LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-545691

Reliable High Performance Peta- and Exa-Scale Computing

G. Bronevetsky

April 5, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Reliable High Performance Peta- and Exa-Scale Computing

Greg Bronevetsky

I. ABSTRACT

As supercomputers become larger and more powerful, they are growing increasingly complex. This is reflected both in the exponentially increasing numbers of components in HPC systems (LLNL is currently installing the 1.6 million core Sequoia system) as well as the wide variety of software and hardware components that a typical system includes. At this scale it becomes infeasible to make each component sufficiently reliable to prevent regular faults somewhere in the system or to account for all possible cross-component interactions. The resulting faults and instability cause HPC applications to crash, perform sub-optimally or even produce erroneous results. As supercomputers continue to approach Exascale performance and full system reliability becomes prohibitively expensive, we will require novel techniques to bridge the gap between the lower reliability provided by hardware systems and users' unchanging need for consistent performance and reliable results.

Previous research on HPC system reliability has developed various techniques for tolerating and detecting various types of faults. However, these techniques have seen very limited real applicability because of our poor understanding of how real systems are affected by complex faults such as soft fault-induced bit flips or performance degradations. Prior work on such techniques has had very limited practical utility because it has generally focused on analyzing the behavior of entire software/hardware systems both during normal operation and in the face of faults. Because such behaviors are extremely complex, such studies have only produced coarse behavioral models of limited sets of software/hardware system stacks. Since this provides little insight into the many different system stacks and applications used in practice, this work has had little real-world impact. My project addresses this problem by developing a modular methodology to analyze the behavior of applications and systems during both normal and faulty operation. By synthesizing models of individual components into a whole-system behavior models my work is making it possible to automatically understand the behavior of arbitrary real-world systems to enable them to tolerate a wide range of system faults.

My project is following a multi-pronged research strategy. Section II discusses my work on modeling the behavior of existing applications and systems. Section II.A discusses resilience in the face of soft faults and Section II.B looks at techniques to tolerate performance faults. Finally Section III presents an alternative approach that studies how a system should be designed from the ground up to make resilience natural and easy.

II. MODELING APPLICATION AND SYSTEM BEHAVIOR

A. SOFT FAULTS

As the feature sizes and voltages of electronic components grow smaller and the numbers of such components grow larger, soft faults are becoming a serious threat to application correctness. Even as today's systems experience soft faults on a regular basis (a 104k node BlueGene/L system suffers from 1 uncorrected soft fault every 8 hours (1)), tomorrow's systems will grow even less reliable (2) (3).

1. MODELING ERROR PROPAGATION

Traditional approaches for analyzing application vulnerability to soft faults use fault injection to determine how faults in individual applications components affect application performance and the correctness of its results.

However, the complexity of real-world applications means that a given fault injection experiment must consist of thousands of application runs to cover all major failure modes. This is expensive for sequential applications and completely infeasible for large-scale parallel applications. In collaboration with Lide Duan, Sui Chen and Lu Peng from Louisiana State University I am developing modular techniques to analyze how soft faults affect applications.

My approach divides applications into major routines and studies the error properties of each. Errors are injected into each routine and its outputs are checked for errors to measure the routine's error vulnerability. Further, errors are injected into operation inputs and the outputs are checked to measure how much the routine amplifies or dampens errors in inputs. Figure 1 shows the vulnerability of the outputs of the Matrix-Matrix Multiplication and SVD Factorization routines in the GNU Scientific Library. The data shows the fraction of entries in the output matrixes that have an error of a given magnitude from 1e-14 to .75 (if the correct value is x and the error magnitude is m, the erroneous value is $x \cdot m$). We are working to combine error vulnerability models of individual routines in numeric libraries to predict the error vulnerability of applications that use them.

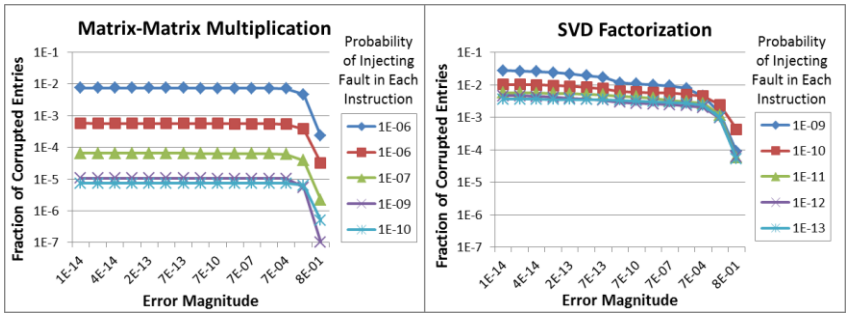


Figure 1: Fraction of Entries in Operation Outputs with Errors of a Given Magnitude

2. ALGORITHMIC RESILIENCE

Because soft faults affect the application state in subtle ways, the resulting corruptions of application results are often extremely difficult to detect. Replication can be used to detect faults by executing the application twice and comparing the results of the replicas. Correction is possible if more than two replicas are used. Although replication works for arbitrary applications, it can be expensive both in power and performance because it doubles or triples the amount of computation performed by the application. It is thus necessary to develop techniques to provide resilience to soft faults at a lower overhead, making it practical for HPC applications.

