Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System

Eric Missimer*, Richard West and Ye Li

Computer Science Department
Boston University
Boston, MA 02215
*Email: {missimer,richwest,liye}@cs.bu.edu *VMware, Inc.*

***Abstract*—This paper presents different approaches for real-time fault tolerance using redundancy methods for multi-core systems. Using hardware virtualization, a *distributed system on a chip* is created, where the cores are isolated from one another except through explicit communication channels. Using this system architecture, redundant tasks that would typically be run on separate processors can be consolidated onto a single multi-core processor while still maintaining high confidence of system reliability. A multi-core chip-level distributed system could therefore offer an alternative to traditional automotive systems, for example, which typically use a controller area network such as CAN bus to interconnect multiple electronic control units. Using memory as the explicit communication channel, new recovery techniques that require higher bandwidths and lower latencies than those of traditional networks, now become viable. In this work, we discuss several such techniques we are considering in our chip-level distributed system called Quest-V.**

I. Introduction

Fault-tolerance in real-time systems has historically been accomplished through redundancy in both hardware and software [1]–[4]. For example Briere et al. explain how in the Airbus A320/330/340 there are "sufficient redundancies to provide the nominal performance and safety levels with one failed computer, while it is still possible to fly the aircraft safely with one single computer active [5]." Redundancy across multiple compute nodes protects against both hardware and software failures. We propose a technique that uses hardware virtualization to sandbox each core of a multi-core processor, along with a subset of machine physical memory and a collection of I/O devices.

This design allows the system to operate as a *separation kernel* [6], where communication between sandboxes occurs through explicit memory channels and inter-processor interrupts. Each sandbox is isolated from the software faults that can occur in another sandbox and any hardware faults that are specific to a single core or subset of memory. This allows the consolidation of computational tasks onto a single multi-core processor while still maintaining a large number of the advantages associated with a traditional distributed system. This approach also has the advantage of having a higher bandwidth communication channel, i.e. memory, than in traditional control networks such as CAN, enabling new recovery techniques. In this paper we focus specifically on different Triple Modular Redundancy [7], [8] (TMR) techniques that can be applied to a virtualized distributed system on a chip.

The next section provides a brief summary of Quest-V, a separation kernel we have been developing for use in real-time mixed-criticality systems [9]. Section III discusses our generic fault *detection* approach. Section IV introduces three fault *tolerance* methods we are currently considering for Quest-V, and each approach is described in detail in Sections V- VII. This is followed by a discussion of our generic fault *recovery* technique in Section VIII. Related work is discussed in Section IX followed by the conclusions and future work in Section X.

II. Quest-V Design

Quest-V is designed for real-time high confidence systems. It is written from scratch for Intel's x86 architecture, and operates as a *distributed system on a chip*. Quest-V uses hardware virtualization technology to partition a one or more cores, a region of machine physical memory and a subset of I/O devices into separate *sandboxes*. Intel's extended page table (EPT) feature on modern VT-x processors is used to partition memory into separate regions for different sandboxes [1].

Hardware virtualization assists in the design of virtual machine monitors (VMMs, a.k.a. hypervisors), which are capable of managing multiple virtual machines (VMs) on the same physical machine. Unlike in traditional VMMs, Quest-V does not require the use of a monitor to schedule VMs on processor cores. Instead, each sandbox runs its own kernel that performs (sandbox-local) scheduling directly on available processor cores.

[1] AMD x86_64 and ARM Cortex hardware virtualization extensions provide a similar functionality.

This approach has numerous advantages. First, because a trusted monitor is not performing VM scheduling, it is eliminated from the normal operation of the system. This results in very few VM exits, resulting in lower overheads due to virtualization. Second, this simplifies the design of a monitor, which now only needs to initialize sandboxes (or, equivalently, VMs) [2], establish shared memory communication channels between sandboxes, and assist when necessary in fault recovery. The reduced complexity of Quest-V's monitor code makes it more easily verified for correctness.

