Big Data in Verification: Making Your Engineers Smarter

David Lacey, Mike McGrath, Alan Pippin, Ed Powell, Ron Thurgood, and Alex Wilson
{david.lacey, mcgrath, alan.pippin, ed.powell, ron.thurgood, alex.wilson}@hpe.com
Hewlett Packard Enterprise, 3404 E Harmony Road, Fort Collins, CO 80528

***Abstract*-Big Data is getting a lot of press these days but does it really apply to ASIC verification efforts? HPE has found that it does apply and can be instrumental in helping make smarter decisions. HPE is taking small steps to capitalize on the concept of Big Data. In this paper, we will share our view of using Big Data concepts in the ASIC verification domain, highlighting several specific examples of how easy it is for the audience to get started with their own Big Data analysis. We will also share our ideas on where we see larger Big Data opportunities being able to help our verification efforts in the future.**

I. Introduction

Big Data is a commonly used term across the industry that is defined by the volume, velocity, and variety of the data collected [1] [2] [3]. However, does everyone really understand how to get value from it in Verification? In this paper, we apply Big Data techniques in Verification and show that it is not a scary topic but rather one that can be a significant help to the reader. We have come to realize that Big Data isn't just about its size, but about the ongoing impact it can have. We need to get our products to market sooner to meet customer needs faster than we are doing today. To accomplish this, we are using this data to drive more accurate forecasts for licenses, compute resources, engineering resources, schedules and equip our verification engineers with critical data to enable them to make smarter decisions for all current and future ASIC projects in our organization. We believe that identifying new data sources, collecting them continuously, and inventing ways to analyze them fits the definition of Big Data and has helped us reach those goals.

We show that gigabytes of data can be analyzed without complicated, time consuming, custom toolsets, or complex MapReduce operations [4] stored in NoSQL databases and does not require Artificial Intelligence to make conclusions about the data. This makes Big Data analysis accessible to any Design and Verification Team by using these techniques on data that can be easily collected and analyzed, no matter what its size.

Our team has been collecting and analyzing data from our ASIC development efforts for years. From this experience, we present practical and real examples of how we are utilizing Big Data in our verification methodologies to drive smarter decisions. Our role is to make sure that the people who need this data to make decisions have the data. To do this, we first focus on how to identify and collect the right data to answer the questions that need to be answered. This helps our team perform verification tasks smarter, not by changing how we write tests or testbenches or checkers, but by helping us increase simulation throughput and performance, monitor DUT performance (bandwidth and latency), and improve test effectiveness through more advanced coverage analysis, enabling us to find bugs faster and reduce our time to market.

Next, we show real examples of how we are presenting and analyzing the data that is collected. The data must be presented in effective ways that lead the user to the answers they need. Practical examples will show how we look for anomalies in the data, how we use data to identify ways to address challenges, and how we move from simply having data to transforming it into information to be acted on.

To illustrate our use of Big Data, we focus on the following areas: Compute Farm Metrics, Performance Data, Test Results Data, and Big Coverage Data. For each area, we show the specific types of data we collect, how we store it, and how we analyze it. The practical questions that we need answered with this data are shared to demonstrate the value that Big Data brings to our team.

II. Collecting The Right Data

Collecting the right data makes all the difference in the types of questions that can be asked and answered to enable verification engineers to be smarter and more efficient. The data in Table 1 has been invaluable in helping us ask and answer these questions that drive smarter decisions.

Table 1: Types of Data Collected

Compute Farm Metrics	Performance Data	Test Results Data	Big Coverage Data
compute farm job data	test name	test metadata, definition	<u>Per Coverage Event</u>
license usage data	traffic mixes	tree revision	name, test ID
disk usage	bandwidth, latency, utilization	pass/fail results & error sigs	value, simulation time
system load	swept variables & parameters	phase/sim/wall durations	<u>Per Test</u>
per site/business/user	per interface	memory requested/used	name, user, cmd line,
every 6 minutes	per topology	garbage collection statistics	date, generator
~400M rows/year ~40 GB/year for 10+ years	~20M rows/year ~20 GB/year for 10+ years	~70M rows/year ~30 GB/year for 10+ years	~100B rows/month ~1 TB/month

III. Compute Farm Data

The compute farm data we collect (outlined in Table 1) captures information every 6 minutes per user/site/business and gives us tremendous insight on the status of our compute farm jobs, detailed license usage data from our license servers, information about our fair share license request queueing from our job manager, compute host loads, and disk usage metrics. This data helps us answer various questions about our license forecasting, license utilization, and compute farm queueing issues, enabling us to make smarter decisions about how to purchase and manage these resources more effectively. The types of questions we have answered with this data and corresponding case studies follow.

A. Asking the right questions

Knowing what the actual vs expected demand of licenses lets us ask questions like: *Why do my job submission pending times seem unusually high today? What is causing this? Why are all my user jobs being starved for licenses when they are supposed to have the highest priority? Why did only half of my tests run over the weekend? Why did one team get more than their fair share of the license allocation in regressions last night? How much disk space and memory will we need in the future? Some user jobs are starving regressions -- exactly which users are they and what are they running?*

Answers to these questions help us understand actual demand versus expected demand. We can tell users things like *"you aren't running 24x7 with 20 licenses, you are running 24x7 with only 15 licenses"*. We can spread licenses out across our teams to get better utilization. When you run out of licenses, you typically spend more money to get additional licenses because you don't have the data to tell you what is really going on.

