Survey of Fault Tolerance Techniques for Shared Memory Multicore/Multiprocessor Systems

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

Abstract— **With the advent of modern nano-scale technology, it has become possible to implement multiple processing cores on a single die. The shrinking transistor sizes however have made reliability a concern for such systems as smaller transistors are more prone to permanent as well as transient faults. To reduce the probability of failures of such systems, online fault tolerance techniques can be applied. These techniques need to be efficient as they execute concurrently with applications running on such systems. This paper discusses the challenges involved in online fault tolerance and existing work which tackles these challenges. We classify fault tolerance into four different steps which are proactive fault management, error detection, fault diagnosis and recovery and discuss related work for each step, with focus on techniques for shared memory multicore/multiprocessor systems. We also highlight the additional difficulties in tolerating faults for parallel execution on shared memory multicore/multiprocessor systems.**

I. Introduction

It has become possible to integrate billions of transistors on a single die with modern nano-scale technology and therefore allow many processing cores to be implemented on the same chip. While this advancement allows software with a large level of parallelism to execute very efficiently on such processors, it has also introduced reliability issues as the small transistors are more susceptible to both transient [2] and permanent [18] faults. This necessitates the implementation of efficient and scalable online *fault tolerance* (FT) techniques to reduce the probability of failures of such systems.

Fault tolerance of programs running sequentially on uniprocessors is well understood and many efficient solutions exist for that purpose. On the other hand, programs running in parallel on shared memory multicore processors present a greater challenge due to shared memory accesses, which are a frequent source of non-determinism. This paper gives a survey of work done on fault tolerance with primary focus on work for shared memory multicore systems.

Section II, discusses the basic concepts of a system, faults, failures and fault tolerance. Then we classify fault tolerance into four different steps: *proactive fault management* (discussed in Section III), *error detection* (discussed in Section IV), *fault diagnosis* (discussed in Section V) and *recovery* (discussed in Section VI). Redundant execution for fault tolerance is discussed in Section VII. We finally conclude this paper with Section VIII.

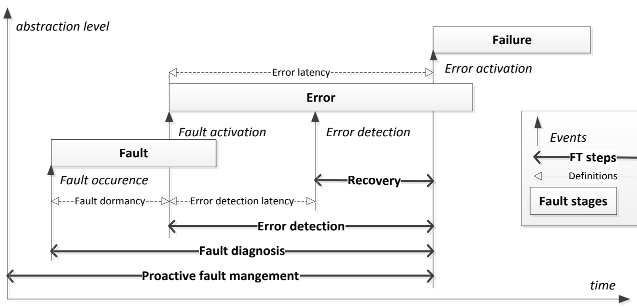

Fig. 1. Fault propagation and fault tolerance

II. Basic Concepts

In this section, we present the basic concepts related to the field of fault tolerance. Our discussion is based on the way a system behaves and interacts with other systems in its environment [38].

A *system* is an entity that interacts with other systems. A system can be hardware based, for example a processor, or software based, such as a running application. A system consists of components which can be systems themselves. The service delivered by a system is its behavior as perceived by other systems using it. The total state of a system is the set of its internal and external states. A system is said to fail when its external state deviates from the correct state. The cause of this failure is **fault**(s) within the system or external to it. Fault propagation is illustrated in Figure 1. When a fault becomes **active**, it would impact the total state of one or more components of the system. The deviation of the total state of a component from the correct state is known as an **error**. When an error propagates to affect the external state of the system, the error is said to be activated. When the error is activated, **failure** of the system is said to have occurred. The time between **fault activation** and failure is known as **error latency**. In other words, a fault might lead to an error which in turn might lead to the failure of the system.

A. Faults

Faults can be classified into four different classes depending on their *pesistence*, *effect*, *source* and *boundary*.

With respect to persistence, a fault can be **permanent**, **intermittent** or **transient**. Permanent faults are continuous in time, while a transient fault is random and occurs for only a short period of time. An intermittent fault is a repetitive malfunction of a device or system that occurs at intervals.

With respect to effect, a fault can be either **activated** or **dormant**. An activated fault is one which has produced an error, while a dormant fault is one that has yet to produce an error. An activated fault can be further classified into **latent** and **detected**, where a latent fault is one which has produced an error that has still not been detected by the system.

