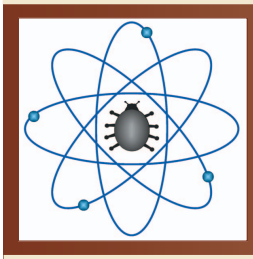
Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate

Michael Grottke and Kishor S. Trivedi
Duke University



Combatting vastly different types of software bugs requires different strategies.

During the Gulf War, 28 US Army reservists were killed and 97 were injured on 25 February 1991 when the Patriot missile-defense system at their barracks in Dhahran, Saudi Arabia, failed to intercept an incoming Scud missile.

This well-known incident occurred due to a software fault, or bug, in the system's weapons-control computer. This event led to a *system failure*—that is, a deviation of the actual system behavior from correct service. It was also a case in which engineers employed multiple techniques to fight the software bug.

BOHRBUGS: REMOVE

Finding and removing software faults is the classic strategy for dealing with them. Fixing bugs in the operational phase is considerably more expensive than doing so in the development or testing phase. Published literature reports cost-escalation factors ranging from 5:1 to 100:1 (B.W. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," *Computer*, Jan. 2001, pp. 135-137).

Therefore, engineers expend much effort on detecting and removing bugs during software development via both dynamic software tests and static techniques like code reviews and walkthroughs. Systematically conducted unit and system tests play an important role in revealing faults that lead to failures during software execution.

However, diagnosing and isolating the underlying fault responsible for an observed failure becomes difficult if the failure can't be reproduced. Software testing is, therefore, mainly suitable for dealing with faults that consistently manifest under well-defined conditions. Testers sometimes refer to such faults as Bohrbugs, an allusion to Niels Bohr's simple and intelligible atomic model (J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. 5th Symp. Reliability in Distributed Systems*, 1986, pp. 3-12).

MANDELBUGS: RETRY, REPLICATE

However, testers do encounter failures they can't reproduce. Under seemingly exact conditions, the actions that a test case specifies can sometimes, but not always, lead to a failure.

Fault activation

To explain this phenomenon, it's useful to take a closer look at how the static fault in the software is connected to the dynamic failure occurrence.

Usually, activating a fault by executing the part of the software where it's located doesn't immediately cause a failure. Rather, it produces an internal condition in the system that deviates from the correct internal condition—referred to as an *error*—even though the user might not perceive this discrepancy. An error can develop into further errors before a failure finally occurs. This functional chain between errors and failure is called *error propagation*.

For example, a fault in an algorithm's implementation can lead to an erroneous computation for specific values of a program variable—a case of fault activation causing an error. The software can use this incorrect result internally for further calculations, in which case the error propagation leads to additional errors. A failure occurs only when the system uses one of these incorrect calculations in a way that influences a perceivable system behavior, or when error propagation causes a failure occurrence.

Based on the relationships between faults, errors, and failures, we can offer two explanations as to why software may behave differently under apparently identical conditions. First, if there's a long delay between the fault activation and the final failure occurrence—for example, traversing several different error states in the error propagation—then it's difficult to identify the user actions that actually activated the fault and caused the failure. Simply repeating the steps carried out a short time before the failure occurrence might not lead to its reproduction.

Second, other elements of the software system—such as the operating system, other applications, or the hardware—can influence a fault's behavior in a specific application. We refer to the set of these elements as the application's *system-internal environment*. For example, inadequate synchronization in multithreaded software can give rise to race conditions, in which the program behavior depends on the relative timing of the threads the operating system schedules. Since a failure only occurs if the operating system schedules the threads in a specific order that the programmers didn't foresee, troubleshooters find it difficult to reproduce such failures and isolate the underlying faults.

A fault can cause the software to exhibit a chaotic and even nondeterministic behavior with respect to the occurrence and nonoccurrence of failures if its activation or error propagation are complex in at least one of these two ways. Software engineers sometimes refer to faults with this property as Mandelbugs, an allusion to Benoît Mandelbrot, a leading researcher in fractal geometry (E.S. Raymond, *The New Hacker's Dictionary*, MIT Press, 1991).

Sometimes, the literature also calls these software faults Heisenbugs. However, Bruce Lindsay, who invented the term, derived it from Heisenberg's Uncertainty Principle, referring to faults that change their behavior when probed or isolated (M. Winslett, "Bruce Lindsay Speaks Out," *ACM SIGMOD Record*, June 2005, pp. 71-79). Since the system-internal environment induces the change in behavior, Lindsay's Heisenbugs are actually a type of Mandelbug.

Unique problems and possibilities

A problem with Mandelbugs is the high probability that a developer won't detect them during testing. Even if a programmer or tester were to execute the parts of the code containing Mandelbugs, they will only cause failures if they meet the complicated conditions related to the system-internal environment. Mandelbugs can therefore go unnoticed until long after the software's release.

Moreover, even if a Mandelbug should cause test cases to fail, reproducing the failures is difficult, as is isolating the underlying Mandelbug. As a consequence, removing the Mandelbug before the software's release might not be possible. Therefore, it's plausible to assume that the majority of the faults remaining in a well-tested piece of software are Mandelbugs. However, the published data is inconclusive, indicating that Mandelbugs account for between 15 and 80 percent of all software faults detected after release (S. Chandra and P.M. Chen, "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2000, pp. 97-106; I. Lee and R.K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans. Software Engineering*, May 1995, pp. 455-467).

On the other hand, Mandelbugs' seemingly nondeterministic behavior makes it possible to deal with them in ways infeasible for dealing with Bohrbugs.

First, when a Mandelbug has caused a failure, a simple retry of the failed action can result in success. This explains the phenomenon that restarting an application or rebooting the system after a crash often solves the problem. We can improve the approach by combining it with checkpointing, a technique that involves regularly saving a snapshot of the application state in stable storage. After a failure, we can restart the application to the latest available snapshot.

Second, adopting replication, that is, using redundant resources—an approach from the hardware reliability field—is possible. Since natural phenomena like physical deterioration cause most hardware faults, failover to an identical component upon failure of the first usually won't lead to a second failure. However, software faults are human-made design errors that lurk in the software code.

Different installations of the same operating system running the same applications should contain the same faults. Scholars, therefore, have questioned whether software replication can offer benefits similar to those for replicating hardware. After all, when executing the commands on a second installation of an identical piece of software, a user would encounter the same fault that led to failure the first time, causing another failure.

Of course, this reasoning implicitly assumes that all faults are Bohrbugs. However, since many software faults are in fact Mandelbugs not manifesting consistently under well-defined conditions, software replication has indeed proven useful. It plays a key role, for example, in the Object Management Group's fault-tolerant CORBA standard.

AGING-RELATED BUGS: REJUVENATE

In recent years, researchers have studied yet another approach to handling software faults. Anecdotal evidence suggests that restarting a program or rebooting a computer *before* the user experiences a failure can be beneficial for avoiding future failure occurrences.

Such proactive measures only make sense if the failure-occurrence rate increases with the runtime; if the rate were constant, then restarting or rebooting wouldn't affect the risk of experiencing a failure. In fact, software systems running continuously for a long time tend to show a degraded performance and an increased failure-occurrence rate, a phenomenon called *software aging*. Consequently, the preventive counter techniques are referred

to as *software rejuvenation* (Y. Huang et al., "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1995, pp. 381-390).

Intuitively, the software-aging phenomenon appears impossible as we are executing software without introducing any changes into its code. Why would the failure-occurrence rate change over time if we don't modify the software code? There are two possible solutions to this puzzle.

First, the aging-related bugs can cause errors to accumulate over time. These error conditions can accrue either within the running application, such as round-off errors in program variables, or in the system-internal environment, such as unreleased physical memory due to memory leaks in the application. In either case, the error conditions don't lead to failures immediately. Otherwise, there would be no aging. The failures occur with a delay.

Second, the total time that the system runs continuously can influence an aging-related bug's activation rate. We can consider this runtime an aspect of the system's internal environment.

Obviously, both types of aging-related bugs are Mandelbugs.

Example: Patriot system

The software fault in the Patriot missile-defense system responsible for the Scud incident in Dhahran was the second type of aging-related bug. To project a target's trajectory, the weapons-control computer required its velocity and the time as real values. However, the system kept time internally as an integer, counting tenths of seconds and storing them in a 24-bit register. The necessary conversion into a real value caused imprecisions in the calculated range where a detected target was expected next.

For a given velocity of the target, these inaccuracies were proportional to the length of time that the system had been continuously running. As a consequence, the risk of failing to track, classify, and intercept an incoming Scud missile increased with the length of time that the Patriot missile-defense system operated without a reboot.

On 21 February 1991, the Patriot Project Office warned Patriot users that "very long runtimes" could negatively affect the system's targeting, implying it should be rebooted regularly. Unfortunately, the Army officials assumed that the users would not continuously operate the Patriot systems long enough for a failure to become imminent; therefore, they did not specify the required rejuvenation frequency.

Costs

While rejuvenation can clean internal error conditions from a system and avoid failure occurrence, it does incur costs.

For example, during a Web server's reboot, the hosted Web site might be unavailable, or if multiple servers provide service at the same site, the running servers must share the load.

In transactions-based software systems, initiating rejuvenation can lose jobs that the system currently serves. The Patriot missile-defense system reboot, which also reset the internal clock to zero, took about 60 to 90 seconds; during this time, the system could not react to incoming missiles.

Rejuvenation, therefore, requires optimal timing. The two main approaches are based on models and measurements.

Model-based approaches use analytic models to capture system degradation and rejuvenation. Under a given rejuvenation policy—such as "Rejuvenate an idle server if at least x hours have passed since the last rejuvenation"—operators can then use the model for determining the optimal time interval x and the dependability measures following from this policy.

The main idea behind measurement-based approaches is to periodically monitor system attributes that might show signs of software aging. For example, a continuous increase in the amount of used physical memory might suggest the existence of memory leaks that would ultimately lead to a system crash. Online-monitoring systems can use the collected data to assess the system's current health and predict aging-related failures (M. Grottke et al., "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, Sept. 2006, pp. 411-420).

Even if software developers don't fully understand the faults or know their location in the code, software rejuvenation can help avoid failures in the presence of aging-related bugs. This is good news because reproducing and isolating an aging-related bug can be quite involved, similar to other Mandelbugs.

Moreover, monitoring for signs of software aging can even help detect software faults that were missed during the development and testing phases. If, on the other hand, a developer can detect a specific aging-related bug in the code, fixing it and distributing a software update might be worthwhile. In the case of the Patriot missile-defense system, a modified version of the software was indeed prepared and deployed to users. It arrived at Dhahran on 26 February 1991—a day after the fatal incident.

Acknowledgments

This work was supported by a fellowship within the Postdoc Program of the German Academic Exchange Service.

Michael Grottke *is an assistant research professor in the Department of Electrical and Computer Engineering at Duke University. Contact him at Michael.Grottke@duke.edu.*

Kishor S. Trivedi *holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University. Contact him at kst@ee.duke.edu.*

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov

Architectures for Online Error Detection and Recovery in Multicore Processors

Dimitris Gizopoulos
Mihalis Psarakis
Dept. of Informatics
University of Piraeus, Greece
{dgizop|mpsarak}@unipi.gr

Sarita V. Adve
Pradeep Ramachandran
Siva Kumar Sastry Hari
Dept. of Computer Science
University of Illinois at Urbana-Champaign, USA
swat@cs.uiuc.edu

Daniel Sorin
Albert Meixner
Dept. of ECE and Computer Science
Duke University, USA
sorin@ee.duke.edu
albert.meixner@gmail.com

Arijit Biswas
TRU Group
Intel Corporation, USA
arijit.biswas@intel.com

Xavier Vera
Intel Barcelona Research Center
Intel Labs Barcelona – UPC
xavier.vera@intel.com

***Abstract*— The huge investment in the design and production of multicore processors may be put at risk because the emerging highly miniaturized but unreliable fabrication technologies will impose significant barriers to the life-long reliable operation of future chips. Extremely complex, massively parallel, multi-core processor chips fabricated in these technologies will become more vulnerable to: (a) environmental disturbances that produce transient (or soft) errors, (b) latent manufacturing defects as well as aging/wearout phenomena that produce permanent (or hard) errors, and (c) verification inefficiencies that allow important design bugs to escape in the system. In an effort to cope with these reliability threats, several research teams have recently proposed multicore processor architectures that provide low-cost dependability guarantees against hardware errors and design bugs. This paper focuses on dependable multicore processor architectures that integrate solutions for online error detection, diagnosis, recovery, and repair during field operation. It discusses taxonomy of representative approaches and presents a qualitative comparison based on: hardware cost, performance overhead, types of faults detected, and detection latency. It also describes in more detail three recently proposed effective architectural approaches: a software-anomaly detection technique (SWAT), a dynamic verification technique (Argus), and a core salvaging methodology.**

Keywords: multicore microprocessors; dependable architectures; online error detection/recovery/repair.

