Are you really confident that you are getting the very best from your verification resources?

Darron May
Design Verification Technology Division
Mentor Graphics Inc
Newbury, UK
darron_may@mentor.com

Fritz Ferstl
CTO
Univa
Regensburg, Germany
fferstl@univa.com

Abstract — **Getting the very best from your verification resources requires a regression system that understands the verification process and is tightly integrated with Workload Management and Distributed Resource Management software. Both requirements depend on visibility into available software and hardware resources, and by combining their strengths, users can massively improve productivity by reducing unnecessary verification cycles.**

I. INTRODUCTION

Your last project had your compute grid running constantly at 100 percent capacity and your simulation licenses were maxed out most of the time. The next chip is twice as large so you must have to double or treble your resources.

Or maybe there is another way based on smart combination of the grid and regression management systems to ensure that every verification cycle is a valid cycle. This paper will show how adding control and visibility to these systems, and then better integrating them, will help your organization get the very best from every verification dollar. The paper will also explain how a regression system can be developed with the infrastructure to control and monitor exactly what is happening. This addition of intelligent automation allows dynamic reaction to the current status and create the maximum throughput and capacity. The diagram in Figure 1 shows the major parts of the complete system that will be explained in this paper.

Not getting the most from technical infrastructure used in a product development cycle can be expensive. The cost, which include software licenses, fixed assets, and wasted time, often can exceed that of the original investment in the infrastructure. This is why it's essential to monitor usage on a job-by-job basis, which requires integration between the license and grid scheduling software to provide for complete visibility. Accounting for metrics such as code churn, bug status as a sign of completion, and performance criteria for bus fabrics or bandwidth within communications applications while integrating the grid, regression and simulation software can paint an even more complete picture of what is happening in the verification environment. This paper will also describe how to combine these metrics to better track both verification resources and overall verification progress, and how to do so within the tight timescales required to produce right-first-time silicon.

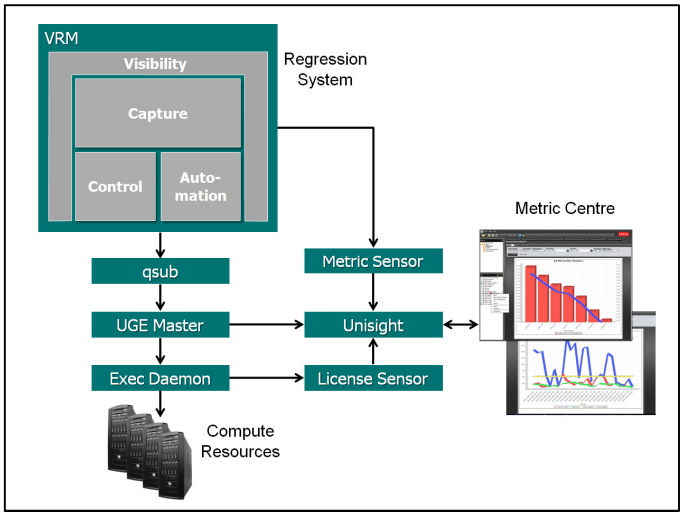

Figure 1 - Block Diagram of Compete System

II. THE REGRESSION SYSTEM

Twin demands increased quality and shortened design cycles put pressure on IP and SoC houses to leverage automation throughout their design and verification processes. In the verification space, these pressures require verification engineers to get more efficient contending with tasks such as coverage closure, bug hunting, smoke and soak testing, all of which are done through running lots of regressions. Regression systems can be heavily scripted, and are often developed and understood by just a small handful of people within a larger organization. Historical baggage is carried over from project to project, and sometimes the majority of users don't even know why they have to run a particular script. It just works until it doesn't.

To get the full benefits of automation, a regression system needs to be able to automate management of seeds for constrained random tests; rerun failed tests automatically, perhaps with more debug visibility; merge coverage across multiple runs; manage tool timeouts; and interface to compute resources. Complexity and capabilities grow over time, a major downfall of regression systems based on scripting alone.

Having the capture, control, automation and status of the regression system wrapped up within one complex script or a series of scripts that call each other can lead to execution and maintenance nightmares. An organization's verification resources should be focused on verifying designs not debugging environment infrastructure.

Using a purpose-built regression system can give verification engineers maximum productivity while reducing the maintenance burden. User productivity can be boosted in most aspects of verification management including capacity, performance, resource usage, turnaround time, preparation, maintenance, results and coverage analysis.

A regression system can be broken down into the ***capture*** and configuration, the ***control*** of when and how the actions are run, ***automation*** of the tedious tasks of gathering data such as coverage data and acting upon results, and finally the ***visibility*** into the regression status and results.

