulus to a DUT (Device Under Test) is a common verification requirement, whether these streams are defined by physical interface signal values, transaction content, or other characteristics. This paper explores a set of architectural choices to be made in providing a solution for such a requirement. Each solution is considered in terms of measurable performance metrics, flexibility of the solution, partitioning of functionality, and ease of management. We will consider whether one can arrive at a single solution reasonably suited for all situations. While the paper focuses on a UVM (Universal Verification Methodology [1]) solution space, the concepts are transferrable to other current high-level verification languages.**

Keywords- UVM; UVC; stacked UVC; logical streams; architecture; performance

I. INTRODUCTION

The need to shape and deliver parallel streams of stimulus to the DUT is a frequent issue. A best-case solution for delivering such stimulus must balance the often conflicting requirements of performance, flexibility, and proper partitioning. In terms of performance, typical metrics are image size and runtime. Primary points of concern for flexibility include ease of managing the number of streams, controlling each stream's characteristics, and modifying any arbitration algorithm. Issues of partitioning include maintaining each verification component as independently reusable, protecting the generally inflexible aspects of protocol from unintended modification, and providing a reasonable user API for managing flexible aspects of the protocol.

II. PROBLEM STATEMENT

There are several reasons one may wish to make use of parallel, logical streams of stimulus, many of which will impose requirements beyond the innate capabilities of most verification languages and methodology base class implementations. There may be a required limit on the rate at which transactions are delivered on a particular stream. There may be periodic operations that need to interleave with the more general stimulus applied. There may be a need to utilize transactions and sequences developed for a higher level of abstraction, perhaps utilizing third party IP. Such third party IP may provide useful sequences that one would like to be able to apply to some other interface protocol, or simply because one wants to maximize reuse of stimulus between system-level and block-level testing on a component.

The base classes provided by UVM (Universal Verification Methodology) and like methodologies have some capacity for managing parallel streams of stimulus. There is provision for arbitration between available sequences in a UVM sequencer, which may be fine for default cases, but may not suffice for more complex situations. SystemVerilog's [2] **dist** constraint type likewise provides some control over relative frequency of one stream versus another. However, that doesn't suit metered delivery concept well.

What is wanted in these cases is typically an approach that allows each logical stream to be independent of, and non-blocking to other logical streams. These stimulus sources may be free running throughout the operational portion of a test or not. They may be native to the interface being tested or not. They may be of like protocol one with another or not (example: PCIe and Ethernet traffic protocols both being transferred across some shared interface of the DUT, whether external or internal.)

III. PREPARE YOUR PAPER BEFORE STYLING

Architectural choices made in developing the testbench can have a large impact on simulation performance. These impacts should be a driving factor in these choices. Simplicity of testbench and test development count, but runtime for tests will have the greater impact on schedule in the end.

Five approaches are considered, and the results compared. The approaches consider two primary sets of choices as to how to achieve parallelization. The first choice is whether to achieve parallelism by forking off multiple sequences within a single sequencer or by constructing a separate sequencer for each stream.

When the application requires conversion from one transaction type to another, a second choice arises as to whether to implement conversion within the higher or lower verification component. Such conversions may be due to applying sequences foreign to the interface verification component, segmentation requirements, or other issues. Here, the choice is whether to convert in the higher or lower verification component.

A. Basic UVC Structure

To begin with, consider the general description of a standard UVM verification component, or UVC [3]. For purposes of this discussion, we will consider the structure of a UVC implementing a single agent. The typical UVM agent consists of a sequencer for creating stimulus, a driver for applying said stimulus to the interface wires, and a monitor for reconstructing transactions observed on the interface. For the purposes of this paper, any response path from driver to sequencer is ignored.

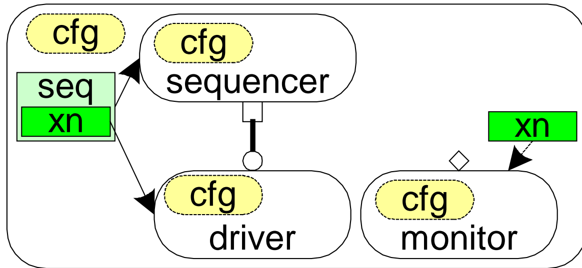

Figure 1. UVC Structure

In Figure 1. through Figure 7. the dark green “xn” items are transactions, and the lighter green “seq” items are sequences. These indicate the object instances constructed at a given point in time. For example, in Figure 1 the “seq” object is constructed in the sequencer, and contains an “xn” object to which both sequencer and driver have a handle. The monitor constructs a separate “xn” object which will be forwarded out its analysis port TLM (Transaction Level Modeling [4]) port.