The source of a fault can be either **software** or **hardware**. Software faults can be for example design faults or malicious attacks like trojan horses.

Lastly, a fault can be either due to a component **internal** to the system or **external** to it.

A fault can produce a number of errors in a computing system, such as, control-flow errors, data corruption errors, logical errors, buffer overflows, memory leaks, data races, deadlocks/livelocks, infinite loops and wild memory writes etc.

B. *Failures*

Failures can be classified into three different classes based on their *domain*, *action* and *consistency*.

In terms of domain, failures can be either **timing** related or **content** related. Timing failures mean that the failing system either responds too early or too late. On the other hand, content failures mean that the content of the information delivered by the system is in corrupt state.

In terms of action taken by a failing system, failures can be divided into **halt** and **erratic** failures. By halt failure, we mean that the system stops responding on failure, while erratic failure means that the failing system keeps responding but in an abnormal manner. Halting on failure is a good property, as errors are not propagated to other systems in the environment. Systems which halt on failure, are known as **fail-stop** systems.

In terms of **consistency**, there can be **byzantine** and **consistent** failures. When a byzantine failure happens, some or all users of the system will perceive different service. On the other hand, for consistent failures, all users will perceive identical service.

Service failure of a system causes a permanent or transient external fault for other system(s) that receive service from that system.

C. *Fault tolerance*

Fault tolerance means to avoid failures in the presence of faults. A system is said to be **fault tolerant** if faults do not affect the external state of that system. It can however allow its components to fail, as long as they do not corrupt its external state. A fault tolerant system must be able to detect errors and recover from them. The time between fault activation and error detection is known as **error detection latency**.

Figure 2 shows the steps which are taken to make a system fault tolerant. These steps are **proactive fault management**, **error detection**, **fault diagnosis** and **recovery**. In coming sections, we discuss these steps and the related work done with special focus on shared memory multicore/multiprocessor systems.

III. Proactive fault management

Proactive fault management means predicting failures of components before they happen and taking precautionary steps to prevent them, as illustrated in Figure 1.

Software rejuvenation [11] is a proactive fault management technique that tries to avoid faults due to software aging. Software aging is the degradation of an application or system with time. Degradation can happen due to resource leakage, such as memory leaks or accumulation of numerical errors for example. In multithreaded applications, deadlocks may also appear due to software aging. Software rejuvenation tends to avoid these aforementioned problems by periodically restarting applications in a clean state.

Another way of performing Proactive fault management is to proactively check for errors in the system, that is check for errors when system is idle for example or by doing system monitoring. One such technique is memory scrubbing [21] in which memories are periodically checked for errors and corrected, even while they are not in use. Another example of a system which uses proactive error checking is [23] which predicts faults through system monitoring. In case of abnormal behavior detection, such as aberrant temperature or disk errors in a node, the tasks executing on it are migrated to a healthy node.

IV. Error detection

Error detection is the process of detecting errors in the system. Timing errors are normally detected by using watchdog timers, while for content errors, redundant execution is normally applied.

A. *Watchdog timers and processors*

A watchdog timer is a timer that is used to check if a system or a subsystem in it, is stuck, for example due to an infinite loop, in which case it triggers a corrective measure by the system.

A watchdog processor is a coprocessor that is used to detect system level errors by monitoring the behavior of the system. A survey of different kinds of watchdog processors is given in [16]. Watchdog processors can be used to check control flow errors. This is done by associating signatures at each node of a program and providing same signatures to the watchdog processor. [15] shows that 90 percent of control flow errors can be detected through watchdog processors with very low hardware and memory overhead.

B. *Redundancy*

Redundancy is a technique in which multiple processing elements are used to process the same data. One such technique is *dual modular redundancy* (DMR) in which two elements are used. An error is detected when the contents of the two processing elements diverge. Another technique is *triple modular redundancy* (TMR) or N-modular redundancy, which in addition to detecting errors can also locate the faulty element through majority voting. Moreover, the system can continue to execute by masking the faulty element. In such

