Use of Portable Stimulus to Verify Task Dispatching and Scheduling Functions in an LTE Switch

Adnan Hamid
Breker Verification Systems
1879 Lundy Ave. #126
San Jose, CA 95131 USA
+1 512 415 1199
adnan@brekersystems.com

Abstract -**The verification of a Long Term Evolution (LTE) software-define switch requires a novel verification methodology. This paper describes one way in which Portable Stimulus can be used to mimic the structure of the software that will eventually operate on the hardware platform, enabling the platform to be verified before the availability of the production software. The approach uses a C++ graph-based scenario model that enables production software routines to be inserted as they become available and enables a seamless transition from simulation to emulation as the design progresses.**

The user had three objectives for the project. First was the ability to create testcases that could be scaled from simple to the most complex examples possible, second, vertical reuse from cluster test to full chip test and third, horizontal reuse from simulation to emulation to post silicon. The user also felt that they needed to have procedural C code that could be inserted into the graph and to help with the generation of the scenarios.

I. INTRODUCTION

There are many trends going on within the semiconductor industry[1]. Each of these alone could cause significant change in methodology, but when they are all converging at the same time, it may result in a complete rethink about certain aspects of the flow. In this paper, we will discuss the verification approach associated with the design of a Software-Defined Networking (SDN)[2] SoC, more specifically an LTE[3] switch, utilizing the emerging Portable Stimulus[4] language and discuss the ways in which software behavior can be mimicked within the verification environment.

The most talked about and probably most significant trend is the slowdown of Moore's law. While we have seen little slowdown in the rate at which the past few nodes have become available, they are getting more expensive and the expectation that each node will provide more transistors that are cheaper, faster and use less power, is no longer a given. In addition, each node is requiring additional and complex physical design steps that are adding to the design time, adding costs and increasing the risk associated with design. This is significant because it means that designs are likely to have a longer lifespan in the field and that means that they have to be more flexible than designs of the past. One design trend that has resulted is that designs are now containing a lot more general purpose compute power and designers can ill afford to have their architectures be unable to adapt and change to the constant transformations that are happening within the industry.

Another change that is affecting design and verification methodologies is the pressure for power reduction which is caused by two factors. The first is that total energy is becoming an issue for all electronic systems and not just those that are battery powered. At the same time, the power densities of chips has reached a danger zone, where causing too much activity in a system can result in extreme thermal impacts that can cause damage to the chip, or result in the product not being able to meet its longevity requirements. Both of these factors have to be included in a comprehensive verification methodology.

The migration of functional capability into software is creating problems for the design and verification of the hardware platform. Simply stated – how do you ensure that the design will meet all of the end-user requirements, when those requirements are not only unknown when the hardware is being designed, but may not be fully defined

until long after the chips have been designed, built and deployed? Software-defined systems are becoming quite commonplace and things that were once thought to require custom hardware are being made a lot more general purpose so that they can be upgraded for new protocols, security requirements and feature sets.

One of the segments in which this trend is accelerating is networking switches. In the most general sense, a switch is a design that connects one set of ports to another set of ports, possibly with multiple protocols and possibly with data transformation along the way. In the past, most or all functionality was built into the hardware, with no central CPU controlling the data flow. Many networking and telecommunications chips (switches, routers, hubs, modems, etc.) have traditionally fallen into this category.

II. The Design

Switches today are being designed with multiple general purpose CPUs coupled with a large array of engines capable of performing arbitrary networking functions. The popular term "software-defined networking" (SDN) has been coined for this type of design. The wireless Long-Term Evolution (LTE) standard is no exception to this trend. Many LTE chips contain multiple processors and present similar design and verification challenges to those seen in smartphones, set-top boxes, and other products that have used SoCs for some time.

However, this type of design requires a completely fresh approach to verification. The verification of each of the components, either individually, or even when integrated into the complete chip, does little to ensure that what has been produced is capable of performing the intended end function. Simply put, without the software, these devices perform no function. Unfortunately, the software has not yet been written, but the design teams need a reasonable level of confidence that the architecture is capable of supporting the needs of the software.

