Automation of Reusable Protocol-Agnostic Performance Analysis in UVM Environments

D. P. Carrington[1], A. J. Pippin[2], T. Pertuit[1]
Hewlett Packard Enterprise
[1]5400 Legacy Drive, Plano, TX 75024
[2]3404 E Harmony Road, Fort Collins, CO 80528

***Abstract*- Performance analysis of a design requires significant effort simply to measure the figures of merit. In order for performance to take a principal role in design and verification processes, the total process of measuring, analyzing, and optimizing performance needs to be easy as possible. We developed a framework named Autoperf that uses design metadata from the verification environment to generate analyses of digital memory system performance against a general and extensible performance model. Using this model, we can process simulation results in a protocol-agnostic way and produce uniform performance measurements across all interfaces, protocols, and devices. We were able to do performance testing at block- and system-level simulation without costly custom measurement development for each testbench. The Autoperf approach simplifies performance verification in simulation to a familiar design verification workflow: component instrumentation, testbench integration, test development, expressing expectations, regression testing, and final sign-off. This contrasts with commercial products which take protocol-specific approaches that require constant manual inspection of graphs to do performance verification. The tools in Autoperf enable us to identify the root cause of performance issues efficiently, to visually check our measured results against the documented architectural intent of the design, and to deliver high-performance solutions using reusable verification components and purchased verification intellectual property.**

I. Introduction

The typical and recommended approach to performance verification is to measure performance frequently and iteratively in order to guide optimization of the design. The task of analyzing the design's performance is ideally the only labor-intensive part of the process. However, in our experience, enabling performance measurement is not always easy. Some key challenges for performance measurement are proprietary technologies or emerging standards such as Gen-Z [1], which have few or no commercial solutions available; custom performance measurement scripts that duplicate functionality present in the verification environment, such as connectivity, routing, or cross-protocol transaction mapping tables; and anticipating the needs of root cause analysis and performance experimentation to search for performance bugs. In order to validate our architected performance goals and to control the total cost of performance verification, we have developed the Autoperf performance verification framework. Some of our key performance measurement needs required special consideration during the development and implementation of Autoperf.

In the prototyping and initial development phases for Autoperf, we anticipated the need to integrate our future designs with proprietary technology and with emerging standards, as well as with established standards. Considering the need for reuse in verification environments, we decided that the implementation of Autoperf required three things. First, a uniform treatment of performance results would reduce the cognitive load for the user of Autoperf, enabling greater usage of the tool. Second, a protocol-agnostic interface to performance data collected by our instrumentation would prevent undesirable coupling between Autoperf and the technology we were using. Third, integrating performance instrumentation with the verification environment through UVM verification components would enable adequate measurement of performance and provide reusable performance instrumentation without adding significant complexity to the testbench.

In response to the challenges that motivated Autoperf, we developed it as an extension to the Universal Verification Methodology (UVM). This methodology consists of a generic performance model, a verification library that extends UVM, simulation post-processing applications which implement performance measurements, and a web application for high-level analysis of performance results. In addition to these essential components of Autoperf, we provided performance data export functionality and a generic performance model API to access performance measurements which enables project-specific scripting. These tools have enabled performance testing, performance bug root cause analysis, and architectural validation of our design's performance. We have successfully reused UVM components with integrated performance instrumentation in multiple verification environments using this methodology. By enabling reuse of performance instrumentation and speeding the measurement and analysis of performance results,

Autoperf has enabled us to validate the performance of multiple logic design projects which have varied in the scope of their performance goals from the minimal to the complex.

II. Description of the Autoperf Framework

The approach in Autoperf is to instrument the design, uniquely identify and perform all of the available measurements and present those measurements in tabular and graphical formats. Autoperf generates and records all measurements it possibly can from the available data and lets the user review the measurements they desire.

The software architecture for Autoperf consists of verification libraries, performance analysis tools, a Vertica [2] database of test performance results, and a web application. The design architecture of Autoperf consists of a protocol-agnostic channel model and a latency model. Understanding these two models lets a DV engineer generate detailed, accurate performance results in a UVM test environment by adding some minimal code to their UVCs (Universal Verification Components).

A. Verification Libraries

Autoperf includes verification libraries in Specman and SystemVerilog to report performance data and metadata from the simulation environment. These libraries support efficient implementations of our custom verification components as well as integrating commercially purchased VIP into Autoperf using vendor-supported callback function mechanisms. The content of the Autoperf library is to permit appropriate UVM components to emit appropriate log messages, in the defined and machine-readable format required by the Autoperf log API.

As a result, most of the Autoperf library is defining the message formats used in the Autoperf log API. Other facilities in the Autoperf library provide for doing the marginal extra bookkeeping needed for performing all performance instrumentation tasks in the verification environment, in addition to the normal function of UVCs. Finally, macros that help define performance test cases are available to help produce large quantities of performance test cases via simple text substitution. In conjunction with properly parameterized UVM sequences for performance test stimulus, the Autoperf verification libraries permit the user to create performance tests that fully verify the design's performance intent.

