# The Computer as Software Component: A Mechanism for Developing and Testing Resource Management Software

Narayan Desai [#1], Theron Voran [*2], Ewing Lusk [#3], Andrew Cherry [†4]

[#]*Mathematics and Computer Science Division, Argonne National Laboratory*
*Argonne, IL 60449, USA*
[1]desai@mcs.anl.gov
[3]lusk@mcs.anl.gov

[*]*Computer Science Department, University of Colorado*
*Boulder CO 80309, USA*
[2]theron.voran@colorado.edu

[†]*Leadership Computing Facility, Argonne National Laboratory*
*Argonne, IL 60439, USA*
[4]acherry@alcf.anl.gov

***Abstract*— In this paper, we present an architecture that encapsulates system hardware inside a software component used for job execution and status monitoring. The development of this interface has enabled system simulation, which yields a number of novel benefits, including dramatically improved debug and testing capabilities.**

## I. Introduction

In [1] and [2], we described a suite of system software components for scalable parallel computers. We laid out an architecture that enabled encapsulation and isolation of the functions provided by process managers, job schedulers, batch queueing systems, accounting packages, and so forth. We presented examples of such components, together with a flexible infrastructure that allows them to securely communicate with one another. We have also produced a software development kit that makes it easy to create new components or new instances of existing ones, thus enabling system administrators to customize the overall component set for particular installations. The resulting software system, called Cobalt [3], is freely available and has been used on a variety of computer platforms. It is in use in production at a number of sites worldwide, with plans to run it on the 100-Tflop and 500-Tflop Blue Gene/P systems Argonne is acquiring.

In this paper, we describe an enhancement we have made to Cobalt that is designed to simulate the hardware on which the system software operates. The simulator is a "normal" Cobalt component, communicating with the other Cobalt components just as they do with one another, and has been implemented by using the Cobalt software development kit.

This idea—creating a *software* component version of the hardware—has several important benefits. First, we can now develop and test our entire system software suite on a single workstation or laptop, without having to bring the production system down for testing. Second, we can vary the load on the simulated machine in a natural way, by submitting job mixes to the real queueing system component, and (provided we have accurately modeled the response of the machine to its software interface) observe its behavior as reflected in the monitoring components that interact with it. Third, since the simulator component has an interface by which we can control its behavior independently of the Cobalt components, we can inject faults of various kinds and see how our system software responds. We can both replay logs of actual system events that resulted in system software problems and create difficult test scenarios to develop and test the responses of Cobalt and other software. Moreover, all of these features are available to the users of Cobalt, who can use these facilities to improve system robustness.

In Section II we describe the larger context in which this work has taken place and summarize related work. In Section III we describe the motivation for our work, the relevant aspects of Cobalt, and the implementation of the Blue Gene simulator (note that one can easily write simulators for other machines as well, given their control inputs and outputs). In Section IV we describe our experiences with the augmented Cobalt system. In Section V, we conclude with some directions for future work.

## II. Context and Related Work

Simulation has long been an important part of the development of system software. The following are just two examples illustrating how system software simulation has proved useful in the high-performance computing community:

- The Blue Gene/L (BG/L) system was simulated on a normal Linux cluster long before actual hardware was available. IBM developed BGLsim [4], which had the ability to model Blue Gene hardware in parallel. This

allowed system software developers to rapidly and accurately prototype their software, so they were ready to run as soon as the physical hardware was available. Unfortunately, this simulator has not been maintained since the hardware has become available.

- The Maui scheduler [5] is a widely used open source scheduler that also provides a simulation mode. It was designed to enable system administrators to try out different scheduling policies on specific workloads and measure their effects on performance. It is not intended, however, to be used to test software correctness.