This paper discusses the verification process used for an LTE base-station chip that was designed using an SoC architecture as shown in Fig. 1. This portion of the LTE base-station chip contains N CPU sub-systems, connected to M task dispatching and scheduling modules by a full crossbar switch. Any CPU can send commands to any module, and minimum(M,N) paths can be active at any one point in time. Each command specifies a series of tasks to be executed, many of which may be dependent on other tasks within the command set. Some of these tasks involve data moves to and from memory which require the services of the DMA engines; others entail invoking various types of offload engines to perform data transformations. The Task Dispatcher is responsible for scheduling the individual tasks and assigning them to the Tasker engines that actually perform the tasks. Complex data structures are used to convey information between each of the task elements.

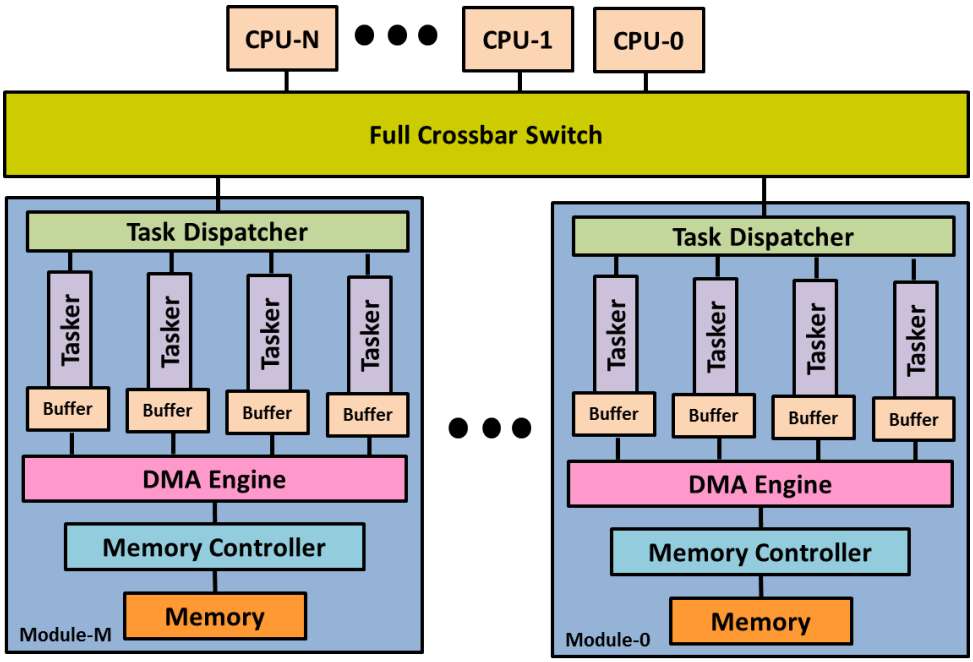

Figure 1: Architecture of the LTE switch

For memory-related tasks, a Tasker can schedule DMA operations to send or receive data from the main memory. Both on-chip and off-chip memory are accessible. For other tasks, the Tasker will handshake with the appropriate offload engine to initiate and complete its work. Although this sounds relatively straightforward, there are many variations and corner-case conditions that must be verified. These include:

- *Cross-covering all combinations of CPUs and modules*
- *Saturating all data paths with heavy concurrent traffic*
- *Exercising the full range of tasks types for each module*
- *Setting up scenarios that make it challenging for a Task Dispatcher to schedule all tasks*
- *Trying all memory types and modes supported by the controller*

The specific focus of this paper is the task dispatching and scheduling functions, which present many of the typical challenges for the verification of complex multiprocessor-based designs.

III. Verification Approach

In addition to the verification objectives outlined in the previous section, the team also wanted to combine simulation and emulation within their verification flow and to have the ability to run the tests on the actual silicon. This severely constrains the methodologies that would be suitable. While methodologies such as SystemVerilog and UVM are very mature in the context of simulation, they do not provide portability to the other verification platforms and cannot be combined with the desire to use parts of the actual software when they have been developed. In addition, the UVM methodology encourages the removal of processors, instead preferring to drive traffic directly onto the busses. This would make it very difficult to provide realistic and legal traffic patterns between the processors and the task dispatchers.