A. Capture

Capturing regression complexity requires abstracting away the running a script with a sequence of commands. To be able to control a regression system and produce actionable data for further actions, a method of separating the control of individual actions from their configuration data is required. Dependences between actions — the tasks of building, optimizing and running simulations — must be defined. The build needs to complete and an optimization performed before the simulations are run. Defining sequential and non-sequential dependences allows multiple parts of the build or multiple simulations to be run in parallel. Parameterization, key to global and local settings, allows actions to be configured to run in different ways. An action could have a setting for debug versus optimized, coverage versus no coverage, or even just the ability to run multiple versions of a particular tool. A system that is based on inheritance can save valuable time capturing the same settings over and over, with only small changes.

The main building block within Questa Verification Run Manager is called the runnable. It can be a group, base type or task. A group runnable allows hierarchy to be constructed, inheriting dependencies and parameters from their parent. Groups can have pre and post actions executed once per group and allowing tasks such as setup and clean-down. Inheritance can also come from a base runnable, thus allowing inheritance to be injected at any level of the hierarchy. The task runnable is the leaf-level action. The commands executed at this leaf level can be launched using any type of shell, the default being the simulator interpreter.

Runnables can be defined as sequential or non-sequential, allowing for either serial or parallel actions. Their execution can also be conditional and repeated using either a count or a list of parameter values. The execution area is a managed directory structure. It is possible to define local files that are either copied or linked into this area so that all paths and references can be relative and used by multiple users. This architecture means multiple regressions of the same design can be in flight simultaneously without each regression stepping on the other.

The runnable configuration allows for constructing a runtime graph of all the actions, and their scheduling and dependencies. Figure 2 shows the runnables and how to define the hierarchy within the XML configuration file.

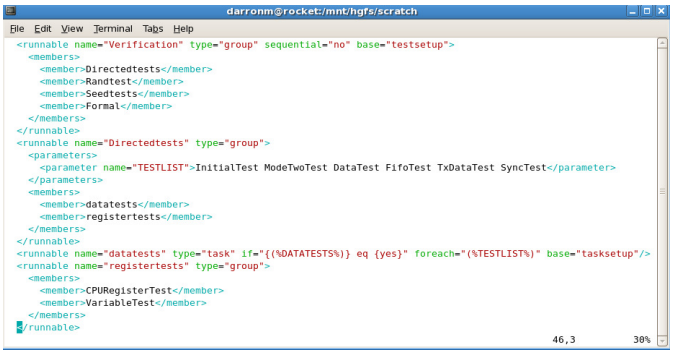

Figure 2 - Runnable Configuration

User questions or overrides can be applied to adjust the parameters, which in turn can completely change the graph. This provides a solid infrastructure for reusing any setup, either on or between multiple projects. And separating control from capture gives maximum flexibility.

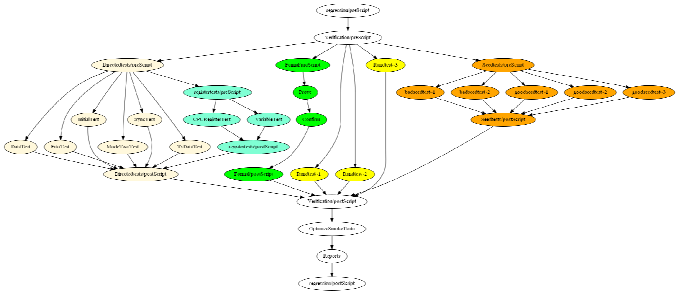

Figure 3 - Execution Graph

Figure 3 shows how the different parts of the defined flow in the XML in Figure 1 become an execution graph. Here we see directed tests colored cream, register tests in cyan, random tests in yellow, good and bad seed tests in orange, and finally formal proof runs in green. Note how the actions are built into a dependency graph so that the underlying execution can control them as separate actions, running them in parallel or applying other restrictions so the actions are run only when ready.

B. Control

Separating capture and execution of actions gives complete control over how and when the actions are executed. The runnable configures how the action is executed, the default being to run the action locally though it could also be to run using 'rsh,' 'ssh' or grid software. Methods can be conditional (based on parameters) and inherited (from a parent or a base), making switching execution method very flexible. The method holds information about how to interact with the execution resource (for example, the grid submission commands); and how to suspend, resume or kill jobs. This layer provides the

user with a simple grid interface making it easy to automatically kill all the submitted jobs for a regression automatically if the regression is stopped. It also allows for array jobs, in which actions are packaged into arrays and passed to the grid in a single command. The grid software can then efficiently unpack and manage the jobs. Job grouping is also available in which actions are grouped together and executed at once. This is extremely useful when job latency and start-up are expensive for short actions, allowing the overhead to be consumed across multiple jobs. All of this is achievable by adjusting a couple of parameters to enable the packing of jobs by the regression system.

Methods have associated queues with a variable maximum running parameter, which allows pre-balancing and scheduling between actions. This allows extra scheduling and job management even before the actions reach the grid or can allow load balancing between machines than are not part of a grid. Time-out's can be set for both queuing and run time of actions via parameters. This automatically manages jobs that hang and jobs that never get started.