In our verification environment, we use UVM-ML in order to support VIP written in Specman and in SystemVerilog (SV) [3]. The original Autoperf codebase is written in Specman. However, by using UVM-ML and proxy objects in SV, we can report performance data and metadata from SV UVCs and even leverage purchased commercial SV VIP. By using the Autoperf verification libraries, and the post-process performance analysis, we avoid needing to write difficult performance checking assertions.

1) Extensions to UVM

Autoperf provides facilities meant to be used by UVM monitor, scoreboard, and sequencer components. Autoperf also provides "mix-ins" for the top level testbench, for sequence items, and for miscellaneous checker components. When this code is included, the UVM testbench is able to report performance measurement data using standard UVM logging facilities. Simulator-specific code is used to open a new output file for storing only Autoperf-related messages. Autoperf libraries exist in both Specman and SystemVerilog. Language-specific mechanisms are used to include Autoperf instrumentation conditionally in each language. In Specman, Autoperf code is included as an aspect.

Each Autoperf-instrumented UVM monitor sends a message to the Autoperf log during test initialization indicating its presence and whether or not it is enabled. This message defines the Autoperf channel that is being monitored. The channel is defined by a unique name, a protocol name (such as UART or PCIe), a boolean that enables bandwidth testing, a bus width, in bits per clock cycle, and a clock period in picoseconds. After test initialization, the monitor will send an Autoperf packet message each time it observes a transaction item. Each packet message contains a flit size, number of flits, a timestamp, an operation name, a number of application data bits, and the name of the monitor observing the packet. This is sufficient for defining bandwidth and utilization measurements based on the Autoperf channel model for interfaces.

When Autoperf is configured to perform latency measurements, packet messages are referred to by other Autoperf log messages. Two types of messages refer to packet messages: transaction messages and packet-packet link messages. A transaction message indicates that a packet was injected to the DUT as part of the test stimulus. A packet-packet link message indicates a cause-effect relationship between two packet messages. Transaction messages can be produced by sequencers (when they process sequence items produced by a test sequence) or by monitors (according to behavior configured inside the monitor). Packet-packet link messages are generally produced by scoreboards.

When Autoperf is enabled, it is best if UVM transaction items contain suitable fields for referring to the packet message associated with that item. This enables a scoreboard to emit a packet-packet link message linking the two packet messages associated with the two items that the scoreboard matches together.

2) Performance Test Definition Facilities

We primarily define performance tests by creating a small Specman file using text substitution that inserts parameter values and strings into a short template. In general, these templates define the mode of the performance test and any parameters needed for the sequence, such as transaction counts, mix weights (usually expressed as a percentage of the stimulus that are read or write requests). Using text substitution, we create a number of these files that express a wide range of operating conditions for the device. In order to sort through the test results later, these files also define metadata for each test using the Autoperf libraries, which support naming each test and adding parameters describing the test's operation.

The Specman file uses language macros defined in the Autoperf verification library to specify test-level metadata. The supported facilities are to set measurement and processing options with the `autoperf_test_config` macro, or to set a test metadata key and value using the `autoperf_test_param` macro. Most string- or integer-valued options can be set using the `autoperf_test_config` macro by providing the name of the option and the value to be set as a string. A user-defined piece of metadata can be added to a test by providing the Specman data type, the name, and the value of the parameter to add. The user-defined metadata will be stored in the database, where it can be searched by key and value.

3) Use with Purchased VIP

We have successfully implemented Autoperf instrumentation on purchased VIP. Purchased VIP commonly has callbacks for IP behaviors such as initialization, observing packets, and matching packets. These callbacks and the general API for the VIP can be efficiently leveraged to provide the needed metadata for Autoperf. For example, an Interface UVC (iUVC) VIP should provide a callback for observing a single packet. The packet object presented to the callback would include fields such as data payload length, packet type, etc. which are sufficient for producing packet messages that describe the channel bandwidth and utilization monitored by the iUVC.

4) Reporting Performance Data

The Autoperf log is mainly used for debugging fundamental issues in the instrumentation of UVCs. Often, it is easier to search for text in the Autoperf log than to examine the proper table in the Autoperf database file. Typically, working through enabling instrumentation in the order that the instrumented data are printed in the log is a convenient workflow that is familiar to DV engineers. The first part of the log contains messages describing the DUT and the interfaces under performance instrumentation: a test message, a configuration message, monitor messages and links between monitors across the DUT. This is all pre-run information and usually occupies a very small part of the log – only a few kilobytes. During the test run phase, Interface UVCs (iUVCs) and Module UVCs (mUVCs) report performance data in the form of packet and packet-packet link messages. This information is all written to the Autoperf log as a series of JSON documents that are each tagged with the type of Autoperf log message that is implemented.