I. Introduction

Dimitris Gizopoulos

Advances in semiconductor manufacturing processes have sustained the validity of Moore's law for several decades both in terms of device counts and delivered performance. Recently, a consensus has been reached in the computing community that the only viable way to keep performance improvement rates within a given power budget is by building *multicore processors* and exploiting massive parallelism. This major paradigm shift in computing comes with several challenges affecting all aspects of hardware and software technologies: circuit design and manufacturing, microprocessor architectures, memory systems, programming languages, compilers, and operating systems.

A major challenge which is now more important than ever before in the history of computing is *dependability* [1]. Traditionally high dependability/reliability was mandatory only for a few applications and systems where cost was not a major limitation. Multicore microprocessors (and memories) are today manufactured in inherently unreliable technologies. *Cost-effective dependability* for general purpose computing systems is now a demand: dependability on multicore chips and computing systems built with unreliable components requires a synergy of effective hardware and software solutions.

978-3-9810801-7-9/DATE11/©2011 EDAA

The most important sources of unreliable hardware operation that can lead to system failures are ([2]): (i) process variability that causes the heterogeneous operation of identical components on the same chip; (ii) soft/transient errors sensitivity of today's very deep submicron circuits; and (iii) accelerated aging/wearout of devices due to their extreme operating conditions. In addition to the problems of hardware errors that can severely affect the correct operation of multicore microprocessors, there is another major threat that continues to worsen. Due to the extreme complexity of multicore processors and the pressure for reduced time-to-market, even after the application of a comprehensive pre-silicon verification and post-silicon validation flow, major design errors/bugs can still exist after the chip enters operation in the field. It is apparent that success of the emerging multicore microprocessor paradigm depends (among many factors) on the effective deployment of online error detection, recovery and repair schemes that can provide low-cost dependability guarantees against hardware errors and design bugs.

In this paper, we summarize of few of the most representative error detection and repair techniques that have been proposed in the literature for the building of dependable multicore architectures. We provide taxonomy of the error detection approaches and compare them based on various criteria: hardware and performance overheads, detection latency, targeted faults and fault coverage. The rest of the paper discusses three such architectural approaches in more detail.

II. Dependable Multicore Processor Architectures

Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera

A. Online Error Detection

Multicore processor architectures incorporate CPU cores, memory arrays (e.g. caches, register files), memory control logic and interconnection logic. Memories that occupy a large portion of processor die can be successfully protected using well-known information-redundancy techniques like error-correcting codes (ECC). Thus, the key element of online error detection is to protect the remainder of the processor: the CPU cores (which dominate the remaining die area), the memory hierarchy control logic (memory consistency), and the interconnection logic. Several online error detection techniques for the aforementioned processor components have been recently proposed. These approaches can be classified in four main categories as shown in Fig.1: (a) *redundant execution* approaches which exploit the inherent replication of processor cores and threads in a multicore processor architecture, (b) *periodic built-in self-test (BIST)* approaches which advocate leveraging the built-in test mechanisms of processors traditionally used for manufacturing testing, (c) *dynamic verification approaches*, and (d) *anomaly detection approaches*.

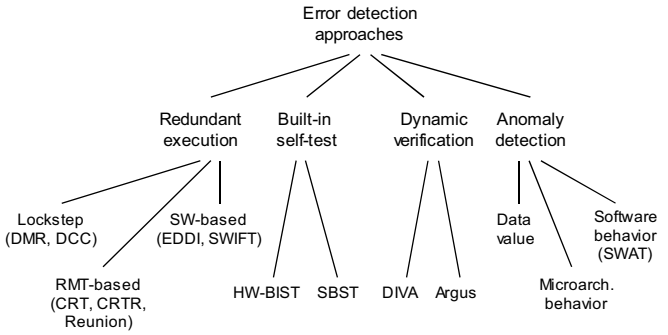


Figure 1. Taxonomy of online error detection techniques.

Redundant execution. In a redundant execution approach two independent threads execute copies of the same program and results are compared. With the advent of multiple on-chip threads in simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures, hardware redundancy techniques such as dual modular redundancy (DMR) and triple modular redundancy (TMR) – which have been studied for a long time but impose very high hardware overheads – have become more attractive; keep in mind that full utilization of cores is not usually feasible and, therefore unused processor cores can execute redundant threads. The two dominant forms of redundant execution in processor architectures are the *lockstep configuration* and the *redundant multithreading (RMT)* with loose lockstepping or without it [3]. In a typical lockstep configuration, identical cores are tightly-coupled in a per cycle or per instruction basis. Aggarwal et al. [4] propose DMR and TMR configurations for CMPs which provide error detection and error recovery through fault containment and component retirement. The proposed technique needs a small amount of extra area to support the reconfiguration-for-repair mechanism (less than 1% in a commodity processor). LaFrieda et al. [5] present a dynamic core coupling (DCC) technique for CMPs which allows arbitrary processor cores to verify each other in a DMR setup avoiding static binding of cores.

Mukherjee et al. [6] propose a redundant execution technique – named chip-level redundant threading (CRT) – that extends RMT technique for single SMT processors to CMP architectures. Similar to RMT, CRT uses loosely synchronized redundant threads reducing the checker overhead. A leading thread in one core is checked by a trailing thread in another core forwarding their results through a dedicated bus. Application on a dual-core SMT processor showed that CRT achieves better results than simple lockstepping the two cores. Also, it achieves better permanent fault coverage than RMT techniques since the redundant threads do not share common resources. Gomaa et al. [7] propose Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) which extends CRT for transient-fault detection in CMPs. CRTR uses a long slack enabled by asymmetric commit to hide inter-processor latency required in CRT. Smolens et al. [8] propose Reunion, a CRT-based architecture that relaxes input replication while preserving the existing memory system, including the coherence protocol and consistency model and reduces comparison bandwidth by compressing the results.

Unlike hardware-based redundant techniques which impose significant hardware overhead, *software-based redundant techniques* can provide a low-cost alternative. Oh et al. [9] proposed EDDI (Error Detection by Duplicated Instruction), a software-based error detection technique in which all instructions are duplicated and appropriate instructions are inserted to check the results. Although more than 100% performance overhead is expected due to instruction duplication, in most cases it is less than 100%; the lower overhead is because the duplicated programs do not have dependencies and thus, the shadow program may fill the empty slot of the pipeline. SWIFT [10] makes several key refinements to EDDI; the major difference is that while the sphere of replication (the domain of redundant execution) in EDDI includes the memory subsystem, SWIFT leaves it out, assuming that the memories are protected by a well-established ECC technique. EDDI and SWIFT are single-threaded approaches that can be applicable in both unicore and multicore processors.

Periodic built-in self-test. The abovementioned redundant execution approaches support *concurrent* error detection because redundant hardware (or software) runs concurrently with the normal one. Another category of error detection approaches leverages the use of built-in self-test (BIST) mechanisms (hardware or software) traditionally used for manufacturing testing. The BIST-based approaches perform *non-concurrent* error detection because the self-test sessions are executed either periodically or during idle time intervals. Hardware BIST techniques [11] are well-established DFT solutions which increase the system testability and relaxes tester's interface speed requirements during manufacturing testing. Shyam et al. [12] utilize existing distributed hardware BIST mechanisms to validate the integrity of the processor components in an online detection strategy. The proposed error detection technique provides high fault coverage (89%) imposing low area overhead (5.8%). Software-based self-test (SBST) has gained increasing acceptance for microprocessor testing the last years and currently forms an integral part of the processor manufacturing test flow [13]. The key idea of SBST is to exploit on-chip programmable resources to execute normal programs that test the processor. Functional test patterns are generated and applied by the processor using its native instruction set, virtually eliminating the need for additional test-specific hardware while the test is applied at the actual operating frequency. SBST has been recently exploited in multicore and multithreaded architectures. Apostolakis et al. [14] apply SBST to bus-based CMPs and propose a test scheduling methodology to exploit core-level execution parallelism and reduce the total test execution time. Foutris et al. [15] extended the SBST methodology of [14] to multithreaded CMP architectures. The proposed methodology speeds up test execution by exploiting execution parallelism and simultaneously increases the fault coverage (88%). Constantinides et al. [16] propose an error detection methodology using periodic execution of SBST tests, assisted by ISA extensions and microarchitectural support. Application of software tests implies a system overhead since the periodic testing time may vary between 5% and 25% of the system time.

Dynamic verification. Another error detection category that does not use redundant execution is *dynamic verification*. These approaches operate at runtime and use dedicated hardware checkers to verify the validity of specific *invariants* assumed to be true in error-free operation. The key point in a dynamic verification approach is to define a comprehensive set of invariants. Dynamic verification was first introduced in dynamic implementation verification architecture (DIVA) [17]. DIVA uses a simple checker core to detect errors in a speculative, superscalar core. DIVA is an excellent low-cost solution for complex superscalar processors where the checker imposes a relatively small area overhead (6% for an Alpha 21264 processor [18]). However, for simpler processors typically used in multicore architectures the complexity of checker core is comparable with that of processor core and it significantly increases the cost. A more recent dynamic verification approach, Argus [19], checks four invariants: control flow, computation, dataflow and memory integrating existing checking mechanisms. Argus architecture imposes less than 17% area overhead to a RISC processor core while still achieves high fault coverage (98%). More details about dynamic verification and especially Argus architecture are discussed in Section IV.

Dynamic verification approaches are also used to validate the cache coherence [20], [21], and the memory consistency [22], [23], [24] of the memory hierarchy system of multicore processor architectures for either online error detection or post-silicon validation. Meixner and Sorin [20], [24] implement low-cost checkers to verify various invariants that a specific memory consistency model must satisfy in error-free operation. Pascual et al. [21] extends a cache coherence protocol to deal with transient faults that affect the interconnection network of a CMP. Chen et al. [22] captures the ordering of shared-memory operations and periodically validates the ordering using a constraint graph. DeOrio et al.,[23] log memory operations in on-chip storage resources and periodically aggregate and check the logs to validate memory consistency in post-silicon validation.

Anomaly detection. These approaches detect faults monitoring the software for *anomalous behavior*, or *symptoms* of faults, using low-cost hardware and software monitors. The anomaly detection approaches can be classified in three categories according to the level of the symptoms they detect ([3]): (a) those that detect data value anomalies [25], like out-of-range values, values not matching with value history, bit invariants, etc., (b) those that detect microarchitectural behavior anomalies [26] like exceptions, cache misses, page faults, etc. and (c) those that detect software behavior anomalies [27], like fatal hardware traps, abnormal application exit, OS hangs, etc. Software behavior anomalies approaches based on SWAT (SoftWare Anomaly Treatment) architecture are discussed in Section III.

Table I compares all the different error detection categories in terms of hardware cost, the extra hardware (if any) required, performance overhead, i.e. overhead imposed to the system due to the additional time for error detection, detection latency, i.e. the time between error appearance and error detection, targeted faults, i.e. the fault types detected by the technique and fault coverage, i.e. the percentage of detected faults.

TABLE I. COMPARISON OF ONLINE ERROR DETECTION TECHNIQUES
Note: DMR occupies twice the number of cores; thus imposes 100% hardware cost (*) and reduces the effective number of cores by one half (**).

