Debugging Communicating Systems: The Blame Game – Blurring the Line Between Performance Analysis and Debug

Rich Edelman
Mentor Graphics
Fremont, CA

Raghu Ardeishar
Mentor Graphics
McLean, VA

***Abstract*—The traditional art of debug is a common occurrence in design and verification. As larger systems are built, a functionally correct system may still need to be debugged by understanding its performance characteristics. Performance analysis can be used effectively for debug, but it can be hard to collect and organize the data required. This paper will discuss a variety of solutions to collect and organize performance data, and discuss some useful metrics and recent situations where they have been useful.**

Keywords—Communicating systems, performance analysis, system-on-a-chip debugging

I. Introduction

As verification problems grow, debugging both the testbench and the device under test become harder. Verification can encompass a wide variety of metrics and criteria, including functional correctness, functional coverage, statement coverage, speed tests and load checks. Debugging a relatively simple block can be time consuming. Debugging a collection of blocks more so and debugging a system can be unmanageable without additional tools and techniques.

For large networks of communicating systems, simple debug has now become complex; encompassing traditional debug alongside performance analysis.

When a system-under-test is built, the individual blocks and sub-systems have already been verified, and are working. When these blocks are put together in a system, new bugs can crop up. Even a system with only two masters can have complex interactions. Larger systems with hundreds of nodes are quite difficult to debug with traditional techniques.

For example, when two masters in an AMBA ACE [1][2] system are put together, their SNOOP behavior may be incorrect. This "failed" behavior is performance that doesn't meet the specification. For example, in this case, master 2 may be participating in the SNOOP protocol, but when a SNOOP result is returned, it ignores the result, and issues an access to memory, despite the fact that master 1 returned a valid SNOOP result.

The problem in this case is quite hard to identify, since master 2 only seems "slower than expected" when it is accessing memory locations that master 1 has already cached. Debugging this problem is very context sensitive.

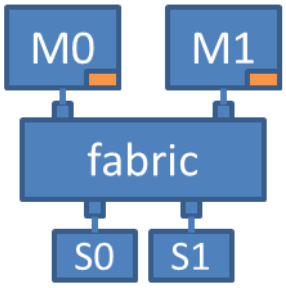

Figure 1 - Two Masters and Two Slaves

These kind of hard-to-debug problems really require collecting the right data and then organizing the data for analysis. Collecting the right data is really just improving observability or visibility. Analysis further processes this observed data, making it possible to see trends or understand complex relationships.

We'll discuss instrumentation techniques that improve visibility, and techniques to capture this instrumentation into databases for further analysis. They will include metrics and methods for understanding communication bottlenecks, and sub-optimal performance due to system level interactions.

Finally, we'll describe use-cases where having performance analysis alongside made debug possible.

II. Background

The key to performance analysis is collecting data to be used, and processing it sufficiently so that it is not overwhelming. For example, on a bus; collecting the number of bytes transferred per clock cycle will yield the bandwidth for each clock cycle. The bandwidth can be averaged or summarized further.

A. Hardware strategies

In hardware terms, solutions to improve visibility include things like embedded probes [3], signature identification, scan, ring buffers, etc. As the hardware executes, node values

are saved on-chip to be retrieved later for error checking or performance analysis..

B. *Simulation strategies*

In software terms, the same hardware solutions are available – mimicking the hardware using software. Additionally, simulation offers potential full visibility to every node in the simulation. This allows for much greater analysis, potentially at the expense of memory and simulation speed.

C. *System Level strategies*

Current debug strategies for System Level Debug consist of waveform and transaction level analysis during simulation or emulation or FPGA prototyping.

Debugging a system-under-test is the same problem as debugging a block; does it work? But the difference between system-under-test and block-under-test is the amount of data and the complexity of the interactions of the item under test [4].

A block might be doing reads and writes; maybe even packet reads and writes, but generally simpler kinds of communication.

A system might have many blocks communicating with each other. The amount of data is much larger and the complexity of interactions is much greater.