A state machine manages each action and the running of all actions as a complete flow. Routines are called at certain times during the life of the regression and action; for example, when an action is scheduled, when an action has started or before the next action is started. Routines have default implementations but can be overridden by the user. So a regression might be set to stop when an event occurs, at a certain wall clock time, at a certain level of coverage or when more than a certain number of actions have failed.

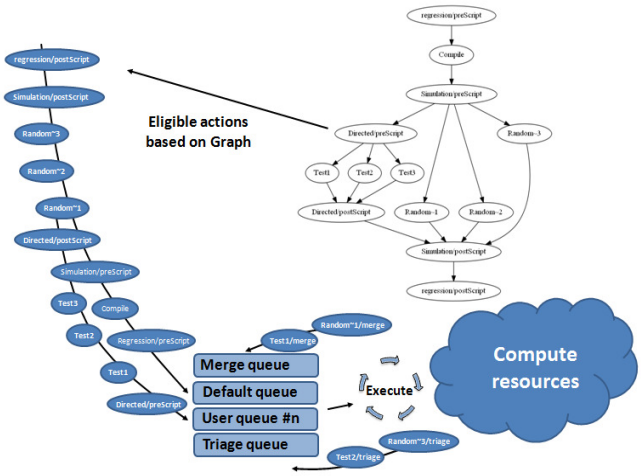

Figure 4 - Execution of actions via queues

C. *Automation*

Automating a system that separates capture and control is straightforward. The regression system needs to find the correct data and enable customized triggering of new verification tasks based off the results of earlier runs. Either the status returned from an action's execution or the UCIS UCDB (Unified Coverage DataBase) test record can be used to indicate the pass/fail status of a completed job. The test record has a TESTSTATUS attribute automatically set to the worst severity that occurred during simulation. Setting the value of an in-built parameter to the resulting UCDB file location will cause the status to be used. Setting an in-built parameter to point to a triage database will cause the failing actions to automatically generate a triage action to extract and store the relevant failure information. The failure can also trigger a re-run of the action with modified parameters that might, for example, enable full visibility and waveform capture for the test. The re-run can be further configured either to run immediately after the failure occurred or later as part of a global re-run at the end of the regression, after automatic analysis on which failures should be re-run. Set up properly, all data necessary to debug and analyze a nightly regression could be available when you come into the office in the morning.

Setting an in-built parameter to point to a merge coverage database file location automatically adds individual test results to a merge queue list. A queue is managed and actions are added, which merge the coverage from the passing tests into the merge UCDB, which in turn makes the incremental results available. If a testplan is being used to drive the verification process, then setting a built-in parameter to define the testplan file will ensure that the testplan is imported and merged with the merged coverage database.

With a constrained random methodology, seed management is required. Using a built-in parameter for the seed allows the seed to be random, then, if a re-run is needed, the seed generated the last time the action was run will be used again. Lists of seeds that caused good and bad behavior can be used to run new regression runs. The output from each action can also be managed automatically using auto-delete to clean up temporary files or output files that are not needed when tests pass, optimizing valuable storage capacity.

D. *Visibility*

Visibility is required to see which actions have completed, which ones are still running, and which actions can be run. User preferences differ, so it's good to provide the verification engineer flexibility in how he looks at regression status. The user interface allows for starting, monitoring and analyzing all of the actions within the regression. A command line is also available to allow the same status information to be visible directly from the shell. This command line can also be used to query the regression configuration to figure out what actions and parameters are available to guild their use of the tool. Regression results are also available in HTML to allow viewing within an external viewer.

Having a regression system that separates the control from the configuration data improves its overall maintenance and user productivity. Major features can be coded into the system itself instead of added as a series of scripts with multiple calling levels, which often lead to a debug nightmare.

III. WORKLOAD MANAGEMENT AND DISTRIBUTED RESOURCE MANAGEMENT

Software for Workload Management (WLM) and Distributed Resource Management (DRM) has become a fundamental building block of compute farms or grids and large-scale technical computing data centers. It is as crucial as the networking infrastructure or file sharing services and provides a similar type of service (and potential bottleneck) to data centers that a conveyor provides to the assembly line in an

industrial manufacturing factory: if it slows down, the production gets severely impeded, and if it stops, then everything comes to a screeching halt. This potential risk is amplified by several major IT trends:

- Technical computing (i.e. computer aided design, simulation, verification and testing) has become the base-line of innovation across all industry sectors, and Electronic Design Automation (EDA) has been on the forefront of this process for significantly more than a decade.
- Technical computing data centers never shrink. They only grow, particularly when it comes to core counts. With that comes growth in job throughputs and number of projects, departments, users or applications being serviced. That, in turn, requires policies ensuring all entities receive their fair share of resources.
- The managed resources comprise an ever more complex microcosm. Servers have heterogeneous architecture (CPU type and core counts, bus and memory architecture, performance data) and may have special purpose devices attached, such as accelerator hardware (NVidia® GPUs or Intel® Xeon Phi™). The network topology and the file storage architecture need to be taken into account to enable optimal performance of applications. And besides CPU, memory and I/O, resources such as software licenses must be managed, a factor of specific importance in the EDA industry.