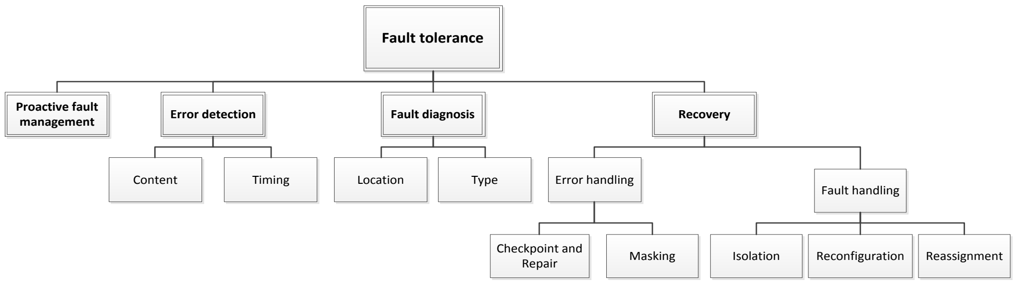

Fig. 2. Classification of steps used for Fault tolerance

systems, the voter also needs to be reliable as it can become a single point of failure.

While N-modular redundancy is used to tolerate hardware faults, N-diversity is used to tolerate software faults, such as, logical bugs left during development. The main idea is that if there is a fault in one version, it can be masked out by using majority voting. Authors in [24] discuss various software fault tolerance techniques using design and data diversity. [7] and [6] are examples of systems which use this technique for tolerating software faults.

For error detection of software running on a single core, fault tolerant systems commonly employ redundant execution at different levels of abstractions, at instruction level [20], process level [22] or virtual machine level [4]. Schemes which work at instruction level have low error detection latencies, while schemes which work at process and virtual machine level allow error to propagate before detecting it. In the absence of faults, these schemes need to make sure that each replica start with the same initial state, executes input data in the same order and perform the same computations. This method is not straightforward to implement, especially for parallel programs running on shared memory multicore/multi-processor systems. SectionVII discusses the related work done to tackle this problem.

V. Fault diagnosis

Fault diagnosis is the process of identifying **location** and **type** of a fault. Location of a fault can be determined either preemptively, that is, before its activation, or after error has been detected due to its activation, as shown in Figure 1.

Failure identification of a fail-stop component is relatively easy as it stops responding when it fails. Time-out is a common mechanism to detect failures of fail-stop components. For example, in a message passing environment, a permanent failure of a processor would be assumed if it stops sending messages.

In TMR systems, faulty component can be located by a majority voter. Another method to locate faulty components is online self-tests. Through this mechanism, a system can find permanent and sometimes intermittent faults in it by testing itself.

Online self tests can be applied concurrently with application execution and therefore can proactively detect dormant faults by locating failed hardware components. Online self tests can be performed using pure hardware built-in self-test (BIST) [39] approaches or using software based self test (SBST) [26]. The benefit of software based approaches is that they do not require any change to the system hardware. Software based techniques are becoming more relevant with increasing number of cores as a core can be dedicated to perform the self tests on the system.

An example of an online self test scheme is [8] which describes and evaluate three different scheduling policies to find permanent and intermittent faults. In this scheme, the online self test can be performed through either special hardware (BIST) or software (SBST). When a test is performed on a processor, it is logically isolated from the rest of the system, while the task which was being executed by that processor is migrated to another processor for continued operation. In the proposed system, only one task can execute at a time on a processor. Self tests are done periodically and the scheduling policies try to select the idle processors or those which are running low priority tasks for testing. Since the test is performed concurrently with application execution, intermittent faults, such as those that occur during burst of a computing in processor, can also be detected.

Type of a fault can be found by using retry/replay methods. For example, in [25], the same BIST test is applied twice in a row. Knowing that transient faults occur infrequently, it can be assumed that transient fault would not occur twice in a row. Hence, if the test fails both times, the failure is considered permanent. mSWAT [40] can also differentiate between a hardware fault and software bugs for a multithreaded program running on a multicore system. After an error is detected, execution is restarted from the last checkpoint. If no error is detected this time, fault is assumed to be transient or a non-deterministic software bug, otherwise a permanent fault or deterministic software bug is assumed. In that case, execution is replayed on different cores. If the same error occurs again, deterministic software bug is assumed, otherwise permanent fault is assumed. In that case, mSWAT does another replay for further analysis to find the faulty core.

VI. Recovery

When a fault is detected, it is important to recover from it. As shown in Figure 1, in a fault tolerant system, recovery must be done before failure of the system occurs. It can be done by either performing **error handling**, **fault handling** or both. Error handling means to eliminate errors from the system without removing the source of the fault. On the other hand, fault handling is the process of removing the source of fault, to prevent reoccurrence of the fault. Error handling is enough for recovering from transient faults, as it is not necessary to locate the source of the fault for transient faults.