Error detection technique	Hardware cost	Performance overhead	Detection latency	Targeted faults	Fault coverage
Lockstep redundant execution	DMR [4]: < 1% for reconfiguration support DCC [5]: 64-entry age table in each core to support master-slave consistency (*)	(**) DCC: 3%-5%	Cycle-by-cycle lockstep	Transient and permanent faults	
RMT redundant execution	CRT [6], CRTR [7], and Reunion [8] require extra hardware: queues, inter-processor communication and checker module	CRT: achieves 13% better performance than a dual lockstep CPU Reunion [8]: 5%-6%	Loose locktep: tens of cycles (due to interprocessor communication and checker latency)	Transient and permanent faults	Better permanent FC than single RMT
Software-based redundant execution	EDDI [9], SWIFT [10]: None	EDDI [9]: Less than 100% in most cases	Low	Transient faults	EDDI: 98.5% SEU
Built-in self-test (BIST)	HW-BIST [12]: 5.8% SBST [13], [14], [15]: None	Depends on the self-test execution frequency [17]: 5%-25%	Test period (maximum)	Permanent faults	[12]: 89% [14]: 91% [15]: 88%
Dynamic verification	DIVA [17]: 6% in a superscalar processor. Much higher in simpler cores Argus [19]: 17% in a RISC core	Low Argus [19]: 3.2-3.9%	Low	Transient and permanent faults and design bugs	Argus: 98% (smaller than DIVA)
Anomaly detection	SWAT [27]: low	Low	95% within 100K cycles 98% within 10M cycles	Transient and permanents faults	> 99%

B. *Online Error Recovery and Repair*

Error recovery techniques are classified into two broad categories: forward error recovery (FER) and backward error recovery (BER). FER techniques detect and correct the errors without requiring to rollback to a previous correct state. This can be achieved only using redundancy, e.g. a TMR lockstep configuration. Backward Error Recovery (BER) techniques periodically save (checkpoint) system state and rollback to the latest validated checkpoint when a fault is detected. Efficient checkpoint and rollback approaches have been proposed in the literature for multicore architectures [28], [29], [30]. These approaches can be classified based on three characteristics: the sphere of BER (register files, caches, memory), the relative checkpoint location (dual or leveled) and the separation of checkpoint and active data (full or partial). *SafetyNet* [28] combines local checkpointing and incremental logging of data and stores updates in special buffers (dual, partial separation by logging). *Revive* [29] uses global checkpointing, flushes cache dirty lines to memory and uses a special directory controller to log memory updates in memory (dual, partial separation by logging). SafetyNet can tolerate fault detection latency up to 1 ms while Revive up to 100 ms. Revive I/O [30] is an extension of Revive that deals with the output-commit problem.

Error repair techniques typically leverage redundancy (either spatial or temporal) to deactivate and isolate the faulty component from the rest of the system. Different approaches [31], [32], [33] have proposed reconfiguration and repair techniques for complex, superscalar processor architectures based on the fact that such architectures include inherent redundancy to provide increased performance and speculative execution. Thus, the redundant and non-essential components could be disabled in order to improve yield and enable graceful performance degradation of the system.

However, the redundant functionality of complex superscalar cores is not usually included in the processing cores of a multicore architecture and therefore the latter does not allow the application of the above repair approaches. Meixner and Sorin [34] proposed *Detouring*, a software-based error repair technique for simple cores which are more attractive for building massively parallel multicore architectures. Given that simple cores do not have sufficient redundancy, the key idea of Detouring is to provide fault tolerance by software: software is modified in order to preserve its functionality but not to use the faulty components. Detours have been proposed for several components, e.g. instruction cache, registers, functional units, etc. Detouring imposes no hardware and performance overhead when the cores are fault-free.

Recent approaches [4], [35]-[38], propose repair techniques allowing reconfiguration for multicore processor architectures. The effectiveness of these techniques should rest on the coordination of fault diagnosis and isolation at several levels, i.e., at circuit level (fine granularity) or at architectural level (coarse granularity) and

the allocation of processing threads to fault-free components. Approaches [4] and [35] adopt the straightforward architectural level solution, i.e. the faulty core is deactivated and replaced by another spare core. Some approaches consider the repair of interconnection network. Collet et al. [35] propose a self-organizing approach that -tests and mutually diagnoses the CPUs, routers, and on-chip memories in a multicore array, isolates and deactivates the defective elements and discovers valid routes.

Recent approaches [36]-[38] consider reconfiguration at finer granularity to reduce performance degradation in high failure rates. *StageNet* fabric proposed by Gupta et al. [36], is a reconfigurable multicore architecture which is designed as a reconfigurable network of processor pipeline stages rather than isolated cores. The pipeline stages act as processing elements and are shared among cores providing inherent fine-grained redundancy. Romanescu and Sorin [37] propose *Core Cannibalization Architecture*, which allows cannibalization of the cores into spare parts, where these parts can be pipeline stages. Powell et al. [38] propose *architectural core salvaging* based on the observation that a multicore architecture can be ISA-compliant, i.e. executes all instructions of the ISA, even if some defective cores cannot execute it entirely. Exploiting cross-core redundancy, the victim thread can migrate to another core that can execute the required operations. Core salvaging approach is presented in Section V.

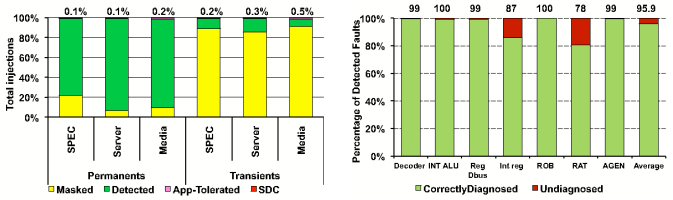
III. SWAT: Designing Resilient Hardware by Treating Software Anomalies

Pradeep Ramachandran, Siva Kumar Sastry Hari, Sarita V. Adve

SWAT (SoftWare Anomaly Treatment) is a comprehensive solution to detect, diagnose, and recover from a variety of hardware faults at very low cost. SWAT is based on two key observations. First, a reliable system must handle only those hardware faults that propagate to software and cause anomalous behavior; the rest can be safely ignored, lowering the incurred cost. Second, despite the impending reliability threat, fault-free operations remain common and must be optimized. SWAT thus detects hardware faults by watching for anomalous software behavior, using zero to low-cost hardware and software monitors. In the rare event of a fault, SWAT invokes a comprehensive diagnosis procedure that isolates the source of the fault in a multicore system, facilitating fine-grained repair or reconfiguration. SWAT performs recovery by using a checkpointing and rollback based mechanism and supports recovery even in the presence of I/O.

A. SWAT Components

Fault Detection: SWAT detects hardware faults by monitoring anomalous software execution (symptoms). These symptom detectors incur near zero overhead in fault free execution. SWAT deploys the following detectors: (1) *Fatal Traps* indicating an illegal software operation, e.g. a divide by zero, (2) *Hangs* indicating an application or system hang, identified with a heuristic hardware hang detector, (3) *Kernel Panics* indicating that the kernel crashed due to a fault, (4) *High OS* indicating an anomalously high amount of contiguous OS activity, (5) *Application Aborts* indicating an application that was terminated due to illegal operations, (6) *Out-of-bounds Detector* flagging loads/stores to addresses outside legal bounds. These detectors can be implemented with existing hardware performance counters and minimal hardware support. Hence, they incur near-zero performance and area overheads. Fig. 2(a) shows their efficacy to detect permanent and transient faults in the core for a variety of workloads. The low rate of Silent Data Corruptions (SDCs, numbers on top of each bar) shows that these detectors are highly effective in single-core [27] and multicore systems [39] (SDC rate between 0.1% and 0.5%). We have also explored the use of software-level program invariants, extracted by using a compiler, as detectors of hardware faults in the iSWAT framework [40]. Our results show that such detectors may be used to further reduce the low SDC rates, demonstrating the customizability of SWAT to the system needs.



(a) Efficacy of detectors (b) Multicore fault diagnosis

Figure 2. Efficacy of SWAT to (a) Detect and (b) Diagnose in-core faults.

Fault Diagnosis: After a symptom is detected, the SWAT diagnosis module takes over to identify the root-cause of the symptom assisted by the recovery module. Simple rollback and re-execution is sufficient to diagnose a transient fault and even recover from it. If the fault is not diagnosed as a transient fault, it can be either a software bug or a permanent hardware fault. To diagnose a permanent hardware fault, SWAT performs a series of rollback and re-executions to compare the traces and isolate the faulty core in a multicore system. If no permanent fault is diagnosed and all the re-executions detect the same SWAT symptom then a software bug is inferred. For permanent hardware faults, SWAT first identifies the faulty core and then proceeds with the fine grained microarchitecture level in-core fault diagnosis.

Multicore fault diagnosis: Multithreaded applications running on multicore systems often share data across threads. This makes diagnosis hard because a fault may escape a faulty core and affect a fault-free core. mSWAT [39] diagnoses the faulty core even in the presence of fault propagation across cores by addressing the following two key challenges: reducing the high cost involved with deterministic replay of a multi-threaded execution, and eliminating the requirement for fault-free spares core for diagnosis. It tackles these challenges by devising a light-weight replay technique that can deterministically replay the execution of each thread in isolation of the other threads. It then uses the isolated deterministic replay to synthesize an inexpensive selective TMR execution only for the purpose of diagnosis. mSWAT successfully diagnoses a large fraction of detected faults. Fig. 2(b) shows the diagnosability of detected faults in various microarchitecture units while running multithreaded media workloads. Over 95% of the faults are successfully diagnosed; all faults escaping to fault-free core are successfully diagnosed.

Single-core fault diagnosis: In most modern systems disabling an entire complex core is wasteful. SWAT therefore proposed a microarchitectural diagnosis procedure called Trace Based Fault Diagnosis (TBFD [41]) that refines the diagnosis further to the microarchitecture level to exploit the built-in microarchitectural redundancy to reconfigure around failed components. TBFD exploits the presence of a fault-free core in the system (identified by mSWAT) to replay and compare the executions on the faulty and fault-free cores. With a sophisticated algorithm that uses violated microarchitecture-level invariants as diagnosis hints, TBFD successfully diagnoses over 98% of the faults to the faulty component, enabling fine-grained repair.

Fault Recovery: Since SWAT, like other symptom detectors, allows the fault to corrupt the architecture state, it relies on checkpointing and rollback support for recovery. While previously proposed hardware checkpointing techniques, such as SafetyNet [28] and ReVive [29], have demonstrated low-cost techniques to recover the architecture state, they have ignored the notorious *output-commit problem*. The only previous recovery solution that handles this problem (ReVive-I/O [30]) relied on software support

and could not guarantee that the committed outputs were protected from in-core faults. SWAT uses a low-cost simple hardware buffer for buffering outputs in hardware and circumvents the limitations of existing solutions. Further, such a solution demonstrates that the detectors need to detect the faults in sub-millisecond durations (instruction latencies of ≤ 100K instruction) in order to keep overheads on fault-free execution from delaying outputs minimal. Previous work has not identified this constraint because output buffering has largely been ignored. Under such a constraint, our SWAT detectors are effective recovering from over 95% of the permanent and transient hardware faults injected into distributed client-server workloads even in the presence of system I/O while incurring less than 5% performance overhead on fault-free execution.

IV. Dynamic Verification of Cores and Memory systems

Daniel J. Sorin, Albert Meixner

There are many ways to detect errors, but many of them are too costly – in terms of power, energy, performance, or area – to be viable for commodity processors. An attractive, low-cost approach to error detection is to dynamically check that certain system-wide invariants are being maintained, and this process is known as *"dynamic verification"* (or "online testing"). By virtue of checking invariants, rather than checking specific components, dynamic verification is independent of the specific implementation and can detect errors due to soft and hard faults as well as errors due to design bugs. Dynamic verification schemes can achieve excellent error detection coverage at overheads in the 1-15% range.

Dynamic verification seems the obvious solution for error detection, dealing with two important, inter-related challenges. The first is that we must determine what invariants to check. Ideally, we would identify a set of invariants that, if checked, would be sufficient for detecting any error in the system. We may need to "divide and conquer", i.e., subdivide the system into components for which we can identify invariants. The second challenge is implementing efficient checkers. Implementation often requires us to reformulate the invariants in a way that is conducive to being checked by hardware. Implementation constraints may also lead to the checker hardware being probabilistic (e.g., uses lossy checksum).

There is a set of recently developed schemes for dynamically verifying multicore processors and in particular focus on dynamic verification of processor cores and cache-coherent shared memory (interconnection network, coherence, etc.). Different sets of invariants have been proposed to check and specific hardware designs check them at runtime. Viability of the schemes has been confirmed by experimental results.

For detecting errors in processor cores, the Argus [19] approach for error detection has been proposed. The key idea behind Argus is that a von Neumann core performs only three activities: control flow (choosing which instructions to execute), computation (performing the computation for each instruction), and dataflow (passing results from a producer instruction to consumer instructions). By checking each of these activities at runtime, Argus can detect virtually any possible error in the core. Checkers have been implemented for each activity, and experimental results have been presented that confirm that Argus detects errors at low-cost.

