A Diskless Chekpointing Approach for Failure Recovery in Multiprocessor Safety-Critical Embedded Systems

Sima Nokarizi[1], Yasser Sedaghat[2], Reza Ramezani[3]

Dependable Distributed Embedded Systems (DDEmS) Lab
Department of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran

sima.nokarizi@stu.um.ac.ir[1], y_sedaghat@um.ac.ir[2], reza.ramezani@stu.um.ac.ir[3]

***Abstract*— Backward recovery is the one of the most important techniques for error recovery in safety-critical systems which are usually based on nonvolatile memories. Since storing checkpoints in hard disks –as a nonvolatile memory- imposes noteworthy timing overhead to the system, diskless checkpointing would be a good solution for low cost fault tolerance in parallel and distributed systems. In this paper an algorithm is proposed which is able to recover a multiprocessor system from failure when up to half of the processors are failed, simultaneously. In contrast to many existing work, in the presented work each processor can have more than one task. The algorithm also by grouping tasks and by coding checkpoints eliminates the need of hard and nonvolatile disks to store checkpoints. The simulation results show the ability of the proposed algorithm in recovering system from failure when up to half of processors are simultaneously failed without using any extra dedicated checkpointing processor. Also compared to the existing approaches, the presented method requires fewer processors.**

Keywords- Fault Tolerance, Backward Recovery, Multiprocessor Error Recovery, Diskless Checkpointing.

I. INTRODUCTION

Embedded systems are usually used to control and build larger systems. Such systems have an important role in safety-critical applications [1]. Any failure in safety-critical applications would cause catastrophe for humans or environment. Thereby, fault tolerance is a key feature in such systems. A system is fault-tolerant when it can continue its operation even in the presence of faults and errors.

Fault tolerance techniques have usually four steps: fault detection, fault location, fault isolation, and finally fault recovery. In fault and error recovery, a faulty system is recovered to a correct and safe situation before the fault occurrence [1].

There are generally two techniques for error recovery: backward recovery and forward recovery. In the forward recovery, when an error occurs, the system continues its operation until it finds an error-free state and then recovers the system to that state. Forward recovery techniques are chiefly based on an N-Version Programming (NVP) technique which has a great energy and computational overhead. Hence, this technique is not suitable for embedded systems [2]. In contrast, backward recovery techniques store fault-free system states (a.k.a. checkpoints) in a safe and nonvolatile memory (such as hard disks). Whenever the system fails, it is recovered to the last fault-free checkpoint and the system operation is continued from that point [2]. This technique has less energy and computation overhead.

In recent decades, parallel and distributed systems have absorbed the attention of many researches, because of their ability in efficiently performing many complex and data-intensive computations. Therefore on one hand embedded systems are enthusiastic to use multiprocessors to achieve higher performance, but on the other hand by increasing the number of processors, the failure probability of such systems increases as well [3]. Hence fault tolerance is even more important in multiprocessor-based embedded systems.

Error recovery via checkpointing can be done in disk-based or diskless manners. In the disk-based approach, checkpoints are stored and read-back from a safe and a nonvolatile disk (especially a hard disk) which incurs a great timing overhead to the system. But in diskless approach, the need of disks is eliminated and instead of such low speed storage devices, fast speed processors' memories are used to store and read-back checkpoints. In this method, the checkpoints and states of a processors is stored in the memory of other processors. When a failure occurs, the faulty processor can be recovered by restoring its checkpoints stored in the memory of a healthy processor [4].

Since disks are known as low speed devices, using them for storing and reading-back checkpoints would increase the total system runtime. On the other hand as diskless approaches uses fast memories, compared to disk-based approach they would have a great impact on system performance. Some fault-tolerant schedulers employs diskless checkpointing scheme to increase system performance [5]. Therefore, in diskless methods it is possible to store more checkpoints and hence reduce the error recovery time [3, 6]. It also would be beneficial to point out that in multiprocessor systems which employ shared I/O and system bus, storing and reading-back checkpoints from disk would become a bottleneck. But in contrast, using diskless approach alleviates this problem.