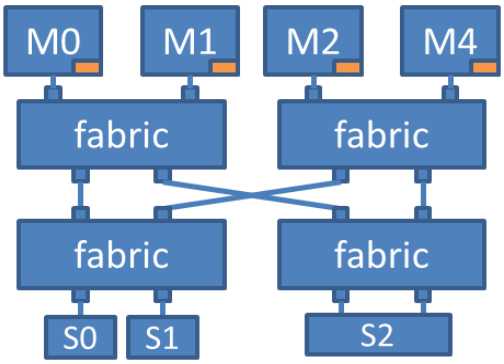

Figure 2- System Level Block Diagram

An example system level model might be verifying streaming video between block A and B, while the CPU runs in the background, sending out email and downloading 10 new large updates for the operating system. This system level model is a consumer device which incorporates a high power computing device in a handheld device.

Debugging the block level verification environment was relatively easy – each verification task was almost a "directed" request – read or write with expected values and expected results or consequences.

Debugging the system level verification environment is relatively difficult, since each verification task is "non-directed". Verification tasks are streams of possibly interacting data which can have unknown effects. Functionally correct blocks may interact with each other in bad ways. One kind of bad interaction is a system which is not operating at peak performance.

This kind of situation is very hard to debug, since there is no error that occurs, simply poorer performance than expected.

D. *Visibility and Controllability*

Visibility and controllability are the basic tenets of good debug, dating back to the beginning of hardware design. Attaching an oscilloscope lead to a board allowed instant visibility. Cutting a wire connection, and reconnecting it to another wire was a fast way to get controllability.

With IC design, this easy visibility and controllability moved from hardware solution to software solution. Simulation allows for modeling hardware, and providing infinite visibility and controllability. Any node can be monitored and any node can be driven.

Waveform debug has relied on monitoring all the "nets" or nodes in a design, and then displaying them on demand later during a debug session. Designs can be simulated "live", allowing selected nodes to be sampled. In both cases low level nodes are examined for correctness, perhaps being correlated with other data.

Transaction level debug has added a layer of abstraction away from waveform debug, allowing collections of multiple nodes to be represented as a single transaction. In AMBA ACE, there are 8 channels, for a total of more than 60 signals. With transaction modeling those 60 signals can be abstracted into basic transactions like READ and WRITE. Using transaction level abstractions has helped with complexity management greatly, but for system level debug we need system level transactions – really system level performance metrics. These metrics are the "visibility" side of the system – what the system is doing; i.e. bytes per second.

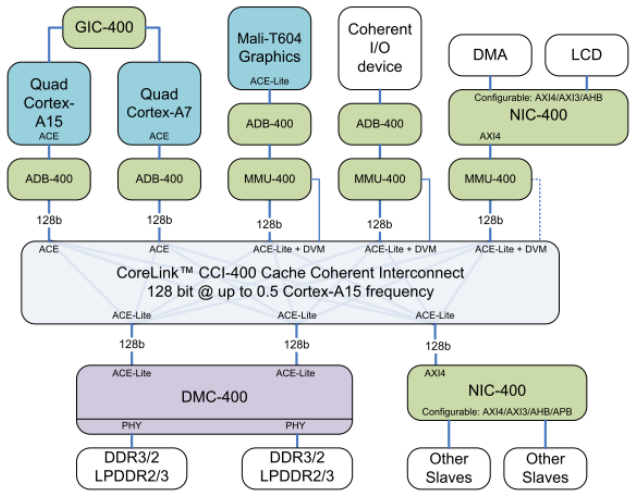

Figure 3 - AMBA 4 400-Series IP

III. Gaining Visibility

Visibility means seeing that is going on. With a system level simulation, there is a lot going on. All nodes cannot be monitored – there will be too much data. A good place to start is to monitor "interesting" busses or communication signals.

Furthermore, these monitored signals should be monitored at the transaction level. Communication between blocks can then be made visible. Once all the communication between blocks is visible, then rules about proper behavior can be extracted.