A. *Error handling*

Two different schemes can be used for error handling, namely **checkpoint and repair** and **masking**. In *checkpoint and repair*, the state of the system is periodically saved (checkpointed) and when an error is detected, it is rollbacked to a previously valid state by using the checkpoint. The benefit of this scheme is that it can be used to tolerate long error detection latencies [12]. On the other hand, in masking, the erroneous components are masked out by using majority voting on the states of redundant components. The state of an erroneous component may be restored by using the state of one of the non-erroneous redundant components [22]. It is a more efficient technique than *checkpoint and repair*, as no rollback is required. However, it is unable to tolerate long error detection latencies. Therefore, introduction of latent faults in such systems needs to be avoided as they may eventually corrupt most of the redundant states to make recovery impossible [36].

Checkpointing can be mainly categorized into coordinated checkpointing and uncoordinated checkpointing. In coordinated checkpointing [30], each process in the system coordinate with each other to take the checkpoint, while in uncoordinated checkpointing, each process separately takes its own checkpoint. The recovery is achieved through a special recovery phase which reforms the global state of the system to perform recovery. Uncoordinated checkpointing is usually avoided however due its proneness to domino effects [37].

Commodity shared memory multicore processors are equipped with memory management unit (MMU) which allows accelerating the checkpointing process by using copy-on-write techniques. It allows incremental checkpointing, that is only saving pages dirtied since the last checkpoint. Authors in [27] and [28] were the first one to implement checkpointing for parallel programs running on shared memory multiprocessor systems by using this technique. Their scheme allows original application to continue execution while checkpointing is performed. This is made possible by giving read only access to the memory pages of the program when starting checkpointing, so whenever something is written to a page for the first time, page fault is trapped by the OS and content of that page saved in the checkpoint besides giving write access back to that page. Authors in [29], improve upon [28] by using translation lookaside buffer (TLB) misses to record data. This avoids the overhead of setting write accesses of pages.

Normally checkpoints are stored in a non-volatile memory due to its reliability. However, schemes like Respec [14] keep the checkpoint as a forked process in linux. This improves efficiency of both storing and restarting from a checkpoint, especially in systems with large amount of RAM. This method is less reliable though. However, its reliability can be increased by using memory scrubbing.

B. *Fault handling*

Fault handling involves **isolation** of the faulty component and recovering the system from the fault. Moreover, tasks which were being computed on a faulty core need to be reassigned to a working core or a spare core. This is known as **reassignment**. Repair of a faulty component can be done in a reconfigurable system through **reconfiguration** [34]. When hardware resources are exhausted, a reconfigurable system might also emulate a hardware component in software [35], so that system continues to perform albeit with degraded performance.

Isolation of a faulty component is done to make the fault originating from it dormant, so that error is not propagated to the other components in the system. In typical shared memory multicore processors, different processes are run on separate address spaces by using the virtual memory system supported by the MMU. This makes sure that a wild write in one process, due to an uninitialized pointer for example, do not affect the execution of other processes in the system. Therefore the virtual memory provides an efficient scheme for isolation and error confinement. However virtual memory alone would not be not enough to confine errors in case when different processes are communicating through shared memory or at kernel level, since the kernel is itself managing the virtual memory. An error in kernel could bring down the whole operating system.

Hive [31] addresses this issue by using independent kernels known as *cells*. In this way, a fault damages only one cell rather than the whole system. To prevent wild writes from one cell to the memory of another one, each cell uses a *firewall* hardware. On failure of a cell, pages writable from that cell are discarded, which prevents any cell from reading data from those pages. This requires prompt detection of failure of a cell, which Hive does by applying an aggressive fault detection scheme, which includes heuristic checks and a distributed agreement protocol.

Hypervisor based fault tolerance [4] takes a step further by running different guest operating systems in isolated environments. This isolation make sure that failure of one guest OS does not affect the other guest OSs. Moreover, authors in [32] and [33] have exploited the isolation provided by a hypervisor to execute device drivers inside virtual machines for fault tolerance and portability. Due to the isolation provided by the hypervisor, a faulty driver does not impact the rest of the system.

