LARGE-SCALE SYSTEM RELIABILITY

Sarah Michalak (LANL), Dennis Abts (Google), Nathan DeBardeleben (LANL), Greg Bronevetsky (LLNL), John Daly (DoD), Armando Fox (Berkeley), Jon Stearley (SNL), David Walker (Princeton), Ravi Iyer (UIUC), Will Jones (Coastal Carolina Univ.)

Large-scale systems are at a crossroads. As the feature sizes of electronics grow smaller and large systems grow to thousands of nodes and millions of cores, the probability that some system component will fail grows steadily. Today's systems have already begun to suffer from this trend, with many systems featuring more than one failure per day, including both fail-stop failures and data corruptions (ex: a 100,000-node BlueGene/L supercomputer suffers from one data corruption every 3-4 hours). This has led many applications and systems to explicitly incorporate fault tolerance features into their code, trading off the cost of fault tolerance mechanisms such as checkpointing against the cost of failures. A major example is the decision by major system vendors to remove local disks from compute nodes, which improves hardware reliability by removing the least reliable system component, while making checkpointing significantly more expensive (~30 min on a typical large system).

Looking into the future, as large-scale systems grow even more complex and include increasing component counts, we anticipate a crisis of reliability where the underlying hardware will become too unreliable to provide useful service. As such, managing reliability at the level of individual components, entire systems and applications must become a key pillar of large-scale system research and development. Since traditional reliability research has primarily focused on preserving the abstraction of perfect reliability in low-fault environments, this new problem of operating in the presence of many faults presents a new generation of system reliability research challenges that requires a significant new research investment.

At its core a given fault or fault tolerance strategy can affect an application in one of three ways:

- Time: the application may run more slowly (e.g. the fault itself or the algorithm to tolerate the fault caused a degradation in system performance)
- Energy: the application will use more energy to produce its result (e.g. checkpointing requires some work to be re-executed and modular redundancy runs an identical application on multiple processors)
- Correctness: the application will produce results that have a lower accuracy (e.g. a memory bit-flip may affect a numerical algorithm's convergence properties)

As such, as part of an integrated fault tolerance strategy we must (i) provide accurate characterization of how an application or system performs with respect to these metrics and (ii) to develop ways to both improve this performance and allow applications and systems to trade off one aspect, such as correctness, for another, such as completion time.

Challenge 1: Understand and control the complex effects of faults on systems

Large-scale systems consist of thousands or even millions of software and hardware components that interact in complex ways. Faults in one component can propagate through other components in many different ways to manifest themselves as a variety of application and system errors. In this context the effects of a given fault may

be different depending on the context and one part of the same application or system may be more or less vulnerable to faults than another. As such, system reliability solutions that treat all faults as equally dangerous and all components as equally vulnerable will be highly sub-optimal, giving up significant performance and functionality. Further, without the knowledge of how individual system components are affected by faults and how those faults travel through the system it becomes very difficult to handle them or identify their root causes. Unfortunately, while today there exist techniques to understand the fault properties of individual software and hardware components, there exists little work on understanding the effect of faults on entire systems. As we work towards developing future generations of cost-effective and productive large-scale systems it is thus critical to overcome this limitation.

A simple example of how our lack of understanding hinders our ability to build reliable systems is the effect of bit flips on applications running on the BlueGene/L supercomputer, where a 100,000 node machine suffers from one L1 cache bit flip every 3-4 hours. Although these flips are detected using a parity code, they can only be corrected by running the L1 cache in write-through mode, which guarantees that there is a valid copy of the cache line in L2 cache. Without knowing anything about the vulnerability of applications to bit flips, the best strategy is to assume that every single bit-flip is fatal to the application. As such, BlueGene/L designers chose to handle all such faults at the hardware and kernel level by providing users with two options: (i) run using L1 write-through mode, which can reduce performance by as much as 50% or (ii) abort the job when a bit-flip is detected, which requires expensive checkpoint/restart operations (on this machine a full-system checkpoint takes ~30 minutes). However, these costs can be significantly reduced if developers understand their application's true vulnerability properties. For example, Monte-Carlo simulations are insensitive to rare data corruptions, meaning that they require no specialized reaction to bit flips. Other applications may be sensitive but may develop efficient strategies for detecting and correcting bit flips if they are informed of them. This was the case for the ddcMD code, where performance was improved by 17% via the use of light-weight checkpointing to correct detected bit flips with no application aborts. Overall, the best strategy to employ on a given physical fault varies widely depending on the fault vulnerability of other system components and requires the cooperation of multiple components, each of which performs the fault detection/correction task that it is best suited for.

This means that cost-effective reliable computing requires a detailed understanding of the effects of component faults on entire systems and cross-component cooperation.

Sub-Goal 1a: Effect of faults on systems - Develop a detailed understanding of how faults propagate through large systems and manifest themselves as errors in other components.

Although component-level reliability has been studied extensively in the context of individual devices, we do not currently have a good understanding of how failures of such components affect other portions of the system or the applications that run on top of it. The lack of such an understanding makes it difficult to predict the portions of systems and applications most vulnerable to failures or to identify the failed system components based on the effects of these failures on the system as a whole.

This is critically important because most work in "classical" fault tolerance focuses on *completely masking* faults from higher layers. As companies such as Google have discovered, at extreme scale this approach is not feasible; we must recognize that *not every fault has equally severe consequences*, and focus our efforts on how to *selectively* mask the most dire faults while potentially allowing higher layers of the system to deal explicitly with other faults. To do this, we will need to develop composable models of system vulnerability to failures that could predict how errors travel through and affect system components and applications. These models would allow us to:

- Design applications and systems to detect and tolerate the most likely or dangerous types of failures, including the degree of reliability truly needed from various components
- Develop tools to quickly identify faulty components, enabling efficient system management
- Predict many types of failure ahead of time to allow proactive fault tolerance strategies
- Apply different masking and recovery strategies to different faults depending on their overall impact on job completion.