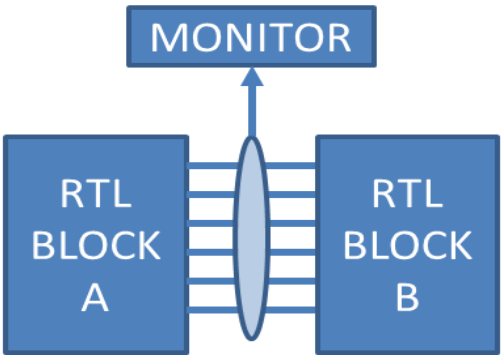

Figure 4 - Bus Monitor

IV. Gaining Controllability

Controllability means controlling what is going on. With a system level simulation nodes deep in the hierarchical block diagram may need to be controlled. Those nodes are already connected to drivers, which must be overridden in order to take over control.

V. Database – time correlated data

With controllability and observability, any interesting node can be watched or set to specific values. Monitored values from two different monitors – two different busses can be placed next to each other to observe their time correlated behavior.

Furthermore, bus monitor data can be placed next to CPU interrupt register value changes, or cache state values. Juxtaposing these different kinds of data – all correlated by time – allows visibility into related behavior.

Traditional debug has been doing this kind of time correlation with waveforms for years. Our monitor techniques and analysis extend this functionality allowing the bandwidth on a node to be displayed alongside the cache state value or the CPU interrupt register value changes.

The amount of data that may be generated from the monitors can be quite large. It may need to be filtered by time or by hierarchical name. Even if this data is filtered there will be large amounts of data that must be stored efficiently. Any proper database can be used, as long as it has a reader and writer, for both storage and analysis. For example an SQL database could be used or other home-grown database.

An advantage of using a standard database is the standard reports that are available. An advantage of a home-grown database is the tuning that can be performed for specific analysis.

VI. Instant gratification –reports during debug

Most reporting and data handling will be provided by a system built for just such jobs – an optimized reporting and charting system for example. But sometimes during simulation certain reports or statistics can be generated live, and help guide debug.

During simulation, a full bandwidth report may be unavailable, but a simple summary of bandwidth on a node can be produced.

VII. Example: Embedded FIFO

A pre-verified FIFO is used deep within the system-under-test. System level debug is continuing – tracking down a slow performing interface. The interface was designed to have a high "Quality Of Service" (QOS), but it is performing very slowly. The system-under-test is functionally correct. All tests are passing, but the total throughput is not what was expected.

As the debugging proceeds, the verification engineer suspects that a FIFO problem exists, and full visibility of that FIFO is needed. A monitor is added to the FIFO – detailed information is collected about the FIFO operation.

The FIFO can be seen to be filling and emptying. It appears to be completely operational. Furthermore, the rate at which transactions are going into and out of the FIFO is optimal. The FIFO appears to be operating at full speed.

But when the FIFO filling and emptying is summarized and put into a performance report, alongside the FIFO element lifetimes, it is apparent that there are elements which go into the FIFO and have very long residency. This long residency is low-latency for that element type.

Those long resident FIFO elements are not getting moved out in a timely fashion, slowing the communication. Upon further debug it is determined that under certain circumstances, the QOS values in the delayed transactions are not being increased as they wait. The system was designed to check the residency of FIFO elements. As the elements age, their QOS value should have been increased. This would have moved them out of the FIFO faster.

The communication slowdown has been debugged as a problem with the QOS updating code. Finally, the bug is discovered: although functionally correct the QOS update value was incorrectly set in the system level test during block initialization. It was initialized to 1, and it should have been initialized to 10.

VIII. Example: Cache Debugging

In an AMBA ACE system (Figure 2), during simulation the connection between master A and the fabric is quite heavily utilized – but only after block A has been put to sleep and then repowered. It appears that after waking up, block A is misbehaving in some way.

The throughput on the link is too low, and does not meet system specifications. It is operating according to the functional specification, but is not operating according to the performance specification. Additionally, the misbehavior only occurs are sleep mode is exited.

The debug process starts with understanding all the traffic on that slow link. The traffic looks fine – reads and writes; nothing out of the ordinary. After the block is put to sleep and repowered, the traffic again looks fine, but it appears that certain lines in the address space are not being cached.

Debug continues on the cache after power-up, and it is determined that the address maps in the cache are not being properly saved during sleep mode. Not all address map ranges have this problem, only ranges beginning with 0xffff0000.