Although result checkers that execute in asymptotically less time than the original algorithm are not possible in general, such checker can be designed for specific algorithms. Since significant amounts of time spent by HPC applications is actually spent in general-purpose libraries I am working on techniques to detect and correct errors that occur in such libraries by leveraging their algorithmic properties. My specific focus is on sparse linear algebra.

Linear algebra constitutes a key class of algorithms that is used widely in scientific applications. These algorithms can be checked via the use of linear error correcting codes (4). Consider the case of matrix-matrix multiplication AB → C, where A, B and C are matrixes. This computation can be checked by multiplying both sides by a check vector c and checking if the resulting vectors are close to each other: $(x^T A)B ?= x^T C$. An error is declared if the difference is too large. Computeational errors can be corrected if the multiplication from the left as above is augmented with another multiplication from the right: $A(By) ?= Cy$ A(By) ?= Cy or if the vectors x and y are replaced with the generator matrix of a linear error correcting code. This check is both effective and efficient: the $O(n^3)$ matrix-matrix multiplication algorithm is checked via constant number of $O(n^2)$ matrix-vector multiplications.

Although very effective for dense linear algebra, this technique works poorly for sparse linear algebra, a domain that is becoming increasingly important for adaptive simulations. This is because in sparse matrixes both the matrix-matrix multiplication and matrix-vector multiplication have the same complexity: $O(n^2)$. In collaboration with Joseph Sloan and Rakesh Kumar from the University of Illinois Urbana-Champaign I am researching

algorithmic resilience techniques that specifically target the unique challenges of sparse linear algebra libraries to improve the resilience of the scientific applications that rely on it.

Our current work focuses on checking sparse matrix-vector multiplication, the backbone operation of sparse linear algebra. The state of the art is to check the operation $Ax \rightarrow y$, where A is a matrix and x and y are vectors, by multiplying both sides by a checker vector and comparing the results: $(c^T A)x ?= c^T(Ax)$. If $c = \vec{1}$ (vector of all 1s), as is common in dense linear algebra literature, performance is poor, averaging 30% overhead across a broad range of matrixes from the University of Florida Sparse Matrix Collection (5). Our work thus focuses on developing alternate vectors that balance performance and accuracy. We have developed several "sparse" techniques (6):

- Approximate Random – Random fraction of c's entries are set to 1 and remaining entries are 0.
- Approximate Clustering – Cluster A's columns by their sums; random assignment of 1s in c is biased to ensure each cluster gets the same number of 1s.
- Identity conditioning – Set c to minimize $\left|c^T A - \vec{1}\right|$, which reduces detection error due to complex matrixes A.
- Null conditioning – Set c in or near the null-space of A ($c^T A \approx \vec{0}$), which reduces detection error due to complex vectors x.

Since each technique works best in different contexts we use a Decision Tree to choose the best algorithm for a given matrix. Figure 2 shows that our approach is both more efficient and more accurate than the traditional check. The F-score is an accuracy metric that balances the rates of (i) faults that were correctly detected and (ii) fault alarms that correspond to real faults. Although the traditional check achieves F-Score of ≥ 0.9 on just 14% of matrixes, our algorithm meets this target for 72% and 88% if the detection algorithm is chosen perfectly by an oracle. Further, while the traditional algorithm has 33% average (26% median) overhead on the matrixes for which it achieves F-Score ≥ 0.9, both our algorithm and the oracle feature 16% average (12% median) overhead on the 72%/88% of matrixes on which each provides high accuracy.

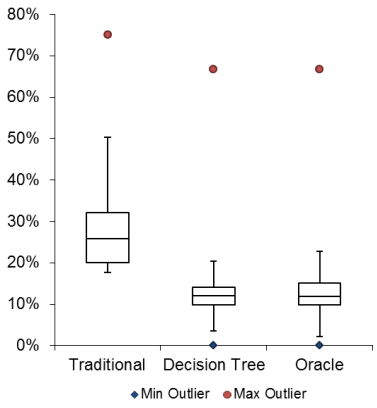


Figure 2: Overhead of Traditional and Our Sparse Checks

B. PERFORMANCE FAULTS

The large number of software and hardware components in HPC systems can result in complex interactions that significantly reduce performance. For instance, if a given chip is far away from the rack's fans, a normal computing load may cause it to overheat. If the chipset or OS respond by throttling its performance, the entire parallel application will slow down because it is stalled on data from the throttled chip. Effects like this can be very difficult to identify and resolve because they span many different components and depend on very specific combinations of events. A key portion of my work is to develop techniques to automatically analyze the behavior of systems and applications to detect, localize and tolerate such performance faults.

Most prior work in this area focuses on building models of individual systems of interest (e.g. Hadoop (7) or PVFS (8)) or workloads in a datacenter (9). While theoretically interesting, it does not offer a way forward to model arbitrary applications and systems. First, research on individual systems relies on human knowledge their design. Given the large numbers of individual systems and applications and their combinations, this methodology cannot model general systems. Conversely, research on large-scale behavior of many applications must deal with very complex behavior, which results in models that have limited utility.

In contrast, my approach looks at systems and applications at a very fine grain, such as individual loops or function calls. At this granularity software behavior is fairly simple and can be described using a few metrics such as cache miss rates and number of instructions per cycle. By combining very accurate models of all components it will be

possible to automatically and precisely model the behavior of entire systems. My research has touched on three aspects of this problem: detection and localization of faults, characterization of the type of fault that is occurring and fine-grained measurement of application behavior.