Sub-Goal 1b: Cross-Layer Reliability - Develop tools and frameworks to enable individual components to participate in a global fault reliability strategy by defining ways for them to interact and share reliability information.

This problem of sharing the responsibility of system reliability across system layers and components can be approached in various ways. One example would be a series of cross-layer reliability interfaces where lower level components export information about their internal faults or performance deviations and higher level interfaces use this information to either tolerate such problems or propagate them to higher system levels. Another approach would be a system-wide monitor that would aggregate information across layers and use statistical analysis to coordinate maintenance actions or predict future failures. Such tools are expected to improve system performance and maintainability as well as significantly reduce the time to achieve full system stability (currently around one year) by identifying marginal components and making it possible to fully utilize the system even before all low-level reliability issues have been addressed.

This infrastructure will have direct implications on the other challenge problems. If the system can export information about the correctness of its results, it is possible to develop novel algorithms that take advantage of such data to improve their reliability or performance. For example, an algorithm that was informed that a given block of memory is unreliable may choose to use it as a cache rather than as primary working space.

Challenge 2: Enable users to reduce the vulnerability of systems and applications to faults

In implementing reliable systems and applications developers are faced with the daunting challenge of identifying and handling a wide variety of system failures and their many possible manifestations. Although an improved understanding of the effect of faults on systems is a critical part of this process, it still leaves developers with a very difficult development task. First, many conventional algorithms are brittle in the face of failures and must be replaced by new variants that are resilient to failures. For example, physical simulations are frequently very tightly coupled, making them very sensitive to any load imbalance or timing variation anywhere in the system (e.g. variation due to performance degradation). In particular the POP ocean model slows down by 30%-1600% in the presence of such variation. Second, many programming models currently available do not help developers to create reliable applications, either because they provide features that make applications more brittle or because they are missing features that may simplify the development of reliable applications. As an example of the former, the common shared memory programming model reduces application reliability by encouraging frequent fine-grained accesses to any byte in memory, which makes it expensive to track and detect errors and encourages applications to become tightly coupled and therefore sensitive to timing variation. An example of the latter is the fact that conventional programming models are missing features such as robust and efficient checkpoint/restart capability and integrated invariant specification and checking.

As reliability becomes an even more critical component of application development, there is a pressing a need for novel reliable algorithms and programming models to improve developers' ability to write reliable applications and systems.

When considering the development of reliable applications, an application's fault vulnerability can be expressed in terms of three components:

- Temporal Sensitivity – dependence by one part of the application on the amount of time taken by another part
- Interaction Locality – the amount and granularity of interactions between application threads and components
- Correctness Sensitivity – the sensitivity of the algorithm to errors in the computation

All three types of locality are individually important to the algorithm's ability to tolerate failures. Applications with good temporal sensitivity will be minimally affected by failures that merely result in timing variation and will be easy to combine with techniques like checkpoint/restart that convert failures into timing variations. In contrast, applications with poor temporal sensitivity will suffer from severe performance degradations due to failures. Interaction locality is critical for limiting the amount and granularity of data exchanged by system components. Such constraints slow the spread of data corruptions through the application and enable designers to use more expensive and accurate fault detection/correction techniques at component interaction sites. Finally, good correctness sensitivity reduces the probability that a given failure will corrupt the final application output and reduces the severity of such corruptions.

As a specific example of how algorithmic research can play a role, a natural-language processing application might rely on an algorithm that, instead of updating a single centralized model as new training data arrives, is able to apply independent updates to separate models and periodically merge or synchronize the models to avoid model drift. Understanding the rate of model drift, or proving bounds on it based on the interval between model synchronizations, are problems that can be addressed by new algorithmic research.

From the programming systems point of view, certain languages and programming models either encourage or discourage various types of locality. For example, since Map-Reduce and Linda decouple data producers and consumers, they encourage the development of applications with good temporal sensitivity. Similarly, message-passing and token-passing dataflow encourage spatial locality by forcing users to explicitly identify all their communication channels. In contrast, since shared memory allows applications to interact at the granularity of individual memory addresses and synchronize using fine-grained primitives, it encourages poor interaction locality and temporal sensitivity. Finally, while numerical analysis theory provides numerical applications with ways to reduce their sensitivity to data corruptions, the same is not available for many other application domains such as databases.

Challenge 3: Measure the reliability and fault vulnerability aspects of real systems

While it is clearly important to enable developers to create reliable systems, this will be of limited use unless users can verify that a given system actually has the reliability properties they need. For example, if a user is considering purchasing hardware that has a certain mean time between failures and some set of common failure modes, they need to find a software stack that provides the highest level of productivity (balance of performance, cost and reliability) in that environment. Further, once a system is installed, users need mechanisms to empirically measure the system's response to faults to make it possible to effectively manage the system and tune its reliability and performance. This capability is of special importance in the context leading edge supercomputing systems that take as much as a year to bring up to "production" status.

The use of reliable systems will require tools to empirically and independently measure system reliability.

To this end we will need to focus our efforts on developing suites of benchmark applications, reliability metrics and physical test environments that can evaluate the reliability of systems and applications with respect to the metrics of Time, Energy and Correctness. Such criteria will make it possible to make general statements about to fitness of individual systems, system components and applications for specific tasks and specific physical environments. It would also allow the industry to track its progress towards providing highly productive operation.