The yellow “cfg” object is a single configuration object to which all components of the agent have a handle. This object may have variable runtime controls for the UVC, and / or state shared between the components.

Sequences may be applied to the sequencer under direction of the test being run, or the sequencer may be assigned a specific sequence to be run for the duration of one or more test phases. Sequences are typically created at the driver’s request, and a single transaction from this sequence sent to the driver. Thus, at any point in time there is but one sequence / transaction on the active side of the UVC (composed of sequencer and driver), and one on the passive (monitor) side. This situation changes as parallel streams of stimulus are created.

B. Where to Parallel

The first question to answer is whether to achieve parallelism by implementing parallel sequences or parallel sequencers. Certainly, issuing parallel sequences on a single sequencer is the simplest to conceptualize.

1) Parallel Sequences

In Figure 2. a new sequence, main_seq, is introduced. The body of this sequence constructs a seq instance for each logical stream, launching each seq as a forked process, and allowing it to run indefinitely. Note that while this sequence is shown as issued by the UVC’s sequencer, it could as easily be running on the testbench’s virtual sequencer. There is always one instance of each stream’s sequence constructed. In this case, which sequence next provides a transaction to the driver is indeterminate, depending only on the order in which the several streams happen to construct transactions. The driver will, however, have handle to at most one transaction. One further transaction instance is constructed by the monitor observing the interface, which it provides via TLM to any interested parties.

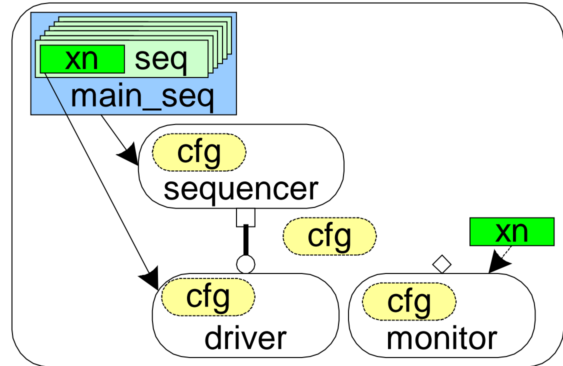

Figure 2. UVC Implementing Parallel Sequences

Of the five methodologies considered, this approach maintains a significant advantage for image size, and is comparatively good in terms of runtime. Little is added to complexity. The UVC itself requires no modifications. What is added is a containing main sequence (shown in blue). Beyond forking off the logical stream sequences, this sequence may need a means to terminate the logical stream sequences when the test’s active stimulus period (run_phase in UVM) is complete. If arbitration is needed, it would also be implemented in this main sequence. This avoids the need to extend the UVC’s sequencer.

Partitioning is not ideal in this case. Arbitration and termination of the logical stream sequences are both managed by the main sequence. Providing separate methods within the main sequence to implement these activities will help. The impact is minimal, but it is something to remain aware of.

Managing the characteristics of each logical stream may require development of unique sequences for each stream. In many cases, a common sequence can be used by applying constraints to fields of the sequence. This simplifies implementation of the main sequence, but it may limit its flexibility. The common sequence can be instanced as an array, simplifying the main sequence. Using a dynamic array will make selection of included streams more efficient.

Use of a common sequence is not a requirement of the approach, then, but shows distinct advantages. Adding or removing streams for a given test can be easily accomplished. Including a control field in the main sequence and constraining it as desired from the test gives a means of determining which logical stream sequences to construct. Type of sequence to construct, or constraints to be applied, may be fixed based on stream number, or may be configurable by further fields in the main sequence.

2) Parallel Sequencers

The alternative of using parallel sequences is to provide a separate sequencer for each stream. The greater complexity of this approach is clear from Figure 3.

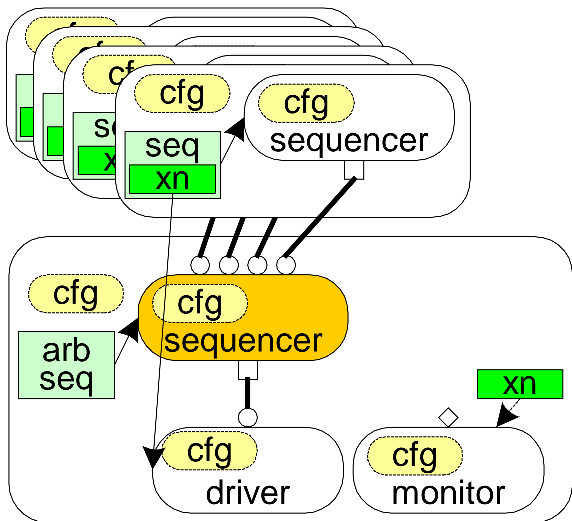

Figure 3. Using Parallel Sequencers

In Figure 3. through Figure 6. orange components require extending the UVC component. Factory methods can be used to replace the original components with their extended versions.

