SLAC–PUB–9761

Hardware Testing and System Qualification: Procedures to Evaluate Commodity Hardware and Production Cluster[*]

John Goebel
Stanford Linear Accelerator Center,
Stanford University,Stanford, CA 94309

Submitted to Linux Journal

[*]Work supported by Department of Energy contract DE–AC03–76SF00515.

Introduction

Without stable hardware any program will fail. The frustration and expense of supporting bad hardware can drain an organization, delay progress, and frustrate everyone involved. At Stanford Linear Accelerator Center (SLAC), we have created a testing method that helps our group, SLAC Computer Services (SCS), weed out potentially bad hardware and purchase the best hardware at the best possible cost. Commodity hardware changes often, so new evaluations happen periodically each time we purchase systems and minor re-evaluations happen for revised systems for our clusters, about twice a year. This general framework helps SCS perform correct, efficient evaluations.

This article outlines SCS's computer testing methods and our system acceptance criteria. We expanded the basic ideas to other evaluations such as storage, and we think the methods outlined in this article has helped us choose hardware that is much more stable and supportable than our previous purchases. We have found that commodity hardware ranges in quality, so systematic method and tools for hardware evaluation were necessary. This article is based on one instance of a hardware purchase, but the guidelines apply to the general problem of purchasing commodity computer systems for production computational work.

Defining System Requirements

It is difficult to maintain system homogeneity in a growing cluster environment. The hardware available to build systems changes often. This has the negative effect of adding complexity in management, software support for new hardware, and system stability. Introducing new hardware can introduce new hardware bugs. To constrain change and efficiently manage our systems, SCS developed a number of tools and requirements to enable an easy fit into our management and computing framework. We reduced the features to a minimum that would fit our management infrastructure and produce valid results with our code. This is our list of requirements:

1. One rack unit (1U) case with mounting rails for 19 inch rack

2. At least two Intel PIII CPUs at 1GHZ or greater
3. At least 1GB of ECC memory for every two CPUs
4. 100MB Ethernet interface with PXE support on the network card and in the BIOS
5. Serial console support with BIOS level access support
6. One 9GB or larger system disk, 7200 RPM or greater
7. All systems must be FCC and UL compliant

Developing a requirements list was one of the first steps of our hardware evaluation project. Just listing 'must haves' as opposed to 'nice to haves' grounded the group. It slowed feature creep, useless additions to hardware, and vendor specific methods for doing a task. This simple requirement culled the field of possible vendors, and reduced a tendency to add complexity were none was needed. Through this simple list, we picked eleven vendors to participate in our test/bid process. A few vendors proposed more than one model, so a total of thirteen models were evaluated.

Starting Our System Testing

The eleven vendors we choose ranged from the largest system builders to the small, screwdriver shops. The criteria for being in the evaluation was to meet the list of basic requirements and send three systems for testing. We needed the systems for ninety days. In many cases, we did not need the systems that long, but it's good to have the time to thoroughly investigate the hardware.

Two of the three systems were racked, the third was placed on a table for visual inspection and testing. The systems on the tables had their lids removed, and were digitally photographed. Later the tabled systems would be used for the power and cooling tests and visual inspection. The other two systems were integrated into a rack in the same manner as all our clustered systems, but they did not join the pool of production systems. Some systems had unique physical sizing and racking restrictions that prevented our being able to use them.

Each model of system had a score sheet. The score sheets were posted to our working group's web-page. Each problem was noted on the website, and we tried to contact the vendor to resolve any issues. In this way we tested the system, the vendors willingness to work with us, and their ability to fix problems. We had a variety of experiences. Some vendors just shipped us another model, some worked through the problem with us, others responded that it was not a problem, and one or two ignored us. This quickly narrowed the systems that we considered manageable.

Throughout the period of testing, if a system was not doing a specific task it was running hardware testing scripts or run-in scripts. Each system did 'run-in' for at least thirty days. No vendor does 'run-in' for more than seventy-two hours, and this allowed us to see failures over the long term. Other labs reported that they too saw problems over long testing cycles.

We wanted to evaluate a number of aspects of all the systems. First, the quality of the physical engineering. Second, how well it operated and if it was stable. Third, measure a system's performance. Last, evaluate the contract, support, and vendor's responsiveness.

Physical Inspection

The systems placed on the table were evaluated by several criteria:

1. Quality of construction
2. Physical design
3. Accessibility
4. Quality of the power supply
5. Cooling design

Quality of Construction

The systems greatly varied in quality of construction. We found bent-over, jammed ribbon-cables, blocked airflow, flexible cases,

and cheap, multi-screw access that were unbelievably bad for a professional product. There were poor design decisions, like a power switch offset in the back of a system that was nearly inaccessible once the system was racked. On the positive side of the experience, there were a few well engineered systems.

Physical Design:

This evaluation would include quality of airflow and cooling, rackability, size/weight, and system layout. Features such as drive bays out the front would also be noted. Airflow is a big problem with the hot x86 CPUs especially in restricted space like a 1U rack system. Some systems had blocked airflow or had little to no circulation. Heat can cause instability in systems and reduce operational lifetimes, so good airflow is critical.

Physical Construction:

Rigidity of the case, no sharp edges, how the system fit together, and cabling, are part of this category. These might seem small, uninteresting factors until you get cut by a system case, or have a huge percentage of 'dead on arrivals' because the systems were mishandled by the shipper and the cases were too weak to take the abuse. We have to use these systems for a number of years, and to have a simple yet glaring problem is a pain and potently expensive to maintain.

Accessibility:

Toolless access should be a standard on all clustered systems. When you have thousands of systems, you are always servicing some. To keep the cost of that service low, parts should be quickly and easily replaceable. Unscrewing and screwing six to eight tiny machine screws slows down access to the hardware. Also, parts that fit so one part does not have to come out to get to another part and easy access to drives are pluses. Some features that we did not ask for, like keyboard and monitor connections on the front of the case are o.k., but not really necessary.