Our work on Cobalt offers a new approach to system software development. Cobalt [1] uses a component architecture based on the SciDAC Scalable System Software Architecture [6] to implement resource management and allocation functionality. We have extended the component model to include a layer for simulating hardware. This extension has allowed us to simulate interactions with Blue Gene control system software while running on completely different (and less exotic) hardware. Thus, we avoid one of the major difficulties facing system software developers: gaining access to scarce computer resources.

The approach of treating the system as a component can be directly applied to the concept of mock objects [7], [8]. Mock objects are test-specific objects that provide the same interfaces as one that cannot be made to work in a test environment. These are of clear use in cases, like ours, where hardware availability is limited.

The exposure of component interfaces also provides an opportunity for fault injection. Fault injection is a technique for testing software behavior under error conditions. It is typically implemented in software by adding a mechanism to internally simulate faults. While this mechanism is valuable for hardening system software against faults, it frequently requires modifying the software being tested; and such modifications unfortunately can mean that the results may not represent actual faults. In component-based systems, however, fault injection can be implemented *directly*, thereby easing the introduction of simulation faults into execution.

## III. Technical Approach

In this section, we elaborate on our motivation for encapsulating system hardware inside the Cobalt component architecture, described below. We then demonstrate how this approach results in improved development, debugging, and testing capabilities.

### A. Cobalt Architecture

Cobalt consists of a set of functional units that provide a specified interface to common operations, described in Figure 1. Each function is provided only by a single component. In the Cobalt architecture, process management, scheduling, queue management, and accounting are provided by individual components. The primary benefits of the use of a component architecture are flexibility and composability. Because components interact with one another using defined interfaces,

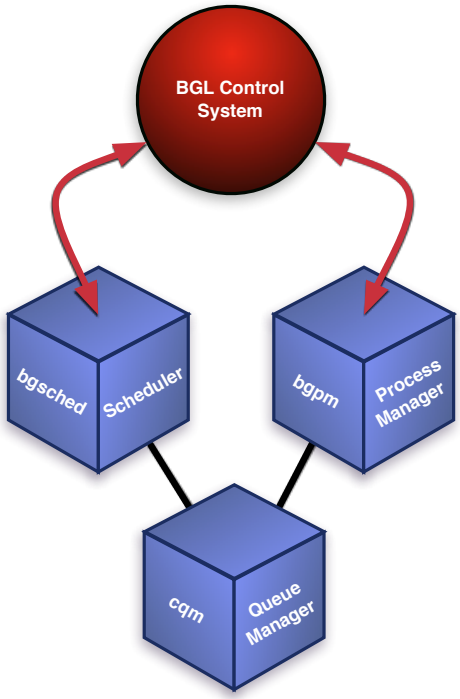


Fig. 1. Old Cobalt Architecture on Blue Gene/L

arbitrary software can consume or provide interfaces. For example, the queue manager component doesn't need to know which implementation of a process manager is used, so long as it provides the process manager interface. This approach facilitates the composition of component capabilities in new ways, enabling the creation of new system functions that work regardless of underlying component implementations.

Component interfaces consist of functions made available via authenticated XML-RPC. Cobalt ships with an SDK that provides convenient methods of implementing new component implementations and clients. It uses this SDK heavily for all components included in releases. As a result of this and the overall component approach, Cobalt is relatively small; it consists of less than 7,000 lines of Python code.

Using a component architecture has worked well for Cobalt, allowing portability from clusters to the Blue Gene/L. The porting effort for Blue Gene/L consisted of the replacement of two components: the process manager and scheduler. In both cases, this reimplementation was necessary because of the allocation requirements of BG/L partitions; nodes can be allocated only in aligned blocks of particular size. This requirement has bearing on both the scheduling and process execution; hence, these two components needed to be implemented specifically for BG/L. The other components of the system were able to be used as is and remain portable across clusters and BG/L.