This approach builds on the stacked UVC approach, with the sequencer of an upper, or "upstairs" UVC connecting to a lower, or "downstairs" UVC's sequencer, rather than its own driver. For this discussion, suppose upstairs and downstairs UVCs to be of the same type. One added requirement for stacked UVCs is a means to deconfigure the driver in the upstairs UVC. No other modifications are required for the upstairs UVC.

In the downstairs UVC, we must provide new TLM connections into the sequencer component, which in turn requires that we introduce a sequence, a sequencer method, or some other means by which to select between those TLM connections. The pull type nature of these connections, where the driver requests an item which the sequencer then provides, may impose certain restrictions on the way arbitration is implemented. In a TDM (Time Domain Multiplexed) application, for example, stream selection is very deterministic. In other cases, the driver may implement some means of indicating which stream is requested. A typical case would be to simply arbitrate in round-robin fashion between all available streams.

Whatever the mechanism, its implementation should take steps to avoid creating another set of sequence or transaction instances. Care should also be taken to avoid constantly requesting new sequences from the upper UVCs, leading to further unnecessary increases in image size at the beginning of the test.

Whether it is easier to manage constructing and connecting sequencers or whether it is easier to manage sub-sequence construction for a given test is debatable point. If one typically applies a default sequence to each sequencer which it runs throughout the active portion of the test, then managing at the sequencer may actually be more intuitive. It can be controlled easily enough by including in the testbench environment the same sorts of knobs that were described for the main sequence used in the parallel sequence approach. This would impact UVM's build_phase and connect_phase, but is otherwise fairly non-intrusive.

Clearly, adding parallel sequencers adds complexity. Data shows that it will also have an adverse effect on image size and performance. Given that complexity, why would one choose such an approach? What advantages does it offer?

One advantage lies in the fact that we can now partition the arbitration as a separate sequence running on a separate sequencer, or as a method of that sequencer if the algorithm is fixed. Having a dedicated sequencer for each stream may simplify the nature of the sequences themselves, or increase the opportunity for reuse of existing sequences.

The primary advantage of the approach is that it allows for the possibility that not all streams are even of the same fundamental type. Suppose a given logical stream may be Ethernet or ATM traffic, and that differing third party IP is utilized for both types. These sequences cannot easily be supplied to a common sequencer. However, having multiple UVCs in the upper level allows the possibility that not all UVCs are of the same kind.

Where there is a need for multiple stimulus types, this approach is mandated. Other situations requiring conversion may require a stacked UVC approach, but not necessarily the parallel instances of the upper layer suggested here.

C. Where to Convert

Supposing a need for conversion, the next question is where conversion is to be done. We can assume at least one upstairs UVC. This UVC may implement some protocol which happens to be transferred across the downstairs interface. It may simply create complete packets while the downstairs UVC is operates on segments or fragments of these packets.

What is being done on the upstairs downstairs path likely needs to be reversed in the monitor components of the UVCs. This paper considers only the active, upstairs downstairs path. Should an upstairs UVC convert to downstairs transaction types, or should the downstairs UVC accept the upstairs transaction type and do the conversion?

1) Downstairs Conversion

UVM advocates performing the conversion downstairs, as a part of extending the sequencer. The changes to the parallel sequencer approach are shown in Figure 4. Upstairs sequences and transactions are shown in blue from here forward.

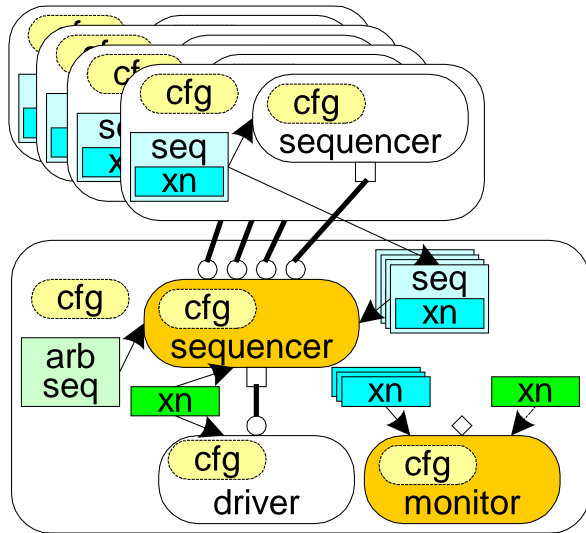

Figure 4. Parallel Sequencer Implementation with Downstairs Conversion

The downstairs sequencer needs further extension. Previously, it had been fit with TLMs to field sequences from upstairs. Now, it must also implement whatever conversions are required. This may be a one-to-many conversion for a segmenting interface. It may be translating from a "foreign" transaction type to native. While the monitor path is not in focus in this paper, it must be observed that the monitor will need to be extended to translate in the reverse direction. Where segmentation is being done, this also introduces an array of partial transactions that are not yet ready to be passed upstairs.