III. Fault Detection

Before we discuss different possible TMR configurations in Quest-V, we will briefly discuss how we detect faults. This approach is used throughout all the subsequently described system configurations. TMR uses a majority voting mechanism to detect an error. In the traditional TMR setup, the results of redundant (replicated) computations are sent to the voter. While we could use this approach, we have decided to use a more aggressive fault detection technique that can detect any deviation in the redundant computation, not only the result. Our approach takes hashes of memory on a per-page basis of all memory modified by the program between synchronization points. We also take a summary hash of all the modified memory. The voter first compares the summary hashes. If the summary hash values are identical, no further error detection actions are taken. If the summary hashes are not identical, the voter iterates through the per-page hashes to determine which pages are different. This allows for faster recovery, described in detail in Section VIII. Currently, we are using the Jenkins one-at-a-time [10] hash function due to its simplicity. For added security, where a malicious sandbox could corrupt a page so that the hash would be the same as the valid page, a cryptographically secure hash function could be used.

Taking consensus on memory hashes is a generic fault detection approach as it just relies on the program's user-space state or the entire sandbox being identical across all instances. This also places a restriction on the types of programs that can be monitored. Their execution across sandboxes must be identical during a fault-free execution. The task or guest can only rely on its own internal state and any data passed to the program through shared memory or by the hypervisor. Redundant tasks or guest code must not make use of process-specific values such as process IDs, unless they are identically replicated, and similarly should not use constructs such as `gettimeofday()` which might differ across replicas. We do not believe this restriction is too prohibitive; however, we plan to remove this restriction in future work.

During execution, the redundant guests or task instances reach specific synchronization points. Depending on whether the redundancy is for the entire sandbox (guest) or only a single task determines whether the synchronization points are when a VM exit occurs or when the replicated task makes a system call. This influences whether synchronization is handled in the hypervisor or a guest kernel. At the synchronization point, the hypervisor or kernel determines which pages have been modified and creates the necessary per-page hashes and summary hash. The memory management unit can be used to help track which pages have been modified as the pages can initially be marked as read only to cause hypervisor or kernel traps on attempted page updates. The hashes are sent to the voter and the sandbox/task continues execution. We do not halt execution, i.e. barrier synchronize the tasks, and wait for the results of the voter as this would add unnecessary overheads. Instead, no external action, e.g. output to an actuator, is taken on behalf of the redundant copies until a majority of identical hashes have been collected. Once enough hashes have been collected and verified, any necessary I/O is performed and the hashes are released.

If an error has occurred but there is still a majority consensus, any I/O is performed, and the hypervisor or kernel is notified as part of fault recovery (see Section VIII). If there is no consensus, an application dependent recovery procedure can be used to bring the system into a useful state. For example, for an autonomous automobile application, the system could safely bring the vehicle to a halt. To detect performance faults, where a sandbox or task does not reach a synchronization point, timeout values can be specified by the application developer. If a synchronization point is not reached within the timeout value it is treated as a fault and the same generic recover procedure is used to correct the performance failure.

IV. Virtualization Based Triple Modular Redundancy

The chip-level distributed system design of Quest-V creates an opportunity to develop new fault tolerance techniques. For example, techniques that exploit high bandwidth, low latency communication between sandboxes can be explored. In this section, we will introduce various techniques and we will highlight their strengths and weaknesses in Sections V, VI and VII.