Knowing our license utilization lets us better allocate and use our licenses. We can see from Big Data which organizations and sites have been using the licenses. We can pinpoint where the problems are coming from and adjust their utilization appropriately.

B. Case Study: Why are my user jobs being starved for licenses?

We had a situation where tests were being starved for licenses. It started with a simple question: *Why was our demand for licenses so high causing user jobs to be starved for licenses during the day?* We pulled up the *Pending Jobs* chart from our web interface (see Figure 1). We saw the demand spiking, causing user jobs to be starved for licenses. Our compute farm metrics database showed a window of time where the problem occurred. This was a transient event. Users felt like it was taking longer for their jobs to run, but without the data, we wouldn't have known if it was a real issue or not. Even if we believed it was real, without the data, how do you track down a transient event like this? By the time you investigate, the issue has cleared up but your engineers continue to be impacted.

After analyzing our available license count against our pending user jobs, we discovered we were being starved (see Figure 2). Our regression jobs are required to obey a buffer limit which assures licenses are always available for users. They can't take licenses reserved for user jobs. Yet, our license availability was clearly falling below this buffer limit. So, who was taking the licenses? Available licenses fell below our buffer limit and license availability, causing response time for user jobs to increase.

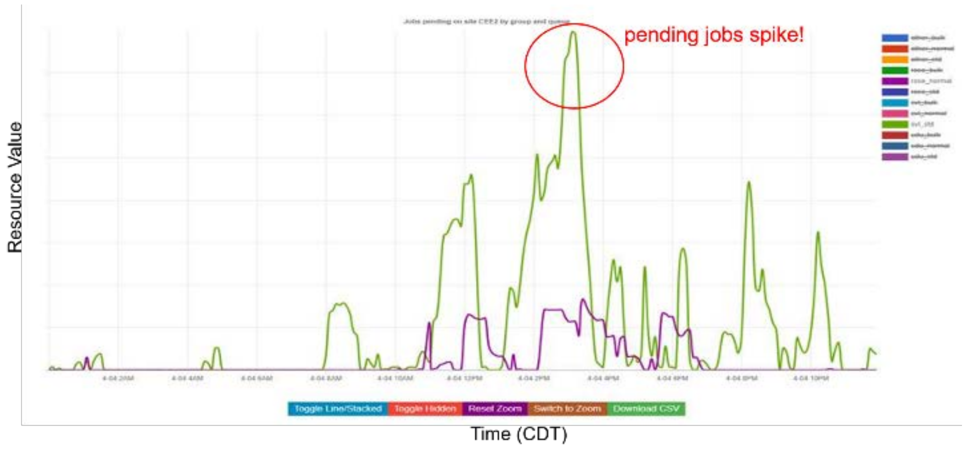

Figure 1. Compute Farm Data – Pending jobs spike

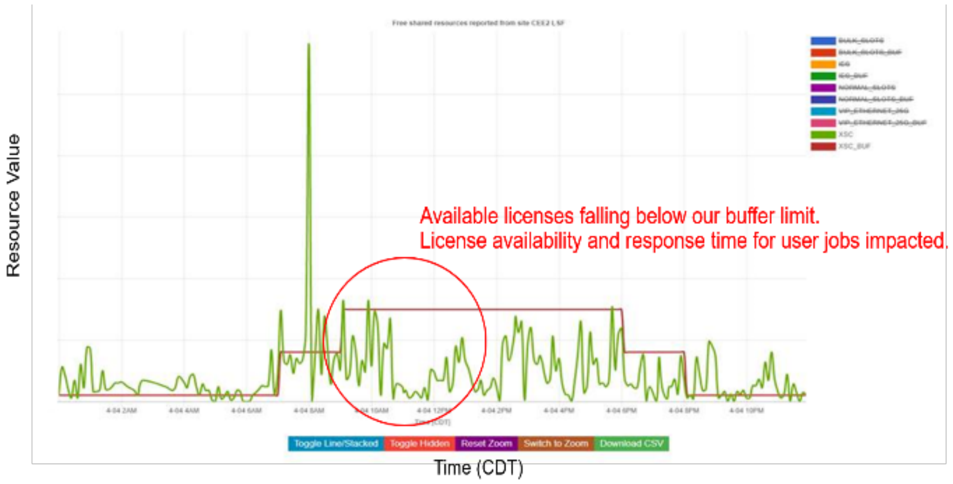

Figure 2. Compute Farm Data – Avail Licenses below buffer

We looked at our license server plots, organized by business, to find out where the licenses were going (see Figure 3). We found that another business was getting more licenses than we were. We also discovered the same business was also using more license resources than they were supposed to (see Figure 4). We now knew which business was taking the licenses and preventing our jobs from running, but we didn't know why. We asked some additional questions like: *Who made the license count jump? Who is running jobs in that business?* We started looking at their jobs for anomalies to point to what they were doing incorrectly. *Were they specifying the license limits properly? Were they asking for and reserving the license resource they were using?* We used live data to know which users were requesting resources, showed them the problem, and had them correct their resource requests, which ultimately solved the issue we were having.