For detecting errors in the memory system, dynamic verification of memory consistency (DVMC) [24] has been proposed. Because a memory consistency model defines the correct behavior of the memory system, the model serves as a complete invariant for dynamic verification. That is, by dynamically verifying memory consistency, we can detect all possible errors. The problem is solved by a divide-and-conquer approach by splitting consistency into sub-invariants. The most challenging sub-invariant for dynamic verification is cache coherence, and a scheme for achieving this goal is presented [20].

V. How to Manage Accidentally Heterogeneous Cores

Arijit Biswas

A. Heterogeneity by Accident

The search for higher performance with optimal power in the era of multicore CPUs has given rise to the notion of heterogeneous cores. The main concept behind such CPUs is that a general purpose one-size-fits-all core is not necessarily the best for optimal power/performance. In some cases, it may be more effective to have different types of cores on a single CPU die in order to best accommodate various needs of different applications. While such CPUs tend to be more difficult to program due to their asymmetric nature, they are nevertheless gaining in popularity in many computing areas.Heterogeneous cores are becoming popular, but multicore CPUs with symmetric general purpose cores still reign supreme. They are easier to program and provide more consistent performance in many instances. However, such non-heterogeneous CPUs may become heterogeneous, not by design, but as a result of permanent defects that may render a core unable to correctly execute certain instructions.

Multicore CPUs devote a large fraction of die area to regular memory structures, mainly caches. Fortunately, caches can be protected from manufacture-time defects using well-known techniques. Thus, the remainder of the die becomes the major source of defect vulnerability. The bulk of this remainder is CPU cores. In the past, such a defect would cause the entire CPU die to be rejected as defective. Recent work has shown promise that such defects are tolerable if they are properly managed.

Managing defective cores is a two-part challenge: defect detection and defect tolerance. In this section, we focus on solutions for defect tolerance. One obvious solution set, which we define as core disabling and core sparing, disables defective cores (disabling) or enables spare standby cores (sparing) in the event of a core defect. Core disabling reduces sales price due to smaller core count, and core sparing consumes precious die area while providing no performance or economic benefit in a non-defective die.

B. Core Salvaging

A more desirable alternative to core disabling and core sparing is core salvaging, which allows defective cores to continue operation. Microarchitectural core salvaging techniques disable defective execution pipelines [32] or schedule operations on alternate or spare resources [42], [31], [33], [37] to avoid utilizing the defective area, suffering performance loss due to defects. Microarchitectural core salvaging exploits microarchitectural redundancy, relying on the ability of a core to execute the entire ISA correctly even in the presence of certain defects. Ideally, the performance impact of defects is minimal, so that the presence of a single core with a tolerated defect on a multicore die is negligible.

Most modern cores contain large amounts of redundant logic which is generally used to improve performance. This redundant logic could potentially be used to compensate for the defective logic if the defect exists in an opportune location to facilitate this. This comes at the cost of being able to test for and isolate the defect to a finer granularity than just a core. It turns out however that such opportunities for taking advantage of this technique are actually quite small and may require significant overhead.

A better solution that offers significantly more benefit than microarchitectural redundancy alone, is called architectural core salvaging [38]. Architectural core salvaging leverages the fact that a single core need not be ISA compatible so long as the CPU as a whole is. What this means is that if a defect on a particular core

renders it unable to execute certain instructions, that core is still usable assuming that we can detect and move the un-executable instructions to a different core.

C. Results

Recent work has shown that microarchitectural redundancy does not cover defects as well as previously thought [38]. This is because large portions of many redundant structures such as multi-entry arrays are actually made up of non-redundant logic such as decoders, buffers, and interconnect which is not covered. Other structures like execution units are often used by multiple instructions, some of which only use a particular unit even though redundant functionality exists. Microarchitectural techniques have been shown to cover only ~10% of the non-cache core area.

This work also shows that architectural redundancy, the same redundancy exploited by architectural core salvaging, can cover significantly more area. While there are always defects that will render a core useless, such as the inability to execute memory operations, most ISAs contain numerous instructions which are used infrequently yet the hardware dedicated to them occupies significant area. A good example includes SIMD instructions. Not all applications include these instructions in their mix. Even many applications that do have them only have them in specific portions of the code. In such cases, it would be possible to trap on any such instruction and migrate the thread to a "good" core.

Further, it may be possible that better thread-scheduling and thread-swapping algorithms may be able to better harness and use this technique. Even with simple thread-swapping algorithms, architectural core salvaging has been shown to cover nearly half the execution units on an IA32-like processor.

Powell et al. [38] also shows that architectural core salvaging becomes compelling as the number of cores on a CMP increases. Further, this technique is orthogonal to core sparing, which would continue to be used if a major defect compromises a core such that even basic instructions could not execute correctly. Combining microarchitectural and architectural core salvaging can cover significant portions of the processor core while gaining nearly the full performance out of a core with minor defects. Powell et al. shows a 21% coverage for protecting only a small handful of structures and pipelines with these techniques [38].

REFERENCES

[1] D.Gizopoulos, S.Mukherjee, "Dependable Computer Architecture", Special Section, Guest Editorial, IEEE Trans. on Computers, Jan. 2011.

[2] S.Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", IEEE Micro, vol.25, no.6, pp. 10-16, Nov.-Dec. 2005.

[3] D.J.Sorin, Fault Tolerant Computer Architecture, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publish., 2009.

[4] N.Aggarwal, P.Ranganathan, N.P.Jouppi, and J.E.Smith. "Configurable isolation: building high availability systems with commodity multi-core processors", ISCA 2007.

[5] C.LaFrieda, E.Ipek, J.F.Martinez, R.Manohar, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor", DSN 2007.

[6] S.S.Mukherjee, M.Kontz, and S.K.Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives", ISCA 2002.

[7] M.Gomaa, C.Scarbrough, T.N.Vijaykumar, and I.Pomeranz, "Transient-fault recovery for chip multiprocessors", ISCA 2003.

[8] J.C.Smolens, B.T.Gold, B.Falsafi, and J.C.Hoe, "Reunion: Complexity-effective multicore redundancy", MICRO 2006.

[9] N.Oh, P.P.Shirvani, E.J.McCluskey, "Error detection by duplicated instructions in super-scalar processors", IEEE Trans. on Reliability, vol.51, no.1, pp.63-75, Mar 2002.

[10] G.A.Reis, et al., "SWIFT: Software Implemented Fault Tolerance", Intl. Symp. on Code Generation and Optimization (CGO), 2005.

[11] G.Hetherington, et al., "Logic BIST for large industrial designs: real issues and case studies", ITC 1999.

[12] S.Shyam, et al.,,"Ultra Low-Cost Defect Protection for Microprocessor Pipelines", ASPLOS 2006.

[13] M.Psarakis, D.Gizopoulos, E.Sanchez, and M.Sonza Reorda, "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3, pp. 4-19, May/June 2010.

[14] A.Apostolakis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors", IEEE Trans. on Computers, vol. 58, no. 12, pp. 1682-1694, July 2009.

[15] N.Foutris, et al., "MT-SBST: Self-Test Optimization in Multithreaded Multicore Architectures", ITC 2010.

[16] K.Constantinides, O.Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation", MICRO 2007.

[17] T.M.Austin. "DIVA: A reliable substrate for deep submicron micro-architecture design", MICRO 1999.

[18] C.Weaver and T.Austin, "A Fault Tolerant Approach to Microprocessor Design", DSN 2001.

[19] A.Meixner, M.E.Bauer, and D.J.Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores", MICRO 2007.

[20] A.Meixner and D.J.Sorin, "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures", HPCA 2007.

[21] R.Fernandez-Pascual, J.M.Garcia, M.E.Acacio, J.Duato, "Low Overhead Fault Tolerant Coherence Protocol forCMP Architectures", HPCA 2007.

[22] K.Chen, S.Malik, and P.Patra, "Runtime validation of memory ordering using constraint graph checking", HPCA 2008.

[23] A.DeOrio, I.Wagner, and V.Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," HPCA2009.

[24] A.Meixner and D.J.Sorin, "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures." IEEE Trans. on Dependable and Secure Computing, vol. 6, no 1, 2009.

[25] P.Racunas, K.Constantinides, S.Manne, and S.S.Mukherjee, "Perturbation-based Fault Screening", HPCA 2007.

[26] N.J.Wang and S.J.Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors", IEEE Trans. on Dependable and Secure Computing, 3(3), pp. 188-201, July-Sept 2006.

[27] M.-L.Li, et al., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design", ASPLOS 2008.

[28] D.J.Sorin et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery", ISCA 2002.

[29] M.Prvulovic, Z.Zhang, and J.Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors", ISCA 2002.

[30] J.Nakano et al, "ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers", HPCA 2006.

[31] P.Shivakumar, S.W.Keckler, C.R.Moore, and D.Burger, "Exploiting microarchitectural redundancy for defect tolerance", ICCD 2003.

[32] E.Schuchman and T.N.Vijaykumar, "Rescue: A microarchitecture for testability and defect tolerance", ISCA 2005.

[33] J.Srinivasan, S.V.Adve, P.Bose, and J.A.Rivers, "Exploiting structural duplication for lifetime reliability enhancement", ISCA 2005.

[34] A.Meixner, D.J.Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," DSN 2008.

[35] J.Collet, P.Zajac, M.Psarakis, and D.Gizopoulos, "Chip Self-Organization and Fault-Tolerance in Massively Defective Multicore Arrays", IEEE Trans. on Dependable and Secure Computing, 2010.

[36] S.Gupta, et al., "The StageNet fabric for constructing resilient multicore systems," MICRO 2008.

[37] B.F.Romanescu and D.J.Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults", PACT 2008.

[38] M.Powell, A.Biswas, S.Gupta, S.Mukherjee, "Architectural Core Salvaging in a Multi-Core Processor for Hard Error Tolerance", ISCA 2009.

[39] S.Hari et al, "Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems", MICRO 2009.

[40] S.Sahoo et al, "Using Likely Program Invariants to Detect Hardware Errors", DSN 2008.

[41] M.Li et al., "Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults", DSN 2008.

[42] F.A.Bower, P.G.Shealy, S. Ozev, and D. J. Sorin, "Tolerating Hard Faults in Microprocessor Array Structures", DSN 2004.

Designing Safety-Critical Computer Systems

Designers must balance costs against an acceptable level of risk when implementing the techniques and devices that reduce the chance of mishap in safety-critical systems.

William R. Dunn
Independent Consultant

The ubiquitous computer is firmly established as the electronic component of choice for designing systems that control safety-critical applications. Such applications can be found everywhere: aircraft fly-by-wire controls, oil and chemical processing, hospital life-support systems, manufacturing robotics, and countless other commercial and industrial applications. As this century matures, developers will increasingly exploit computing's power in safety-critical applications that directly touch us all: steer-by-wire automotive systems, automated air- and surface-traffic control, powered prosthetics, and so on.

However, these computer-based systems raise the ongoing concern that they might fail and cause harm. Indeed, past computer failures have produced catastrophic results, most famously the notorious Therac 25, a therapeutic computer system intended to heal but which inadvertently killed and maimed patients before being forced off the market.[1]

The safety of computer-based systems is of long-standing and continuing interest to computing professionals. As research continues in this area, proposed system concepts and architectures—deemed safe by their developers—have been found to be impractical for real-life engineering applications that can place lives, property, or the environment at risk. Such dependable, seemingly safe, concepts and structures fail in practice for three primary reasons: Their originators or users

- have an incomplete understanding of what makes a system "safe,"
- fail to consider the larger system into which the implemented concept is to be embedded, or
- ignore single points of failure that will make the safe concept unsafe when put into practice.

Reviewing the fundamental definitions and concepts of system safety provides a framework for addressing these shortcomings. Exploring the systematic design of safety-critical computer systems in engineering practice helps to show how engineers can verify that these designs will be safe.[2]

DEFINING SAFE

The notion of safety is most likely to come to mind when we drive a car, fly on an airliner, or take an elevator ride. In each case, we are concerned with the threat of a *mishap*, which the US Department of Defense defines as an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.[3]

The *mishap risk* assesses the impact of a mishap in terms of two primary concerns: its potential severity and the probability of its occurrence.[3] For example, an airliner crash would affect an individual more severely than an automobile fender-bender, but it's much less likely to happen. This assessment captures the important principle that systems such as cars, airliners, and nuclear plants are never absolutely safe. It also provides a design principle: Given our current knowledge, we can never eliminate the possibility of a mishap in a safety-critical system; we can only reduce the risk that it will occur.