### B. Motivation

System software is difficult to design, develop, and test, even under the best of circumstances. The unique character-

istics of modern high-performance computing environments compound these difficulties. The use of exotic hardware in HPC systems has become common in the past few years, ranging from special-purpose networks to integrated systems such as the IBM Blue Gene platform. As a result, establishing dedicated test environments that duplicate the hardware of production environments is impractical and cost-prohibitive. And since production HPC systems are heavily used and often governed by allocation policies, use of these resources for extended testing is generally not a viable option.

Yet the availability of reliable system software is of utmost importance on HPC systems. This fact places system software developers in a precarious position. Software robustness is critical, but building identical testing systems for system software development is not practical. Dedicated developer access to production resources competes with application groups and, hence, occurs infrequently.

This issue was our primary motivation in this work. We found that the Cobalt development process was severely hampered by this tension. This was especially apparent when BG/L was new and Cobalt was first ported to it. In several cases, bugs encountered soon after deployment could easily have been found with access to development resources. These problems resulted in additional downtime of the production system for Cobalt testing—clearly not an efficient use of the machine.

Recognizing the desirability of having the same level of flexibility with respect to the Blue Gene control system as we had become accustomed to with our higher-level components, we began to view the Blue Gene control system as a component with which other components interact. With our new hardware component approach, the ability to properly simulate hardware has greatly improved testing practices, enabling us not only to test BG/L functionality with Cobalt on other systems, but also to run automated regression testing on a regular basis. As the Cobalt code-base grows and more features are added, automated testing has become important, since manual "spot checking" proved to be inadequate in identifying potential problems.

### C. Simulation of the Blue Gene/L Control System

The use of component interfaces makes it possible to combine the functionality offered by multiple components in a variety of significant ways. Hence, it is straightforward to write sophisticated administrative tools with a minimal amount of work. Moreover, the tools are portable across diverse architectures.

The Blue Gene/L control system is the software that handles all administrative interactions with the BG/L rack, from partition allocation and setup to job execution. This system is provided by IBM and is closed source. Cobalt uses this software to execute jobs and to poll for current hardware state information.

In order to directly support the Blue Gene simulator without using different code paths from inside Cobalt, we created a new component interface for the Blue Gene control system.

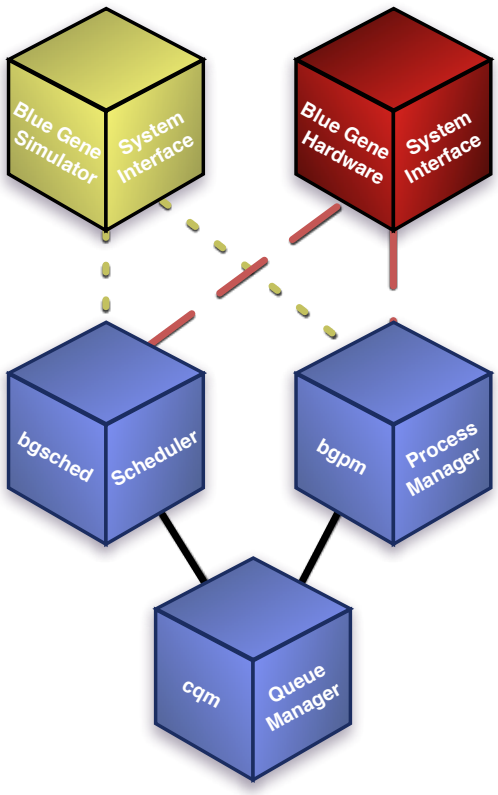


Fig. 2. Blue Gene/L Simulator Architecture

This implementation consists of two parts. First, we added a thin wrapper on top of the Blue Gene control system that calls it directly, providing the functionality we currently use. The simulator implements the same interface. This architecture is shown in Figure 2. As seen in this diagram, the process management component of Cobalt interacts directly with the simulator or the hardware abstraction layer.

The addition of this abstraction barrier allows us to run precisely the same Cobalt code against a simulator as we would on actual hardware. Since the simulator implements the same interface, Cobalt components can address either implementation transparently.