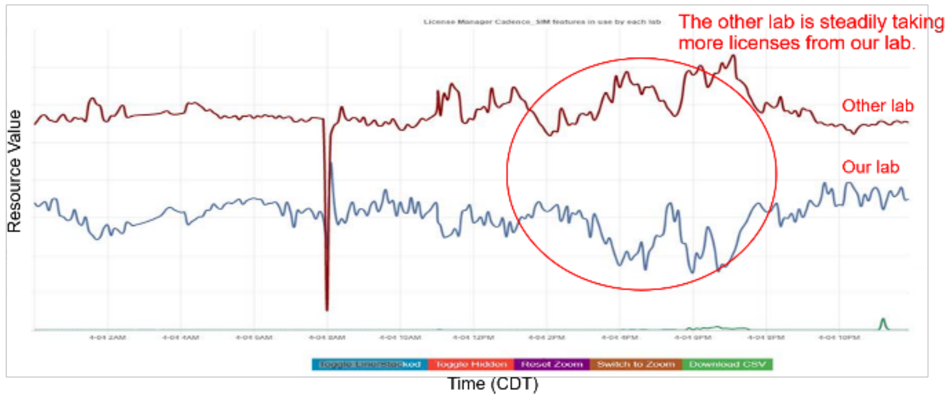

Figure 3. Compute Farm Data – Business License Usage

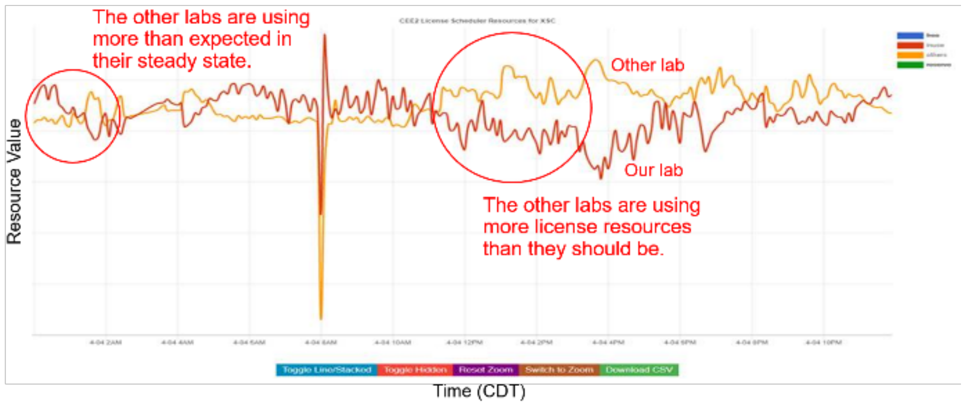

Figure 4. Compute Farm Data – Business Resource Usage

We also found a second repetitive issue you can see in Figure 1. Each day at certain times, a spike similar to the original one was found. One problem (being starved) led us to solve this secondary problem too. *What are these license demands? Where were they coming from?* We found that all our continuous integration jobs were set to start at the same times during the day across all our projects, causing spikes in license demand. We were able to flatten that demand by spreading these jobs throughout the day.

Big Data helped us ask these questions, and the plots helped us answer them. This enabled our engineers to get back to work quickly, increased our job throughput, decreased user job pending times, and raised productivity across the business.

C. Case Study: Why aren't we using all of our licenses?

Our regressions weren't running enough tests, so our first thought was that we needed more licenses. After reviewing our license utilization over the past several months, we found that we were never using 100% of our licenses while the job manager showed that we regularly use all but a few of our licenses. We then looked at the various license states reported by the job scheduler and found that a good chunk of our jobs were in a reserved license state, meaning the test had requested a license but hadn't actually used it yet. The two graphs in Figure 5 and Figure 6 demonstrate this disparity. Figure 5 shows the state of all licenses seen by the scheduler. About a tenth of our licenses at this time were tied up in reserved (green) and very few free licenses (blue) at many points. Figure 6 shows that the license server reported we only reached about 90% license utilization. We decided to investigate these jobs in the reserved state and found multiple issues that resulted in better license usage.

In the first issue, our simulation job flow is split into multiple parts: compilation, elaboration, simulation, and post simulation scripts. Only some of these steps require a license and in this case we realized that our post simulation scripts, which spent most of their time compressing and copying files, were causing the scheduler to reserve a license when none was needed. We revised the tools managing our jobs to return licenses after the simulation portion had completed to open that license back up, leaving our post simulation tools to complete without holding up other jobs.

The second issue was whether the compilation step could be improved. The vendor tools required a license for this step, but it was a lengthy process that only needed a license for the initial setup. We worked with the vendor and used our Big Data analytics to demonstrate the license bottleneck this imposed on us. The vendor made the appropriate changes so that compilation didn't require a license the entire time which freed up more of our jobs to acquire available licenses.

The overall benefit here was better cycle throughput per simulation license without purchasing additional licenses.