IX. EXAMPLE: MORE CACHE DEBUG

During system level debug, the cache on master M0 is misbehaving, but it is unclear what is going on by just examining the cache behavior. The cache state is incorrect, but the incorrect behavior appears to be caused by a SNOOP initiated by master M1 to specific ranges.

The cache state value changes are instrumented as transactions, and the bus transactions on each master are instrumented as transactions. These "streams" of transactions can then be time correlated as in Figure 5.

Once the cache state changes are put in context (time correlated) with the transactions on the masters, it becomes clear that a specific master transaction ordering is causing the problem.

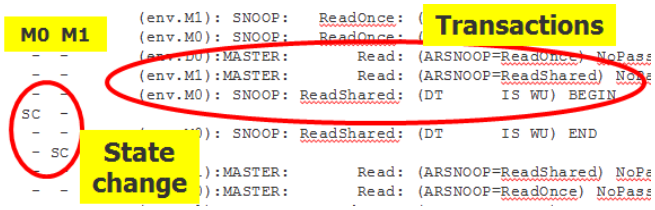

Figure 5 - Time Correlated Cache Behavior

For this address, the cache line state in cache M0 is moving from SC to undefined, and M1 is moving to SC. It appears that the second SNOOP caused this transaction. More debug will be necessary, but the time and preliminary cause has been identified. The system was functionally correct – there was no data corruption, but the cache state value change is causing a performance problem.

X. CAPTURING DATA

As described above, providing visibility and correlating the information can help debug performance problems. Capturing the data – providing the visibility can be done in many ways. Below three techniques are outlined; a simple pin monitor, a UVM transaction monitor and a VIP monitor.

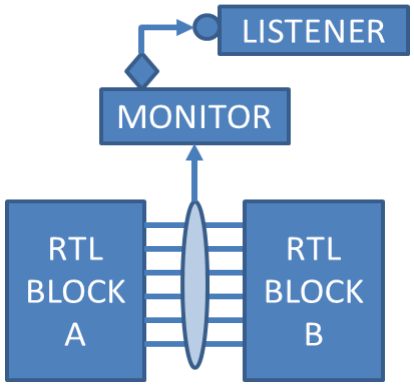

Figure 6 - Transaction Level Monitor and Listener

A. *Pin wiggle data capture*

A simple bus can be monitored, and at the appropriate trigger, the signals can be captured, then saved to a database, printed or returned as values to the caller. The monitor below is a simple monitor with three outputs. Each time it is called it returns when it sees a "transaction". The return values of 'rw', 'addr' and 'data' are the transaction values.

```
task monitor(output bit      rw,
             output bit[31:0]addr,
             output bit[31:0]data);

  @(posedge CLK);
  while ((READY!==1) || (VALID!==1))
    @(posedge CLK);

  rw = RW;
  addr = ADDR;
  if (rw == 1) begin // READ
    @(posedge CLK);
    while ((READY!==1) || (VALID!==1))
      @(posedge CLK);
    data = DATAO;
  end
  else          // WRITE
    data = DATAI;
endtask
```

This data capture is powerful and simple. It avoids UVM overhead; it avoids constructing a class object for each data sampled. It may suffer from a too low-level approach, essentially collecting interesting pin wiggles, where interesting is determined by ready/valid.

B. *UVM data capture*

A UVM monitor can be created which monitors a bus, then creates a transaction. The transaction can then be published on an analysis port. The analysis port is connected to "listeners" who are interested in the results of the monitor.

For example and simple way to monitor a transaction using a UVM based monitor is listed below.

```
class monitor extends uvm_component;
  `uvm_component_utils(monitor)

  uvm_analysis_port #(transaction) ap;
  virtual abc_if vif;
  ...
  function void build_phase(uvm_phase phase);
    ap = new("ap", this);
  endfunction

  task run();
    forever begin
```

```
        transaction t;
        t = new("t");
        vif.monitor(t.rw, t.addr, t.data);
        ap.write(t);
      end
    endtask endclass
```

This data capture is powerful, abstracting transactions according with the data provided by a pin wiggle monitor.