This paper focuses on diskless backward recovery in multiprocessor systems such that each processor can run one or more tasks. In these systems, each processor has

978-1-4799-1972-7/15/$31.00

its own dedicated memory and these memories are used instead of hard disks to store checkpoints. The goal is to recover system from failure when multiple processors fail, simultaneously. In the proposed method each processor stores its checkpoints in the memory of other processors. Hence, when a processor fails, its tasks can be recovered by restoring tasks' states stored in the other processors memory. In order to reduce the size of checkpoints an XOR-based coding scheme is employed too.

The paper is organized as follows: Section 2 reviews some related work. Section 3 describes the proposed method and demonstrates how processors are grouped and how checkpoints are coded, compressed, and stored in the memory of other processors. Section 4 reveals the simulation details and evaluates the proposed method and finally section 5 concludes the paper.

II. Related Work

As mentioned earlier, there are two disk-based and diskless approaches for storing checkpoints in backward recovery technique. Many authors [2, 7-11] have focused on disk-based checkpointing and tried to reduce the associated overheads. Also some other techniques such as incremental checkpointing [12], buffering checkpoints [13], compressing checkpoints [14], and memory exclusion [15] are proposed to reduce the time and memory overhead in disk-based checkpointing technique.

In disk-based checkpointing scheme, a long time is spent to write and store checkpoints in the disk. As indicated by [16], it usually takes 5 to 15 *ms* to access hard disks whereas in DRAM memories this time is 40 to 80 *ns*. It shows that in spite of using some techniques to reduce the overheads of disk-based checkpointing approach, its overhead compared to diskless approaches is still much greater and would be a bottleneck for systems which use backward recovery for fault tolerance.

Diskless checkpointing technique was first introduced in [17] to overcome the overheads of storing and reading-back checkpoints. This technique eliminates the need of low speed hard disks for saving checkpoints which eventually leads to reduce the time takes to store checkpoints as well as reduces the interval of saving checkpoints [3]. This technique is also implementable in multiprocessor systems which use processors' memory to store checkpoints.

As mentioned before, since access time to internal memory is much faster than hard disk (it is almost 100,000 times faster [16]), diskless methods use internal memory for storing checkpoints. Reducing the time required to store and restore checkpoints leads to gain a better performance [18].

There are many methods for diskless checkpointing technique. Neighbor-based and coding-based are two of the most famous methods of this technique [19, 20]. Neighbor-based method is very simple. It stores a copy of space address and registers as a checkpoint in the memory of the neighboring processor. When a processor fails, it is recovered by restoring its state from its neighbor processor's memory [19].

Neighbor-based method has three approaches: mirroring, pair neighbor, and ring neighbor [20]. In the mirroring approach a processor which only stores checkpoints (a.k.a. checkpointing processor) is dedicated to each processor which only runs tasks (a.k.a. application processor). In this approach the checkpoints of the application processor is stored in the memory of the checkpointing processor. The mirroring approach cannot recover failure when both application processor and checkpointing processor fail simultaneously. In the pair neighbor approach, each two processors constitute a pair and each processor sends its own checkpoint to its paired processor and vice versa. Hence, no dedicated checkpointing processor is required. This approach cannot tolerate coincident failure of both processors, neither. Finally ring neighbor approach does not need any dedicated checkpointing processor. In this approach, all processors constitute a virtual ring and each processor sends its own checkpoint to the next processor in the ring. In this approach it is impossible to recover system from failure if one processor and its next neighbor processor fail, simultaneously. Memory consumption is one of the most significant drawbacks in neighbor-based approaches [20], since a full copy of tasks data and states is stored in the memory of the other processors.