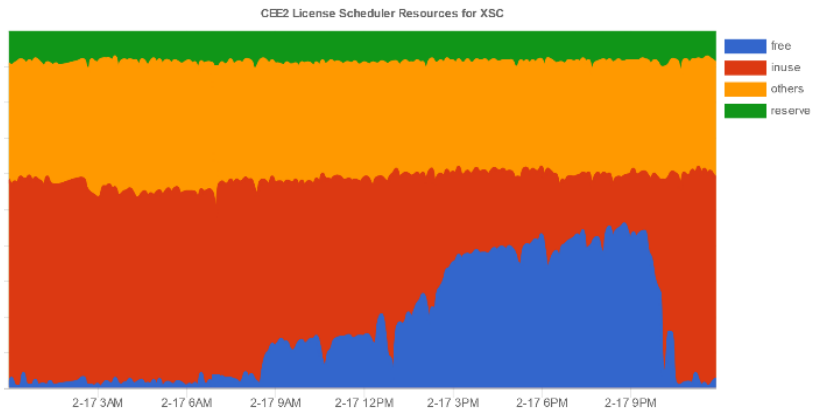

Figure 5. Compute Farm Data – Job Scheduler License Usage

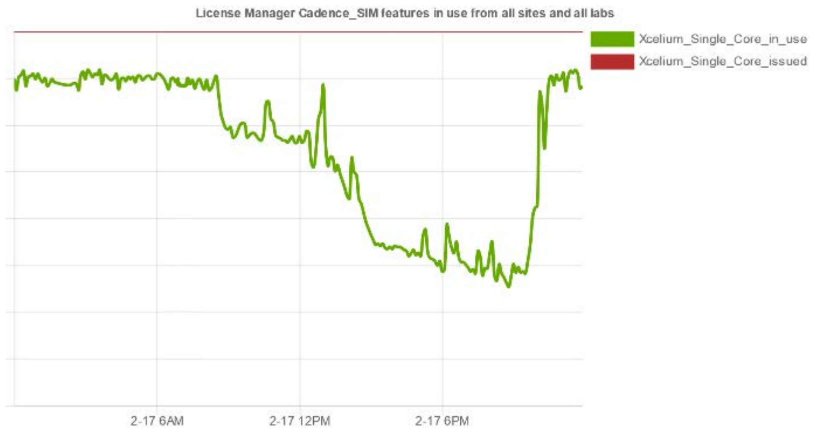

Figure 6. Compute Farm Data – Vendor License Usage

IV. Performance Data

The performance data we collect (outlined in Table 1) enables analysis of different aspects of our design's performance while also providing insights to aid debugging performance issues. We utilize an internally developed tool called Autoperf [5] to obtain and analyze this data. The approach in Autoperf is to instrument the design, uniquely identify the available measurements, and present those measurements in tabular and graphical formats. Autoperf generates and records all measurements it can from the available data and lets users review the measurements they desire.

Autoperf uses unstructured data [6] and aggregates corresponding measurements from multiple tests from the same testbench. The use of unstructured data lets users create new analyses as needed instead of requiring significant effort in advance to plan and execute a new performance item and its data model. The software architecture for Autoperf consists of verification libraries, performance analysis tools, a Vertica database [6] of test performance results, and a web application [5]. This data helps us answer various questions about the performance of our DUT over time, enabling us to catch performance problems as they happen. The types of questions we have answered with this data and a corresponding case study follows.

A. Asking the right questions

Measuring the performance and latency on each interface across each performance workload, configuration, and testbench lets us ask questions like: *Am I hitting the performance metrics we need to hit? When did we stop hitting the performance metrics we needed to hit? What set of swept parameters across multiple tests gave me the best performance? Have we met our performance requirements for each specific traffic class?* Answers to these questions help ensure that every testbench meets its expected performance targets and helps us isolate when design changes cause performance to drop. Our performance data helps us spot anomalies in our design in real time as they happen.

B. Case Study: Why did our data mover bandwidth take a sudden drop between this week and last week?

Figure 7 shows the Input versus Output Bandwidth (BW) of a typical data mover operation, with each data point coming from specific performance tests run over a period of time. The ratio of input to output BW is related to the size of the transfer request. We would have expected to see a tight clustering of points for each transfer size around the expected BW numbers. Instead, this plot showed that something changed that caused our BW to fall off dramatically. We run, record, analyze, and plot data from our performance test runs in our nightly regressions. If the numbers fall out of expected ranges (using various outlier detection methods), a failure is reported for the verification engineers to look into. We leverage the power of Big Data to automatically tell us when something goes wrong. We don't have to manually analyze our performance test results each day. We have scripts and processes running on the data that will tell us when something goes wrong or has anomalies in the data, like in this case.

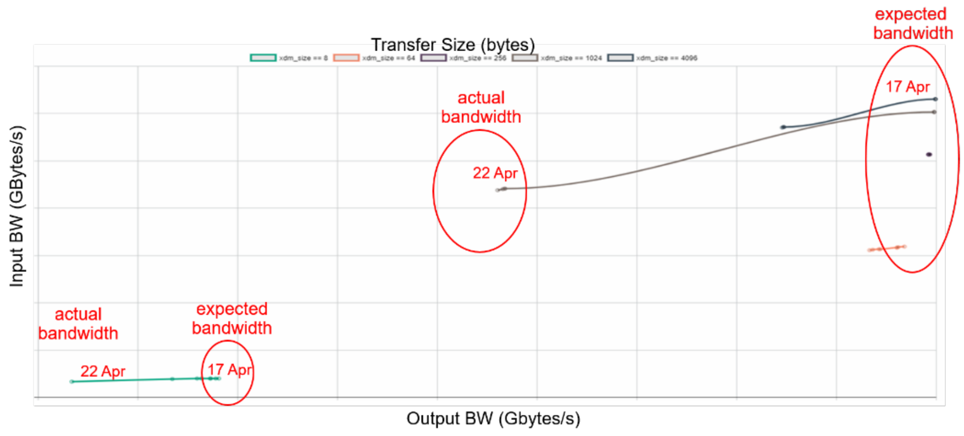

Figure 7. Performance Data Case Study – Expected versus Actual performance drop

We analyzed the data from April 17th by looking back at the auto-generated plots from that day, and were surprised to see that our data transfer commands were all being read at the beginning of the simulation as shown in Figure 8. In a real system, these would be spread out over the course of the transfer. This was a stimulus bug that we didn't catch the first time we setup these simulations, leading to an unrealistic data transfer scenario and unrealistic performance numbers.