Redundancy in either data and/or execution can be used to detect Byzantine errors, e.g. soft errors causing bit-flips, possibly as a result of the system being exposed to radiation. Separate Quest-V sandboxes can support

[2]Initializing a sandbox requires setting up an appropriate hardware partition of processor cores, memory and I/O devices. Devices themselves can be shared but resource partitioning is typically done only once, at boot-time.

redundant executions of a process or guest. Whenever an external action needs to be taken, e.g. sending a message to an actuator, a consensus mechanism, such as Triple Modular Redundancy [7], [8] (TMR), can be used to ensure that faulty values do not propagate to the device. We will focus our discussion of fault tolerance techniques in Quest-V on those that follow the TMR approach. In what follows, we describe three different fault tolerant Quest-V system configurations, which depend on where the voting mechanism and device driver are located:

- **Voting mechanism and device driver in the hypervisor** – Each process submits its results to the hypervisor via a hypercall or through emulated devices. The hypervisor waits for the results or a timeout, compares the results and sends the majority to the device.
- **Voting mechanism and device driver in one sandbox** – A single sandbox acting as an *arbitrator* has sole access to the device. The arbitrator is responsible for comparing the results of redundant computations, which are distributed across the other *computation* sandboxes. Communication between the arbitrator and each computation sandbox is via a separate shared memory channel, which is protected by extended page table (EPT) mappings. That is, no two computation sandboxes can access each other's private channel to the arbitrator, thereby preventing a faulty computation sandbox from corrupting the results of another sandbox.
- **Voting mechanism distributed across sandboxes and device driver is shared** – Each sandbox that contains a redundant process also has shared access to the device. Each redundant process compares its own state to the redundant copies to determine if a fault has occurred and recovers if necessary. The non-faulty sandboxes elect a leader to send the data to the device. Communication occurs in pair-wised private shared memory channels as described in the previous configuration.

In the following three sections, we discuss further details of our proposed approach for each of the above configurations.

A. Voter – Single Point of Failure

The final voter in a TMR setup can be a single point of failure. The voter could malfunction and select the minority result or simply output a different result than its inputs. All the configurations above could suffer from the voter malfunctioning. A common solution is to use three voters [11] and the output of each voter is the input to the next stage of the computation. However, eventually a single output that is to be sent to the device needs to be determined (assuming the device does not support three results and will perform its own TMR). Techniques have been developed to protect single points of failure such as voters. For example, Ulbrich et. al used arithmetic encoding techniques to protect the voters in TMR [12]. Furthermore, while the redundant sandboxes might not have direct access to the device in the first and second configurations described above, they could have read-only access to ensure that the voting mechanism is behaving correctly. If the redundant sandboxes reach a consensus that the voting mechanism is behaving incorrectly the device driver and voter could be re-initialized or the device could be mapped to a new sandbox. We plan to explore these techniques in future work.

V. Hypervisor Voting and I/O

In this configuration, the voting mechanism and device driver are located in the hypervisor. This conflicts with the design philosophy of Quest-V, as the hypervisor should ideally be as simple as possible. Furthermore, placing a device driver in the hypervisor could make the entire system vulnerable if the device driver is incorrectly implemented. However, if we overlook this, TMR fault tolerance can be applied to operating systems other than Quest-V, e.g. Linux, without the need to modify any source code. Specifically, the hypervisor could host three or more redundant guests that communicate the results of the computation through an emulated I/O device. Besides voting and performing I/O, the hypervisor is responsible for encapsulating each guest to ensure that they remain in loose lockstep. This approach is depicted in Figure 1.

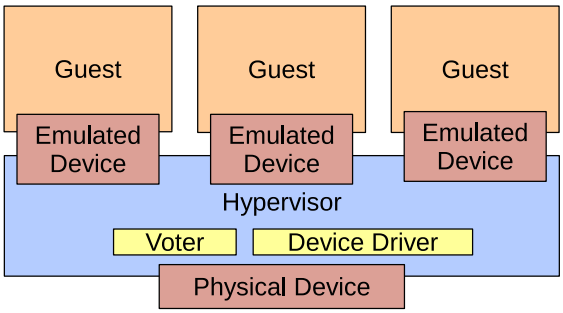

Fig. 1: Voting mechanism and device driver in the hypervisor.