The key requirements of any data-center-based WLM/DRM system can be summarized with a few key words: It needs to be ***dependable*** to avoid costly downtimes. It needs to be ***responsive*** and ***scalable*** to meet throughput requirements and keep utilization near the optimum. And it has to be ***flexible*** to adapt to the changing infrastructure complexities and allow for implementation of policies reflecting the operational goals of an organization. The following subsections will discuss each of these attributes in more detail and through data points and experiences gathered from typical installations of Univa® Grid Engine™, which is a proven WLM/DRM system widely used across leading organizations in all industry sectors including EDA.

A. Dependability

Uptime of 99.9% or even more is a pre-requisite to getting the most out of a technical computing infrastructure. The system is up not only when its daemon hierarchy (see Figure 5 for an example of a typical WLM/DRM architecture) is running, but also when it is actually ready to accept, schedule, dispatch and execute jobs. The following describes what is required to ensure these sorts of high uptimes:

- Continuous uptime and service even during reconfiguration.
- Fail safety in case of partial system failure: If, for example, the scheduling component has an issue (failure or slowness) then it still needs to be possible to accept new jobs or process already running jobs.
- Rapid service readiness in case of system restart: If central controller components of the system are being restarted (e.g. for an upgrade of the software) then service readiness needs to be reached as quickly as possible. A system being restarted with hundreds of thousands of jobs in its queues should provide service within a few minutes.

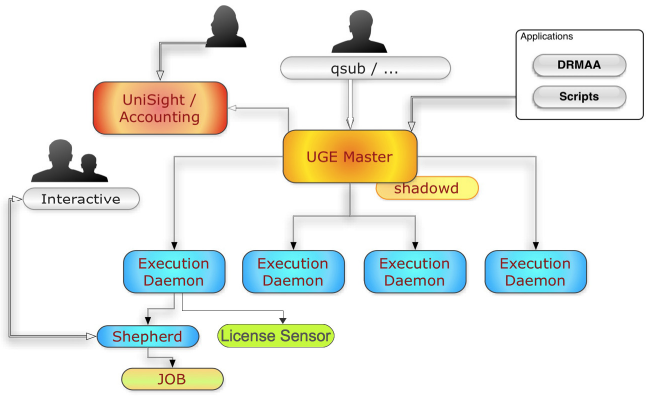

Figure 5 - Grid architecture diagram

B. Responsiveness

Responsiveness affects two dimensions in the operation of a central computing infrastructure. One is "to feed the beast" so it can deliver results at the expected rate. The other is to provide good end user and administrator experience. Engineers interacting with the system as well as its system administrator have a job to do and it isn't waiting for responses from the system.

When it comes to injecting workloads at a rate required by state-of-the-art high throughput-clusters, consider that submission rates reach 200 jobs per second from the command-line and can exceed 1,200 jobs per second through an API. It is important to note that one such job may potentially represent millions of tasks when organized as a so called array job (see Scalability below). With such rates, it is possible to feed the system for days or even weeks worth of execution quite literally within a few seconds.

Features like multi-threaded request processing, tailored handling for status queries, and easy configuration changes or job submissions are crucially important to end user experience. Status queries (on particular jobs, parts of the jobs or all jobs) are very common operations and must perform well while having minimal impact on job submission, scheduling and dispatching, accounting gathering or operations changing the configuration of the system.

C. Scalability

Core counts in today's commercial production clusters can exceed 150,000, some companies have dozens of clusters, each with tens of thousands of cores. In either case, the system needs to be capable of providing close to 100% utilization. At the top range, a moderately sized cluster might handle 100,000,000 jobs per month. This means between 50 to 60 jobs per second that need to run through the full job life-cycle from start to end, 24x7. Life-cycle steps include: job submission with

verification, scheduling of workflows while considering policies and load metrics, dispatching to execution servers, accounting and reporting while a job executes, as well as post-mortem and persisting out each of the above steps to enable restoring status in case of failures.

Maintaining such throughput numbers is only possible with some key architectural provisions. One aspect is that multi-threading needs to be utilized inside the system for parallel processing of activities and to take advantage of multi-core architectures of modern server infrastructure. Another crucial point is the persistent store for status data which needs to meet high performance and dependability requirements. Ideally, sites need to have a choice of approaches like in-memory databases, transactional databases or flat file storage

It's also important to take advantage of intrinsic workload efficiencies. If large numbers of workloads operate on different parts of a design's data but otherwise are exactly identical, then it should be possible to group them into a so called array job. This array job should then only represent a single job in the DRM/WLM system, which gets instantiated for every piece of data being processed. This ensures minimal memory footprint or impact on scheduling times allowing millions of such array tasks to coexist at the same time without overloading the system.