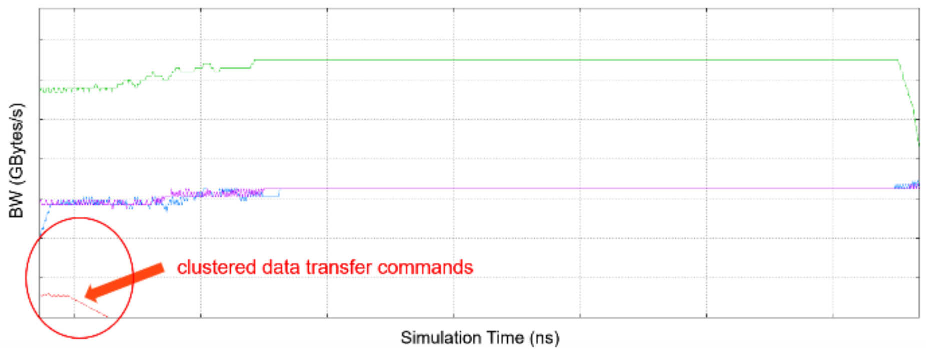

Figure 8. Performance Data Case Study – Issue Identified

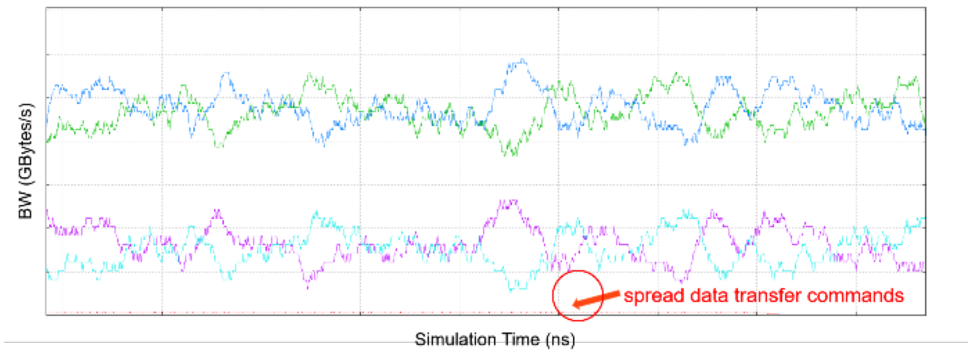

Figure 9. Performance Data Case Study – Issue Fixed

We analyzed the data from April 22nd by looking back at the auto-generated plots from that day and saw that our data transfer commands were now being read throughout the entire simulation as expected as shown in Figure 9. The performance drop flagged by our analysis scripts and automated plots helped us recognize we had a problem and showed the effect of fixing it. There were really two bugs here: First, we didn't realize our performance was higher than expected initially. Second, we weren't spreading out our data transfer commands throughout the entire simulation which is the proper modeling of real software behavior. Without this data and our automated performance analysis tools operating on that data, we would not have caught this or been able to see and debug it as quickly as we did.

V. Test Results Data

Our test results data (outlined in Table 1) captures information from every simulation and verification job that is executed. This data helps us answer various questions about our design and verification environment, enabling us to make smarter decisions related to it. The types of questions we have answered and a corresponding case study follows.

A. Asking the right questions

Knowing how many simulation cycles we are running on each testbench lets us ask questions like: *Are we properly balanced between each testbench? Are any testbenches being starved for cycles? Are any using more than their fair share of cycles?* Answers to these questions help us ensure every testbench is being run with the expected priority and volume of testing set by our project priorities.

Knowing how effective our regression testing is lets us ask questions like: *What are our pass and fail rates? How many tests are we running per night per testbench? What is our time to first failure? What tests are finding the most bugs? What types of bugs are we finding (functional: assertions, checkers; verification: testbench issues, test errors; etc)? What are our bug open and close rates?* We utilize pass/fail rate and bug open/close rate curves help us know how mature the DUT is and how stable it is as we approach the end of our milestones.

Knowing how the bug rate compares with verification cycles and cycles over time lets us ask questions like: *Are we running the right tests in regression that can find the bugs? How do bug find rates vary with verification cycles and cycles over time? Are we done finding the easy bugs (requiring fewer cycles to hit)? Are we spending our time hunting for corner case bugs now (requiring many cycles to hit)? How mature and stable is the DUT?* Answers to these questions help us predict our schedule with greater accuracy, enabling us to allocate resources more effectively.