In coding-based methods, m-out-of-n existing processors are used as checkpointing processor to code and store checkpoints of the application processors. Therefore the checkpoints of the faulty processors can be decoded and calculated again by using both checkpointing and application processors. Coding-based methods have two steps. In the first step, each application processor calculates its own checkpoints. Then checkpoints are coded and stored in the memory of the checkpointing processors which solely codes, stores and decodes checkpoints. Parity and Reed-Solomon are the most widely used coding techniques in coding-based checkpointing methods [19].

Parity technique requires only one dedicated checkpointing processor to store checkpoints of all application processors. The j^{th} byte of the checkpointing processor is the result of taking XOR from j^{th} byte of checkpoints of all application processors. When a processor fails, the checkpointing processor recovers the faulty processor by using its coded checkpoint and the checkpoints of other healthy processors. This technique reduces the size of checkpoint, but to recover a faulty processor, all other processors must be healthy and also synchronous [20]. Reed-Solomon technique uses mathematic operation to code the checkpoints and it requires m checkpointing processors to recover m simultaneously failed processors. This technique is very complex and incurs a huge timing overhead, but is can recover more than one faulty processor [3, 19].

Some authors have proposed diskless solutions by combining two aforementioned methods. For example in [3, 18, 20], on one hand, similar to neighbor-based methods, do not need any dedicated checkpointing processor and checkpoint of any given processor is stored in the memory of other application processors. On the other hand, they similar to coding-based techniques use some coding algorithms to reduce the size of the checkpoints.

III. Proposed Method

In this paper a new diskless checkpointing strategy is proposed which does not require any dedicated

checkpointing processor and also can tolerate more than one simultaneous processor failure in multiprocessor systems in which each processor can have more than one task.

A. Assumptions

The proposed strategy deals with a multiprocessor system consisting of P_0, P_1, ..., P_n processors. Each processor has its own dedicated memory and can directly connect to all other processors by message passing. Due to reliable communication channel it is supposed messages are transferred reliably as well. Sending and receiving of messages takes no time and also similar to previous work it is supposed there is no cache memory.

In contrast to previous work [3, 18, 20], which suppose each processor has only one task, the proposed method supposes each processor can have more than one task.

From the fault type perspective, it is supposed that faults are transient and processors are not failed permanently. But it is supposed more than one processor would fail simultaneously. Also it is assumed that taking and saving checkpoints has a very low cost, therefore their cost is ignored.

B. The proposed Checkpointing and Recovery Technique

The proposed method is a combination of neighbor-based and coding-based approaches which stores checkpoints in the memory of other processors and therefore the need of hard disks for storing checkpoints is eliminated. Our method also uses a simple XOR-based coding technique to reduce the size of the stored checkpoints in a way that in spite of compressing checkpoints, it can recover faulty processors even when up to half of processors are failed simultaneously.

It is supposed there are n processors in the system and each processor can have one or more tasks. The proposed method groups processors. The number of groups is equal to the number of simultaneous faults that the system must tolerate. For example if k simultaneous faults should be tolerated, k groups are created in a way that each group contains $\lfloor n/k \rfloor$ processors. Remaining processors are located at the last group. It is supposed only one failure occurs at each group. By increasing the number of groups, the number of recoverable simultaneous failures increases as well.

At first, each processor calculates its checkpoint and stores it in its dedicated memory. To tolerate k simultaneous faults, after grouping processors into k distinct groups, An XOR is taken from the first tasks of processors of a given group and the result is stored in the memory of the first and second processors of the next group (these two processors run their assigned tasks and also stores the coded checkpoints of the previous group. For brevity we refer to these processors as checkpointing processors). This process is repeated for all corresponding tasks in all processors of all groups. Figure 1 shows this process. Therefore the Parity of all groups is calculated and stored in the memory of two checkpointing processors in the next group. In each group one faulty processor is recoverable (this faulty processor can even be the checkpointing processor).