Power

We tested the quality of the power supply using a Dranetz-BMI Power Quality Analyzer (see sidebar). Power correction is often noted in the literature for a system, but we have seen radically different measurements relative to the published number. For example, one power supply that was published to have a power factor correction of .96 actually had a .49 correction. This can have terrible consequences when multiplied by 512 systems. We tested the system at idle and under heavy load. The range of quality was dramatic and an important factor in choosing a manageable system.

The physical inspection, features, cooling and power-supply quality test weeded out a number of systems early. Getting these out of the way first reduced the number of systems that we had to do extensive testing on, thereby reducing the amount of time for testing in general. System engineering, design, and quality of parts ranged broadly. Moving to the next testing stage would also cull the herd and result in systems that we have been pleased to support.

Testing via Software

Run-In

Run-in (often called burn-in) is the process that manufacturers use to stress systems before they put them in the field. It is used to find faulty hardware. There are a number of open source run-in programs. One common program is the Cerberus Test Control System <http://sourceforge.net/projects/va-ctcs/>. It is a series of tests and configurable wrapper scripts designed originally for VA Linux Systems's manufacturing. Cerberus is ideal for run-in tests, but we also developed specific test based on our knowledge of system faults. We were successful in crashing systems with our scripts more often than using a more general tool such as Cerberus. Testing using programs developed from system work experience can be a more effective than using Cerberus alone, so consider creating a repository of testing tool.

Read the instructions carefully and realize that run-in programs can damage a system; you assume the risk by running Cerberus. Also, there are a number of knobs to turn, so consider what you are doing before you just launch the program. But if you are going to build a cluster, you will need to test system stability, and run-in scripts are designed to test just that quality.

At the time that we were testing systems, two members of our group wrote own run-in script based on some of the problems that we have seen in our production systems. Unlike benchmarks, which try to measure system performance and often have sophisticated methods, the run-in script is a simpler process. The system is put under load and either passes or fails. A failure crashes the system or reports an error, 'passes' often do not report information. We also ran production code, which uncovered problems. Production code should always be run whenever possible. In our evaluations, we had a few failures. One of the systems that passed the initial design inspection tests with flying colors failed under heavy load.

Whenever a system was not being actively evaluated, it was in run-in, so we far exceeded the seventy-two hour run-in time that is the maximum manufacturers can afford to test the systems.

Performance

There are a plethora of benchmark programs. The best benchmark is to run the code that will be used in production just like it is good to run production code during run-in. This is not always possible, so a standard set of benchmarks is a decent alternative. Also, standard benchmarks establish a relative performance value between systems, which is good information. We do not expect a dramatic performance difference in commodity chipsets and CPUs, but there are performance differences when different chipsets/motherboard combinations are involved, which was the case in this testing trail.

We also wrote a wrapper to a number of standard benchmarking tools, and packaged it into a tool called HEPIX-Comp (High

Energy Physics - Compute). It is a convenience tool, not a benchmark program itself. It allows a simple 'make server' or 'make network' to measure different aspects of a system. For example, HEPIX-Comp is a wrapper for the following tools (among others):

- Bonnie++
- IOZone
- Netpipe
- Linpack
- NFS Connectathon package
- Streams

Understanding the character of the code that will run on the system is paramount to evaluation through standard benchmarking. For example, if you are network constrained, a fast frontside bus is less important than network bandwidth or latency. These are good benchmarks that measure different aspects of a system. Streams, for example, measures the I/O memory subsystem throughput, which is an important measure for systems with hierarchical memory architectures. Bonnie++ measures different types of read/write combinations for I/O performance. There a many others in HEPIX-Comp.

Many vendors report performance which tends to give the best possible picture. For example, sequential writes as an I/O performance measure is pretty rosy compared to random, small writes, which is closer to reality for us. Having a standardized test suite run under the Linux installation that is used in production establishes a baseline measurement. If the system is tuned for one benchmark, it might perform the benchmark well at the expensive of another system performance factor. For example, systems tuned for large block sequential writes hurts small random writes. A baseline benchmark suite will at least show an 'apple to apple' comparison, although not the potently best performance. So this is by no means a perfect system, but it is one more data-point in an evaluation that characterizes system performance.

All the data was collected and placed on internal webpages created for the evaluation and shared among the group. We met once a week and reported on the progress of the testing. After our engineering tests were complete, we choose a system.

Non-Engineer Work

Non-engineering factors (contractual agreements, warranties, and terms) are critical to the success of bring in new systems for production work. The warranty terms and length affects the long-term cost of system support. We also try to assess the financial health of the company. A warranty does little good if the vendor is not around to honor it.

Another aspect of that couples the non-engineering work with the engineers is the acceptance criteria, which is seldom talked about until it is too late. These criteria determine the point in the deployment that the vendor is done and the organization is willing to accept the systems. This should be in writing in your purchase order. If the vendor drops the system off at the curb, and later during the rollout period some hardware related problem surfaces, you need to be within your rights to tell the vendor to fix the system problem or remove the systems. On the vendor side, a clear separation of what is a hardware and what is a software problem needs to be clear. Often a vendor will have to work with the client to determine the nature of the problem, so the cost of that will need to be built in to the price of the system.

The Result

The success of the method outlined in this article is apparent in how much easier, and therefore cheaper, it is to run the systems we chose after doing this extensive evaluation. We have systems that we purchased without doing the qualification outlined here. There has been a lot fewer problems after the better evaluation, and we are able to get more work done in other areas, like tool writing and infrastructure development. And we are less frustrated, as are our researchers, having good hardware in production.