We have also written single-purpose performance post-processing tools that artificially create an Autoperf log from a test case when this is more expedient than implementing updates to legacy VIP components. By using this log as an interface to the rest of the Autoperf analysis flow, we were able to do significant performance analysis with no investment into measurement – only investing in instrumentation.

B. Performance Analysis Tools

The Autoperf tools are comprised of a number of loosely coupled protocol-agnostic applications, of which the performance measurement applications are the most important. These applications process the simulation results after the simulator has exited. To use Autoperf, invoking the post-processing application is sufficient to run all of the needed steps to perform a performance analysis of that test.

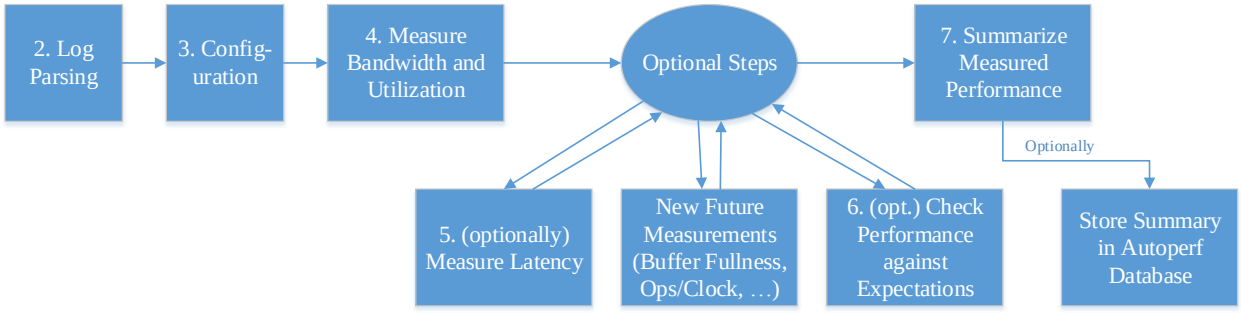

Figure 1: Steps in the Autoperf Post-processing Application, showing the use of the Autoperf component applications

1) Autoperf Post-processing Application

The post-processing application is a program that comprises all of the steps that need to be run in order to fully analyze a simulation that defines a performance test. Generally, the Autoperf post-processing only needs to be run if the simulation ran without the verification environment detecting errors. Running the Autoperf post-processing flow will measure the bandwidth and utilization achieved by the design, optionally measure latency, optionally check that the performance met the pre-defined criteria, and optionally store the results of doing these measurements into the database. The specific steps are diagrammed in Figure 1.

2) Autoperf Log Parsing Application

The log parsing application is a program that reads the Autoperf log file which was created during simulation. The log parsing always needs to be done in order to produce a database file that can be processed efficiently by the measurement applications. The process of log parsing checks that the performance data reported by the verification environment is associated with correctly formed metadata constructs in the Autoperf performance model. A modest (usually tens of megabytes) in-memory cache is used for monitor and recent packet information in order to accelerate processing packet data and packet-packet link information. The log parsing application converts all of the metadata and data into a structured, table-based format in a SQLite database. The resulting SQLite database is typically much smaller than the original log. In our workflow, we can compress and save this SQLite database in order to be able to re-generate detailed test performance results later.

```
hp_autoperf_test_message_s-@3466 { "test_name" : "myblk_generic_stream_rd_100pct", "config_file" : "myblk_config",
"generic_stream_test" : "rd_100pct", "latency_test" : false, "read_percent" : 100, "read_trans" : true, "write_trans" : false,
"other_trans" : false, }
hp_autoperf_config_message_s-@3467 { "avg-interval" : 256, }

hp_autoperf_mon_message_s-@30627 { "mon_name" : "sys.sve.my_blk_env.serdes_env.tx_agent.mon", "mon_shortname" :
"SERDES->MYBLK", "mon_test_bw" : true, "mon_max_bw" : 2147483647, "trans_fabric_id" : "SERDES", "mon_bus_width" : 256,
"mon_clock_period" : 1000, }
hp_autoperf_mon_message_s-@30624 { "mon_name" : "sys.sve.my_blk_env.noc_env.rx_agent.mon", "mon_shortname" : "MYBLK-
>NOC", "mon_test_bw" : true, "mon_max_bw" : 2147483647, "trans_fabric_id" : "NOC", "mon_bus_width" : 256,
"mon_clock_period" : 1000, }

hp_autoperf_mon_mon_pair_message_s-@30672 { "source_mon" : "sys.sve.my_blk_env.serdes_env.tx_agent.mon", "dest_mon" :
"sys.sve.my_blk_env.noc_env.rx_agent.mon", "mon_mon_test_ql" : true, }

hp_autoperf_pkt_message_s-@31048 { "pkt_head_timestamp" : 2599500, "pkt_tail_timestamp" : 2600500, "pkt_flit_count" : 2,
"pkt_flit_size" : 256, "pkt_transid" : 0, "pkt_op" : "RD", "pkt_data" : 0, "pkt_fabric" : "SERDES", "mon_name" :
"sys.sve.my_blk_env.serdes_env.tx_agent.mon", }
hp_autoperf_trans_message_s-@31050 { "trans_id" : 1, "trans_timestamp" : 2599500, "trans_monitor_startpoint" :
"sys.sve.my_blk_env.serdes_env.tx_agent.mon", "trans_target_addresses" : "0x0000000004120080", "trans_type" : "READ", }
hp_autoperf_pkt_message_s-@31051 { "pkt_head_timestamp" : 2599500, "pkt_tail_timestamp" : 2600500, "pkt_flit_count" : 1,
"pkt_flit_size" : 256, "pkt_transid" : 0, "pkt_op" : "Read", "pkt_data" : 0, "pkt_fabric" : "NOC", "mon_name" :
"sys.sve.my_blk_env.noc_env.rx_agent.mon", }

hp_autoperf_pkt_pkt_link_message_s-@30549 { "scbd_name" : "sys.sve.my_blk_env.checker.serdes_noc_scbd", "source_mon" :
"sys.sve.my_blk_env.serdes_env.tx_agent.mon", "dest_mon" : "sys.sve.my_blk_env.noc_env.rx_agent.mon", "source_pkt_head_ts" :
467500, "dest_pkt_head_ts" : 472500, "response_turnaround" : false, "latency_flow_key" : "MY_BLK", }
```