1. FAULT DETECTION AND CHARACTERIZATION

In collaboration with Ignacio Laguna and Saurabh Bagchi from Purdue University and Bronis R. de Supinski from LLNL I have developed a technique to detect bugs and system faults HPC applications by observing the effect of these phenomena on application behavior (10; 11). Our approach, called "Automata-Based Debugging for Dissimilar Parallel Tasks" (AutomaDeD) divides application execution into individual code regions separated MPI calls. For each such region we record metrics such as execution time and performance counters. Since each code region runs multiple times during a single application execution AutomaDeD builds a probability distribution of the observed values, as illustrated in Figure 3.

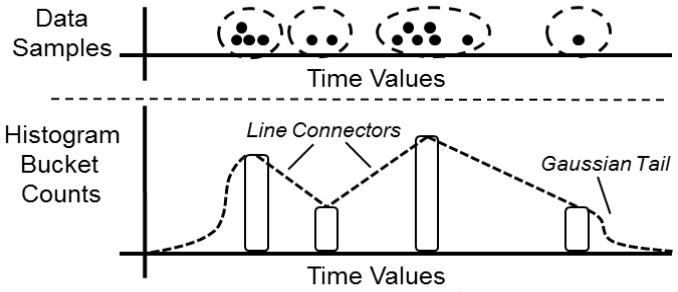


Figure 3: Building a Probability Distribution from Observations

After the model is trained on representative application runs it is applied to production runs. The elapsed time and performance counters observed while a code region executes are compared to the probability distribution of these values collected during training. If the probability of a given observation is low, it is declared to be an error and sent to the developer or system administrator for further review. Our experiments show that this approach can accurately detect faults and localize the fault time and location (process and code region. Figure 4 illustrates this by showing the degree to which one execution of the NAS BT benchmark deviates from normal behavior when another application is executed concurrently (models a bug in the MVAPICH-0.9.9 task launcher). It shows that while the interfering application is executed BT's behavior is significantly abnormal (high deviation scores) and returns to normal behavior (low deviation scores) when the interfering application terminates.

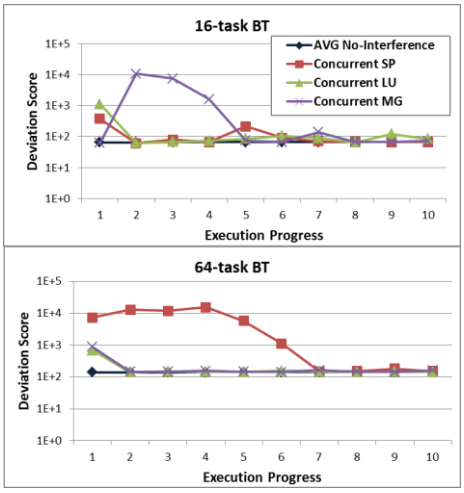


Figure 4: Deviation from Normal Behavior

The fault location and time provided by AutomaDeD can be used to detect the problem and help focus developers or automated tools on its most significant effects. Since multiple faults may have the same causes or may afflict the system in the same way it, AutomaDeD has been enhanced to provide even more useful information by collecting similar faults together. AutomaDeD does this by asking the administrator for examples of what each given fault type looks like. A fault type is described in terms of a set of sample runs of a given application that are believed to be affected by the fault. Further, the administrator provides a set of non-faulty sample runs. AutomaDeD then builds a classifier based on the execution time and performance counter observations taken for each code region during the different runs. When a production run of the application is affected by a fault the trained classifier then determines which fault type is occurring. This guides administrators and automated tools more precisely to the fault's root cause.

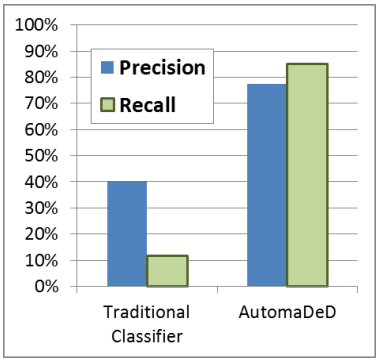


Figure 5: Accuracy of Classifying Faults

While this simple approach has intuitive appeal, in the context of system faults it works very poorly. As Figure 5 shows, traditional statistical classification algorithms are used only 40% of the fault classifications are correct (Precision) and just 10% of the real faults are classified correctly (Recall). This is because system faults have a very irregular effect on the application behavior. Figure 6 shows that during faults only a small fraction of events (code regions executions) are abnormal (points with a high "Abnormality Value"). This is because the influence of a system fault depends strongly on how the OS schedules the application and the faulty software and on the resource needs of application code regions (e.g. code with few memory accesses is not affected by memory performance faults). We solved this by developing a novel feature extractor that trains the fault classifiers using only the highly abnormal observations. This technique, which identifies structure in observations without supervision, is correct in 85% of its predictions (Precision) and real faults are correctly classified 78% of the time (Recall), as Figure 5 shows.