Another alternative would have been to write custom software test cases that would run on the CPUs. The generation of the software would be complex given the number of levels of task handoff, redirection between the levels of processing capabilities and memory banking scheme being used. In addition, given that the exact workloads were not known and many such testcases would be necessary to get to the confidence levels required, it was decided that a more efficient approach would be to use a methodology that would enable the generation of an arbitrary number of test cases, of various levels of complexity, and have the tool be able to generate all of the necessary code to run on the processors and to create any necessary coordination with outside events.

The verification team chose to solve these challenges using Portable Stimulus. Portable Stimulus is based on graphs that define the control and data flows for all usage scenarios supported by the SoC, and is currently being standardized within the Accellera Systems Initiative, Portable Stimulus Working Group. Portable Stimulus would allow them to seamlessly transition between simulation testbenches, software-driven test cases, and hardware platforms such as emulation as indicated in Fig. 2. They also chose to use a C++ graph-based scenario model to cover the intended functionality of the modules. The use of C++ made it trivial to call existing C routines, including chip initialization sequences, from appropriate points in the scenario model. Finally, the team chose the TrekSoC[5] portable stimulus solution from Breker Verification Systems to generate the test cases required to thoroughly exercise as many paths in the scenario model graph as possible.

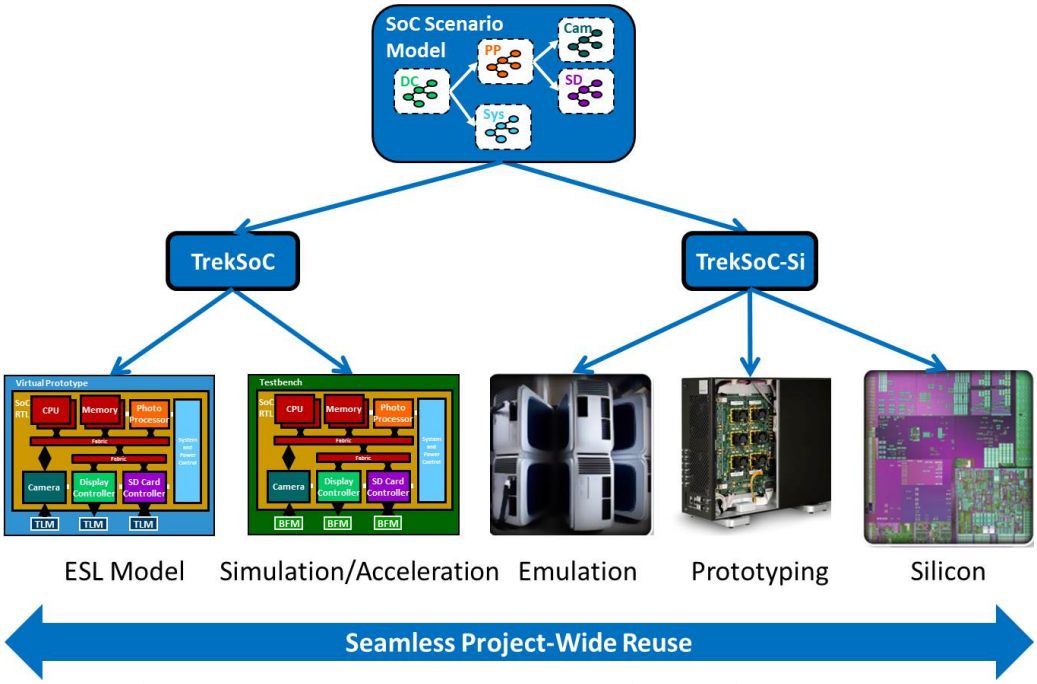

Figure 2: Portable Stimulus enables retargeting tests to multiple execution environments

Since the primary goal of the project was to verify the Task Dispatcher, it was modeled at the highest level of detail so that all aspects of its operation could be covered. The modeling contains the way in which tasks can be joined together to form commands, the data structures that are used to convey information between the tasks, and the interactions between the taskers and the memory management components, including the DMA engines. The general approach is to generate a number of commands, each of which has an arbitrary number of interdependent tasks. These are pre-generated and stored in the program that is to run on the CPUs. The generation of the commands should conform to all of the rules placed on the software, including the ways in which memory is allocated, transferred into the taskers and subsequent results transferred back into main memory.