Figure 2: Sample log contents in type-tagged relaxed JSON format accepted by the Autoperf Log Parsing application

3) Autoperf Analysis Configuration Application

The analysis configuration application is a program that provides configuration of options to the measurement applications. It supports a general configuration file for testbench-specific configuration and further test-specific configuration supplied by the test case. The resulting set of options can be printed on the standard output in order to ease debug. This program is sometimes used interactively to query and update the default options for presenting a particular performance database file. The use of this configuration utility enables re-doing the same analysis steps later without having to specify many command-line options to set the desired behavior of the measurement tools.

4) Autoperf Bandwidth and Utilization Measurement Application

The bandwidth and utilization measurement application is a program that analyzes the packet messages output by `uvm_monitor` instances. The Autoperf channel model defines an algorithm for doing these bandwidth and utilization

calculations. The averaging interval over which bandwidth is calculated is defined by the user using a factor that is an integer number of clock cycles. The clock period for each interface is multiplied by the factor to get the time interval. The bandwidth and utilization measurements use the same averaging interval factor. We have found that a default averaging interval factor of 64 clock cycles produces usable graphs in the majority of cases. In a minority of cases, an increased averaging interval should be used to smooth the time-series on the graph and/or provide increased precision for bandwidth and utilization results. Examples of increased averaging intervals include DUTs which have a high ratio between the highest and lowest clock domain frequencies, or for long system-level tests. The value 64 is recommended as the minimum practical value for this setting as it reduces the precision of utilization measurements to 1.6%.