Published by the IEEE Computer Society

Risk reduction adds to system cost, however. Indeed, in some applications—such as nuclear energy—ensuring safety can dominate total system cost. When creating a safe system, minimizing this expense forces us to compromise to the extent that we expend resources to reduce mishap risk, but only to a level considered generally acceptable.

ACCEPTABLE MISHAP RISK

Generally, the public at large establishes the acceptable risk for a given mishap type in terms of its willingness to tolerate the mishap as long as it occurs infrequently. Statistics for various common mishaps and their average frequency represent acceptable risk—and can range anywhere from 10^{-2} to 10^{-10} incidents per hour. The relative rarity of such occurrences explains why, despite the tragic events underlying these statistics, most of us can feel relatively safe while driving a car or flying on an airliner.

This data can also give designers of safety-critical systems a sense of what constitutes safe and unsafe. For example, if they design a safety-critical computer system and project that it will have a 10 percent chance of catastrophic mishap per hour of operation, they know that the design is unsafe, and they must lower the mishap risk to an acceptable level.

Fortunately and wisely, system designers do not decide what constitutes an acceptable level. Instead, they rely on safety standards framed as public law or that result from the work of industrial associations, professional societies, and safety-related institutes that embody the general public's consensus of acceptable risk. For example, two widely used safety standards—the US government's Mil-Std-882D[3] and industry's IEC 61508[4]—provide detailed guidelines regarding acceptable risk.

THE COMPUTER SYSTEM

Typically, virtually any computer system—whether it's a fly-by-wire aircraft controller, an industrial robot, a radiation therapy machine, or an automotive antiskid system—contains five primary components.

The *application* is the physical entity that the system monitors and controls. Developers sometimes refer to an application as a plant or process. Typical applications include an aircraft in flight, a robotic arm, a human patient, and an automobile brake.

The *sensor* converts an application's measured physical property into a corresponding electrical signal for input into the computer. Developers sometimes refer to sensors as field instrumentation. Typical sensors include accelerometers, pressure transducers, and strain gauges.

The *effector* converts an electrical signal from the computer's output to a corresponding physical action that controls an application's function. Developers sometimes call an effector an actuator or final element. Typical effectors include motors, valves, brake mechanisms, and pumps.

The *operator* is the human or humans who monitor and activate the computer system in real time. Typical operators include an airplane pilot, plant operator, and medical technician.

The *computer* consists of the hardware and software that use sensors and effectors to monitor and control the application in real time. The computer comes in many forms, such as a single board controller, programmable logic controller, airborne flight computer, or system on a chip. Many computer systems, such as those used for industrial supervisory control and data acquisition, consist of complex networks built from these basic components.

System designers rely on safety standards that embody the general public's consensus of acceptable risk.

HAZARD ANALYSIS

The safety issues and design methodology associated with these networks and their complex structures strongly resemble those that apply to any simple computer system. Thus, we can study such a system to gain insights about basic design techniques that we would apply to more complex systems.

In the basic computer system, developers fully define the application, including all hardware, software, and operator functions that are not safety-related. Because the basic computer system employs no safety features, it probably will exhibit an unacceptably high level of mishap risk. When this occurs, solving the design problem requires modifying the operator, computer, sensor, and effector components to create a new system that will meet an acceptable level of mishap risk.

The design solution begins with the question, How can this basic computer system fail and precipitate a mishap? The key element connecting a failure in the basic system to a subsequent mishap is the *hazard*,[3] defined as any real or potential condition that can cause

- injury, illness, or death to personnel;
- damage to or loss of a system, equipment, or property; or
- damage to the environment.

Hazard examples include loss of flight control, nuclear core cooling, or the presence of toxic mate-

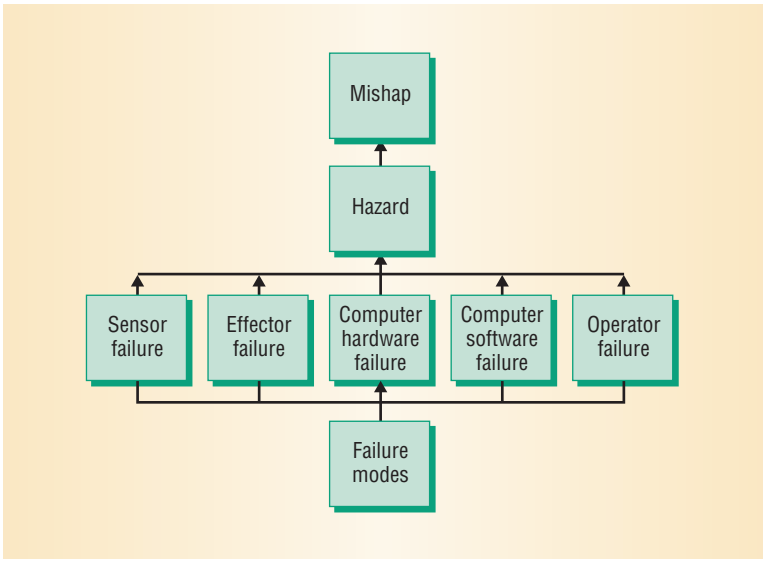


Figure 1. Mishap causes. System designers identify the application's attendant hazards to determine how system-component failures can result in mishaps.

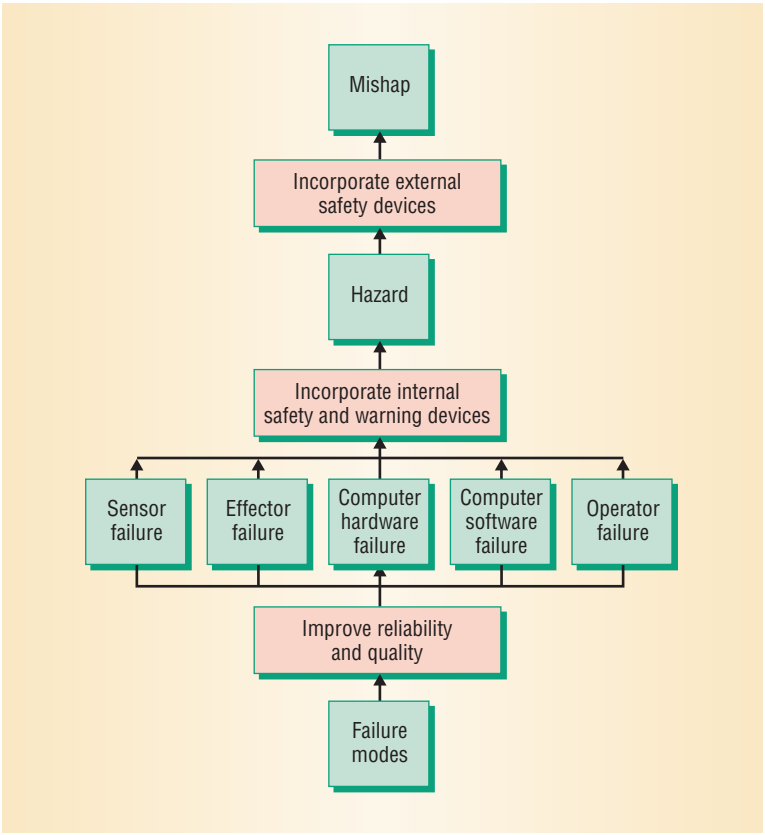


Figure 2. Risk mitigation measures. Designers can modify a system to reduce its inherent risk by improving component reliability and quality and by incorporating internal or external safety and warning devices.

rial or natural gas. All such hazards reside in the application.

Thus, system design focuses first on the application component of the system to identify its attendant hazards. Then designers turn their attention to the operator, sensor, computer, and effector components. To determine how these components can fail and cause a mishap, the designers perform a failure-modes analysis to discover all possible failure sources in each component. These include random hardware failures, manufacturing defects, programming faults, environmental stresses, design errors, and maintenance mistakes.

These analyses provide information for use in establishing a connection between all possible component failure modes and mishaps, as Figure 1 shows. With this analytical background in place, actual design can begin.

A SIMPLE EXAMPLE

Consider a basic computer system used for electrically heating water. In this case, the application is a steel tank that contains water. The effector, a computer-controlled electric-heater unit, heats the water. A temperature sensor measures the water temperature and transmits a corresponding signal back to the computer. Software in the computer maintains the water temperature at 120°F by turning the heater on if the sensed temperature dips below this setting and by turning the heater off if the temperature climbs above the setting.

That the water this system stores might overheat presents one hazard. A potential mishap could occur if the water overheats to the boiling point and causes the tank to explode. Another potential mishap could occur if a person opens a water tap and the overheated water, under high pressure in the tank, scalds that individual as it exits the faucet and flashes into steam.

Several failures can create this hazard. The temperature sensor might fail and inaccurately signal a low temperature. The heater unit might fail and remain on permanently. Computer interface hardware might fail, permanently signaling an "on" state to the heater. A computer software fault, possibly originating in an unrelated routine, might change the set point to 320°F. The operator might program an incorrect set point. Component failures might also occur because of

- a maintenance error such as the repair person installing the wrong temperature sensor,
- an environmental condition such as the heater being placed in an overly warm environment that causes a chip failure, or
- a design failure that results in using the wrong sensor for the selected operating temperature.

This hot water system, as it stands, has an unacceptable risk of mishap.

MISHAP RISK MITIGATION

Given the system's high risk of mishap, design attention turns to modifying it to mitigate this risk. Designers can do this in three ways:

- improve component reliability and quality,
- incorporate internal safety and warning devices, and
- incorporate external safety devices.

Figure 2 shows how and where applying these mishap-risk-mitigation measures can alleviate the computer system mishap causes shown in Figure 1.

Improving reliability and quality involves two measures: improving component reliability and exercising quality measures that will avoid or eliminate the sources of component failure. Reliability improvement seeks to reduce the probability of component failure, which in turn will reduce mishap probability.

A widely used and effective approach for improving reliability employs redundant hardware and software components. Redesign can remove component reliability problems that stem from environmental conditions.

Other sources of component failure such as personnel error, design inadequacies, and procedural deficiencies are more elusive. IEC 61508 includes these sources of failure in a general category described as *systematic failures* and recommends various quality-oriented approaches for avoiding or eliminating them.

Although reliability and quality measures can reduce mishap risk, they normally will not lower it to an acceptable level because component failures will still occur. When a project requires additional risk mitigation steps, *internal safety devices* form the next line of defense. An example of an internal safety device is the thermocouple circuit, which shuts off the gas supply in a home heating furnace should its flame go out. Developers implement these devices in both hardware and software. Internal safety devices not only reduce the effects of hardware and software faults but also provide a barrier against systematic failures, including personnel errors, design inadequacies, and procedural deficiencies.

Even after designers have taken these measures, system failures can still occur, resulting in mishaps. *External safety devices*, which can range from simple physical containment through computer-based safety-instrumented systems, provide a last line of defense against these residual failures. These devices provide protection when the application experiences a hazardous event.

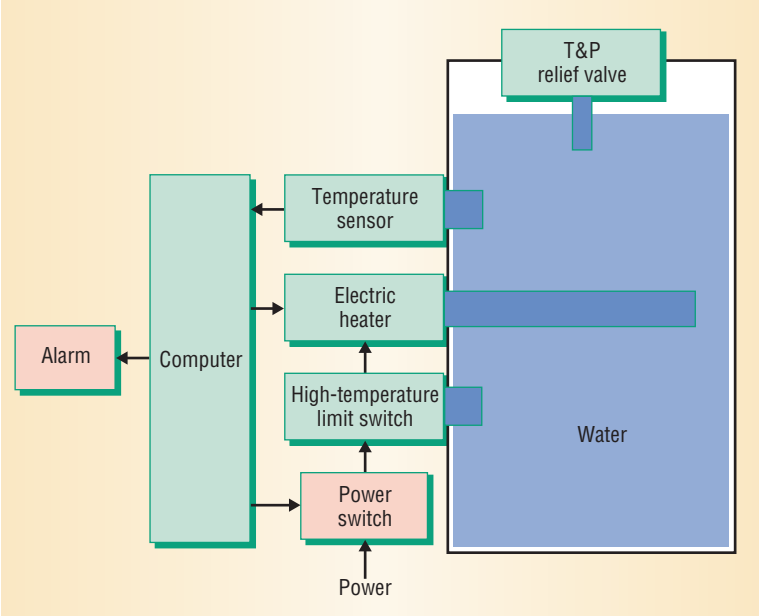


Figure 3. Applying risk-mitigation measures. The addition of safety devices such as a high-temperature limit switch and a temperature-and-pressure (T&P) relief valve has reduced the computer-controlled water heating system's operational risk.