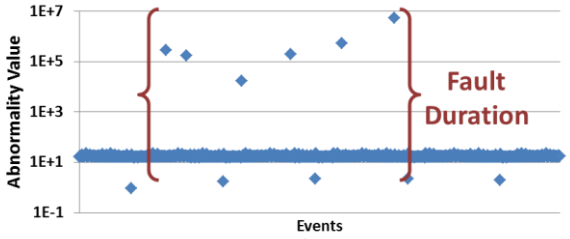


Figure 6: Abnormality of Events During a Fault

2. DIFFRACTIVE PROFILING

To create fine-grained models of application and system behavior it is necessary to measure it very precisely. For instance, the counters that measure energy use on the Intel Sandybridge architecture are only updated at a 1 millisecond granularity and other measurement tools are even coarser (12). Since most function calls and loops are significantly shorter than this, it is not possible to directly measure their energy use. Statistical profiling overcomes this by inferring the average number of events during a very small code region from many coarse measurements.

Figure 7 illustrates this approach. Observation code is executed periodically at a coarse granularity using periodic interrupts (one interrupt for many code regions). To measure each code region's execution time, the observation code examines the Program Counter (PC) at the time of the observation to identify the currently executing code region. It then builds a histogram of code region observation counts. Each region's estimated average execution time is computed by multiplying the application's overall execution time by $\frac{count\ of\ region}{\text{application total count}}$. This estimate converges to the true average as the number of observations increases. Statistical profiling has also been extended to counter-based metrics other than time.

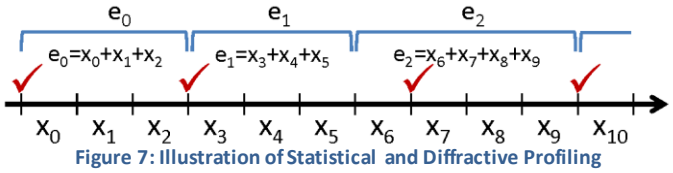


Figure 7: Illustration of Statistical and Diffractive Profiling

Unfortunately, statistical profiling requires a large number of samples to produce an accurate estimate (13), which can be expensive for coarse-grained measurements such as energy counters or for very-fine grained code regions such individual basic blocks. I am collaborating with Marc Casas-Guix from LLNL to improve the convergence speed of statistical profiling via a technique called Diffractive Profiling. This technique explicitly tracks the code regions executed between adjacent observation points, recording the number of times each region executed. The number of events between adjacent observations is modeled as the sum of events in each code region. Thus, if x_i represents the number of events that occur during code region i, and e_j is the number of events between observations j and $j+1$, the number of events that occur during the first measurement period in Figure 7 is $e_0 = x_0 + x_1 + x_2$. Taken together these observations induce the linear system in Figure 8. As the number of observation increases the system's solution converges to the average number of events in each code region. Figure 9 shows that this is more accurate than traditional statistical profiling. Across three different applications and observation periods ranging

$$\begin{bmatrix} 1 & 1 & 1 & \square & \square & \square & \square & \square & \square & \square \\ \square & \square & \square & 1 & 1 & 1 & \square & \square & \square & \square \\ \square & \square & \square & \square & \square & \square & 1 & 1 & 1 & 1 \\ & & & & & \vdots & & & & \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_9 \\ \vdots \end{bmatrix} = \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \end{bmatrix}$$

Figure 8: Linear System Solved in Diffractive Profiling

from 1µs to 100ms, the error of diffractive profiling is orders of magnitude smaller. Intuitively, this is because statistical profiling is equivalent to the solution of the linear system where only the last code region between two observations is known. This corresponds to the system where each row contains a 1 in the column of the last observation, rather than an accurate count of all code regions, which is significantly less accurate.

We are currently using diffractive profiling to measure the energy use of small code regions. Further, we are extending it to infer additional metrics about code regions that are not possible with statistical profiling, such as the standard deviation of the event counts. Finally, to explore the dependence of the number of events in each code region on the application's execution history we are also extending diffractive profiling to infer the exact number of events during every execution of each code region.

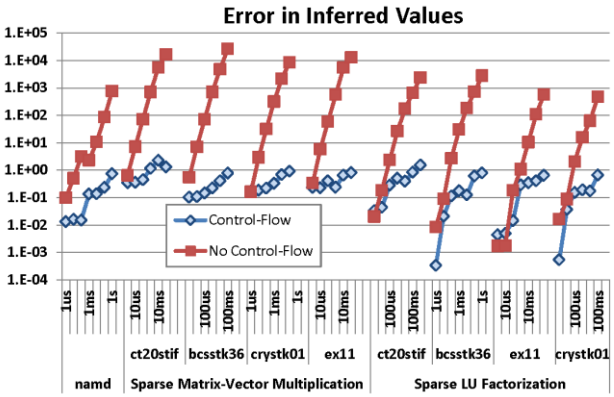


Figure 9: Diffractive Profiling is More Accurate than Statistical

III. MODEL GUIDED SYSTEM MANAGEMENT

My work focuses on enabling systems and applications to productively operate in the face of faults. While it is critically important to address the resilience needs of legacy applications, it is also important to develop new ways to design systems and applications so that they are inherently flexible and resilient. To achieve this goal it is necessary to (i) dynamically schedule application components to computing resources and to (ii) make optimal scheduling decisions by predicting their outcomes before they are made. This is difficult for traditional applications and systems for two reasons. First, they are typically organized around some fixed work management policy that assigns work without considering the connections between the algorithm's current needs or the capabilities of the currently available resources. Second, even if they are designed to be flexible, it is difficult to productively exploit this flexibility because the outcomes of various scheduling decisions are difficult to predict in practice.