The primary benefit of this approach is that it limits component modifications to the downstairs UVC. However, it does introduce potential issues in terms of properly partitioning operations. We still have the need for an arbitration sequence. Now, we add the need for a sequence or method per inbound TLM to perform the conversion. The choice of sequence or method hinges on how fixed the operation is. If the operation is fixed, a method is preferable, as it better hides the operation from the user. If it is more variable, a sequence provides the user with better control over the variability.

2) Upstairs Conversion

Given that a UVC typically translates from the higher abstraction of a transaction class to the lower abstraction of driven signals, it may feel more consistent to perform conversion upstairs. This more neatly partitions the tasks required. The upstairs UVC creates the stimulus and converts it to the transaction type of the downstairs UVC, an activity analogous to that of a standard driver component. The downstairs UVC deals solely with issues of arbitration, driving the selected transaction on the wires.

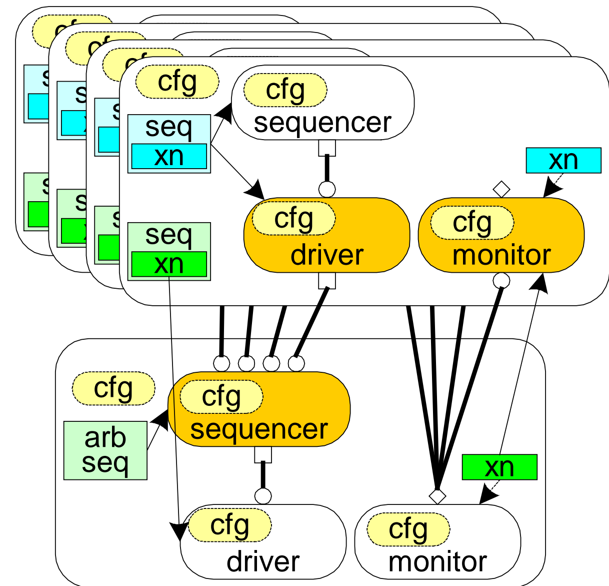

Figure 5. Parallel Sequencer Implementation with Upstairs Conversion

The upstairs UVC is shown with a driver component in Figure 5. Ideally this component would not be needed. However, sequencers are strongly typed such that the sequencer's TLM necessarily matches the transaction type provided by its sequences. Short of completely rewriting the sequencer, this is not something easily changed. Thus, we instead extend the upstairs driver, replacing its usual interface with another TLM.

This new TLM is of the same nature as the sequencer TLM, but takes transactions of the downstairs UVC's type. The driver is thus required to provide all the methods of this request/response type TLM. Methods may be left as unimplemented callbacks (task foo(); endtask) if one can be certain the functionality will not be required. However, maximum reusability will be achieved by implementing all such methods completely.

On the passive side, all that has really happened is that the monitor extension is done upstairs rather than downstairs. The TLM added to the upstairs monitor is again typed for the downstairs transaction type. This extension will need some means of identifying which downstairs transactions belong to its stream and which can be ignored.

This is the most complex solution to implement. It is also the most flexible. Returning to the case of logical streams with differing protocols, consider the conversion issue. If conversion is done downstairs, then we must have multiple arrays of TLM connection, each dedicated to a specific protocol. The downstairs monitor will likewise require at least one TLM per upstairs transaction type. Further, the transform methods become more complex, as they now must determine both the type being transformed and how that transform is to be accomplished.

There is a partitioning advantage with upstairs conversion. Each driver or monitor extension deals only with a single sort of conversion. The downstairs UVC remains wholly agnostic to what sort of stimulus its transactions derive from, concerned

only with arbitrating between whatever comes in, and blindly delivering all observed results upstairs.

D. *Simplification*

There is nothing intrinsic to the need for conversion that requires parallel sequencers upstairs. Both upstairs and downstairs conversion can be done with a single upstairs UVC. That upstairs UVC can then implement a parallel sequence approach just like the first approach considered.

This is shown with upstairs conversion in Figure 6.