D. Flexibility

A system needs to be able to adequately represent the heterogeneous computing infrastructure of a site, from various hardware components to services like file sharing. Especially for the latter, the system needs to be extensible and flexible in the resource pool description, while providing a high degree of out-of-the-box information and metric gathering to simplify configuration.

The cluster configuration and status information (on the one hand) and the workload request profiles (on the other) need to be tied together by a rich set of policies reflecting operational and business goals. Examples for what should be possible to express in such policies are:

- Which type of job fits where and how much resource can it use?
- When can jobs run concurrently on a server and when are they exclusive?
- Which project or department or user or category of workload is entitled to how much of the cluster resources and to which resources in particular?
- Are these soft limits which can be exceeded or undercut but should average out over time (so called fair-share) or are these fixed quotas which must not be violated?

It's no exaggeration to state that all investments in a technical computing data center are a potential waste of money if the drid hasn't been optimized. Not that even running 5% below the optimum on a 24x7 basis equates to 18 days of downtime. So every tenth of a percent counts. Sites using unsuitable and badly tuned WLM/DRM software are known to often reach only 60% of what they could accomplish, and many don't have the ability to even measure what their utilization numbers look like. (See Section VI: Monitoring Metrics.)

IV. MANAGING SOFTWARE LICENSING

Spending on software licenses can exceed tens of millions of dollars matching or exceeding the investments in the hardware infrastructure. Organizations struggle to get a grip on license utilization. Many organizations have subsidiaries across the world, each with their own pool of licensed software. All these subsidiaries can have different working hours and usage patterns for the licenses they own. Asking questions like "Where is excess capacity?" or "Where is insatiable demand?" is a key benefit using license orchestration software in conjunction with the grid system.

Still, licenses inevitably sit idle. Detecting these cases and letting workloads in one part of the company borrow unused licenses from another is the second key functionality that license orchestration software has to solve and where it has to work tightly with the workload management system. Policies representing the operational goals of organizations have to guide this process because more than one job usually will compete for a free license at any given point in time and some projects are more important than others. The license orchestration software needs to have the flexibility to setup corresponding policies that answer questions like:

- Which type of job has access to particular licensed applications and licenses features?
- Which department, project or user has access to how many licenses?
- Is the license entitlement of an entity (e.g. a project or a user) a hard limit or is it a target value to be approximated on average over time (fair-sharing)?
- What is the priority order through which jobs get access to licenses and what are the influence factors?
- What is the desired behavior if a pending, high priority job requires a license which a running, low priority job currently occupies (pre-empt or not and if then how)?

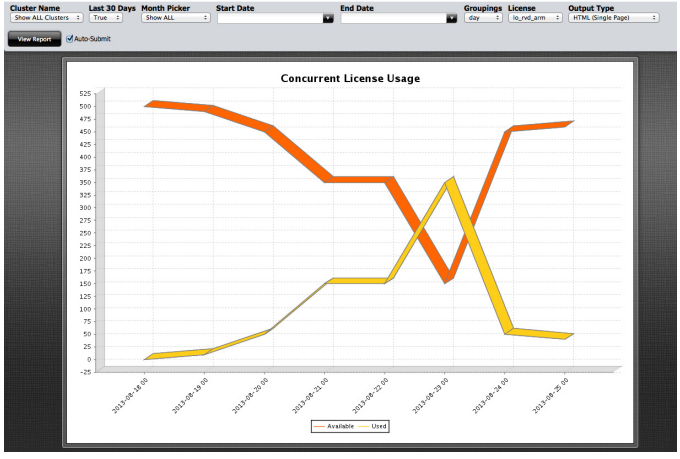

Figure 6 - license reporting showing impact of license usage orchestration

Tackling such challenges could be seen as a requirement for a WLM/DRM system but in practice it is a task which has to stretch across the clusters which a company operates around the globe, each of which runs its own instance of a WLM/DRM system. Hence license orchestration becomes a standalone task and products exist which are addressing it. An example is Univa® License Orchestrator, which is tightly integrated with the grid software and thereby provides solutions for all of the use cases discussed above. With such a combined solution companies have been able to realize immediate cost savings of 20% and often much more. (See Figure 6.)

V. Integration Between Regressions And Execution

The regression system separates execution from capture with a particular method describing transparent ways to execute the actions or jobs. A runnable can have several methods defined with parameters, each conditionally selecting which method is used. Methods can be written to execute the actions locally, via ssh or rsh commands or via WLM/DRM software. The default method causes actions to be run locally. Below in Figure 7 we see the example of a conditional method used to run actions using the ssh command on a remote host. The if attribute holds a test for when the method is enabled, in this case the parameter MODE has to equal "homegrid" for the method to be used.