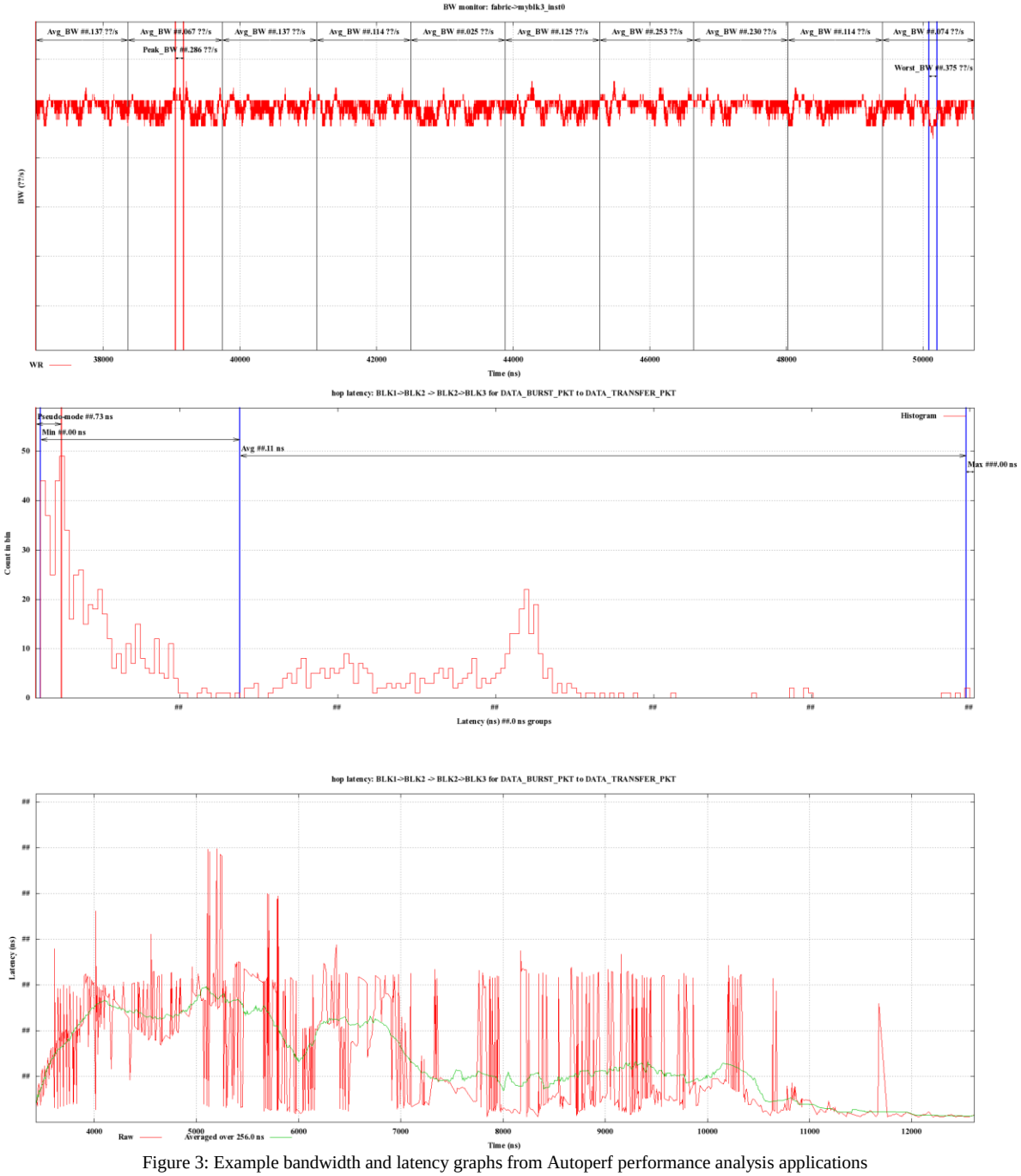

Figure 3: Example bandwidth and latency graphs from Autoperf performance analysis applications

5) Autoperf Latency Measurement Application

The latency measurement application is a program that analyzes packets that are grouped together into a related unit by packet-packet link messages output by scoreboards. The cause-and-effect relationships of the packet-packet link messages are interpreted as edges on a graph, and a graph algorithm identifies packet groupings that form identical flows. These packet groupings are inspected by another graph algorithm that identifies latency measurements of interest, including point-to-point latency across the DUT, latency across each block, and round-trip latency for each request and response pair. This analysis can be performed in low expected asymptotic time. The latency measurement application produces a latency histogram and a graph of latency versus time for each latency measurement identified by the algorithm.

We have used the latency measurements generated by this script to identify and track idle latency and dynamic latency for our designs under a variety of conditions. In general, idle read latency is a key performance indicator for memory subsystems. We have particularly tracked idle read latency and automatically identified cases where a design change unexpectedly increased read latency. By using point-to-point latency measurements across the DUT, we were able to isolate that increase in latency to the request path in a particular block. This enabled a rapid fix to this performance issue.

6) Autoperf Performance Expectation Testing Application

The performance expectation testing application is a program that generates a performance pass or fail result for a test case. Each test case can define its expected performance levels as a minimum or maximum value for a measurement. In the case of bandwidth or utilization measurements, performance patterns that should proceed as sequences of relatively long bursts or pauses can be specified as peaks and valleys that must occur, where a peak or valley is defined as the performance going past a particular value and staying there for a defined amount of time. This application is used by creating a spreadsheet defining the expected values. A script translates the spreadsheet into a text file suitable for check-in to version control defining the expectations in a database format.

7) Autoperf Performance Reporting Application

The performance reporting application is a program that prints a textual summary of performance results after the measurement applications have analyzed the test. This tabular textual summary is useful for understanding the overall behavior of the test. This application also stores the summary of performance data to the Autoperf database.

C. Performance Database

An unstructured Vertica database of summarized performance information is used to store and analyze the results from multiple performance tests. Unstructured data is used to implement user-defined metadata properties for tests more efficiently. The database is structured to enable a metrics generation workflow that consists of searching for tests by date, test name, and by user-defined parameters; identifying the measurements that those tests have in common; and constructing a graph based on measurement and parameter values found in that set of tests.