To overcome these difficulties a programming model must make it easy for developers to

- Divide their overall algorithm into individual tasks
- Identify the state of each task and the information exchanged between tasks
- Describe each task's input data in a way that can be easily analyzed by a modeling algorithm

Given an application written in this fashion it becomes possible for a runtime system to adaptively schedule its tasks on available computing resources. Further, because task interactions and inputs are explicitly specified, it is significantly easier to make intelligent scheduling decisions by using statistical models to predict how each task will behave in a given execution scenario.

To explore the design of such programming models and systems I am designing a prototype system called "Minions". To an application developer a Minions application is a sequential recursive program where function calls are discrete units of computation. Each function's arguments are specified as strings and it may operate only on its own local state. Functions interact with each other via calls and returns (returned data may have arbitrary structure). The Minions runtime then schedules individual tasks (function calls) on available computing resources. This approach revolutionizes the design of dynamic systems because it naturally makes available key information about tasks and their relationships. This makes easy to use statistical modeling techniques to predict the behavior of tasks and make scheduling decisions. For instance, task arguments are strings because such simple data structures are easily handled by state of the art statistical modeling techniques. Further, the constraints on the

state of Minions tasks are designed to enable concurrent execution of tasks on arbitrary computing resources. Finally, explicit task boundaries make it easier to measure task performance and behaviors.

The design of Minions is fundamentally different from prior approaches. Previous work attempts to extract from legacy applications information that is needed to create statistical models. In contrast, this information is available in Minions directly, with no special effort. Further, because it includes function arguments, creating workload-sensitive models is natural. In contrast, such models are extremely difficult for prior approaches, which have no direct way to get a description of the application's workload. The key advantage of Minions is that while tools that work with legacy applications must perform very complex analyses schedule and model them, Minions makes these capabilities natural. This makes it possible to fully explore model-guided applications and system management techniques in isolation from other concerns, guiding the designs of the next generation of application and system designs.

Figure 10 illustrates the Minions design, which is partitioned to enable components focused on system modeling, measurement, runtime optimizations and task scheduling to interact easily and to to combine various component implementations in a single runtime. Minions will be given a set of tasks (names and arguments known), compute resources (performance capabilities known) and runtime actions that it can apply to the tasks (e.g. executing a task on a given architecture or co-locating a set of tasks on the same core or node). Minions will use the Experiment Selector module to choose a subset of task/action combinations that are representative of the others. These will be given high priority to ensure they are executed before the others. For each run it will select one or more measurements to be made of the task's behavior and resource needs. These will range from simple metrics such as CPU utilization to complex inferences that may require multiple task executions (e.g. statistical sampling).