To achieve effective mishap risk mitigation, developers usually strive to apply all three of these mitigation measures concurrently to create a layered approach to system protection. Because even the most lavish project has limited development resources, designers should apply all three types of risk mitigation in a balanced way to reduce mishap risk. In addition, risk mitigation efforts must be distributed evenly across the system's sensor, effector, computer, and operator components because a single neglected failure in any one part of the system can make the aggregate mishap risk totally unacceptable.

THE EXAMPLE REVISITED

Returning to the hot-water system example, upgrading the basic computer system to incorporate safety devices can reduce the system's risk. To reduce risk, the water-heater application uses all three of Figure 2's risk-mitigation measures in three protective layers.

Domestic water heater manufacturers generally employ hardware components with reliability superior to that of everyday household components. Manufacturers take extraordinary quality measures to assure the heater tank's structural integrity. Although these reliability and quality measures can reduce component failure probability and therefore mishap risk, they do not by themselves make the system safe—which is why heater manufacturers add both internal and external safety devices.

Figure 3 shows an internal safety device, the high-temperature limit switch. This device interrupts electric power to the heater when the water temperature, measured by an independent tem-

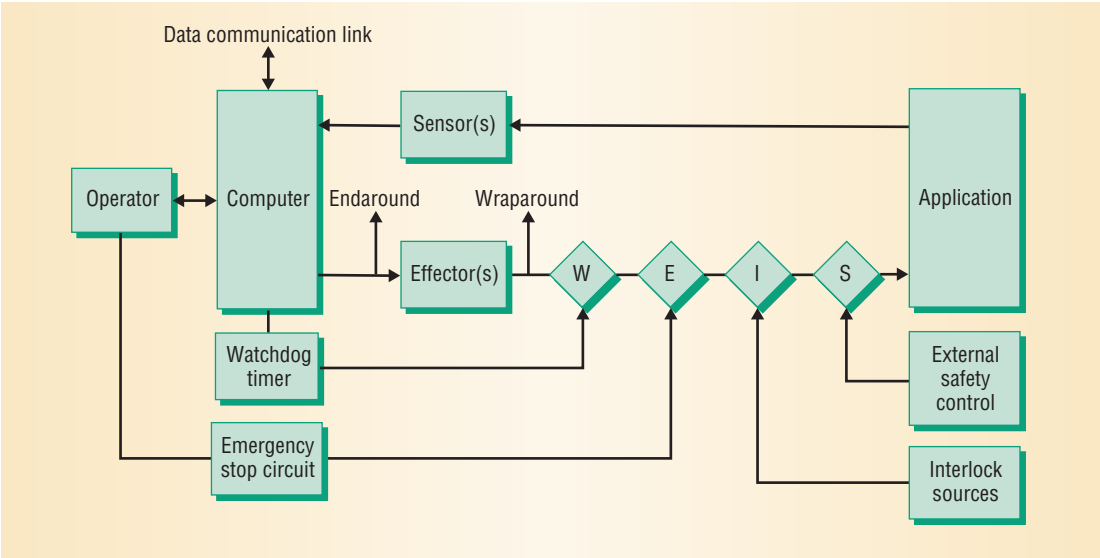


Figure 4. Risk mitigation methods. Designers have added several risk-mitigation devices to this system, including a watchdog timer, emergency stop circuit, and interlocks that inhibit effector actions unless specific external conditions are satisfied.

perature sensor, reaches a preset level. This temperature-limit switch thus provides protection for failures in any of the sensor, effector, computer, or operator components.

Additional internal safety protection can also be written into computer software. To give this software authority over hardware, the manufacturer interposes a power switch—which can be controlled by a computer output port—between the effector power source and the effector. The safety software can detect failures in the various components and cut power to the effector when it detects a failure.

These devices can thus further reduce mishap risk, but they still fall short of lowering it to an acceptable level. Manufacturers achieve additional risk protection by fitting an external safety device to the tank. The *temperature and pressure relief valve*—the T&P valve shown in Figure 3—is an external safety device that relieves tank overtemperature, which can lead to a scalding mishap, and overpressure, which can cause an explosion mishap.

ADDITIONAL SAFETY DEVICES

Figure 4 generalizes the water heater example by showing a basic computer system that has been modified to include risk-mitigation techniques found in real-life applications. One such technique, the emergency stop circuit, inhibits effector outputs by forcing the system into a safe state—as shown by the line in Figure 4 that connects the operator component to the diamond-enclosed E.

Systems often employ interlocks that will inhibit effector action unless some specific external physical conditions are satisfied. The switch that stops the cooking when a user opens a microwave oven door is one example of an interlock. As the water-heater example suggests, designers can reduce mishap risk in a system by using a computer to detect component failures and modifying effector controls to bring the system to a safe state. The design can incorporate various approaches to detecting failures in individual sensors, including reasonableness tests, informational redundancy, state estimators, and analytical redundancy.[2]

As Figure 4 shows, to detect effector failures, the design can use a wraparound in which the effector output feeds back into the computer to verify that the output matches the system command. The same basic approach uses endarounds to verify computer I/O integrity. When the system detects wraparound or endaround mismatches, it signals the effector to shift to a safe state. A failure in the forward computer-to-effector path may, however, prevent the shift. For this reason, developers usually build an additional, independent safety control into the system to neutralize the effector output when it detects wraparound or endaround mismatches.

Finally, most industrial controllers employ a watchdog timer circuit between the computer and effector output. The computer continuously refreshes this circuit with hardware- and software-generated electrical pulses. As long as these pulses continue, the circuit keeps the effector output connected to the application. If the pulses cease through hardware or software failure, the circuit times out, and the system inhibits further effector output.

FAIL-OPERATE SYSTEMS

In fail-safe systems, hardware, software, or an operator detects a failure and modifies effector output so that the system enters a safe, generally nonoperating state. Most real-world applications are fail-safe systems. Many computer systems, however, such as fly-by-wire aircraft control systems,

must continue safe operation after one or more components have failed. These *fail-operate* computer systems achieve their fault-tolerance capability through redundancy.

One fail-operate approach uses a backup system that can take over the computer's safety-critical functions should the system fail. For example, computer-controlled fly-by-wire airliners, such as the Airbus A320 family, can fly—but at great risk—using primitive mechanical controls as a backup should the computer system fail.[5] As is the case in the Airbus application, a system's performance usually degrades when used in backup mode. A second approach simply replicates components so that if a given component fails, the system includes one or more duplicates to continue the required function.

Although component redundancy is a simple concept, the details of implementing it are not. First, the design must replicate virtually every critical component in a system, including computers, sensors, effectors, operators, power sources, and interconnects. Second, the design must incorporate a redundancy-management process into the fail-operate system's hardware, software, or operator components to detect failures when they occur, isolate the failed component, and reconfigure the system so that one or more healthy components will replace or mask the failed counterpart.

These failure, detection, isolation, and reconfiguration processes can quickly become complex, resulting in system development costs that far exceed those of the corresponding basic computer system.[2] For this reason, component redundancy becomes a practical design option only when a backup system is infeasible or when performance must be maintained following one or more component failures.

To design a fail-operate system, many developers use a two-step process in which they first select a redundant hardware structure or architecture and subsequently flesh out this framework with the appropriate redundancy management hardware and software processes. This two-step process is impractical, however, because the system's redundancy-management scheme—*not* its redundant structure—primarily governs the achievable risk level associated with a redundant computer system.

Consequently, designers must resort to a cut-and-try process that will meet a required risk level and, at the same time, satisfy the usual engineering economies of cost, power, weight, and so on. The preferred approach therefore begins with the basic, nonredundant system hardware structure and incrementally introduces redundancy and redundancy-management processes until a fail-operate system emerges that meets the desired safety goal.[2]

Fail-operate computer systems achieve their fault-tolerance capability through redundancy.

EVALUATING SAFETY-CRITICAL COMPUTER SYSTEMS

After the designers have applied measures to mitigate mishap risk to a basic system, they must determine if the modified system design meets an acceptable level of mishap risk. They can use three analytical techniques to make this determination.

In *failure modes and effects analysis* (FMEA), the designer or analyst looks at each component in the system, considers how that component can fail, then determines the effects each failure would have on the system.[2,6] This analysis seeks first to verify that there is no mishap-producing single point of failure in the system because such a potential point of failure would nullify the benefits of applying mitigation measures elsewhere in the system.

Fault tree analysis (FTA) reverses this process by starting with an identified mishap and working downward to identify all the components that can cause a mishap and all the safety devices that can mitigate it.[7,8] This downward decomposition process builds a graphical structure called a fault tree.

In contrast to FMEA and FTA, which are both qualitative methods, *risk analysis* (RA) is a quantitative measure that yields numerical probabilities of mishap.[2,7] To perform RA, the analyst must determine the component failure probabilities for the hardware, software, and operator components in the fault tree.[2,6,7] In accordance with standards such as Mil-Std-882D[3] and IEC 61508,[4] designers usually estimate failure probabilities on a per-hour basis.

If the system consists of redundant components, designers calculate its unreliability—the probability that it will not operate over the span of one hour. Next, they determine mitigation failure probabilities for the fault tree's hardware, software, and operator safety devices. If a mitigation device includes redundant components, designers determine its unavailability—the probability that it will not mitigate if required.

The designers assign these component- and mitigation-failure probabilities to elements in the fault tree, then propagate them upward to yield a figure for mishap risk. If this results in an unacceptable figure, they must implement additional mitigation measures. As a side benefit, the fault tree shows where to add these measures in the system. If, on the other hand, the risk calculation yields an acceptable result, the design is ready for additional vali-

dation steps[5] such as in-depth risk assessment, testing, and field trials to assure that the system, when implemented, will be safe.

Although it may seem obvious, a developer's concerns about a safety-critical system's continuing safety do not end with design and implementation. Indeed, a vigorous system safety program must be in place throughout the system's operational life to ensure that mishap risk is maintained at or below the level achieved in the original design.[3,4]

Achieving risk reduction in a new or existing computer system design requires dealing with *all* the system's components: hardware and software, sensors, effectors, operator, and—most importantly—the primary source of harmful energy or toxicity, the application. After identifying the application's hazards, determining component failure modes, and introducing appropriate risk mitigation measures, designers analyze the modified system to obtain a risk estimate. If risk remains unacceptable, they must take additional risk mitigation steps until the modified system has an acceptable level of mishap risk. ■

References

1. N. Leveson and C.S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, July 1993, pp. 18-41.
2. W.R. Dunn, *Practical Design of Safety-Critical Computer Systems*, Reliability Press, 2002.
3. *Standard Practice for System Safety*, Mil-Std-882D, US Dept. of Defense, 2000; http://www.geia.org/sstc/G48/882d.pdf.
4. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, IEC 61508, Int'l Electrotechnical Commission, 2000.
5. N. Storey, *Safety-Critical Computer Systems*, Addison-Wesley, 1996.
6. W. Goble, *Control Systems Safety Evaluation and Reliability*, ISA, 1998.
7. T. Bedford and R. Cooke, *Probabilistic Risk Analysis: Foundations and Methods*, Cambridge Univ. Press, 2001.
8. *Fault Tree Handbook*, NUREG-0492, US Nuclear Regulatory Commission, 1981; www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/ sr0492.pdf.