The database schema is in third normal form. A table describes each test case by recording the date and time it was run, along with any user-defined metadata associated with the test. Additional tables record the kinds of bandwidth, utilization, and latency measurements associated with that test. Yet more tables uniquely identify the points at which bandwidth and utilization are monitored, so that those can be correlated across test cases, and uniquely identify device flows whose latency was measured. This enables running queries that search for tests that have particular properties, and then joining those queries against the tables of measurement data in a way that the data returned by the query consist entirely of measurements of the same DUT behavior.

D. Web Interface

A web application provides a user-friendly interface to the performance database. Users can search test records using the domain-specific test parameters they defined in their performance test plan and analyze the search results individually or as a group. During analysis, users can examine the recorded summary data for tests, refine their search, or create graphs of performance data that are presented in a format suitable to the domain. The web interface implements the metrics generation workflow described in the previous section.

These example graphs show the kinds of analysis that can be done from the web interface by aggregating results from multiple tests.

Figure *4*: Example of multiple-test aggregate performance analysis from the Autoperf web interface

E. Channel Utilization and Bandwidth Modeling

In Autoperf, memory system throughput is modeled as data flowing through channels. A UVM monitor component that is instrumented for Autoperf corresponds to one or more Autoperf channels. An Autoperf channel has three essential components: a bus width (number of wires), a clock period indicating how often those wires' output is sampled for data, and a flit packing algorithm determining the approximate format of header of data on the bus. The channel model has three use cases: high-speed serial links, synchronously clocked interconnects, and bidirectional buses. The channel model transparently supports dynamic clock frequency scaling by approximation. It can support dynamic link width reduction. It provides both bandwidth and utilization measurements.

The calculation done with the channel model is conceptually simple. Each packet message refers to which channel it uses. The packet message contains attributes defining its flit width, flit count, and number of data bits. The total number of bits in the packet is calculated as the product of the count and width of the flits in the packet. The total number of cycles in the packet is calculated by dividing the total number of bits in the packet by the width of the interface. Dead or partly used cycles can be added in the middle of a packet by the flit packing algorithm. Finally, the calculation keeps track of whether the channel is utilized or not on each cycle by checking whether at least one flit is transferred on a particular cycle, and produces a number of data bits transferred on each cycle.

The monitor's protocol name is used to reference a configuration file that defines the flit packing algorithm in use for the interface. Three packing rules are implemented: "perfect", "split-header", and "sideband-header". This algorithm is responsible for dividing the overhead and data bits between the flits of the packet and defining how much of the interface is used on each cycle of a packet. For instance, the "perfect" rule sends all of the header bits, followed by all of the data bits, adding dead padding bits at the end of the packet if needed. The "split-header" rule enforces that header bits and data bits cannot be sent on the same cycle, effectively adding up to one extra cycle of overhead to a packet. The "sideband-header" rule is similar to the "perfect" rule, but evenly divides the overhead and data bits across all cycles of the packet. This last rule is most suitable for implementing support for dynamic lane width reduction but requires that the monitoring iUVC be aware of the current link width status in its performance instrumentation code.

In the most basic use case of the channel model, the flit width is set to 1 and the number of flits is set to the number of bits in the packet. The flit packing algorithm dynamically figures out the number of cycles that the packet uses. Alternatively, the flit width can be set to the bus width and the number of flits is set to the number of cycles needed to transmit the packet. In a dynamic lane width reduction case, the flit width should be set to the minimum number of lanes of the bus that can be active at once. The number of flits in the packet should be increased by the monitor if only part of the bus is active, based on the number of 'dead flits' that are sent on the inactive portion of the bus while that packet is being transmitted.

F. *Device Latency Modeling*

In Autoperf, device latency is modeled using a graph relation between monitored packets. The performance instrumentation code that needs to be added to the verification environment ends up being simple. There is only one rule to follow in order to enable Autoperf latency measurement: chains of cause-and-effect need to be unbroken from request to response. If a request causes intermediate requests and responses, those should also be modeled. For instance, a read-snoop-writeback-response flow can be modeled for Autoperf latency measurements using the diagrammed relationship shown in Figure 5.