VII. Redundant execution for fault tolerance

Process level and virtual machine level fault tolerant systems apply redundant techniques for fault tolerance. This requires deterministic execution of the redundant components with respect to each other. For this purpose, these systems need to cater for non-deterministic events, such as interrupts, signals, DMAs and non-deterministic functions, such as time of the day. As an example, [13] uses hardware performance counters to count instructions so as to identify the point at which

TABLE I

COMPARISON OF DIFFERENT METHODS FOR DETERMINISTIC REDUNDANT EXECUTION OF SHARED MEMORY MULTITHREADED PROGRAMS

Property / Technique	**Language based (e.g., SHIM)**	**Record / Replay (e.g., Karma)**	**Deterministic execution of programs (e.g., Calvin)**
Scalability	Reasonable	Reasonable	Poor
Programmability	Difficult for arbitrary programs	Easy	Easy
Deadlock prevention	Can be difficult to prevent	Does not prevent	Mutex-based deadlocks can be eliminated

an interrupt occurred in the primary replica and execute the interrupt at the same point of execution in the other replicas.

In multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses and these accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve and therefore an active area of research.

Comparison of the different methods that can be used for deterministic redundant execution is shown in Table I. One way for executing replicas in a deterministic fashion is to use deterministic parallel languages. Examples of such languages are StreamIt [44], SHIM [5] and Deterministic Parallel Java [1]. However, porting programs written in traditional languages to deterministic languages is difficult as learning curve is high for programmers used to programming in traditional languages. Moreover, in languages which are based on the Kahn Process Network Model, such as SHIM, it is difficult to write programs without introducing deadlocks [41].

Deterministic redundant execution at runtime can be done either through hardware, software or a combination of both. Some hardware schemes use record and replay method for achieving deterministic execution. In this method, all interleavings of shared memory accesses by different processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [10] and Karma [42]. These schemes intercept cache coherence protocols to record interprocessor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording, Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well.

Unlike the record/replay method, Calvin [9] executes programs deterministically, that is, given the same input, a program always has the same output. It does so by executing instructions in the form of chunks and later committing them at barrier points. It uses a relaxed memory model, where instructions are committed in such a way that only the total store order (TSO) of the program has to be maintained. An advantage of this method is that mutex-based deadlocks can be eliminated [43]. Moreover, no inter-replica communication is required, thus making this method more dependable than record/replay. The disadvantage of this method though is scalability, as it depends upon barriers to commit chunks.

The disadvantage of existing hardware methods for deterministic execution is that they are applied at system level. They cannot for example, perform deterministic execution of different applications running on a system. Capo [17] is the first scheme to address this issue. It implements a virtualization layer that allows different applications to use the hardware resources for deterministic replay. Non-deterministic events, such as interrupts and signals are handled by the software while for shared memory access interleavings, the underlying hardware for deterministic replay can be used.

Besides hardware methods, software only methods for deterministic execution also exist. One such method is CoreDet [3] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Therefore, it is similar to Calvin, but implemented in software. Since it is implemented in software, it has a very high overhead, 1-11x for 8 cores, as compared to 0.5x-2x for Calvin.

Kendo [19] is a software approach that works only on programs without data races, that is, those that access shared memory only through synchronization objects. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its clock becomes less than those of the other threads. Clock is calculated from retired stores, is paused when waiting for a lock and resumed after lock is acquired. Since this method requires global communication among threads for reading clock values, it also has limited scalability.

Respec [14] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rollbacks and re-execute from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which can induce a large overhead.

VIII. CONCLUSION

In this paper we discussed related work done on online fault tolerance techniques with focus on techniques for shared memory multicore/multiprocessor systems. We have discussed steps which are taken to achieve fault tolerance, which are *proactive fault management*, *error detection*, *fault diagnosis* and *recovery*. Proactive fault management is a precautionary step to prevent failures of components in the system, whereas error detection is performed to detect errors before they lead to failure of the system. We also discussed fault diagnosis techniques which are used to locate failed components and to check the type of a fault. Moreover, we discussed recovery techniques such as checkpoint and repair, reconfiguration and reassignment. Finally we discussed related work to perform deterministic redundant execution of parallel programs running on shared memory multicore/multiprocessor systems, which is still an active area of research.

ACKNOWLEDGMENT

This research is supported by Artemis through the SMECY project (grant 100230).