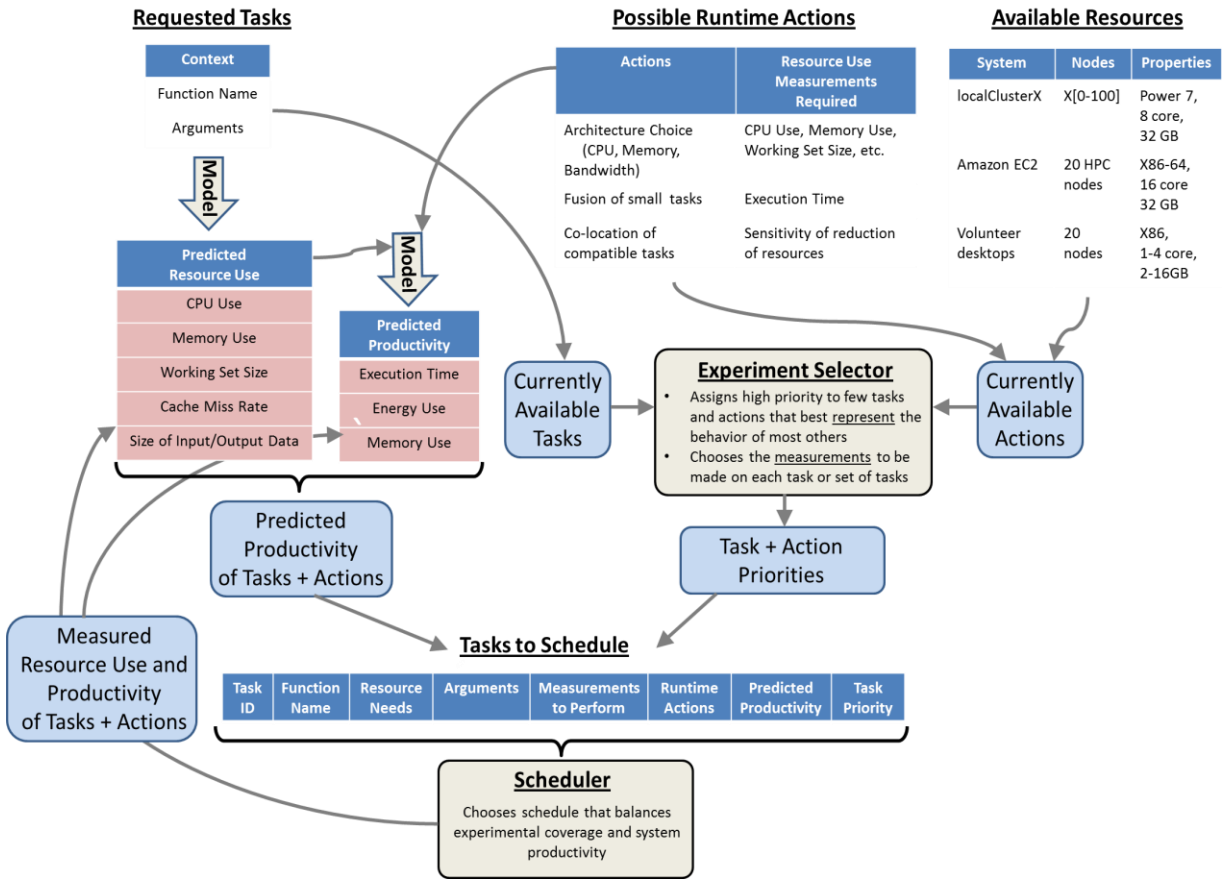


Figure 10: Workflow of Model-Guided Work Management

The resulting set of tasks, with their associated priorities and measurements will be provided to the Scheduler, which will assign them to available compute resources. As tasks complete, their measurements will be fed back to the runtime and used to build (i) Models that associate task arguments to their behavioral properties and (ii) Models that map task properties to the productivity (e.g. execution time, energy use, cost) of a given runtime action on the task. By predicting the outcomes of various runtime actions on tasks in the queue, these models will enable the scheduler to intelligently choose tasks and runtime actions to maximize system productivity.

The ultimate vision of this work is to develop a system that provides highly efficient, adaptable and resilient performance to real-world HPC applications. By demonstrating the benefits of a model-guided dynamic system and application design, I hope to influence the design of other programming models and runtime systems to bring the benefits of this approach to the entire HPC application community.

IV. SUMMARY

The growing power of HPC systems comes at the price of increasingly lower reliability and higher system complexity. To continue DoE leadership in HPC into the Peta- and Exa-scale eras we must make systems resilient to these phenomena. My work is developing a detailed understanding of the effects of faults on real HPC systems, and enabling the design of application and systems that can tolerate them. **My ultimate goal is to help create a new generation of highly-productive and cost-efficient HPC systems to enable novel DOE science applications.**

V. BIBLIOGRAPHY

1. *Extending Stability Beyond CPU Millennium: a Micronscale Atomistic Simulation of Kelvin-Helmholtz Instability.* **J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd and J. A. Gunnels.** 2007. ACM/IEEE Supercomputing Conference.

2. **ITRS.** *International Technology Roadmap for Semiconductors.* 2010.

3. **Peter Kkogge, editor & study lead.** *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems.* s.l. : DARPA IPTO, 2008.

4. *Algorithm-Based Fault Terance for Matrix Operations.* **Abraham, Kuang-Hua Huang and J.A.** 6, 1984, IEEE Transactions on Computers, Vol. 33.

5. *University of Florida Sparse Matrix Collection.* **Davis, Timothy A.** 1994, NA Digest, Vol. 92.

6. *Algorithmic Approaches to Low Overhead Fault.* **Joseph Sloan, Rakesh Kumar and Greg Bronevetsky.** 2012. International Conference on Dependable Systems and Networks (DSN).

7. *Large-Scale System Problem Detection by Mining Console Logs.* **Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan.** 2009. ACM Symposium on Operating Systems Principles (SOSP).

8. *Black-Box Diagnosis in Parallel File Systems.* **Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi and Priya Narasimhan.** 2010. USENIX Conference on File and Storage Technologies (FAST).

9. *Analysis of Application Heartbeats: Learning Structural and Temporal Features in Time Series Data for Identification of Performance Problems.* **Reed, Emma S. Buneci and Daniel A.** 2008. ACM/IEEE Supercomputing Conference.

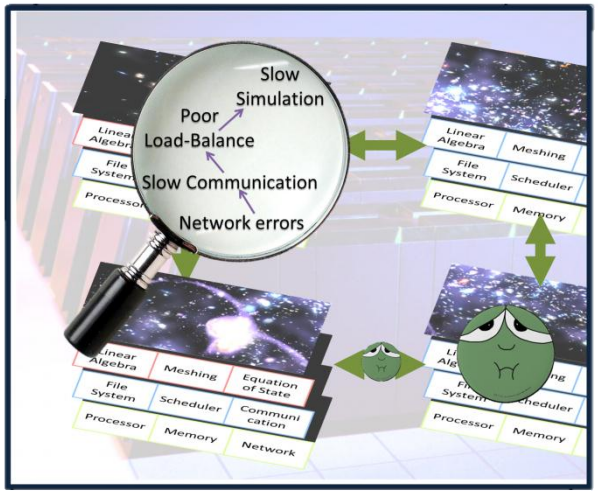
10. *AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks.* **Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi and Bronis R. de Supinski.** 2010. International Conference on Dependable Systems and Networks (DSN).

11. *Automatic Fault Characterization via Abnormality-Enhanced Classification.* **Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi and Bronis R. de Supinski.** 2012. International Conference on Dependable Systems and Networks (DSN).

12. *Accurate Energy Attribution and Accounting for Multi-core Systems.* **Sebi Ryffel, Thanos Stathopoulos, Dustin McIntire, William J Kaiser and Lothar Thiele.** 2009. USENIX.