Fig. 3 shows the graph that describes how the commands are composed of a number of interrelated tasks that are to be sent to the schedulers. While the details of this graph are not being made available due to the proprietary nature of the product being verified, the figure shows how graphs can be composed hierarchically. This figure captures the sequential dependencies of the command structure. Behind each of the nodes in the graph is the code that describes the data structures and how they can legally be filled.

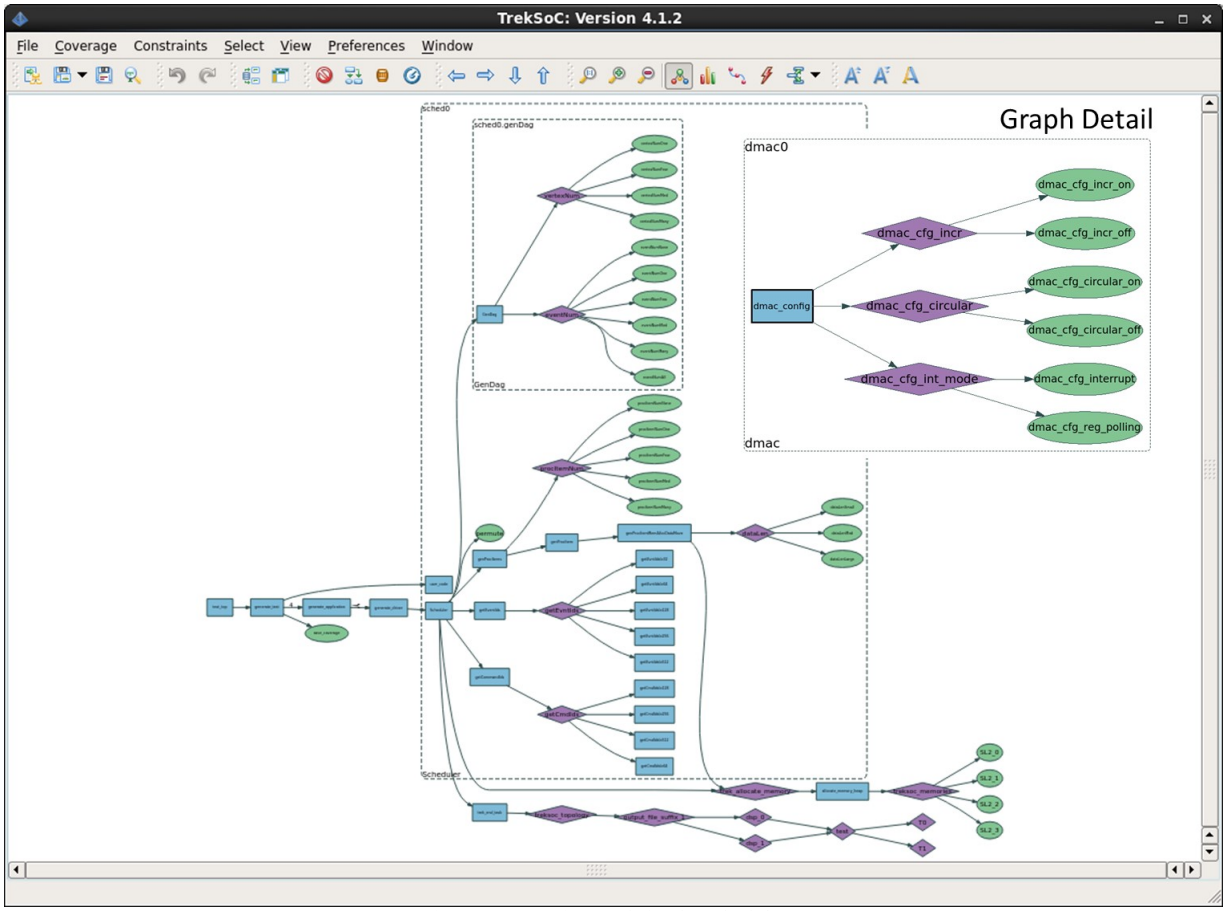

Figure 3: Scenario Graph for the scheduler

One such data structure is the one that describes the necessary DMA transfers in order to get the appropriate data into and out of the task solver. Generic code for a DMA block is shown as an inset to Fig. 3. Graphs are executed from left to right and from top down. So for the initialization of the DMA operation, (blue boxes define sequences) three things are defined as happening sequentially. Each of those involves a choice (purple trapezoids are decisions) of an action to take (Green ovals define leaf actions). Within the options, dma_config_inc defines the way in which the addresses are incremented, dma_config_circ defines if a circular buffer should be used and finally dma_config_intr defines if an interrupt is to be raised at the end of the operation. In each case there are two options possible, with the feature being turned on or off for the first two configuration parameters and the last one defining if interrupts or polling should be used. Each of these decisions is decided randomly unless other constraints have been placed on the graph. Sample code associated with building the descriptors for the DMA operations is shown in Fig. 4.

```
typedef struct {