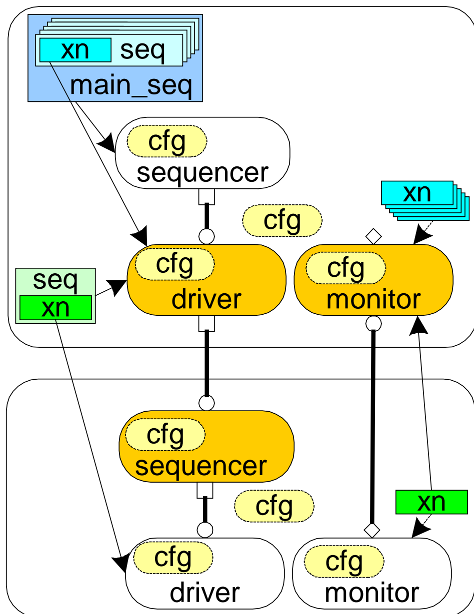

Figure 6. Parallel Sequence Implementation with Upstairs Conversion

The primary benefit is a reduction in the number of UVCs and TLMs in our testbench, reducing overall image size. It may also be observed that managing the response return path on the active side is much simplified, as is the managing of reconstruction on the monitor side.

Similar simplification can be achieved with downstairs conversion. This will introduce the need to manage the interface UVC sequencer extension so as to avoid allowing one stream to block others. Again, the thing to watch is that one does not wind up requesting an infinite series of upstairs sequences and thus increasing the image size unnecessarily.

The case of a segmenting interface UVC can be construed as a case for this approach, although contained in a single UVC. This is shown in Figure 7. This could also have been implemented as a stacked pair of UVCs. However, where segmentation is integral to the interface this is cleaner.

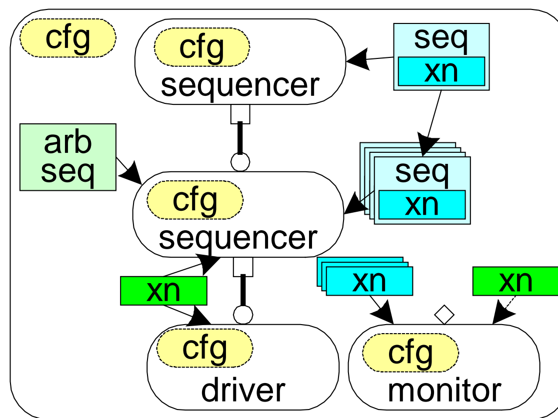

Figure 7. Segmenting UVC Implementation

E. *Performance Assessment*

Experiments were run to compare these approaches. Each testbench used a UVC developed by Paradigm Works and implementing Altera's Avalon Memory-Mapped Interface for the downstairs UVC, and another implementing PLDA's EZDMA protocol for the upstairs UVC(s) when present. To better ensure that observed variations were due to the means used to obtain parallelism and not unrelated factors, all tests ran transactions of one fixed size, consisting of a single phase write to one common address. Where conversion occurred, it was always one-to-one.

For each approach, tests were run across a matrix of number of streams versus total number of transactions, so as to obtain a profile of any trends. Each test was run singly on the same machine, using the same simulator to further minimize unrelated variation. Image size is shown in Figure 8. Performance, normalized as CPU seconds per microsecond of simulation time, is shown in Figure 9. In the latter case, the ten transaction results are eliminated, as they are of limited value beyond indicating the fixed overhead of the test. These figures are specific to the approach of using parallel sequences with conversion done downstairs.