While the simulator provides an identical interface, its behavior is quite different from the standard implementation in three ways. First, no actual process execution occurs as a result of requests; the simulator produces only the external appearance of execution. Second, we added operational constraints similar to those required on Blue Gene hardware. For example, if a request attempts to use a partition that is already in use, the simulator logs an error message locally and returns an error message to the requester. Third, we added the ability for the simulator to reproduce common Blue Gene control system failures or failures of the Blue Gene hardware itself. The combination of these three features affords an unprecedented level of flexibility in development and testing.

When the Blue Gene simulator receives a request, it first checks whether running that request would result in a job failure. For example, if the new job overlaps with a currently running job, it both returns an error code to the requester and logs an error message locally. During the job execution, it simulates the real system by showing the partition in the busy state. When the job has completed, it sends a notification to the caller.

When the system is configured to simulate control system failures, it can produce results similar to errors encountered on the actual system. One such situation manifested itself early in the life of BG/L, wherein partitions would be left in an inconsistent state after an application exited. Because we are simulating the behavior through the interface, these errors and many similar ones of this class can be easily replicated. This turn-key replication of known system faults allows us to test Cobalt against these and other error cases on demand.

This use of simulation is considerably different than that used by other resource managers. In general, simulation is used to assess the performance and effectiveness of scheduling algorithms. Our goals for the Blue Gene simulator are completely different. We aim to properly exercise the Cobalt code base as accurately as possible by simulating the responses of the Blue Gene control system. Because accuracy of system and Cobalt behavior is the primary goal, simulation must occur in real time, since consumers of the interfaces provided by the simulator may perform timeouts.

In some ways this limits the utility of this approach, since it is impossible to model schedules over large periods of time. At the same time, this approach allows us to validate the interactions between the resource manager and the underlying system, even under fault conditions.

### D. Simulator Control

In order for the simulator to properly exercise Cobalt, it must be able to produce all of the observed behaviors of the Blue Gene control system. To this end, we have introduced the notion of job outcomes. Each different job outcome simulates a different result that a running job can have on the system. The most common outcome is proper execution, where a job runs for some period of time, less than the requested run time. This simulates proper job execution and submission, where the user has requested enough time, and the program runs properly.

Several other types of outcomes are also available. An overrun outcome models the case where a user has not requested enough time for their job. Many system or application failures manifest themselves in this way, for example if application performance drops or I/O issues occur. Another outcome we have implemented models control system failure, where a job fails and leaves the involved nodes in an unusable state. As more failure pathologies are observed, they can each be implemented as an outcome.

Using this set of outcomes, we can test Cobalt against each of these fault modes upon demand. By controlling the particular outcome as the simulator simulates job execution, we can ensure that Cobalt is properly hardened against all failure modes we have previously seen and recorded.

Users can control the application of outcomes to a workload by using percentages or raw failure counts. For example, the simulator can be configured to correctly execute 90% of jobs, with the other 10% using the control system failure outcome. This will result in a gradually growing pool on unusable Blue Gene nodes. With this behavior, we can ensure that Cobalt properly functions in the face of problems we have previously experienced, even if we have not actually experienced these problems recently.

## IV. Experiences

Introducing a simulator component into Cobalt has dramatically changed the way we are able to develop, debug, and test Cobalt. We can now begin running our post-installation test suite against releases *prior to* their deployment. Previously, we had to run these tests after deployment, during a maintenance window. Such a window could last up to four hours, depending on the magnitude of the testing performed. During these periods, no user jobs could run; hence, the system was being wasted from the perspective of users. The first major benefit of the simulator component was the dramatic reduction of duration of these windows. After the initiation of simulator-based testing, Cobalt maintenance windows were reduced to one hour.

Moreover, since testing no longer competes with applications for cycles, we have been able to perform more invasive and time-consuming tests. Running more comprehensive tests has resulted in increased reliability in production mode.