This approach builds upon hypervisor-based fault tolerance (HBFT) [13]–[16]. HBFT is an example of a primary-backup based fault tolerance system that uses a hypervisor to encapsulate a guest virtual machine so the state of the entire system can be easily manipulated and saved. This allows the primary and backup, which run on different machines, to be easily synchronized without relying on any information from, or modification of, the operating system or application. The execution of

the guests is divided into time epochs. At each epoch, the primary and backup are ensured that they are in the same state. This is either accomplished by having an active backup that is kept in lockstep [13], having a dormant backup and transferring the changed state from the primary to the backup [14], or having an active backup that is not kept in lockstep and any discrepancies between the two executions are handled at the epoch boundaries [15]. During execution, the hypervisor buffers all output until an epoch has been reached and the primary and backup have been synchronized. This ensures that if the primary fails, the backup can perform the same output without duplicated output being seen outside of the system.

HBFT, as it stands, can only handle crash failures, where the primary node halts when it fails [17], [18]. If the primary fails, the backup guest begins execution to ensure high availability. The HBFT model can be extended to recover from Byzantine errors by adding a third redundant guest. At epoch boundaries, i.e. synchronization points, the hypervisor examines the state of each guest and if a single guest differs, the state can be corrected. This approach would not require any modification of the guest operating system or application and could even be applied to closed-source software. By combining all the guests onto a single multi-core processor, we cannot recover from errors such as power failures that bring down the entire processor, but we can recover from Byzantine errors in a much more efficient manner as data can be transferred between guests at a much faster rate.

This approach does have some disadvantages compared to the other two approaches. First, as previously mentioned, the hypervisor must be much more advanced. For example, it must support emulated devices and be able to recover the entire guest operating system. Also, every sandbox must be performing the exact same operations. It is not possible for one sandbox to run non-safety critical tasks that the other sandboxes do not run as the hypervisor has no information about what state belongs to which task. We will see in the next two sections that this limitation does not apply to the other approaches.

VI. Arbitrator Sandbox

In this configuration, the voting mechanism and device driver are located in an *arbitrator* sandbox. The redundant computations are performed in three or more guests as depicted in Figure 2. Communication between guests is explicit through shared memory channels. These could be set up statically by the hypervisor at boot-time, or dynamically, at run-time. Unlike the approach described in Section V, the fault tolerance is at the application-level as opposed to the entire guest. This has the advantage that each guest does not need to be identical, e.g. only a subset of the applications running in each guest need to be replicated. Applications that are not safety-critical can be executed in just one guest. This approach requires operating system support, to hash the application pages and communicate the results to the arbitrator sandbox.

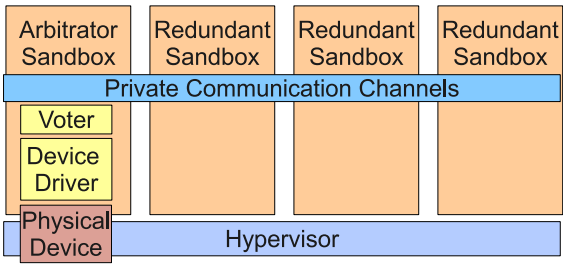

Fig. 2: Voting mechanism and device driver in an arbitrator sandbox.

The main advantage to this approach is that the hypervisor remains as simple as possible. The hypervisor does not have to emulate any devices, nor does it contain any device drivers. All necessary devices are isolated from the redundant guests via hardware virtualization extensions, as described in our earlier work [9]. Not only is the hypervisor simple but it is kept out of the control path during normal execution, thus avoiding the costs of VM exits. Only during recovery would it be necessary to drop down into the hypervisor. Also, as previously stated, the granularity of redundancy is much finer as the redundancy is at the application-level as opposed to the guest-level.