13. *Towards a Methodology for Deliberate Sample-Based Statistical Performance Analysis.* **Hollingsworth, Geoffrey Stoker and Jeffrey K.** 2011. Workshop on High-Level Parallel Programming Models and Supportive Environments.

Reliable High Performance Peta- and Exa-Scale Computing



Novel Ideas

As the size and complexity of HPC systems grows they become less reliable and perform more erratically. We are developing a modular methodology to analyze effect of faults on applications and improve application resilience:

- Resilience for numerical libraries
- Modular error propagation analysis
- Detection, localization and characterization of performance faults
- Very fine-grain performance analysis
- Model-guided system management

Impact and Champions

Failures and performance degradations significantly reduce the productivity of HPC systems by reducing confidence in simulation results and increasing the time and cost to producing each scientific result. This project will enable applications to operate productively on complex, unreliable HPC systems by modeling the behavior of application components during normal and faulty operation and identifying their reliability needs. Further, it will develop novel algorithms to improve application resilience.

PI: Greg Bronevetsky, LLNL

Milestones/Dates/Status

	Scheduled	Actual
Detection and localization of performance faults	11/10	11/10
Characterization of performance faults	5/11	5/11
Soft fault detection in linear algebra libraries	12/11	12/11
Model-guided system prototype	3/12	3/12
Fault detection and characterization in generic applications	9/12	
Generic framework for hybrid algorithmic- and resilience-based fault tolerance	5/13	
Resilient model-guided system deployment	5/14	

Reliable High Performance Peta- and Exa-Scale Computing

Greg Bronevetsky

Supercomputers are growing increasingly complex, as reflected both in the exponentially increasing numbers of hardware components (LLNL is currently installing the 1.6 million core Sequoia system) as well as the variety of software and hardware in a typical system. At this scale component failures and degenerate cross-component interactions become a regular occurrence. The resulting faults and instability cause HPC applications to crash, perform sub-optimally or even produce erroneous results. Since full system reliability will become prohibitively expensive for Exascale systems, we will require novel techniques to bridge the gap between the lower reliability provided by hardware systems and users' unchanging need for consistent performance and reliable results.

Previous resilience research has developed fault detection and tolerance techniques. However, they have seen very limited real use because (i) the effects of complex faults on real systems are poorly understood (e.g. soft faults or performance degradations) and (ii) it is difficult to implement resilience for a real system in a modular fashion. With little idea of what needs protection and few tools to implement whole-system resilience from building blocks, developers have little opportunity to make applications resilient. My work addresses this problem via a modular methodology for analyzing the behavior of applications and systems during normal and faulty operation and modular system design techniques that will improve application resilience and adaptability.

MODELING APPLICATION AND SYSTEM BEHAVIOR

MODELING ERROR PROPAGATION

I am working to model the propagation of errors through applications by dividing them into individual routines and using fault injection to observe how their outputs are affected by errors injected during their execution or into their inputs. Figure 1 shows the vulnerability of the outputs of the Matrix-Matrix Multiplication and SVD Factorization routines in the GNU Scientific Library. The data shows the fraction of entries in the output matrixes that have an error of a given magnitude from 1e-14 to .75 (if the correct value is x and the error magnitude is m, the erroneous value is $x \cdot m$). We are working to combine error vulnerability models of individual routines in numeric libraries to predict the error vulnerability of applications that use them.

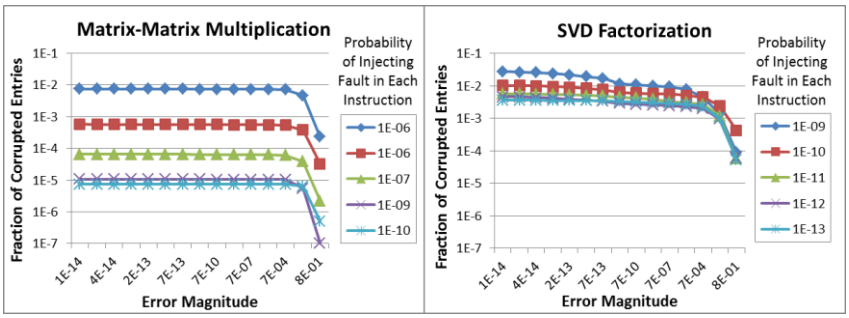


Figure 2: Fraction of Entries in Operation Outputs with Errors of a Given Magnitude

ALGORITHMIC RESILIENCE

There has been extensive work on detecting and correcting errors in dense linear algebra applications via checkers that are asymptotically faster than the original algorithms. Unfortunately, these techniques are expensive for sparse linear algebra because in this context their cost is asymptotically the same as the original algorithm. I am developing error checks for sparse matrix-vector multiplication $Ax \rightarrow y$, the backbone of sparse linear algebra, by using the identity $(c^T A)x ? = c^T(Ax)$. The traditional check, which uses $c = \vec{1}$ (vector of all 1s), has 30% average overhead and frequently misses errors. We have

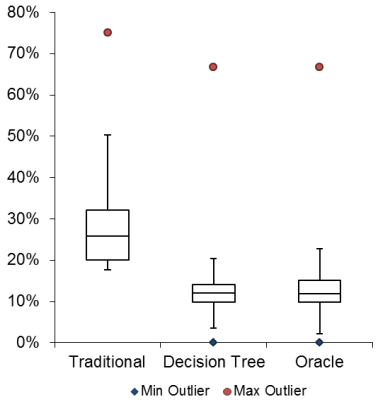


Figure 1: Overhead of Traditional and Our Sparse Checks

combined several techniques for choosing c that include sparsely sampling 1s and 0s or that condition c to minimize $|c^T A - \vec{1}|$ or to be near A's null-space. As Figure 2 shows, by choosing the right technique for each matrix detection overhead shrinks to just 10%, and 12% if the choice is made using a decision tree. Further, the technique is much more stable than the traditional check, with consistently high detection accuracy across many error rates.