Knowing our test distributions lets us ask questions like: *Am I getting the right distribution of tests? Are any being starved or not running as often as they should be? Is each team receiving the necessary amount of simulation cycles? For these cycles, is each team getting the right distribution of tests to bring the greatest value from those runs?* With this Big Data analysis, we learned that we needed to start our longer running tests and tests that failed in the last regression first. This gives our engineers feedback from those tests sooner, reducing the overall testing cycle time.

Knowing if our test simulation performance is optimal or not enables us to ask questions like: *What factors are causing performance degradation in our simulations?* Teams want to understand if they are getting the optimal performance from their simulation runs. We found instances where our simulation performance was degrading over time. Besides engineers, licenses are the most expensive resource we consume, so improving simulation performance lets us avoid over purchasing licenses by using the ones we have more efficiently.

Knowing our CPU and memory usage lets us ask questions like: *What CPU and memory configurations do we need to upgrade my farm? What CPU configurations would be optimal for our current workloads? Has the CPU utilization of our verification workloads remained high or have there been deviations over time? What memory configurations would be optimal for our current workloads?* We've found that by analyzing the maximum, minimum, and average memory utilization, we can make these determinations and purchase exactly what we need.

Knowing how many license hours were used over the previous year lets us ask questions like: *What should my license forecast be for the next year? How many license hours were used on a given project? How many license hours were used over all projects and teams?* Based on this data, we increased the accuracy of our forecasts versus actual license needs.

B. Case Study: Why are my tests running slow and taking longer than expected?

We analyzed our test results data and found our simulations were running 6x longer than normal (see Figure 10).

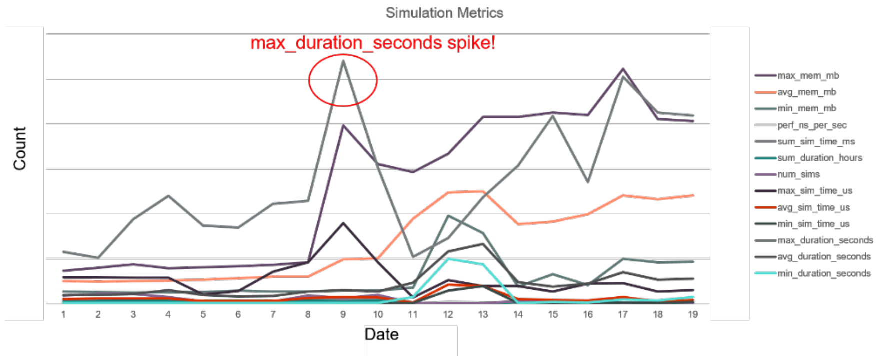

Figure 10. Test Results Data Case Study – Simulation Metrics – max_duration_seconds spike

We then analyzed all of our simulation metric plots for that same day and time to see if anything jumped out at us. We found that there was a direct correlation between the increased run time and the number of garbage collections done during the impacted simulations. Our queries revealed the exact revision of code that introduced the dramatic increase in the number of garbage collections. We found a change in our testbench that led to a dramatic increase in the amount of memory required to run the simulation. Since we hadn't increased the amount of memory requested by the test, the simulator was garbage collecting to reclaim enough memory to meet the required memory demand. This thrashing of memory led to a 6x slowdown in all of our simulations for this block. We optimized the testbench and increased the memory being requested by the test to reduce the number of garbage collections required, which fixed the problem as shown in Figure 11.

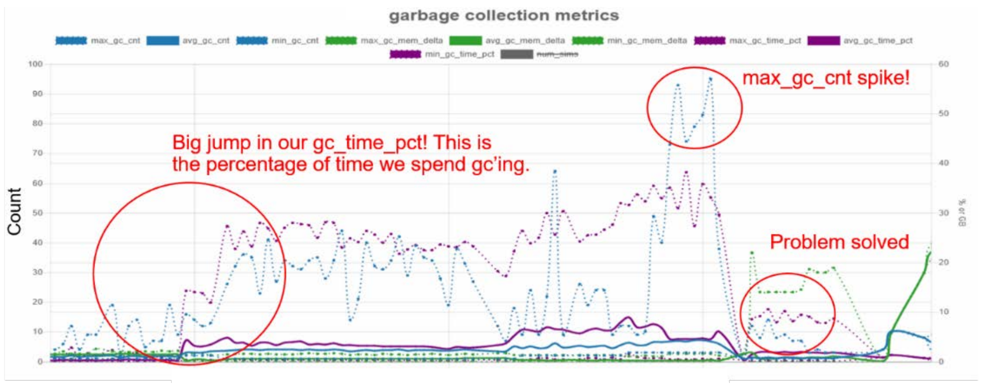

Figure 11. Test Results Data Case Study – Garbage Collection Metrics – Problem and Resolution

VI. Big Coverage Data

Applying Big Data methods to functional coverage offers several advantages over the SystemVerilog covergroups that engineers frequently rely on. Our work in this area is incomplete but ongoing and has potential to make coverage data more useful. Typical covergroups just record whether a specific condition within a block has occurred, but with a sufficiently large database (several terabytes or more) it becomes practical to record all value changes and events for the interesting signals within a design, such as state registers, busses, enables, and stall signals. If enough context is recorded with each value change then complex coverage questions can be more easily answered, coverage work can be more interactive and flexible, and the quality of tests and regressions can be better measured.