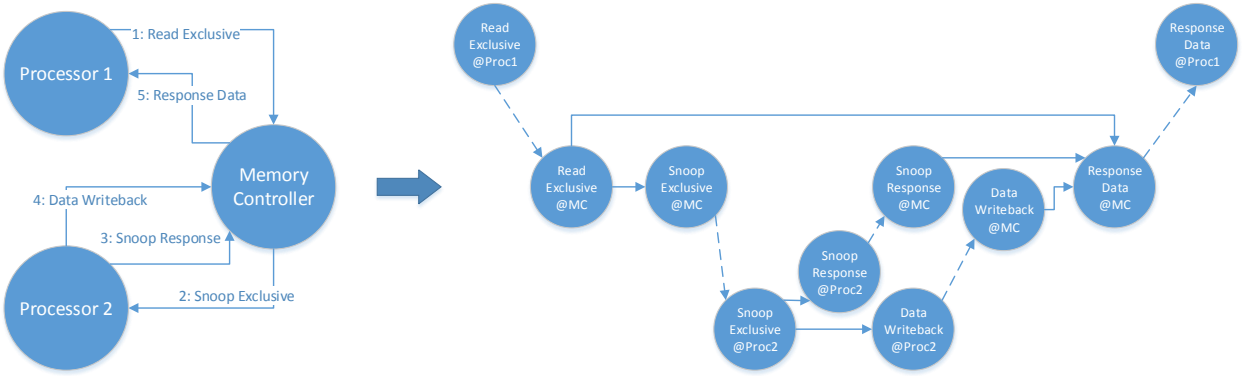

Figure 5: Example Read-Snoop-Writeback-Response flow showing latency graph modeling in Autoperf

Assuming that the network-on-chip (NoC) connecting these processors and the memory controller is the DUT, the verification engineer needs to know the latency measurements across the NoC. These correspond to the dashed lines in Figure 5. If there are several sub-components in the NoC, there might be intermediate steps between the processors and memory controller. In this test, the application is being run on Processor 1. Therefore, the latency from Read Exclusive to Response Data also needs to be known. In order for that latency to be correct, the solid lines corresponding to request-response delay in the Memory Controller and Processor 2 need to be accurate. These may be verification placeholder components only, in which case the modeled delays need to be tuned. If, for instance, part of the Memory Controller component is also part of the DUT, then the latency measurements made should be adjusted so that the verification engineer knows how much of the total latency is due to the DUT. By indicating to the latency model which components are verification placeholders, the latency measurement application can automatically make the measurements that provide the most value to the performance verification engineer.

III. Use of the Autoperf Framework

Using Autoperf for performance verification in simulation follows a workflow familiar to engineers who have experience with simulation. The steps of using Autoperf are component instrumentation, testbench integration, test development, expressing expectations, regression testing, and final sign-off. We realize benefits from Autoperf during each of these phases. We also have processes to validate that the results we compute from Autoperf accurately reflect the simulated device, known as measurement validation.

In the component instrumentation phase, the verification libraries are used to add functionality to UVCs. An example testbench is integrated with that UVC functionality by enabling any needed Autoperf functionality for the newly enabled UVC. Next, an existing test is used to validate the measurements performed by the UVC functionality. During this measurement validation, the measurement data are closely inspected to ensure that the interface is accurately modeled. Once this step has been performed, we can use the Autoperf framework to execute a performance test plan. Performance testing usually begins when the RTL model is almost functionally complete. When combined with a thorough program of verify-as-you-go, this ensures that the performance tests are done on a model that is valid and verified.

During test development, the Autoperf framework is used by DV engineers to check approximately that the performance test case is doing what they expect. Functional coverage and assertions are also used to ensure that the performance test case exercises the intended device corner cases, and functional checking is almost never disabled in case a functional bug is uncovered during performance testing. These performance test cases are defined in a comprehensive performance test plan that seeks to identify whether any of the expected performance-critical use cases for the DUT will fail to meet the design goals. For example, sequential and random access across memory addresses can result in different performance characteristics for caches, but both cases must perform at the expected level.

IV. Benefits of the Autoperf Framework

The key benefit of Autoperf is that performance testing can begin sooner, at lower levels of integration, to catch gross performance mistakes. Because Autoperf tools and UVCs can be reused without modification in any project, it only takes a couple of hours to begin performance testing using Autoperf. Furthermore, doing performance testing in block environments helps to enable performance testing in system environments because of the reuse of UVCs from block to system level environments. This is true for both bandwidth and latency performance measurements. The Autoperf instrumentation libraries and latency model permit full reuse of UVC instrumentation code from block to system environments.

Another important benefit of Autoperf is that performance issue root-cause analysis (RCA) is made easier by the detailed performance measurements enabled by the framework. One key ability that improves RCA is detailed latency measurements. For instance, when a request enters the DUT, how long does it take to leave the DUT? The answer to this question is a one-way latency measurement. Having latency measurements of the request time and the response time separately narrows down where an extra register stage or wait state was added during timing closure. Autoperf's further detailed latency measurements reduce manual effort in debugging latency increases.

Using Autoperf has let us reduce the amount of time we spend on re-checking performance results because of the ability to automate checking performance expectations. While the process is not perfect, we can avoid checking the results of a huge majority of our performance tests by only looking at performance tests that fail basic tests of our performance expectations, such as not hitting full utilization at the expected bottleneck for the test. We can then make a more thorough investigation of performance issues.

We can also validate our system architecture at a higher level by using the Autoperf web interface to analyze results from multiple tests together. One basic test is to graph the bandwidth versus latency of read requests in multiple tests. When combined with a performance test plan that varies the expected latency of read requests, we can see how the system's performance is affected by load.