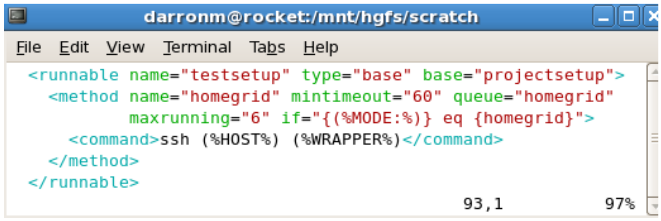

Figure 7 - Method for run actions via ssh

Special attributes and parameters built into the method control how to interface with the grid software. The gridtype defines the grid system being used so that the regression system knows how to pause, continue and delete jobs. The GRIDOPTS parameter contains all the default switches required to submit a job for a particular grid type and allows the regression system to adjust the switches to make use of other features within the grid for example array jobs. The method used to interface with UGE is shown below in Figure 8; again it has an if attribute. The method in Figure 7 and Figure 8 could be put into a single runnable and the setting of the MODE parameter to the required value would switch the execution between the ssh and grid execution.

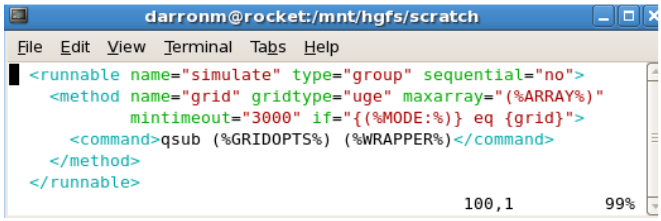

Figure 8 - Method for grid integration

The 'qsub' command is the executable used to submit actions to the grid and by setting the 'maxarray' attribute to more than one will cause the regression system to pack the actions into arrays for submission to the grid. One of the most important aspects of submitting to the grid is the rate at which actions can be sent. With array jobs, 1000s of actions can be packed into a single submission command, and therefore allowing 10,000s of jobs to be submitted at a very fast rate. This is possible by simply adjusting the value of a single attribute called the maxarray value.

VI. Monitoring Metrics

Having pertinent reporting and monitoring tooling entails three aspects: gathering comprehensive metrics, providing analytics to distil useful reports from that data, and having a user interface allowing for easy navigation. We'll discuss these in the following three subsections.

A. Data Gathering

The most commonly used metrics should be reported by the WLM/DRM system by default and the monitoring and reporting analytics software should provide default reports extracting valuable information from that standard data. Key metrics of an infrastructure are the utilization of servers and the embedded resources such as the CPUs and cores or main memory and virtual memory utilization. These and other common metrics are available in most workload and resource management systems or license orchestration software.

In addition there is often a case for site-specific data. Or there can be a need to extend the set of reported metrics by integrating a regression system and the WLM/DRM software. Accomplishing such metric extensions requires that the WLM/DRM system provides plug-in interfaces allowing for inserting any data that might be desired to track. An example of how easy it can be to expand metrics reporting was shown in Figure 1 with a metric or load sensor interface. Those load sensors can report metrics data relative to the host they are running on or any host in the cluster and for the cluster globally as well. Below is a simple example for such a load sensor written in Bourne shell:

```
#!/bin/sh
end=false
while [ $end = false ]; do
        read input
        # read anything from stdin and stop if it
        # says "quit"
        if [ "$input" = "quit" ]; then
                end=true
                break
        fi
        # else report a metric for this host
        echo $HOST:mymetric:5
        # and one for  the cluster globally
        echo global:clustermetric:100
done
```

Load reporting will be triggered periodically by sending carriage return characters. The above example will report static values for two metrics every time, one for the local host and the other for the cluster globally. The only other required step is registering the metrics inside the grid configuration.

B. Analytics

Beyond comprehensive reporting data it's necessary to accurately analyze all usage data as it is associated with the concrete workloads, projects, departments or users having utilized resources. Usage reporting needs to be held against policies defining resource access in the workload management or license orchestration system to check on correct implementation of the policies or to analyze the reasons for deviation and take corrective actions.

Another angle to be represented in reporting is to highlight underutilized resources or, conversely, overbooked resources which lead to long waiting times for jobs and thus delays in producing the results the site is expecting to gain from the computing infrastructure. Figure 9 and Figure 10 show the embedded UniSight™ analytics and reporting system that can be used to report on policies such as fair-share or on idle resources..