  unsigned int LINK_en : 1;  // DMA config bits
  unsigned int CONFIG_INCR : 1;
  unsigned int CONFIG_CIRC : 2;
  unsigned int CONFIG_INTR : 1;
  unsigned int pad :11;

  U16 LENGTH;                   // length of transfer
  U32 SRC_ADDR;                 // source address
  U32 DST_ADDR;                 // destination address

  U32 LINK_ADDR;                // address of next descriptor
( if LINK_en )
} Descriptor;

trek_memory_address_t build_chain ( trek_memory_address_t
link_addr ) {
  // allocate memory to construct the descriptor
  trek_block_t blk = var::trek_make_block ( sizeof
( Descriptor ) );
  Descriptor *p = (Descriptor *) blk.Ptr();

  // handl the link field
  if ( link_addr.valid() ){
    p->LINK_en = true;
    p->LINK_ADDR = link_addr;
  } else {
    p->LINE_en = false;
  }

  // pick source and destination locations
  int len = trek_random(1, 100);
  p->LENGTH = len;
  p->SRC_ADDR = trek_allocate_memory (len);
  p->SDT_ADDR = trek_allocate_memory (len);

  // configure the transfer
  p->CONFIG_INCR = 0;
  p->CONFIG_CIRC = 0;
  p->CONFIG_INTR = 1;

  // allocate descriptor as constant memory
  // so we don't have to built it at run time
  trek_memory_address_t addr = trek_allocate_constant (
          sizof(Descriptor),blk,TREK_ALIGNMENT_32BIT );

}
```

Figure 4: Code defining the legal ways to specify the DMA transfers

By solving this complete graph within TrekSoC, a valid command can be created, as shown in Fig. 5. Here we see a number of tasks and the dependences that exist between them. This would be output by the tool as a block of code that would be executed on one of the main CPUs. When executed on the hardware the collection of tasks and dependencies would be sent to the scheduler and each of the nodes would execute on a tasker.