Having a single sandbox vote and send the results to the device driver does have its limitations. First, a sandbox is necessary to contain the voting task. While the task performing the voting would require a low overall utilization, the other approaches do not require a separate sandbox just for voting. Another limitation is that while we gain the ability to do task-level redundancy we do so at the cost of having to add the redundancy support into the operating system. While this is possible for an operating system such as Quest-V, it becomes increasingly difficult for a more complex system such as Linux.

VII. Shared Voting

A third configuration is to have the voting process and device shared across all sandboxes. This approach is similar to the second approach in that it has a smaller granularity of redundancy, e.g. at the application level, and therefore the redundant sandboxes do not have to be executing identical sets of tasks or operate in lockstep. It also requires operating system support to hash memory pages and communicate the results to different sandboxes. This approach avoids the need for

a special arbitrator sandbox. Each sandbox takes a vote on the results of the other sandboxes, again communicating through private shared memory channels. Of the sandboxes that have a value equal to the majority vote, a temporary leader is elected, which performs the actual I/O. This approach is depicted in Figure 3.

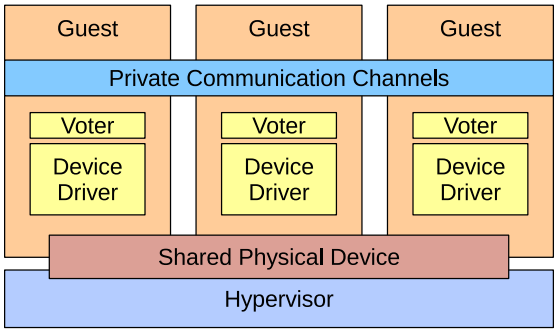

Fig. 3: Voting mechanism and device driver are shared across sandboxes.

There are two possible ways that the shared driver could be implemented. One is that the device is directly mapped to all participating sandboxes all the time. This has obvious security and safety concerns as a faulty sandbox could at any time send erroneous output signals through the device. As previously mentioned in Section IV, the other sandboxes could be monitoring the memory associated with the device to determine if such behavior occurs and if so, signal to the hypervisor that the device should be unmapped from the faulty sandbox. This could be acceptable if a few erroneous I/O messages were permitted and the device could be reinitialized.

A more secure method would be the following: whenever an I/O operation needed to be performed, each sandbox sends to the hypervisor the sandbox it wishes to be the leader to perform the I/O operations. The hypervisor then temporarily grants that sandbox the write privileges to the device while still permitting the other sandboxes to have read access. Once the I/O operation is complete the sandboxes signal that the write privileges should be revoked. The granting and revoking of write privileges performed by the hypervisor only occurs if a majority of the sandboxes signal that it should occur. In this way, a sandbox would have to fail specifically after it was granted write privileges and before the privileges were revoked. Such an approach requires some support from the hypervisor as the hypervisor has to be able to dynamically map and unmap a device to different sandboxes. It would also involve a large number of VM exits which would be required to temporarily grant and then later remove access to the device. Also, how interrupt service routines should be handled is unclear as they often require direct access to the device and occur asynchronously.

VIII. Recovery

We have a few key requirements for our recovery procedure. First, it should be as generic as possible, so that it can be used across multiple applications. Second, the recovery procedure should be applicable to both an entire operating system running within a sandbox and to a task running within an operating system. Obviously there will be some differences, mostly complications due to operating system recovery, but the general approach should be the same. This allows us to share a similar code base between operating system and task recovery, reducing the code base size and the possibility of errors.

Initially, our recovery procedure involved taking snapshots of the changed state at each synchronization point, similar to a rollback procedure. However, instead of rolling back to the last known good state we would instead roll forward using the snapshot of a correct instance to bring an incorrect instance up to the same state. However, in our preliminary evaluations, the snapshot procedure dominated the overhead associated with recovery to the point that rolling back and rolling forward had nearly identical recovery times. We therefore decided to abandon taking snapshots at synchronization points and perform recovery without them.