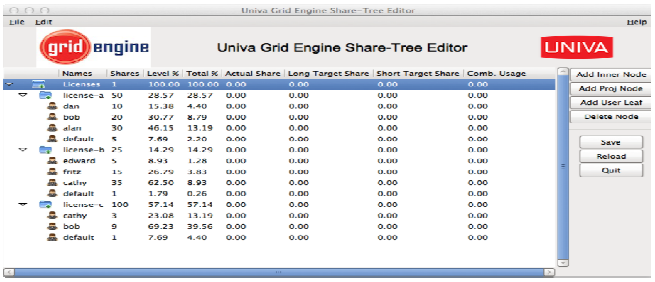

Figure 9 - Reporting on meeting the fair-share policy in the cluster

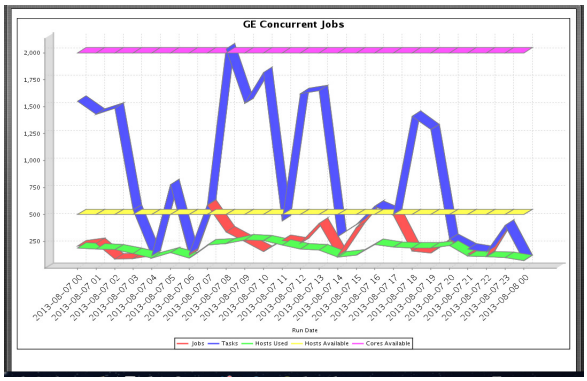

Figure 10 - Reporting idle resources

Another crucial point is that the amount of data collected by a large throughput cluster can be huge within just hours and it is not uncommon for cluster administrators to want to look back for weeks or even months. In addition, we have already discussed that companies often have several of these clusters in operation and they may have license orchestration being employed in parallel. So the reporting and analytics framework needs to be capable of aggregating all that data consistently into a data warehouse and then provide efficient means to mine the data for pertinent reports. Careful database and analytics design is required in order to enable reporting based on clusters that service 100,000,000 jobs per month or more.

C. Reporting and Analytics Interface

With the vast amount of data being collected and the versatile sets of metrics being represented, it is crucial that the reporting and analytics interface provides various avenues for how to approach dissecting the data and turning it into reports. Examples for useful analytics schema as employed by users of UniSight™ are:

- A comprehensive set of efficient default reports for the most important cluster characteristics, e.g.
  - Host inventory and usage
  - Job list, memory usage and run/wait times
  - Queue utilization
- Variations of ad-hoc analytics which allow to create custom reports, such as:
  - Job, host and license usage analyses
  - "Top Down" for all of the above, i.e. starting with all data in the multi-dimensional data space and filtering out what's not of interest
  - "Bottom Up – Saiku" analysis for all of the above, i.e. starting with an empty report and adding data from the multi-dimensional data space being of interest

All variations of reporting should be easily accessible and customizable via a single sign-on web interface such as the one shown in the screen shot in Figure 11.

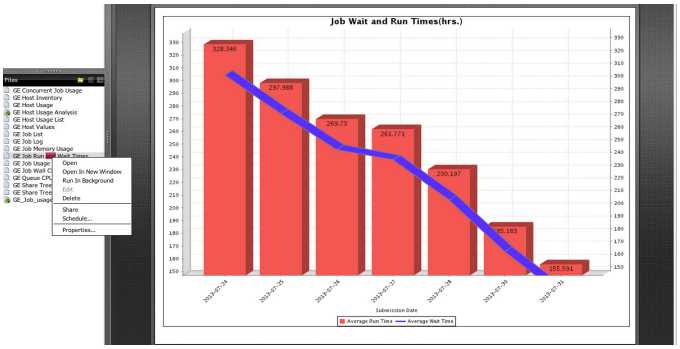

Figure 11 - Reporting and analytics screen shot

With such infrastructure it is easy to see how it is possible to add application specific metrics to the list of data that is gathered automatically. In the case of verification there are many metrics that can be gathered and reported next to data such as hardware machine and software license usage to give a fuller picture of the compete process. Trending such data over the period of the verification process and having it available in one place allows all the stakeholders to be able to make the right decisions dynamically.

For example a job run successfully on the grid doesn't mean that the results of the application are successful. With the regression system and grid software integrated it is possible to transfer information about tests that are run, pass/fail information, number of good/bad seeds, types of test i.e directed or constrained random. All of this information can be

transferred to the analytics and reporting system using the load sensors explained above allowing higher level job metrics to be viewed with the grid information over time. Further metrics not associated with the jobs but with the project can also be transferred like the number of lines of HDL code, the number of lines of code that have changed (code stability) and open/closed bug count numbers. Viewing all of these metrics together over time gives a complete picture of the verification process as it progresses as described in the 2012 DVCon paper "Metrics in SoC Verification" [1].

VII. CASE STUDIES

The following section details case studies of implemented systems similar to the ones explained in this paper utilizing a regression system with its integration to grid software. These are all summarized in the table in Figure 12.