Once a listener is built and connected to the monitor using the analysis port, it can print or record the transaction to a database. It could also simply summarize the average bandwidth. It is an "analysis component" that can perform any kind of analysis desired.

C. *VIP data capture*

Using a Verification IP (VIP) to capture the monitored data allows for the captured transaction to be a very precise replica of the proper protocol – with no effort. The original VIP creator creates a monitor, and will supply a UVM based transaction on an analysis port.

For example, for AXI, an AXI VIP can be used to recognize either phase level transactions, complete transactions or both. Those transactions can be published on an analysis port for further processing, just like the UVM monitor (Figure 7).

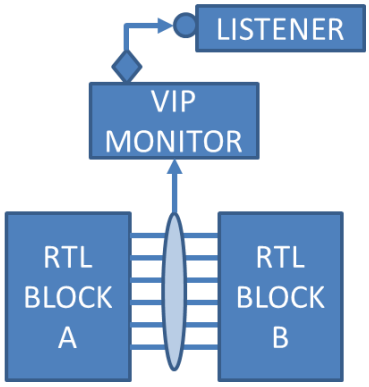

Figure 7 - VIP Monitor and Listener

XI. DATA SELECTIVITY

The monitor collects information. It is up to the verification engineer to monitor at the correct level of detail. For example, in the case of the AMBA ACE protocol, there are 8 channels (Figure 8). The channels are READ ADDRESS, READ DATA, WRITE ADDRESS, WRITE DATA, WRITE RESPONSE, SNOOP ADDRESS, SNOOP RESPONSE and SNOOP DATA. Taken together these 8 channels represent an ACE connection point.

In certain debug situations, only a single channel might be of interest, and in other debug situations two or more channels might be of interest.

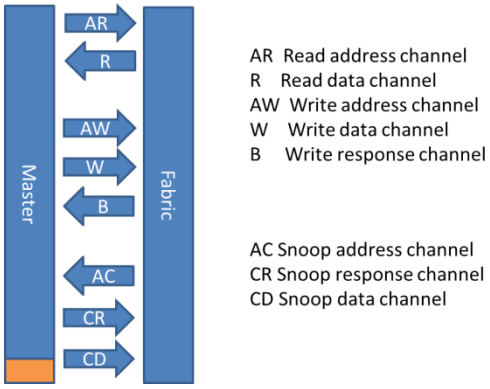

Figure 8 - ACE Master and Fabric

A performance chart may be generated of all transactions on an ACE bus; or just the complete READ transactions (READ ADDRESS + READ DATA); or just the READ ADDRESS transactions.

XII. METRICS AND METHODS

After data is collected metrics can be used to summarize and organize that data. Most metrics will be application specific.

Bandwidth may be measured as bytes transferred per second, or per clock, or per request. Latency will be measured as pure delay. The latency delay might be allocated to sub-segments of the transfer path. It might be lumped as a single number.

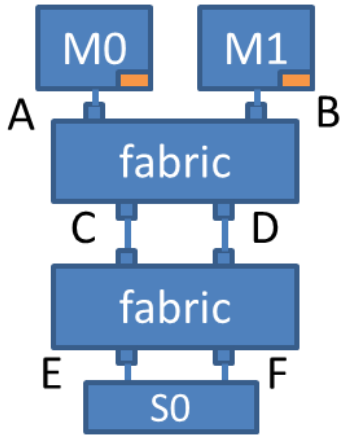

Figure 9 - Simple Path Example

Bandwidth can be calculated as a simple number – 12.5 MB/second at node E (Figure 9).

Bandwidth at node E can be calculated as a single number averaged over time, as in Figure 10.