The point of taking snapshots was to allow one sandbox or task to be recovered without interfering with the execution of the sandbox that is being used as the correct instance. The snapshot pages could be read without the fear that the correct instance would modify them while the recovery procedure was occurring, as the correct instance would actually be using different memory frames at the time of recovery. If we do not take snapshots, then we run the risk that the correct instance modifies a page while we are copying it for recovery. This is the same issue that occurs during live migrations of virtual machines [19]. The solution is to divide the migration, or in our case, the recovery procedure into different phases. First, pages are pushed from source (correct instance) to destination (recovering instance), and if a page is modified, it is re-pushed. Second, the source is stopped and pages are copied without the need to be concerned about consistency. Finally, as the migrated virtual machine executes, any pages that have not been pushed are pulled as they become necessary. Different live migration strategies balance these phases. We can use a similar approach to recovery which basically involves performing a live migration on a sandbox or application. However, instead of halting the source instance, we allow it to continue running. We will explore what balance of the three phases is most appropriate for a recovery.

IX. Related Work

Achieving triple modular redundancy through multiple executions of a task has been previously explored by Reinhardt and Mukherjee [20], and Döbel, Härtig and Engel [21]. Reinhardt and Mukherjee developed a simultaneous and redundantly threaded processor, in which hardware is responsible for error checking and recovery. While this alleviates software developers of fault tolerance concerns it also adds extra overhead by replicating components that are not safety-critical. Furthermore, specialized hardware features must be available.

Döbel et al. developed a software based approach to task replication on top of the L4 Fiasco microkernel [21]. Their approach involved a master-controller task, which monitored the execution of redundant tasks. The controller handled CPU exceptions, page faults and system calls made by the redundant tasks and ensured they had identical state at these points. This is similar to our first approach of using a hypervisor as the master controller. As with our other two approaches, it operates on a per-task basis rather than an entire guest.

X. Conclusions and Future Work

We have presented three fault tolerant configurations based on TMR. Such techniques are made possible by the unique design of the Quest-V separation kernel. We have focused on TMR as it is a common fault tolerance mechanism used to handle soft errors.

Beyond implementing and comparing the previously described techniques, one of the main challenges remaining is protecting the voter. As briefly discussed in Section IV, techniques such as arithmetic encoding can be used to protect voters in TMR. We have also discussed the possibility of having redundant sandboxes monitor the results of the voter, by having read-only access to a device driver and signaling the monitor if an error is detected. We plan to compare these techniques as part of (real-time) online fault detection and recovery in Quest-V.

References

[1] A. Avizienis, "The N-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, no. 12, pp. 1491–1501, 1985.

[2] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The Mars approach," *Micro, IEEE*, vol. 9, no. 1, pp. 25–40, 1989.

[3] R. Keichafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *Computers, IEEE Transactions on*, vol. 37, no. 4, pp. 398–404, 1988.

[4] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[5] D. Briere, C. Favre, and P. Traverse, "Electrical flight controls, from airbus a320/330/340 to future military transport aircraft: A family of fault-tolerant systems," in *The Avionics handbook*, C. R. Spitzer, Ed. CRC Press, 2001.

[6] J. Rushby, "The design and verification of secure systems," in *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, CA, Dec. 1981, pp. 12–21.

[7] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[8] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[9] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2014, pp. 201–212.

[10] B. Jenkins, http://burtleburtle.net/bob/hash/doobs.html.

[11] N. Rollins, M. J. Wirthlin, P. Graham, and M. Caffrey, "Evaluating TMR techniques in the presence of single event upsets," in *Military and Aerospace Programmable Logic Devices International Conference*, 2003.

[12] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE, 2012, pp. 49–60.

[13] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 1–11.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 2008, pp. 161–174.

[15] M. Lu and T.-c. Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 534–543.

[16] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[17] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[18] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," 1994.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[20] S. K. Reinhardt and S. S. Mukherjee, *Transient Fault Detection via Simultaneous Multithreading*, ser. ISCA '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/339647.339652

[21] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 83–92.