Based on these initial experiences with the simulator, we recognized its potential for providing a number of innovative capabilities affecting software hardening, correctness checking, and system debugging. To this end, we made several enhancements to the simulator.

First, we added consistency tests to detect improper use of hardware. For example, Blue Gene nodes can only run a single job simultaneously. If two jobs are simultaneously executed on the same nodes, the simulator will signal a job failure, and log an error message. The detection of this error mode, and several others like it, have enabled us to detect a number of latent problems that could cause failures during normal workloads.

Next, we added the ability to simulate particular faults as a part of a test workflow. Because all interactions between Cobalt and the hardware system cross a component interface, we can log requests and responses to be played back later. When system failures occur, we can use the traces collected to add these fault modes to the simulator.

The use of the component-based simulator has improved aspects of Cobalt's development, testing, and debugging. In this section, we discuss each in detail.

### A. Development

One major advantage to moving system interactions into a component interface is that developers no longer need to deal with the system interfaces directly in the main components,

such as the scheduler and process manager. Hence, developers can focus on the logic required for the component, without having to worry about system interfaces.

The simulator component also removes the need for separate code paths for development and actual execution. Separate code paths basically double the complexity of any piece of code, which in turn increases the probability of introducing bugs in the development process.

Developers are now able to run Cobalt on inexpensive and common hardware. Not only does this approach lower the cost of developing and testing new features, but it also improves the reliability of prereleases by enabling more thorough testing prior to their initial installation on exotic hardware.

### B. Correctness Testing

System software correctness is a characteristic that is hard to test rigorously. Typically, system managers perform a series of basic tests after system upgrades. Often, these are the only testing that deployed system software receives on a regular basis.

The availability of the simulator makes these tests executable at any time. Also, because the simulator implements system use consistency checks, test workloads provide accurate insight into the correctness of Cobalt's scheduling and allocation behavior. Moreover, the availability of system activity traces provides a path to easy improvement of these tests.

In addition to improving the the quality of Cobalt, these tests provide insight into the suitability and stability of new Cobalt releases prior to deployment. Because system administrators are ultimately responsible for overall system stability, this capability is key. Moreover, all of these tests can be performed without the consumption of any time on the production system.

Once we integrated use of this simulator into our daily processes, the number of bugs found during post-upgrade tests fell dramatically, because the tests run after upgrades can now be run on the simulator using a synthetic workload. Most important, system administrators can now gain confidence in new releases of Cobalt without exposing users to any risk of decreased system stability.

### C. Debugging

Debugging system software on scalable systems is the most difficult task faced by developers. Problems are frequently scale-related and may occur only occasionally. These issues can be caused by subtle hardware and software interactions that are unexpected and difficult to replicate. The addition of a component interface between the resource manager and system hardware provides a new set of capabilities for debugging these problems.

First, this interface can be monitored. Because the interface is exposed, common code in Cobalt can be used to directly capture traces of system events. This tracing functionality can be used to record standard workloads or unexpected behavior. The patterns in these activity logs can be used to understand unexpected system behavior.

The behavior of this interface in fault cases also can be mimicked by the simulator. The simulator supports custom job execution outcomes, which are patterns of activity that correspond with known system behavior. As new patterns of failure are encountered, administrators can use activity traces to prepare new execution outcomes for use in simulation. These outcomes can be used as a part of a standard test suite to ensure that Cobalt is appropriately hardened against faults of this type. As more faults are experienced and a library of "faulty" outcomes is compiled, the range and effectiveness of this simulator-based debugging grow.

Moreover, when problems are encountered, the availability of these execution traces provides the ability to execute a fault on demand and the ability to replicate the fault on inexpensive hardware. Together, these abilities dramatically reduce the time and expense needed to discover the root cause of system problems. Problems can be debugged offline while production applications continue to run in the stable environment.