MODEL GUIDED SYSTEM MANAGEMENT

My work focuses on enabling systems and applications to productively operate in the face of faults. In addition to addressing resilience in legacy applications, it is also important to develop new ways to design systems and applications so that they are inherently flexible and resilient. To achieve this goal it is necessary to (i) dynamically schedule application components to computing resources and to (ii) make optimal scheduling decisions by predicting their outcomes before they are made. Although critical for Exascale, these capabilities are not common in today's HPC applications. I am working on a new dynamic system that both demonstrates how to design applications to be dynamically optimizable and is itself a proxy application that can be used to evaluate different modeling, monitoring and optimization approaches. Figure 3 illustrates its overall design.

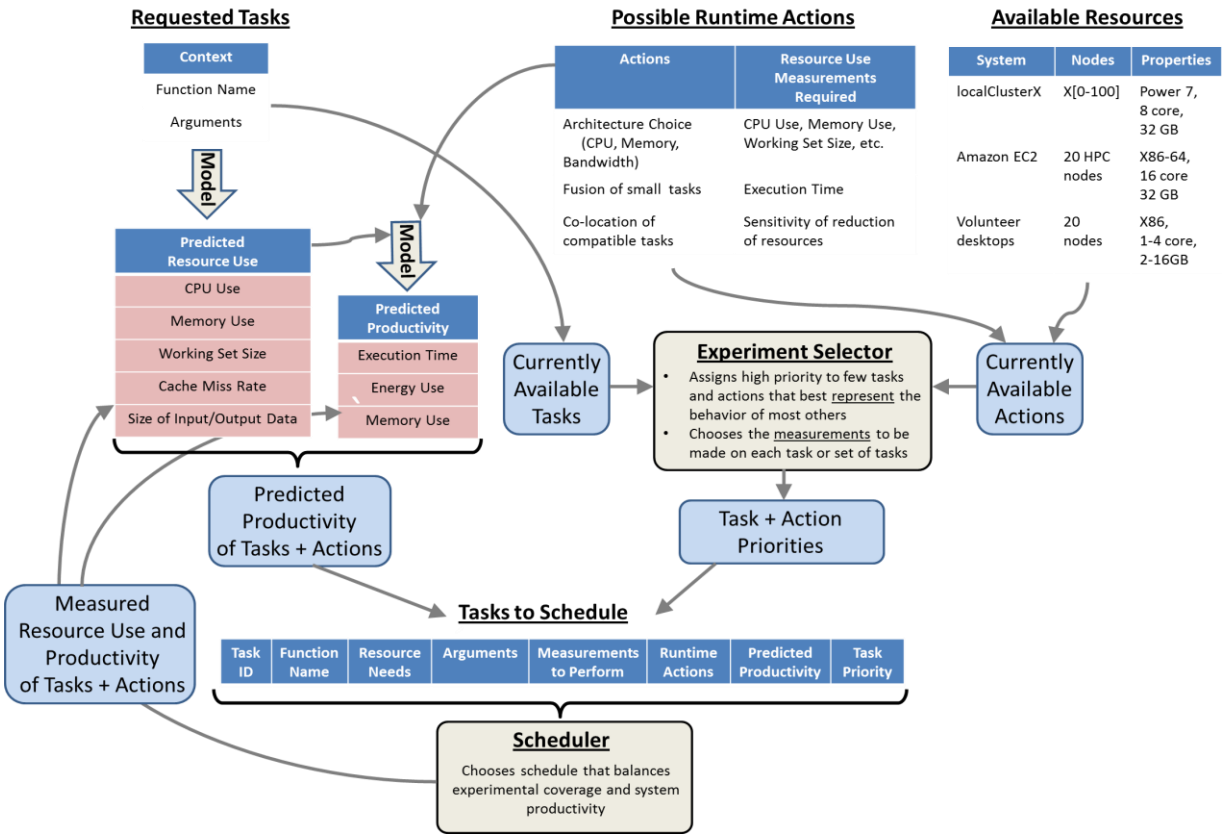


Figure 3: Workflow of Model-Guided Work Management

Work is decomposed into many individual tasks, the arguments of which are explicitly identified by the developer. The developer also identifies one or more runtime configuration decisions that can be made to optimize each task's performance (e.g. setting voltage or task co-location). The modeling system uses empirical observations of task performance under various configurations to predict the optimal configuration of each subsequent task. The system is organized in a modular fashion into a (i) task-specific statistical model, (ii) a model focused on hardware behavior, (iii) an experiment selector that prioritizes the execution of representative combinations of tasks, configurations and measurements, and (iv) a scheduler that optimizes task execution on available resources.