William R. Dunn, *formerly the director of the University of Southern Colorado's research station at NASA Ames Research Center, is now an independent consultant. His research and engineering interests include design and validation of safety-critical flight- and ground-based digital systems. Dunn received a PhD in electrical engineering from Santa Clara University. Contact him at wrdunn@syv.com.*

How the Hidden Hand Shapes the Market for Software Reliability

Ken Birman, Coimbatore Chandersekaran, Danny Dolev, Robbert van Renesse

***Abstract*— Since the 18th century, economists have recognized that absent government intervention, market forces determine the pricing and ultimate fate of technologies. Our contention is that the "hidden hand" explains a series of market failures impacting products in the field of software reliability. If reliability solutions are to reach mainstream developers, greater attention must be paid to market economics and drivers.**

I. Introduction

We suggest in this paper that reliability issues endemic in modern distributed systems are as much a sign of market failures as of product deficiencies or vendor negligence. With this in mind, we examine the role of markets in determining the fate of technologies, focusing on cases that highlight the interplay between the hidden hand of the market (c.f. Adam Smith) and consumer availability of products targeting software reliability.

The classic example of a market failure involved the early development of client-server computing products. Sparked by research at Xerox PARC that explored the basic issues in remote procedure call, industry embraced the client-server paradigm only to discover that the technology was incomplete, lacking standards for all sorts of basic functionality and tools for application development, debugging and operational management. Costs were very high and the market stumbled badly; it didn't recover until almost a decade later, with the emergence of the DCE and CORBA standards and associated platforms.

Our market-oriented perspective leads to a number of recommendations. Key is that work intended to influence industry may need many steps beyond traditional academic validation (even when evaluation is exceptionally rigorous). To have impact, researchers need to demonstrate solutions in the context of realistic platforms and explore the costs and benefits of their proposals in terms of the economics of adoption by vendors and by end-users. Skipping such steps often inhibits promising technologies from reaching commercial practice. On the other hand, we also believe that not all work should be judged by commercial value!

Birman and Van Renesse are with the Department of Computer Science, Cornell Univerisity, Ithaca NY 14850; Dolev is at Cornell on leave from the Hebrew University, Jerusalem, Israel.; Chandersekaran is at IDA in Washington, DC. Emails {ken,rvr,dolev}@cs.cornell.edu; cchander@ida.org. This work was supported in part by grants from NSF, DARPA, AFRL, AFOSR and Intel.

II. The Hand of the Market

The distributed systems reliability community has offered a tremendous range of solutions to real-world reliability problems over the past three decades. Examples include:

- *Transactions and related atomicity mechanisms*, for databases and other services [7][4].
- *Reliable multicast*, in support of publish-subscribe, virtually synchronous or state-machine replication, or other forms of information dissemination [1].
- *The theory of distributed computing* [3][6].

Our own research and industry experiences have touched on all of these topics. We've derived fundamental results, implemented software prototypes, and led commercial undertakings. These experiences lead to a non-trivial insight: market acceptance of reliability technology has something to do with the technology, but far more to do with:

- *Impact on the total cost of building, deploying and operating "whole story" solutions.*
- *Credibility of the long-term vision and process.*
- *Compatibility with standard practice.*

Whether or not a project is attentive to these considerations, market forces often decide the fate of new ideas and systems. By understanding factors that influence this hidden hand, we can be more effective researchers.

Of course, not all research projects need to achieve commercial impact. Thus we're driven to two kinds of recommendations: some aimed at work that will be measured by its direct impact on the marketplace; others aimed at work seeking theoretical insights, where direct commercial impact isn't a primary goal. In the remainder of this document, we flesh out these observations, illustrating them with examples drawn from the major technology areas cited earlier.

III. A Brief History of Reliable Computing

In this section, we discuss the three major technology areas listed earlier. Our focus is on scenarios where the hidden hand played a detectable role.

A. Transactions and related technologies.

The early days of data storage systems were characterized by the exploration of a great range of design paradigms and methodologies. Two major innovations eventually swept most other work to the side: relational databases and the relational query algebra, and the transactional computing

model. We'll focus on the latter [7][4].

Transactions were a response to conflicting tensions. Database platforms need high levels of concurrency for reasons of performance, implying that operations on servers must be interleaved. Yet programmers find it very difficult to write correct concurrent code. Transactions and other notions of atomicity offered a solution to the developer: under the (non-trivial!) assumption that data is stored in persistent objects identifiable to the system (database records, Java beans, etc), transactions offer a way to write programs "as if" each application runs on an idle system. Transactions also offer a speculation mechanism, through the option of aborting a partially executed transaction. Finally, they use abort as a simple and powerful fault-tolerance solution.

Transactions have been a phenomenal market success and are a dominant programming model for applications where the separation of data and code is practical. However, not all programs admit the necessary code-data separation. Overheads are sometimes high, and scalability of transactional servers has been a challenge. Thus attempts to import the transactional model into a broader range of distributed computing settings, mostly in the 1980's, had limited success. Users rejected the constrained world this demanded.

To highlight one case in which a market failure seems to have impacted the transactional community: transactions that operate on multiple objects in distinct subsystems give rise to what is called the nested transaction model, and typically must terminate with a costly two or three-phase commit. Most distributed systems require this mechanism, and it is also used when replicating transactional servers. Multi-phase commit is well understood and, at least for a while, was widely available in commercial products. But products using these features were often balky, unwieldy and costly. Today, popular standards such as Web Services transactions permit the use of complex nested transactions, but few products do so. Most only offer singleton atomic actions side by side with other options (such as "business transactions", a form of scripting tool). The market, in effect, rejected the casual use of transactions that access multiple servers.

In the next two sections, we'll look more closely at reliability options for applications that have traditionally been unable to use transactions. Examples include time-critical services, lightweight applications with in-memory data structures and distributed programs that cooperate directly through message passing (as opposed to doing so indirectly, through a shared database).

B. Reliable multicast.

Three of the authors were contributors in the field of reliable multicast technologies. For consistency, we'll summarize this work in light of the transactional model. Imagine a system in which groups of processes collaborate to replicate data, which they update by means of messages multicast to the members of the group (typically, either to *all operational members*, or to a *quorum*). Such a multicast must read the group membership (the current *view*), then deliver a copy of the update to the members in accordance with desired ordering. We can visualize this as a lightweight transaction in which the view of the group is first read and data at the relevant members is then written. Joins and leaves are a form of transaction on the group view, and reads either access local data or fetch data from a suitable quorum.

In our past work, we invented and elaborated a group computing model called *virtual synchrony* [1] that formalizes the style of computing just summarized. The term refers to the fact that, like the transactional model, there is a simple and elegant way to conceive of a virtually synchronous execution: it seems as if all group members see the group state evolve through an identical sequence of events: membership changes, updates to replicated data, failures, etc. The execution "looks" synchronous, much as a transactional execution "looks" serial. On the other hand, a form of very high-speed ordered multicast can be used for updates, and this can sometimes interleave the delivery of messages much as a transactional execution can sometimes interleave the execution of operations.

Virtual synchrony has been a moderate success in the harsh world of commercial markets for reliability. One of the authors (Birman) founded a company, and virtual synchrony software it marketed can still be found in settings such as the New York Stock Exchange and the Swiss Exchange, the French Air Traffic Control System, and the US Navy's AEGIS warship. Other virtual synchrony solutions that we helped design and develop are used to provide session-state fault-tolerance in IBM's flagship Websphere product, and in Microsoft's Windows Clustering product. We are aware of two end-user development platforms that currently employ this model, and the CORBA fault-tolerance standard uses a subset of it.

The strength of virtual synchrony isn't so much that it offers a strong model, but that it makes it relatively easy to replicate data in applications that don't match the transactional model. One can, for example, replicate the data associated with an air traffic control sector, with extremely strong guarantees that updates will be applied in a consistent manner, faults will be reported and reacted to in a coordinated way, and the system will achieve extremely high levels of availability. These guarantees can be reduced to mathematics and one can go even further with the help of theorem proving tools, by constructing rigorous proofs that protocols as coded correctly implement the model.

Yet virtual synchrony was never a market success in the sense of transactions. End users who wish to use virtual synchrony solutions have few options today: while the technology is used in some settings and hidden inside some major products, very few end-user products offer this model and those products have not been widely adopted.

Virtual synchrony has a sibling that suffered a similar market failure. Shortly after virtual synchrony was introduced, Leslie Lamport proposed Paxos, a practical implementation of his *state machine model*, which also guarantees that processes in a system will see identical events

in an identical order. Over time, he implemented and optimized the Paxos system, and it was used in some research projects, for example to support a file system.

Paxos is closely related to virtual synchrony: there is a rough equivalence between the protocols used to implement group views (particularly for networks that experience *partitioning*) and the Paxos data replication model. Paxos updates guarantee stronger reliability properties than virtual synchrony multicast normally provides, but at a cost: Paxos is slower and scales poorly relative to most virtual synchrony implementations. Like virtual synchrony, Paxos never achieved broad commercial impact.

Thus, we see reliable multicast as a field in which a tremendous amount is known, and has been reduced to high quality products more than once, yet that seems plagued by market failure. We'll offer some thoughts about why this proved to be the case in Section IV.

C. Theory of distributed computing

Finally, consider the broad area of theoretical work on distributed computing. The area has spawned literally thousands of articles and papers, including many fundamental results widely cited as classics. Despite its successes, the theory of distributed computing has had surprisingly little direct impact. Unlike most branches of applied math, where one often sees mathematical treatments motivated by practical problems, the theory of distributed computing has often seemed almost disconnected from the problems that inspired the research. Moreover, while applied mathematics is often useful in practical settings, distributed systems theory is widely ignored by practitioners. Why is this? Two cases may be helpful in answering this question.

1) Asynchronous model.

The asynchronous computing model was introduced as a way of describing as simple a communicating system as possible, with the goal of offering practical solutions that could be reasoned about in a simple context but ported into real systems. In this model, processes communicate with message passing, but there is no notion of time, or of timeout. A message from an operational source to an operational destination will eventually be delivered, but there is no sense in which we can say that this occurs "promptly". Failed processes halt silently.

The asynchronous model yielded practical techniques for solving such problems as tracking potential causality, detecting deadlock or other stable system properties, coordinating the creation of checkpoints to avoid cascaded rollback, and so forth. One sometimes sees real-world systems that use these techniques, hence the approach is of some practical value.

But the greatest success of the asynchronous model is also associated with a profound market failure. We refer to the body of work on the *asynchronous consensus* problem, a foundational result with implications for a wide range of questions that involve agreement upon a property in an asynchronous system. In a seminal result, it was shown that asynchronous consensus is impossible in the presence of faults unless a system can accurately distinguish real crashes from transient network disconnections [3].

Before saying more, we should clarify the nature of the term "impossible" as used above. Whereas most practitioners would say that a problem is impossible if it can never be solved (for example, "it is impossible to drive my car from Ithaca to Montreal on a single tank of gas"), this is not the definition used by the consensus community. They take *impossible* to mean "can't always be solved" and what was actually demonstrated is that certain patterns of message delay, if perfectly correlated to the actions of processes running an agreement protocol, can indefinitely delay those processes from reaching agreement. For example, a fault-tolerant leader election protocol might be tricked into never quite electing a leader by an endless succession of temporary network disconnections that mimic crashes. As a practical matter, these patterns of message delay are extremely improbable – less likely, for example, than software bugs, hardware faults, power outages, etc. Yet they are central to establishing the impossibility of guaranteeing that consensus will be reached in bounded time.

FLP is of tremendous importance to the theoretical community. Indeed, the result is arguably the most profound discovery to date in this area. Yet the impossibility result has been a source of confusion among practitioners, particularly those with just a dash of exposure to the formal side of distributed computing, a topic explored in [5]. Many understand it as claiming that it is impossible to build a computer system robust against failures – an obvious absurdity, because they build such systems all the time! For example, it is easy to solve consensus as an algorithm expressed over reliable multicast. In effect, a multicast platform can be asked to solve an impossible problem! Of course, the correct interpretation is a different matter; just as the systems they build can't survive the kinds of real-world problems listed above, FLP simply tells us that they can't overcome certain unlikely delay patterns. But few grasp this subtlety.

This isn't the only cause for confusion. One can question the very premise of proving the impossibility of something in a model that is, after all, oversimplified. In real systems, we have all sorts of "power" denied by the asynchronous model. We can talk about probabilities for many kinds of events, can often predict communication latencies (to a degree), and can exploit communication primitives such as hardware-supported multicast, cryptography, and so forth. Things that are impossible without such options are sometimes possible once they are available, and yet the theory community has often skipped the step of exploring such possibilities.

2) Synchronous model.