The verification libraries used by Autoperf are very easy to configure. Because the instructions that come with Autoperf are very comprehensive, enabling basic performance measurements when reusing existing UVCs is very quick. We were able to start performance testing in five test environments for performance testing in just a week thanks to reuse of UVC performance instrumentation and Autoperf's performance measurement.

Autoperf lets us to avoid a couple common mistakes in evaluating the performance of the design. Doing performance testing against the RTL with the full design verification environment in place prevents against common mistakes in simulation that can result in measurement errors, as described by Jain[4]. This methodology avoids issues due to an unverified or invalid model. The use of UVM and verification languages avoids issues due to an improper implementation language or a poor random number generator. The use of standard analysis techniques in the Autoperf toolset help us avoid a subset of mistakes in overall performance evaluation, by performing some analysis of the data derived from simulation using a validated performance analysis toolset.

V. Limitations of the Autoperf Framework

Autoperf lacks key functionality to analyze processor performance, such as measuring instructions per clock or pipeline or buffer fullness. New measurement and analysis capabilities would need to be implemented for Autoperf to be able to measure these important figures of merit for verifying processor performance.

Autoperf is a fundamentally classical digital logic performance measurement framework. It is not designed to analyze the analog portions of electronic designs. This restriction permits Autoperf to implement algorithms related to classic digital logic, such as assuming that one bit is transferred per wire per clock cycle.

Autoperf has not been proven to work in an emulation environment; only simulation has been used so far. As Autoperf is a verification library that extends classical UVCs in a logging capacity only, we expect that Autoperf-powered performance tests will be amenable to emulation speed-up.

Table 1: Results of a brief study of the overhead of using Autoperf data logging in simulation

Test	Autoperf Simulation Runtime	Sim without Autoperf	Runtime Overhead	Autoperf Analysis Runtime	Autoperf Measurements Done	Analysis Overhead	Total Overhead
BLOCK_1	257.8s	248.8s	3.5%	30.27s	BW, LATENCY	11.74%	15.8%
BLOCK_2	275.4s	295.5s	-7.3%	6.67s	BW	2.42%	-4.5%
CLUSTER_1	1061.7s	916.0s	13.7%	17.65s	BW	1.66%	17.8%
CLUSTER_2	4912.4s	4094.7s	16.6%	297.63s	BW	6.06%	27.2%
SYSTEM_1	8784.5s	8602.4s	2.1%	41.90s	BW	0.48%	2.6%

Autoperf runs in a simulation environment. The performance impact on simulations of adding Autoperf data logging is about 6% to 9%, as measured by commenting out the statement that prints Autoperf data messages to the Autoperf log file. So far, we have not optimized the performance of Autoperf data logging, however, analysis suggests that Autoperf logging speed could be increased by a factor of 5, reducing runtime overhead to 1-2%. This analysis was done by doing a linear fit of the above data against the size of the general simulation log file and against the size of the Autoperf log file. This calculation suggested that significant overhead was incurred by just the statement printing to the Autoperf log file, in excess of the cost of file I/O. Optimizing just this output should be effective in increasing the Autoperf log file output rate.

The analysis applications of Autoperf run as a post-processing step. The runtime of these analysis applications is, on average, approximately 4.5% of the simulation run time. Overall, the total computer time overhead of using Autoperf is about 12% of simulation runtime, not including the runtime of any additional code needed to collect logged Autoperf data, which was not measured.

Finally, the Autoperf web interface lacks any facility to superimpose graphs, add data series, do variability analysis, or do any power set experimental design analyses. These can be implemented, but have not been implemented so far. Jain recommends these types of analysis as ways to avoid significant mistakes in performance evaluation [4].

ACKNOWLEDGMENT

We would like to recognize Derek Alan Sherlock's excellent work in performance analysis. His insights and complaints motivated us to improve our approach to pre-silicon performance. We would also like to recognize the contributions and involvement of other engineers involved in performance at Hewlett Packard Enterprise, whose engagement contributed to the motivation for Autoperf and made staffing possible: Chris Brueggen, Jason Wells, Michael Schroeder, and Craig Warner. Finally, we thank David Lacey for encouraging the publication of our results with Autoperf.

REFERENCES

[1] Gen-Z Consortium. (2018). *Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access.* [online] Available at: https://www.genzconsortium.org/ [Accessed 7 Dec, 2018].

[2] Vertica. (2018). *Big Data Analytics on Premise, in the Cloud, or on Hadoop | Vertica*. [online] Available at: https://www.vertica.com/ [Accessed 6 Nov. 2018].

[3] T. Pertuit, D. Gibson, and D. Lacey, "Is Specman Still Relevant? Using UVM-ML to Take Advantage of Multiple Verification Languages," DVCon Proceedings, 2018.

[4] R. Jain, *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*, Littleton, MA: John Wiley & Sons, 1991.