Source Node	MB/s
A	10
B	20
C	25
D	35
E	45
Total	*135*

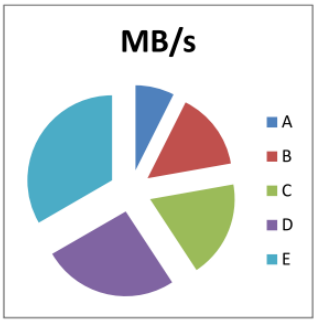

Figure 10 - Average Bandwidth

Additionally, bandwidth can be calculated for many time ranges, as in Figure 11

Source Node	MB/s	MB/s	MB/s	MB/s	MB/s	MB/s	MB/s	MB/s
A	10.0	9.0	8.1	7.3	8.7	7.9	11.8	10.6
B	20.0	18.0	16.2	14.6	17.5	15.7	23.6	21.3
C	25.0	22.5	20.3	18.2	21.9	19.7	29.5	26.6
D	35.0	31.5	28.4	25.5	30.6	27.6	41.3	37.2
E	45.0	40.5	36.5	32.8	39.4	35.4	53.1	47.8
Total	*135.0*	*121.5*	*109.4*	*98.4*	*118.1*	*106.3*	*159.4*	*143.5*

Figure 11 - Bandwidth sample in time ranges

Bandwidth at a node can also be shown divided or "attributed" – so that contributions from various sources can be seen.

Bandwidth contributions at node E may be more interesting – explaining that on node E, traffic is due to some originating node X. This graph can answer the question – "Which source is causing the most traffic at node E?" In Figure 12 such an attributed graph is shown.

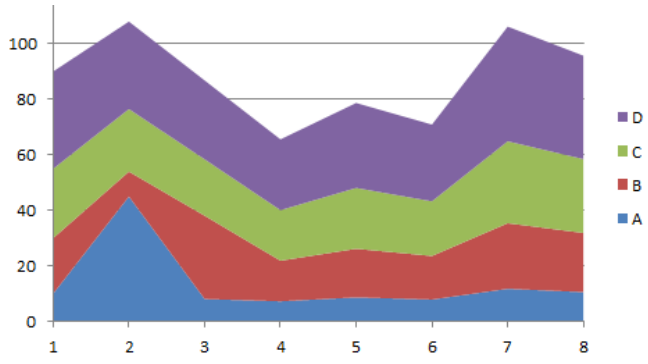

Figure 12 - Attributed bandwidth

Although only bandwidth was discussed, there are many other metrics which might be useful. The specific metric that is useful in a given case is left as an exercise for the reader.

XIII. Linking and Relating

The attributed data above is very hard to collect. For example, on an AMBA AXI fabric as a request transfers between master and slave, then another master and slave and so on, until it reaches its destination; the AXI "ID" field can be used to tag the request with an identifying number.

Normally as the request transits the fabric, the ID bit width is extended to allow for extra bits of information, so that the return responses can be routed back to the originator. Unfortunately, on any given fabric this "ID-extension" rule is not always followed, or is modified in each fabric component. The fabric and the components are self-consistent. They do the proper routing. What is not possible is to identify a request from the outside. A monitor of the system cannot relate the transactions to each other. For example, a request at node E might be identified as the request that originated from node A. Without a "label" or "tag", the READ request at node E does not have an "originator". The monitor cannot attribute the node E read to the origination point: node A.

The complexities of ID management and protocol specific routing are beyond the scope of this paper. But, for better debug and performance analysis, some scheme of consistent request and response ID management should be used.

Two implementations for managing requests and responses using IDs are below.

The first possible solution is the similar to a unique prefix of a routing system. As a request passes through a node, some node ID is prefixed to the existing ID. As the request transits the fabric, its ID becomes a prefix routing path. This solution is built into the hardware, and may be necessary to route responses. An AMBA fabric may this kind of "ID expansion" built in.

If it is built in, then it comes for free in terms of the debug and performance analysis system. A performance monitoring system simply needs to record the request and response, along with the ID. The prefix of the ID determines the node, and the path through other nodes.

Unfortunately, the prefix system is implemented on hardware, and may become unwieldy with a large, multi-level fabric, as each node will add bits to the ID. Some of the issues can be solved by simply treating each possible path through the system as an integer. As a request transits the fabric, the two things are recorded; two numbers: the ID and the integer number of this path. The next time the ID goes through a node, the ID and the particular path integer is recorded. Generally there will be fewer than 2^32 paths (or 2^64)

A second solution is an external table or stack. This stack is a record of each path or node that the request has passed through. Each time a request passes through a node the node ID is pushed onto the stack along with the node number. Each request builds a "path stack" managed in a central location. This solution may or may not be committed to hardware – thus may only be available during verification. It will consume memory – each outstanding transaction in the system will have a "stack allocated" to it. This solution will scale well with large numbers of transactions and a large fabric.