References

[1] http://dpj.cs.uiuc.edu.

[2] R. C. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, vol. 1, pages 17 –22, March 2001.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, vol. 38, pages 53–64, March 2010.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[5] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 264–272, New York, NY, USA, 2005. ACM.

[6] B. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[7] R. Kapitza H. P. Reiser, F. J. Hauck and W. Schrder-Preikschat. Kemari: Vm synchronization for fault tolerance. In *European Dependable Computing Conference*, pages 67–68, 2006.

[8] O. Heron, J. Guilhemsang, N. Ventroux, and A. Giulieri. Analysis of on-line self-testing policies for real-time embedded multiprocessors in dsm technologies. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 49 –55, 2010.

[9] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333 –334, February 2011.

[10] D. R. Hower and M .D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 265 –276, 2008.

[11] Y. Huang, C. Kintala, N. Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *IN PROCEEDINGS OF IEEE INTL. SYMPOSIUM ON FAULT TOLERANT COMPUTING, FTCS 25*, 1995.

[12] W W. Hwu and Y. N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *Computers, IEEE Transactions on*, vol. 36, pages 1496 –1514, December 1987.

[13] C .M. Jeffery and R .J .O. Figueiredo. Towards byzantine fault tolerance in many-core computing platforms. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 256 –259, 2007.

[14] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. *SIGPLAN Not.*, vol. 45, pages 77–90, March 2010.

[15] A. Mahmood and E .J. McCluskey. Watchdog processors: Error coverage and overhead. *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput., FTCS-15, Ann Arbor, MI.*

[16] A. Mahmood and E .J. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, vol. 37, pages 160 –174, 1988.

[17] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. *SIGPLAN Not.*, vol. 44, pages 73–84, March 2009.

[18] S. Nomura, M. D. Sinclair, C. Ho, V. Govindaraju, M. Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. *SIGARCH Comput. Archit. News*, vol. 39, pages 201–212, June 2011.

[19] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, vol. 44, pages 97–108, March 2009.

[20] G .A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243 – 254, 2005.

[21] A. M. Saleh, J. J. Serrano, and J. H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *Reliability, IEEE Transactions on*, vol. 39, pages 114 –122, April 1990.

[22] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, pages 135 –148, 2009.

[23] G. Vallee, C. Engelmann, A. Tikotekar, T. Naughton, K. Charoenpornwattana, C. Leangsuksun, and S. L. Scott. A framework for proactive fault tolerance. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 659 –664, 2008.

[24] Z. Xie, H. Sun, and K. Saluja. A survey of software fault tolerance techniques.

[25] A. Sanyal, S. Alam, and S. Kundu. Bist to detect and characterize transient and parametric failures. *Design Test of Computers, IEEE*, vol. 27, pages 50 –59, 2010.

[26] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software-based self-testing of embedded processors. *IEEE Trans. Comput.*, vol. 54, pages 461–475, April 2005.

[27] K. Li, J. F. Naughton, and J. S. Plank. Real-time concurrent checkpoint for parallel programs. *SIGPLAN Not.*, vol. 25, pages 79–88, February 1990.

[28] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pages 874–879, August 1994.

[29] J. Liao and Y. Ishikawa. A new concurrent checkpoint mechanism for real-time and interactive processes. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 47 –52, 2010.

[30] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, vol. 34, pages 375–408, September 2002.

[31] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[32] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, vol. 6, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[33] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[34] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *SIGARCH Comput. Archit. News*, vol. 35, pages 470–481, June 2007.

[35] A. Miele A software framework for dynamic self-repair in embedded socs exploiting reconfigurable devices. *International Conference on Automation, Quality and Testing, Robotics*, vol. 2, pages 1–6, 2010.

[36] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pages 448–465, May 2006.

[37] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[38] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, pages 11 – 33, 2004.

[39] H. Al-Asaad, B. T. Murray, and J. P. Hayes. Online bist for embedded systems. *IEEE Des. Test*, vol. 15, pages 17–24, October 1998.

[40] S. K. S. Hari, M. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 122 –132, December 2009.

[41] B. Jiang, E. Deprettere, and B. Kienhuis. Hierarchical run time deadlock detection in process networks. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 239 –244, October 2008.

[42] A. Basu, J. Bobba, and M D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[43] E. D. Berger T. Liu, C. Curtsinger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, October 2011.

[44] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.