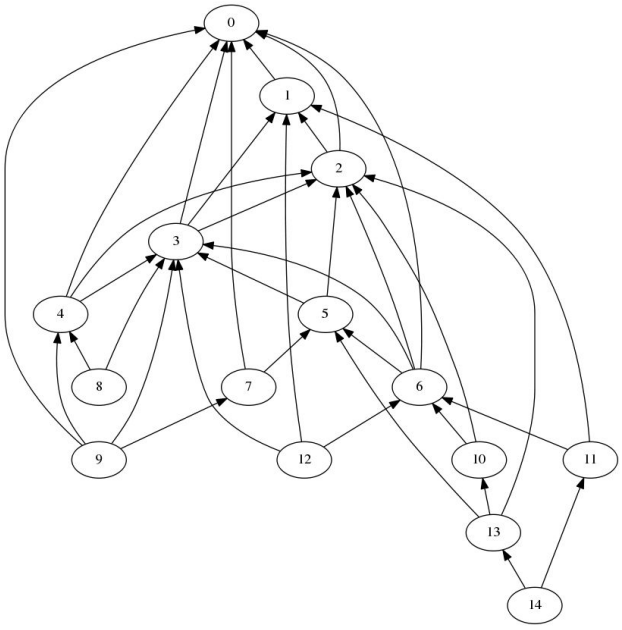

Figure 5: Command Graph generated from TrekSoC

This one command graph being dispatched to the task schedulers would hardly constitute a realistic load. In reality many of these would be initiated based on information being streamed into the chip. In order to add the next layer of complexity, it is necessary to schedule multiple commands as typified by Fig 5. This requires making sure that each command has non-overlapping memory regions, that the timing of the system is realistic in terms of the actually amounts of processing each of the operations would require and that the results of each task can be fully verified after the operation. TrekSoC does this by performing system-level scheduling as shown in Fig. 6.