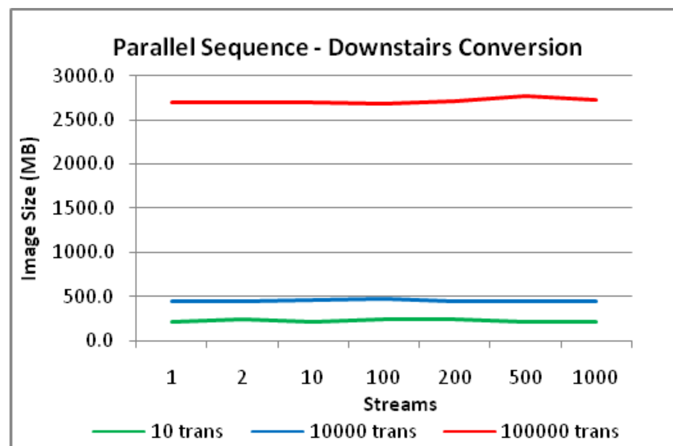

Figure 8. Image Size for Parallel Sequence Implementation

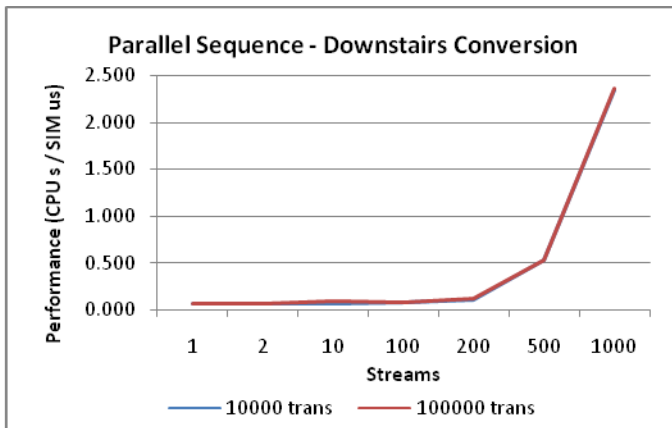

Figure 9. Performance for Parallel Sequence Implementation

Of particular note, image size does not show any signs of being impacted by parallelization of sequences. Further, while performance clearly degrades when there is massive parallelism, the trends are nearly identical for both the 10k and 100k run lengths. This suggests that any overhead induced during testbench initialization is relatively small.

For the parallel sequencer implementation, image size is shown in Figure 10. Performance is shown in Figure 11. The downstairs conversion option is again chosen. Whether using parallel sequences or parallel sequencers, moving conversion upstairs does not significantly change the trends seen in these graphs, although the specific values differ. Here, we see that while image size is still pretty constant, there is a distinct uptick when we arrive at the 1000 stream case.

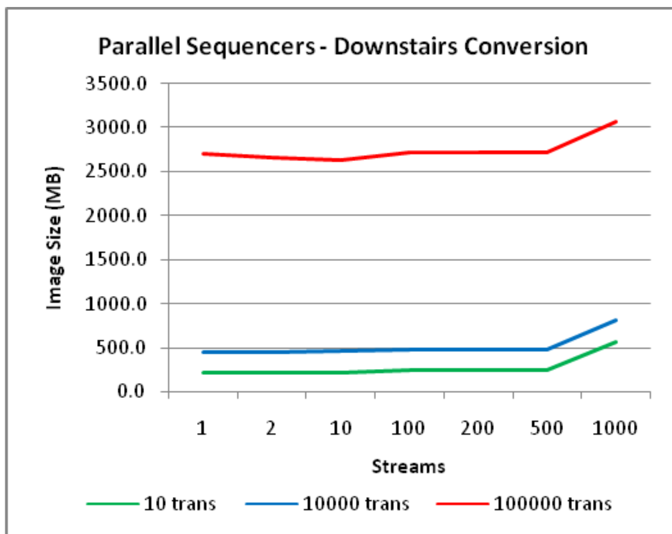

Figure 10. Image Size for Parallel Sequencer Implementation

The situation for performance is more interesting, and demonstrates the primary performance issue seen with parallel sequencers: There's a lot of zero time inertia to overcome as the number of streams rises. Notice that at the 1000 stream data point, the 10k performance data is more than double what is observed at 100k. The 10 transaction measurements (not shown) come in nearly 100x worse than the 10k. There's a great deal of time being spent during the build and configure phases with this approach. That's a factor to consider in the architectural decision. Necessity may dictate a parallel sequencer approach even where massive parallelism is expected. But, there's a huge upfront cost that applies to every simulation run. Plan for it.