These capabilities enabled us to harden Cobalt against BG/L hardware and control system failures. Achieving this same outcome manually would have required considerably more time and effort.

## V. Conclusions and Future Work

Testing has become an increasingly difficult task as the scale of systems has increased. We believe that the combination of component interfaces with simulation and hardware interaction provides a new series of tools that help developers to better cope with system complexity and scale. Using this mechanism, developers and users alike have benefited from increased portability, making Cobalt easier to test separately from the production system. The method of simulator implementation used—the addition of a *component interface*—adds an examination point where fault behavior traces can be gathered. These behavior traces have proved indispensable in replicating faults on demand. Because these traces can be captured from a single fault occurrence and executed arbitrarily on the simulator, root cause analysis and debugging are substantially eased.

We have identified several areas for future work. For example, we plan to provide faster-than-real-time simulation for a subset of Cobalt. Because of the mechanisms employed by our simulation environment, this change will be problematic; it is likely that we will have to reimplement the way we interact with the system clock and time-sensitive system calls. Also, it will only be useful for modeling scheduling performance, as opposed to system robustness.

Our simulation approach can be extended to other platforms. In each case, a software component that models hardware behavior must be implemented. Because simulator accuracy is the goal, it will likely be impossible to write a generic system simulator. By their nature, these simulators must be system-specific. We plan to implement similar simulators and hardware components for Cray XT4 systems, clusters, and SICortex systems.

Another area that can be improved is the recognition of anomalous events. Currently, these must be manually recog-

nized and then translated into a job outcome the simulator can use. It is highly likely that some fault modes may not cause an obvious failure and thus may escape notice. Detection of these faults could provide more data for use in proactive hardware maintenance or early detection of software problems.

Work is under way to interface Cobalt with the CIFTS fault-tolerant backplane [9]. This software is a common mechanism to propagate and use system and software fault data. The mechanisms described in this paper will likely be directly applicable to this new data source.

## ACKNOWLEDGMENTS

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Other support was from the DOE SciDAC program under award #DE-FG02-04ER63870, NSF MRI Grant #CNS-0420873, and the NSF sponsorship of the National Center for Atmospheric Research.

## REFERENCES

[1] E. Lusk, N. Desai, R. Bradshaw, A. Lusk, and R. Butler, "An interoperability approach to system software, tools and libraries for clusters," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 401–407, Fall 2006.

[2] N. Desai, R. Bradshaw, A. Lusk, E. Lusk, and R. Butler, "Component-based cluster systems software architecture: A case study," in *Proceedings of the 6th IEEE International Conference on Cluster Computing (CLUSTER04)*. IEEE Computer Society, 2004, pp. 319–326.

[3] N. Desai. Cobalt Web page. Argonne National Laboratory. [Online]. Available: http://trac.mcs.anl.gov/projects/cobalt

[4] L. Ceze, K. Strauss, G. Almasi, P. Bohrer, J. Brunheroto, C. Cascaval, J. Castanos, D. Lieber, X. Martorell, J. Moreira, A. Sanomiya, and E. Schenfeld, "Full circle: Simulating linux clusters on linux clusters," in *LCI 2003: Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution*, 2003. [Online]. Available: citeseer.ist.psu.edu/ceze03full.html

[5] D. B. Jackson, H. L. Jackson, and Q. Snell, "Simulation based hpc workload analysis," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 47.

[6] A. Geist. SciDAC scalable system software Web page. Oak Ridge National Laboratory. [Online]. Available: http://www.scidac.org/ScalableSystems

[7] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: Unit testing with mock objects," in *Proceedings of the eXtreme Programming and Flexible Processes in Software Engineering Conference (XP2000)*, 2000.

[8] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "jmock: supporting responsibility-based design with mock objects," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 2004, pp. 4–5.

[9] P. Beckman. CIFTS Web page. Argonne National Laboratory. [Online]. Available: http://www.mcs.anl.gov/research/cifts