Industry	Sub process	Productivity	Pre-VRM/Analysis	With VRM/Analysis	Benefits
IP Developer	Nightly regression test time	Throughput	28 hours	2.5 hours	9 X faster
	Results and Coverage Analysis	Turn-around	2 hours	20 minutes	6 X faster
	Regression file cleanup	Capacity	15 minutes	30 seconds	30 X faster
Automotive	Nightly regression test maximum	Throughput	40 tests	320 tests	8 X more tests
	Nightly regression test setup time	Turn-around	30 minutes	2 minutes	15 X less time
	Nightly regression addition time	Turn-around	60 minutes	5 minutes	12 X less time
	Nightly regression Script Files	Turn-around	10 files	1 file	10 X easier
	Nightly regression Results Analysis	Turn-around	>1 hour	<1 minute	60 X faster
Memory	Project #1 regression time	Throughput	120 hours	23 hours	> 5 X faster
	Project #2 regression time	Throughput	17.5 days	2 days	> 8 X faster
	Regression file cleanup	Capacity	Space exceeded	Complete	Regressions Complete
Semiconductor	System regression test redundancy	Throughput	38.9 hours	9.4 hours	> 4X faster
Gaming	Reduce runtime variation	Throughput	33 hours	11 hours	3 X faster
Micro-processor	Job clubbing	Throughput	350 minutes	125 minutes	~ 3X faster

Figure 12 - Real examples of productivity improvements

The first three examples are from an IP developer, the automotive and memory industries. Each of these examples saw regression throughputs increase from between 5X and 9X due to the fact that their new regression set-up allowed them complete control on each and every test that was run, the architecture of their old system was restrictive and limited their ability to run some tests in parallel. Introducing a structured approach to capturing regressions not only allowed them improve verification throughput but also improved turn-around time on tasks such as test setup time, test maintenance, test clean-up by between 6X and 15X. Regression clean-up being one such task that was time consuming and automation achieved a 30X improvement. In the other example non-effective management of disk space was causing some incomplete overnight regressions, this resulted in no data to analyze in the morning and tests having to be run again.

The fourth and fifth examples are from the gaming and semiconductor industries; these are examples of how automation and visibility of the effectiveness of each test can allow new regressions to be selective in what is run in future. The 3X improvement was achieved by the ability to analyze quickly and easily the runtime variation introduced by constrained random tests. Initially a variation of between 2X to 10X was seen across the same test with different seeds, being presented with this information allowed the regression system to automatically pick seeds from the quicker higher achieving coverage tests to provide the improvements. More than a 4X improvement in regression times where gained by a user in the semiconductor industry by the automation of optimization. In this case test ranking or grading was carried out to find tests that where redundant so that new regressions could be run to achieve the same coverage but with fewer tests.

The last example shows the effectiveness of job clubbing within the micro-processor industry where large numbers of shorter jobs need to be run. The latency and start-up of a job use to take five minutes and 1000 two-minute simulations needed to be run on a 20 CPU farm. Without clubbing this required 50 runs in batches of 20 on the 20 CPUs at 7 minutes each totaling 350 minutes in runtime. By grouping the simulations into batches of 10, each new job took 10 times two minutes simulation minutes, plus the five-minute start-up, a total of 25 minutes. With clubbing this resulted in 100 jobs of 25 minutes over the 20 CPUs therefore requiring five runs and a total run time of 125 minutes, close to a 3X improvement in throughput just by changing a parameter in the regression system.

VIII. CONCLUSION

This paper has shown how important it is to have full control over what tests run within your regression, making sure that each test is run with the goal of improving overall verification. Once the regression system has the knowledge of what needs to be verified, a grid system is required to ensure that the compute and software resources are being utilized to their full. Integration between the regression and grid systems is important to ensure that there are no inefficiencies in transferring what needs to be done to the execution engines. Regression systems need to separate capture from control to allow verification to be carried out in a manner ensuring that every verification cycle is being executed to improve the progress towards completion and there is no redundancy. This paper has detailed how this can make massive improvements to the throughput of regressions with some real examples shown in the table below.

A grid system is also required to get the best from the compute resources and to ensure that they are fully utilized and used in a fair way between the teams that need to use them. Just a very small percentage down time can lead to days of verification being lost. Software license utilization is just as important as hardware utilization and having a system that can monitor and feedback the usage numbers across the organization can lead to better utilization. The integration between the grid and regression system is important to allow jobs to be queued as quick as possible. The use of job arrays and the ability of the regression system to bundle smaller jobs in larger jobs can lead to massive throughput gains.

Finally a system that allows the gathering of metrics from regression, grid, software and user metrics, and then displaying them in a single system, allows a full picture of what is happening within the verification environment. This visibility allows correct decisions to be made dynamically to ensure that silicon is delivered right first time and on time.

IX. REFERENCES

[1] Metrics in SoC Verification, DVcon 2012, Andreas Meyer and Harry Foster.