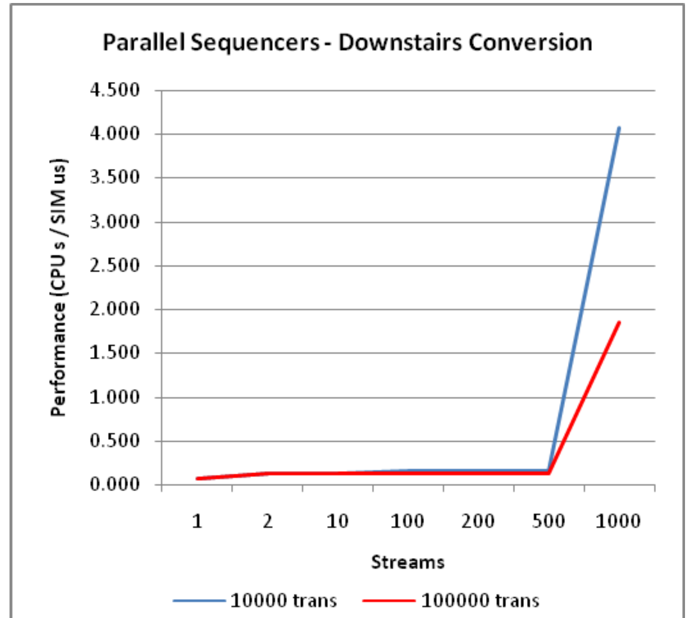

Figure 11. Performance for Parallel Sequencer Implementation

Next, we compare and contrast the five solutions in terms of these same performance metrics. Once again, we'll consider image size first, shown in Figure 12. The results are much as one would expect. Adding conversion has greater impact on the image size than does the choice of where it is accomplished. Barring the massively parallel cases, the cost is pretty much the same for all conversion approaches. In these latter cases, the parallel sequencer approach adds another 400MB or so to the image size.

Comparative performance results are shown in Figure 13. Here, one begins to see some significant distinction between approaches, particularly in the more parallel cases. Overall, it appears that parallel sequences have some advantage over parallel sequencers particularly for large-scale parallelization.

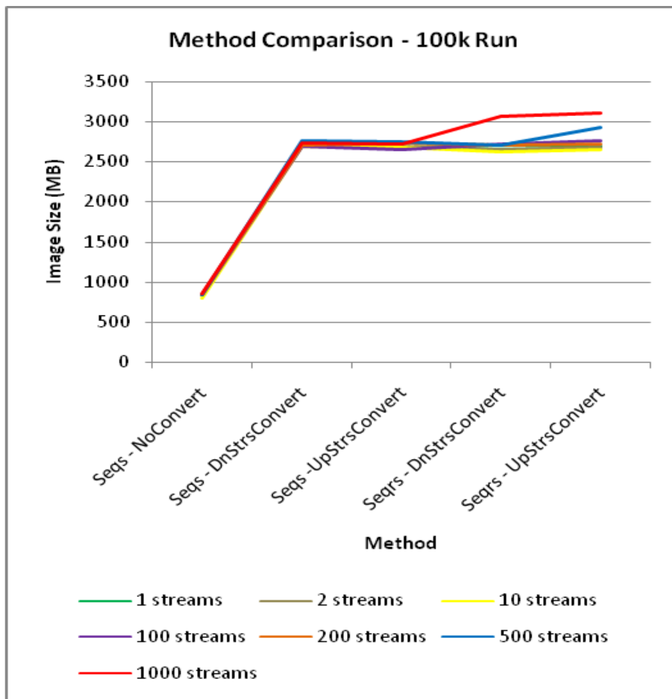

Figure 12. Methodology Comparison: Image Size

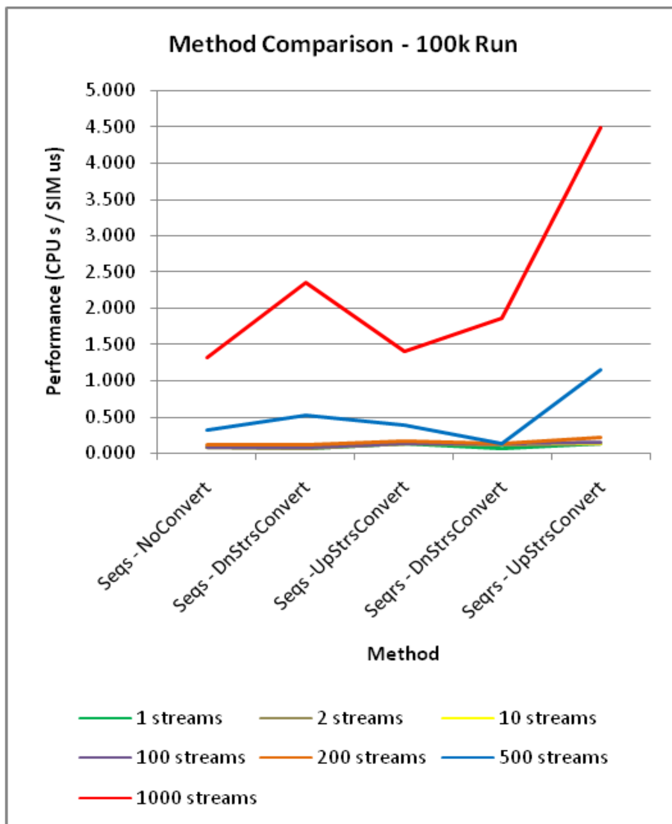

Figure 13. Methodology Comparison: Performance

Upstairs conversion, while more complex, is advantageous. Comparing only the "Seqs – DnStrsConvert" and "Seqs – UpStrsConvert" cases, there appears to be an inflection point in the number of parallel streams below which doing conversion downstairs is actually more efficient in terms of run time. Up to and including the 200 stream test, downstairs conversion performed better, running up to 2x faster in some cases. However, starting with the 500 stream case, the results toggle, and upstairs conversion begins to outperform downstairs.