A. Asking the right questions

Examples of more complex coverage questions are those that involve multiple blocks in a large design, or questions about the frequency, duration, or sequence of events. A microprocessor engineer might ask, *"Are pipelines A, B, and C all stalled at nearly the same time?"*, *"Is this bus arbitration scheme fairly serving all requestors?"* or *"Is fifo X more than ¾ full at a time when many cache misses have occurred in the past N cycles?"* With appropriate tools, the data for tests can be visualized through line charts that are similar to waveforms, and engineers can compose coverage questions step-by-step, visualizing the result after each step. A first step might chart all the time periods when a fifo is more than ¾ full, and a second step might use those time periods to mask a chart of the rolling average of cache misses. This work can be done interactively, composing new questions without the need to immediately re-run regressions to see answers. Once a coverage question is composed, it can become a metric for choosing the best tests to put in regressions, or for eliminating redundant tests from regressions. Metrics can automatically measure the quality of regressions over time as the design and tests change.

B. Four Major Components

A Big Data approach to coverage can include four major components:

1. A fast database with several terabytes of storage.
2. A way to instrument the verification testbench to log activity in the database.
3. A web application to explore, measure, and present the data.
4. A REST data service to give other tools, scripts and web pages access to coverage metrics.

To record data from simulations in the database, we need to instrument the verification testbench, and this process must be easy or engineers will not take time to do it. To record simple value changes of signals and registers, we create XML files that specify module names and the hierarchical paths to specific signals/registers within those modules. A script reads the XML files and converts them into executable Verilog code that detects value changes as a simulation runs and records them in a compact ASCII file. After a simulation completes, a different script reads the ASCII file and stores its data in the database. For events that are more complicated than value changes, such as CPU instructions issuing, we provide a small library of instrumentation subroutines, and engineers can call these subroutines from their testbenches. The subroutines can record metadata along with the required simulation time, value, and signal name to create richer data. For instance, when an instruction issues, the metadata could include the opcode and source/destination register IDs. The simulation performance impact of instrumentation varies with the number of instrumented signals and the activity factors of those signals, and is similar to the impact of enabling covergroups.

Database tables let us record the full context of events in our simulations. We can retrieve not just a hierarchical signal name and the simulation time when it changed to a new value, but also metadata, test name, date of simulation, simulation command-line, person who ran it, RTL version and more. So much data can be recorded that even large databases need a strategy to avoid exceeding the available storage. In practice, not all simulations need to record data. A representative subset of nightly regressions can record data by default, and a command-line switch can turn on recording for other simulations on demand. The recorded data can have configurable expiration dates, since old data loses its importance as the design changes. Expired data is automatically removed. The database might contain data derived from user-defined metrics (i.e. average FIFO occupancy) that requires far less storage than the raw data that it is derived from, and derived data can be given a longer expiration period than raw data. This preserves important information for long periods while minimizing storage of raw data.

To benefit from the data, there must be a practical way to visualize it and calculate results from it. Our approach is a web-based application that lets engineers define functions that operate on the data, and creates charts of both raw data and functions. The goal is for engineers to define functions that measure how well a test performs its intended purpose. A large set of primitive functions are built into the application, and user-defined functions are built up from primitives and from other user-defined functions. Primitives provide arithmetic operations (i.e. `add`, `subtract`, `integrate`), logical operations (i.e. `and`, `or`, `greaterThan`), and time-base operations (i.e. `timeShift`, `length`). Some primitives consume waveforms and return new waveforms, like `add` or `mask`, while others consume waveforms and return a scalar value, like `max` or `integrate`. Likewise, user-defined functions can return either waveform or scalar data. Waveforms are visualized with line charts, letting us see the activity in a test over the course of simulation time. Scalar values are visualized with bar charts, letting us compare how tests or groups of tests performed against user-defined metrics. Bar charts that group tests by their configuration settings could answer the question, *"What configuration settings are best at stressing features of the floating-point unit?"* Bar charts that group tests by week could answer the question, *"As weeks pass by, are our regressions getting better at stressing the floating-point unit?"*

The user-defined functions that engineers create are valuable metrics for understanding our simulations and we want outside scripts and tools to have access to those metrics. A REST data service can provide that access, so that any script that knows the names of a specific metric and a test filter (test filters select which tests to calculate the metric across) can ask how the selected tests performed against that metric and receive a reply across the network. A watchdog process can use the service to check on the health of nightly regressions and automatically e-mail an alert if a metric shows an unusual change. Web pages can use the service to update tables of coverage information. Potentially, an adaptive testing algorithm could use the data service to check the metrics of past simulations before deciding what tests to run next.

Though our work on big data coverage is not complete, we've seen some challenges that come from this approach. First, the ability to visualize coverage data is most useful if charts can be generated interactively, within a couple minutes or preferably within seconds. The web application must be written with performance in mind. For example, if a particular, intermediate calculation is used on new data every day, then it is worthwhile to automatically precalculate and cache intermediate results in the database so that when higher-level functions are run interactively, they can use the cached results to reduce calculation time. Second, to limit storage requirements we must thoughtfully choose what data to record and how long to keep it. Tools must make it easy to adjust the storage policy.

Big data coverage can reduce the need for traditional functional coverage, replacing many SystemVerilog coverpoints with user-defined metrics that are more powerful and flexible. However, it is unlikely to replace corner-case coverpoints that are hit rarely and unpredictably. That is because we don't want to instrument 100% of our simulations, overwhelming our database. Traditional functional coverage is a more efficient tool for those coverpoints.