The confusion extends to a second widely used theoretical model: the *synchronous model*, in which we strengthen the communications model in unrealistic ways. This is a model in which the entire system executes in rounds, with all messages sent by correct processes in a round received by all other

correct processes at the outset of the next round, clocks are perfectly synchronized, and crashes are easily detectable. On the other hand the failure model includes *Byzantine failures*, namely cases in which some number of processes fail not by crashing, but rather by malfunctioning in arbitrary, malicious and coordinated ways that presume perfect knowledge of the overall system state.

A substantial body of theory exists for these kinds of systems, including some impossibility results and some algorithms. But even for problems where the Byzantine model makes sense, the unrealistic aspects of the synchronous model prevent users from applying these results directly. For many years, Byzantine Agreement was therefore of purely theoretical interest: a fascinating mathematical result.

Byzantine Agreement has experienced a revival illustrative of the central thesis of this paper. Recently, Castro and Liskov [2] and others reformulated the Byzantine model in realistic network settings, then solved the problem to build ultra-defensible servers that can tolerate not just attacks on subsets of members, but even the compromise (e.g. by a virus or intruder) of some subset of their component processes. Given the prevalence of viruses and spyware, this new approach to Byzantine Agreement is finding some commercial interest, albeit in a small market. In effect, by revisiting the problem in a more realistic context, this research has established its practical value.

IV. DISCUSSION

We started with a review of transactions, a reliability technology that had enormous impact, yielding multiple Turing Award winners, a thriving multi-billion dollar industry, and a wide range of remarkably robust solutions – within the constraints mentioned earlier. Attempts to shoe-horn problems that don't fit well within those constraints, on the other hand, met with market failures: products that have been rejected as too costly in terms of performance impact, awkward to implement or maintain, or perhaps too costly in the literal sense of requiring the purchase of a product (in this case, a transactional database product) that seems unnecessary or unnatural in the context where it will be used.

The discussion of reliable multicast pointed to a second, more complex situation. Here, a technology emerged, became quite real, and yielded some products that were ultimately used very successfully. Yet the area suffered a market failure nonetheless; while there are some companies still selling products in this space, nobody would claim a major success in the sense of transactional systems.

Our experience suggests that several factors contributed to the market failure for multicast platforms:

1. Multicast products have often been presented as low-level mechanisms similar to operating system features for accessing network devices. As researchers, this is natural: multicast is a networking technology. But modern developers are shielded from the network by high level tools such as remote procedure call – they don't work directly with the O/S interfaces. Thus multicast presentation has been too primitive.
2. These products were often quirky, making them hard to use. For example, our own systems from the 1990's had scalability limitations in some dimensions that users found surprising – numbers of members of groups, or numbers of groups to which a single process could belong, and performance would collapse if these limits were exceeded. Had we addressed these limitations, those systems might have been more successful.
3. The field advanced, first pushing towards object-oriented platform standards such as CORBA, and then more recently towards service oriented architectures such as Web Services. Multicast solutions didn't really follow these events – with the notable exception of the CORBA fault-tolerance standard. But the CORBA standard was a peculiar and constraining technology. It was limited to an extremely narrow problem: lock step replication of identical, deterministic server processes. This determinism assumption is limiting; for example, it precludes the use of any sort of library that could be multithreaded (an issue even if the application using the library is single-threaded), and precludes applications that receive input from multiple sources, read clocks or other system counters, etc. In practice such limits proved to be unacceptable to most users.
4. The products developed for this market were forced to target a relatively small potential customer base. The issue here is that replication arises primarily on servers, and at least until recently, data centers have generally not included large numbers of servers. Thus purely from a perspective of the number of licenses that can potentially be sold, the market is comparatively small.
5. These products have not made a strong case that developers who use them will gain direct economic benefit – a lower total cost of application development and ownership – relative to developers who do not use them. We believe that such a case can be made (as explained below), but this was not a priority for the research community and this has left developers facing a "reliability tax" – an apparent cost that must be born to achieve reliability.
6. Product pricing was too high. Here we run into a complex issue, perhaps too complex for this brief analysis. In a nutshell, to support the necessary structure around any product (advanced development, Q/A, support) a company needs a certain size of staff, typically roughly proportional to the complexity of the code base. Multicast is not a simple technology and the code bases in question aren't small or easy to test, particularly in light of the practical limitations mentioned in points 1 and 2, which made these products unstable for some uses. In effect, they are expensive products to develop and maintain.

Now consider prices from the customer's perspective. A data replication framework is a useful thing, but not

remotely as powerful a technology as, say, an operating system or a database system. Thus one should think in terms of pricing limited to some small percentage of the licensing cost of a database or operating system for the same nodes. But pricing for operating systems and databases reflects their much larger markets: a revenue stream is, after all, the product of the per-unit price times the numbers of units that will be sold. Thus multicast products are limited to a small fraction of the number of machines, and a small fraction of the product pricing, of a database or similar product – and this is not a level of income that could support a thriving, vibrant company. But if vendors demand higher per-unit licensing, customers simply refuse to buy the product, seeing the cost of reliability as being unrealistically steep!

7. These products have often demanded substantial additional hand-holding and development services. In some sense this isn't a bad thing: many companies make the majority of their income on such services, and indeed services revenue is what kept the handful of multicast product vendors afloat. And it may not be an inevitable thing; one can easily speculate that with more investment, better products could be developed. Yet the current situation is such that the size of the accessible market will be proportional to the number of employees that the company can find and train, and that most product sales will require a great deal of negotiation, far from the "cellophane-wrapped" model typical of the most successful software products.

In summary, then, the ingredients for a market failure are well established in this domain. The bottom line is that for solutions to ever gain much permanence, they would need to originate with the major platform vendors, and be viewed as a competitive strength for their products. This has not yet occurred, although we do believe that the growing popularity of massive clusters and data centers may shift the competitive picture in ways that would favor products from the vendors. The issue here is that building scalable applications able to exploit this price point involves replicating data so that queries can be load-balanced over multiple servers. This is creating demand for replication solutions – and hence opening the door for technologies that also promote fault-tolerance, security, or other QoS properties.

Yet, having worked with vendors of products in this area, we've also noted commercial *disincentives* for commoditized reliability. The issue is that many platform vendors differentiate their systems-building products by pointing to the robustness of their components. They offer the value proposition, in effect, that by executing an application on their proprietary product line, the user will achieve robustness not otherwise feasible. If end-users can build robust distributed systems without needing expensive reliability platforms, platform vendors might lose more revenue through decreased sales of their high-margin robust components than they can gain by licensing the software supporting application-level solutions.

And what of the theoretical work? Here, we believe that researchers need to recognize that theory has two kinds of markets. One is associated with the community of theoreticians: work undertaken in the hope of shedding light on deep questions of fundamental importance and of influencing future theoretical thinking. Impact on the commercial sector should not be used as a metric in evaluating such results.

But we also believe that the experience with practical Byzantine solutions points to an avenue by which the theory community can have substantial impact. The trick is to tackle a hard practical problem that enables a completely new kind of product. We believe this lesson can be applied in other settings. For example, real systems are stochastic in many senses. The research community would find a rich source of hard problems by looking more explicitly at probabilistic problem statements framed in settings where networks and systems admit stochastic descriptions. Results translate fairly directly to real-world settings and would be likely to find commercial uptake.

V. RECOMMENDATIONS

Not every problem is solvable, and it is not at all clear to us that the market failure in our domain will soon be eliminated. However, we see reason for hope in the trends towards data centers mentioned earlier, which are increasing the real value of tools, but want to offer some observations:

- Researchers need to learn to listen to consumers. On the other hand, one must listen with discernment, because not every needy developer represents a big opportunity. This is particularly difficult during dramatic paradigm shifts, such as the current move to net-centric computing. When such events occur, it becomes critical to focus on early visionaries and leaders, without being distracted by the larger number of users who are simply having trouble with the technology.
- We haven't made an adequate effort to speak the same language as our potential users, or even as one-another. For example, if our users think of systems in stochastic terms, we should learn to formalize that model and to offer stochastic solutions.
- Practical researchers have often put forward solutions that omit big parts of the story, by demonstrating a technique in an isolated and not very realistic experimental context. Vendors and developers then find that even where the technology is an exciting match to a real need, bridging the resulting gaps isn't easy to do. In effect we too often toss solutions over the fence without noticing quite how high the fence happens to be and leaving hard practical questions completely unaddressed. Only some vendors and developers are capable of solving the resulting problems.
- We need greater attention to our value propositions, which are too often weak, or poorly articulated. Technology success is far more often determined by

economic considerations than by the innate value of reliability or other properties.

- Our work is too often ignorant of real-world constraints and of properties of real-world platforms. For example, if a solution is expensive to deploy or costly to manage, potential users may reject it despite strong technical benefits.
- Our community has been far too fond of problems that are either artificial, or that reflect deeply unrealistic assumptions. The large body of work on compartmentalized security models is an example of this phenomenon.
- Real users seek a technology "process" not an "artifact", hence those of who develop technical artifacts need to be realistic about the low likelihood that conservative, serious users will adopt them.
- Concerns about intellectual property rights have begun to cloud the dialog between academic and commercial researchers. The worry is typically that an academic paper, seemingly unfettered by IP restrictions, might actually be the basis of an undisclosed patent application. If that patent later issues, any company that openly adopted the idea faces costly licensing. Hence companies either avoid dialog with academic researchers or limit themselves to listening without comment, lest they increase their exposure.

These observations lead to a few recommendations, which we focus on work aimed at the real world:

- Developers need to build demonstrations using real platforms if at all possible, and ideally to evaluate them in realistic scenarios.
- Results should make an effort to stress value in terms buyers will understand from an economic perspective. Even research papers should strive to show a credible value proposition.
- If the development team isn't in a position to provide long term product continuity, development and support, it should try to disseminate solutions via vendors, or to work with vendors on transitioning.
- Academic research groups should work with their University licensing officers to try to clarify and standardize the handling of software patents that the University might seek, in the hope that industry teams considering dialog with academic research groups will see IP ownership issues as less of a threat. The huge success of the Berkeley Unix project and its BSD licensing approach is a model that other academic research teams might wish to study and try to emulate.

VI. CONCLUSIONS

We've reviewed market forces that can have a dramatic impact on the ultimate fate of technologies for reliable computing, with emphasis on technical areas in which the authors have had direct involvement. Our review led to several kinds of insights. One somewhat obvious insight is that not all forms of academic research are of a nature to impact the commercial market: some work, for example, demonstrates the feasibility of solving a problem, and yet can't possibly be offered as a free-standing product because one couldn't conceivably generate a revenue stream adequate to support a sensible development and support process. Customers buy into a company's vision and process – it is rare to purchase a product and never interact with the company again. Teams that don't plan to create such a process shouldn't expect to have commercial impact. Yet academic groups are poorly equipped to offer support.

A second broad class of insights relate to the way that we pose problems in the reliability arena, demonstrate solutions, and evaluate them. We've argued that even purely academic researchers should pose problems in ways that relate directly to realistic requirements, demonstrate solutions in the context of widely used platforms, and evaluate solutions in terms that establish a credible value proposition.

A third category of suggestions boil down to the recommendation that researchers should ponder market considerations when trying to identify important areas for future study. This paper pointed to two examples of this sort –recent work on practical Byzantine agreement, and the exploration of stochastic system models and stochastic reliability objectives, arguing that these are both more realistic and also might offer the potential for significant progress. But many problems of a like nature can be identified. Others include time-critical services ("fast response" as opposed to "real-time"), scalability, and trustworthy computing (construed broadly to include more than just security).

Finally, we've suggested that the reliability community would do well to heal the divisions between its theoretical and practical sub-areas. While academic debate is fun, often passionate, and can lead to profound insights, we need to communicate with external practitioners in a more coherent manner. The failure to do so has certainly contributed to the market failures that have heretofore marked our field.

VII. REFERENCES

[1] K.P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Springer; 1 edition (March 25, 2005)

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4):398--461, November 2002.

[3] MJ Fischer, NA Lynch, and MS Paterson. Impossibility of distributed consensus with one faulty process. JACM, 32(2):374--382, 1985.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[5] R Guerraoui, A Schiper. Consensus: The Big Misunderstanding. Proceedings of the 6th IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)

[6] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, 4 (3), pp.382-401

[7] D. Lomet. Process Structuring, Synchronization and Recovery Using Atomic Actions. ACM Conference on Language Design for Reliable Software, Raleigh NC. SIGPLAN Notices 12, 3 (March 1977) 128-137.