It should also be noted that at these extremes, given the more generally observed degradations in performance, the differences are going to be felt much more strongly than in those earlier cases. The loss of .06s/us will be felt nearly as much as the loss of 1.1 s/us. It will depend on whether there is any possibility that you will ever need to support so large a number of streams. If there is one fundamental take away from this, it is that where the potential for massive parallelization of stimulus exists, the approach chosen becomes all that much more critical.

IV. Conclusion

Based on the results obtained with regard to performance metrics, as well as the relative complexity of the several approaches considered, there are some conclusions that can be drawn. First, it is clearly preferable to achieve parallelization through sequences rather than sequencers. The issue of where to convert is a bit murkier, but in general, downstairs conversion is to be preferred. It provides a reasonably straightforward means of handling conversion, and allows a sufficient degree of partitioning. The exception would be when massive parallelization is expected.

At the outset, we stated an interest in arriving at a universal solution that would fit all circumstances. The nearest we have to such a solution is that of a stacked approach utilizing parallel sequencers with any conversion done in the upper layers. However, this approach is also the most complex of those considered, and the worst performer. As such, our recommendation is that such an approach be used only when absolutely necessary.

A. *Example Application*

The examples used to obtain performance comparisons were highly contrived. What about a real-world example? Portions of a testbench for a device transferring Ethernet transactions over Altera's Avalon Streaming interface is shown in Figure 14.

Here, there are actually two levels of parallelization happening, both with conversion involved. The upstairs UVC is an Ethernet Frame Generator, producing a series of Ethernet transactions for each stream implemented. The sequences are free running, and provide means to manage the bandwidth utilized by each stream individually. The downstairs UVC implements the Avalon Streaming interface. This interface performs segmentation and provides physical support for up to 256 channels. Segmentation is handled by the addition of a second sequencer which implements the one to many transform as a method of the sequencer, and provides sequence based arbitration between available packets.

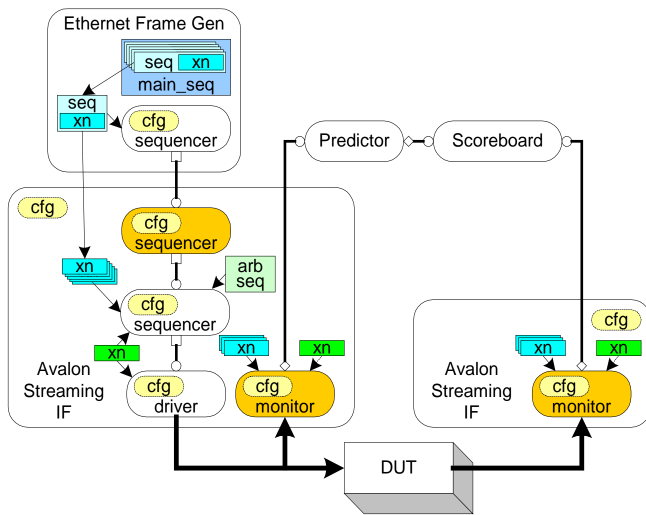

Figure 14. Testbench example

In order to support the Ethernet packet format, the upper sequencer of the Avalon UVC must be extended to repackage the Ethernet UVC's transaction into one native to the Avalon UVC. The monitor must also be extended to provide translation of the reassembled Avalon transactions back to Ethernet transactions. The monitor extension can be reused on the DUT farside, where a second instance of Avalon UVC is used in passive mode.

V. SUGGESTIONS FOR FUTURE WORK

Further efforts should be undertaken to evaluate the impact of return path implementation and managing multi-stream conversion issues on the passive (monitor) side. Another concern to be addressed pertains to what standard hooks and methods might be incorporated in the several UVM components to facilitate the conversion processes cleanly. Finally, replication of these or similarly structured experiments across the several simulation platforms would be advisable.

REFERENCES

[1] Accellera, "Universal Verification Methodology (UVM) 1.1 User's Guide," Accellera Organization. Napa, CA, May 2011.

[2] IEEE, "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language," IEEE Computer Society, New York, NY, Dec 2009.

[3] Accellera, "Universal Verification Methodology (UVM) 1.1 User's Guide," Accellera Organization. Napa, CA, May 2011, pp.3–5.

[4] Open SystemC Initiative, Translaction Level Modeling Library, Release 1.0.