VII. BIG DATA TOOLKIT

While tools exist in the industry to help manage Big Data in Verification [7], we've created our own Big Data Toolkit mostly comprised of open source and a few commonly available commercial tools. We've created scripted flows around these tools to collect, analyze, and present the data that guides our decisions. We've summarized how we use these tools across these different tasks in Table 2.

TABLE 2: A Big Data Toolkit

Collect		**Analyze**	**Present**		
Structured DBs	*Unstructured DBs*	*Data Queries Tools*	*Web Frameworks*	*Plotting*	*Textual*
MariaDB	Hadoop	Perl/Python ORMs	JavaScript/Jquer	Excel*	Perl
Postgres	Vertica*	(DBIx/SQL Alchemy)	Python/PHP	GNUPlot	Python
MySQL		Excel* with pivot tables and filtering	(Django/Yii)	ChartJS	Ruby
		MySQL Workbench	Jenkins	Grafana	
			Custom		

** = commercial tool*

A typical flow to maximize your ROI with minimal effort using the Big Data Toolkit above involves defining a database schema, collecting the data, inserting the data into a database, and presenting the data for analysis. To accomplish this you would first design the schemas around the data you want to capture in MySQL Workbench or an Object Relational Mapping (ORM) tool like DBIx or SQL Alchemy. You would then write scripts in Perl or Python that extract the data you need (either from logfiles, recurring Jenkins jobs that sample the data, or other data sources) and insert that into the database schema. Next, you would write query scripts in Perl or Python that provide a rich command line interface to the user to query the database with ease. You would also ensure the scripts can return the queried data in an ASCII table, CSV, or JSON format. You would then setup a web server using a custom web framework that uses a plotting engine like ChartJS or Grafana that defines the set of plots you want to see. ChartJS parses the data from your query scripts, and Grafana parses the data directly from your database (avoiding the need to create query scripts). These tools produce interactive charts that can be customized by the user to see the data they want, and can be exported to Excel for further analysis. We've found that this tool flow can be created by a small team with a few resources in about one engineer month of effort and maintained by a few engineer hours a month.

VIII. FUTURE USE CASES

Although we are using Big Data to answer these types of questions every day in our development activities, there is even more we want to do. In order to stimulate thoughts for the reader, we will briefly share some of the opportunities we will explore in the future.

Coverage data is often stored by EDA vendors in structured databases and is heavily summarized. Working with EDA vendors to store this data in higher performing unstructured databases using the Big Coverage Data ideas above would help us answer more questions about how effectively our tests are reaching deeper state spaces within our designs. We would like to see improvements in intelligent test grading.

Our Test Results data is currently stored in a structured MySQL database. We'd like to store it as unstructured data and use Hadoop to analyze it. This would enable us to store more dynamic and varied information per test. Engineers would then decide what test attributes (test modes, testbench configurations, error signatures, regression durations and conditions, failing modes, register programming, coverage points hit, etc) they want to store per test and analyze when they need it most (while debugging a failure, closing coverage, etc).

Additionally, we would like to expand our use of Big Data to get even more value from our cycles. Big Data can help our team drive improvements in intelligent stimulus generation and coverage driven stimulus. From an analysis standpoint, we see excellent opportunities in stimulus generation to leverage modern artificial intelligence (AI) and machine learning (ML) techniques. For data visualization, we will be exploring improvements such as heatmaps to summarize and visualize the data and the anomalies in that data. To date, we have been analyzing data within the domains in which the data was generated. There are additional opportunities to take a bigger picture view by querying and merging Big Data across multiple data domains. Data is often segregated into multiple databases and performing queries across these databases will let us gain even more insight.

IV. Conclusion

While even more can be done with Big Data than was discussed here, we have shown how anyone can quickly jump into the world of Big Data and leverage these concepts in their own environments to bring value to their teams. We have shown how asking the right questions about the data you collect can lead to insights that make your engineers smarter, letting them overcome design and verification challenges. With the Big Data Toolkit, mostly comprised of open source and freely available tools, readers can create their own Big Data infrastructure using databases for storage, query libraries for analysis, and web frameworks for presentation and analysis. What questions can you ask to make your verification more intelligent? What data can you start collecting today to answer those questions? Now is the time to collect and go beyond simply answering questions with Big Data. Now is the time to enable that data to drive action and change to improve your design and verification abilities.

References

[1] D. Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety," 2001.

[2] SAS, "Big Data. What it is and why it matters.," [Online]. Available: https://www.sas.com/en_us/insights/big-data/what-is-big-data.html. [Accessed 1 4 2018].

[3] Oracle, "What is Big Data?," [Online]. Available: https://www.oracle.com/big-data/guide/what-is-big-data.html. [Accessed 1 4 2018].

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 2004.

[5] D. P. Carrington, T. Pertuit and A. J. Pippin, "Automation of Reusable Protocol-Agnostic Performance Analysis in UVM Environments," in *DVCon*, 2019.

[6] Vertica Systems Inc., "The Vertica Analytic Database Technical Overview White Paper," 2010.

[7] S. Sikand, "Functional Verification Big Data Analytics — Real-Time Global & Local Insights," IC Manage, unpublished.