With either of the solutions above, a collection of "paths" can be built, and the history of where a request or response goes can be understood.

A simple, general purpose monitor can be constructed which understands this occupancy graph, and can provide "attribute data" for the performance reports. No monitor needs to be built with any knowledge of the fabric architecture or any specific path mappings.

Some may propose build a monitoring system with intelligence about fabric routing paths and protocol specific rules. These are "smart" monitors. These monitoring systems are fraught with peril, for their complexity is as large as the hardware being tested. They are hard to reuse, being specific for a fabric, and they are difficult to test – there are many corner cases.

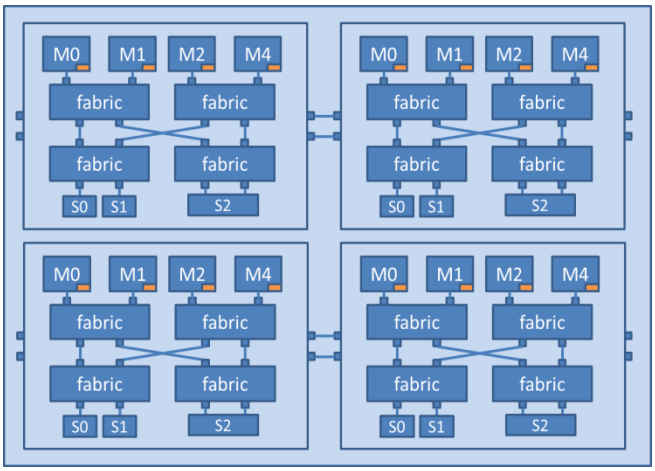

Figure 13 - Large Path Example

XIV. Conclusion

The authors thank Gordon Allan who creates so many good visualizations and always has an inspiring way to look at data and Avidan Efody who always has interesting real world debug problems.

The techniques described in this paper suggest monitoring "interesting" places in a system, saving the monitored data as transactions for further processing. The transactions are saved in an efficient database waiting further processing.

Some of the monitored interesting places are interconnect. These monitors watch the flow of requests and responses. Each protocol monitored may have different levels of detail which are interesting for the debug or performance analysis in question.

Some of the monitored interesting places are design specific; for example the state of the cache. These monitors record the behavior of some feature for later correlation with other monitored behavior.

These techniques are quite general, although much of the examples in this paper are AXI and ACE based, any system of communicating nodes can apply the ideas.

When the monitored data is saved into an appropriate database, many kinds of math can be performed on it to generate reports. These reports may be protocol specific or application specific. Each system will have its own special behaviors.

Data flow – collections of data passing through a node – has sub-flows that are interesting to understand. Without special attributes or other techniques the sub-flows cannot be identified. In order to provide maximum debug and performance analysis, each request and response must have a unique identifier, and an "occupancy path". Any fabric is a large, complex interconnect communication system. It must provide ways to allow good debug, including IDs for each (path, request/response).

Don't write "smart monitors". Collect the right data at the right time, and then put all the smart ideas into interpreting the data, and producing performance analysis reports to help debug.

Fabric SoCs are really large communicating systems (Figure 13). They have complex interactions. These kinds of systems must mix debug and performance analysis in order to understand

XV. References

[1] Ashley Stevens, *"Introduction to AMBA® 4 ACE™ and big.LITTLE™ Processing Technology"*, June 6, 2011 (updated July 29, 2013)

[2] ARM AMBA® AXI™ and ACE™ Protocol Specification, February 22, 2013, ARM IHI 0022E (ID022613)

[3] T. Gheewala, *"CrossCheck: A Cell Based VLSI Testability Solution"*, DAC 1989.

[4] R. Edelman, A. Gonier, *"Stripeviewer – Using Transactions to Effectively Debug Large SoC Designs"*, IPSOC 2013