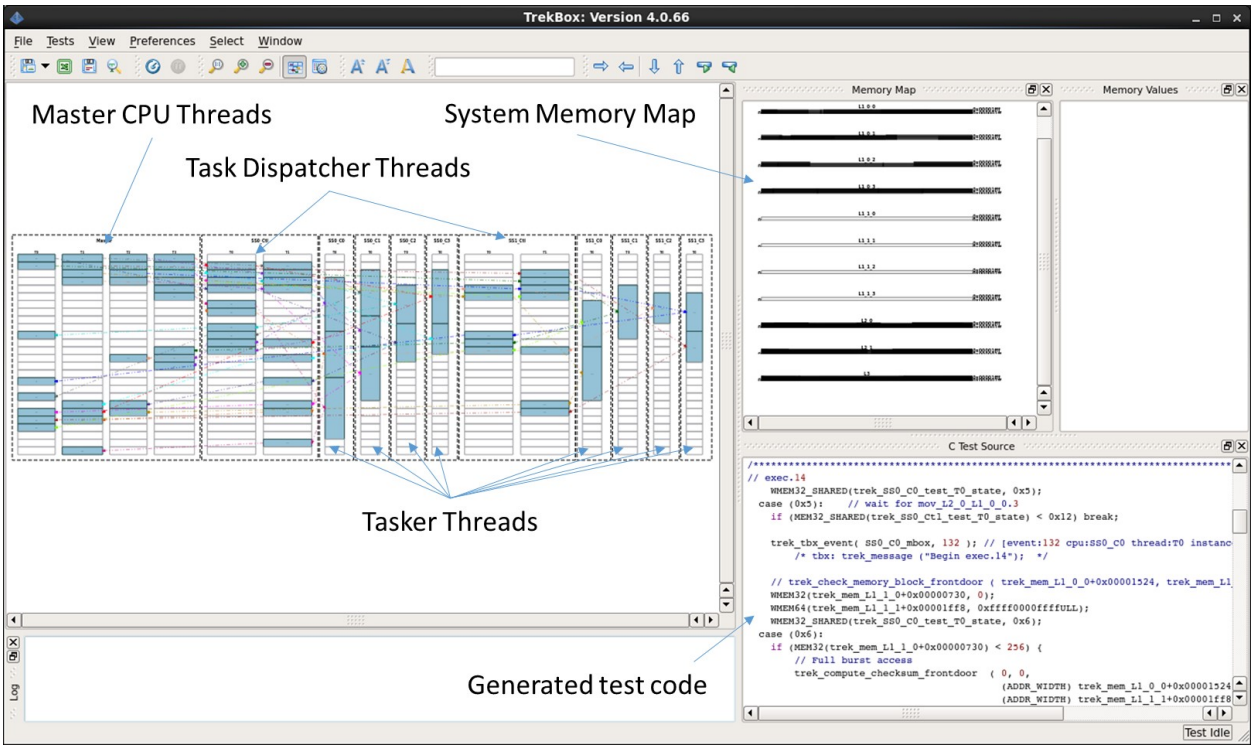

Figure 6: Schedule operations at the system level

The four columns on the left correspond to each thread running on the general purpose CPUs. Each thread takes some of the pre-built commands and sends them off to the task dispatchers, which start attempting to execute each of the commands, each shown as a tasker thread. Some of those tasks are blocked by shared resources, such as the DMA operations. As commands are completed and the task schedulers are ready to accept new commands, they are dispatched by the general purpose CPUs. This schedule it built before the actual timing of the system is known and is purely based on resource availability. The various views of the system are interlinked such the actual code associated with a task is shown, lower right, whenever a task is selected along with the memory regions that are currently in use.

Once such a scenario has been executed on the simulator or emulator, the timing becomes known and can be back annotated onto the schedule at which point the user will be able to assess the efficiency of the system and identify areas in which the architecture may be stifling performance, or where certain sequences of tasks may be causing problems for the scheduling algorithms. This type of chart is also useful for being able to determine the sequence of hardware events, such as when power domains have been shut down and brought back up again. Being able to ask queries of the tool, such as have two power domains been brought up with a certain timeframe could identify potential brown-out situations within the chip.

IV. Results

At this point in time the project is not complete. The first step for the verification team was to model the command sequence dependencies as shown in Fig. 3. As soon as this was done, they were able to execute the graph and have synthetic commands being output by the tool, an example of which was shown in Fig. 5. All of this was

completed before any software had been written and before the design had been completed. The verification team manually checked several examples to gain confidence in the generator. This was the first big advantage that the team found. They were able to develop and debug this important aspect of the testbench independently and when hardware verification was started, few additional bugs were found in the generator.

The first stage of hardware verification was with a single CPU and single task cluster using simulation. Single commands were sent from the CPU to the cluster and basic functionality was verified. Then without any modification in the model, they were able to add additional CPUs, additional task clusters and start some more complex tests, still using single commands. This was all accomplished by changing configuration settings within the tool. Then they switched to a simpler configuration of CPUs and task clusters and started to send multiple commands to the scheduler.

When they started to run the first examples of multiple simultaneous commands using a configuration more closely resembling the actual silicon, simulation became too slow and they migrated to emulation to continue executing additional complex test cases. This phase continues at the time of submission of this paper. Errors have been found in the hardware that involve very deep state space execution, and the cases have been isolated enough that it is unlikely that they would have been found if directed testing had been used.

The same strategy is expected to be used when the actual silicon becomes available and the exact timing of the chip could be compared against simulated results. It is not known at this point how extensive their utilization of this capability will be, or if they will use it to identify areas for additional verification, such as power verification or architectural performance evaluation. It is also not clear if the will have any time in their schedule to act on any performance limitations that are found in the architecture or if this information will be fed into the next generation of the chip.

The user has been very happy with the abilities that this verification strategy has provided. Portable Stimulus promises both horizontal and vertical reuse and that capability has been demonstrated. The initial graph for the command structure was created in just a couple of days, demonstrating the ability for users to grasp the general concepts of graph-based verification very quickly. The customer was also grateful for the ability to develop and debug significant aspects of the testbench independently of both the hardware and software.

REFERENCES

[1] Brian Bailey. Electronics Butterfly Effect, http://semiengineering.com/electronics-butterfly-effect/ (2015)

[2] How SDN LTE could simplify network management. http://searchsdn.techtarget.com/tip/How-SDN-LTE-could-simplify-network-management (2016)

[3] 3rd Generation Partnership Project, https://sites.google.com/site/lteencyclopedia (2016)

[4] Accellera Systems Initiative, http://www.accellera.org/activities/working-groups/portable-stimulus (2015)

[5] Breker Verification Systems, http://www.brekersystems.com/products/product-overview (2015)