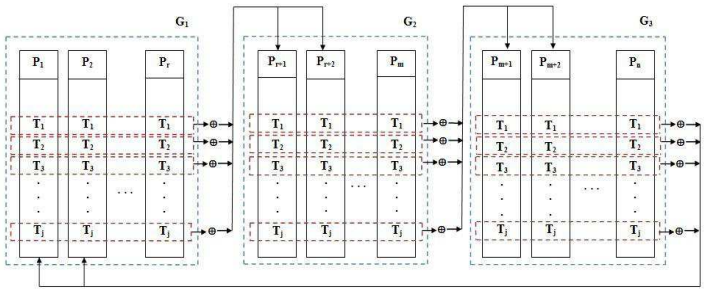

Figure 1-Coding and storing checkpoints to recover multiple failures

The intuitive behind storing parities of a group into two processors of the next group is to have a backup checkpointing processor when one of them is failed and therefore increasing the system reliability. In this way if only one failure occurs in each group, then at least one of the checkpointing processors in each group would be healthy to be used for error recovery. The worst case scenario is the situation in which k failures occur simultaneously and as only one processor fails in each group; all k faulty processors are still recoverable.

The recovery process is as follows. Whenever a processor fails, a message is sent to the first checkpointing processor in the next group. If the checkpointing processor is healthy, the recovery process starts. Otherwise the message is sent to the second checkpointing processor. As always at least one of these checkpointing processors is healthy, then a message is sent to the all other healthy processors in the group of the faulty processor. As each processor always has checkpoints of its own tasks, an XOR is taken from checkpoints of all application processors of the group and then the result is sent to the checkpointing processor of the next group. The checkpointing processor, using its coded checkpoints and the received checkpoints, decodes and recovers the checkpoint of the faulty processor and sends it back to the previous group to recover the faulty processor. The recovered processor recalculates and stores checkpoints of its tasks to be used for other processors failure in the future.

It is noteworthy to state that storing the XOR result in more than two processors does not have any positive impact on increasing the number of tolerable failures in each group. Because as there are more than one unknown checkpoint data, even having more checkpointing processors do not help to decode XOR.

IV. Simulation and Evaluation

A. Simulation

To evaluate the proposed method, a simulation environment is implemented by C# language. Each task has an individual checkpoint which its size depends on the variables and states of the task. For simplicity, in the simulation it is assumed that the size of checkpoint of each task is 25 bits.

The implementation has three functions. As the *ini_start()* function in Figure 2 shows, after assigning

tasks to processors and calculating their checkpoints, the processors are grouped based on the maximum number of tolerable failures.

```
1. // group_length: number of processors in each group
2. // fault_count: number of simultaneous failures that should be recovered
3. // Processors: a collection of system processors
4. // Groups: a collection of processor groups
5. ini_start()
6. {
7.     group_count = fault_count
8.     group_length = Processors.count / group_count
9.     for i=1 to Processors.count
10.    {
11.        Assign some tasks to Processors[i]
12.        for j=1 to Processors[i].Tasks.count
13.            τ = Processors[i].Tasks[j]
14.            Processors[i].Tasks[j].CP = Checkpoint(τ)
15.        Index = ((i-1) / group_length) +1;
16.        Groups[Index].Add(Processors[i])
17.    }
18. }
```

Figure 2 – Assigning tasks and grouping processors

After grouping processors, the first and the second processors of each group (checkpointing processors) in addition to run their tasks are nominated to store the coded checkpoints of the previous group processors. Afterwards parities of corresponding task in processors of each group are calculated and sent to the checkpointing processors of the next group. The process of calculating, coding and storing tasks' checkpoints in the checkpointing processors memory is done through *ini_CC()* function which is depicted in Figure 3.

```
1. // NextGroup: index of the next group to store checkpoints of current group
2. // CP, TaskCP: Tasks Checkpoint
3. ini_CC()
4. {
5.     for i=1 to group_count
6.     {
7.         if i < group_count
8.             NextGroup = i + 1;
9.         else
10.            NextGroup = 1;
11.        MaxTask = Maximum Task Count in Processors of Groups[i]
12.        for j=1 to MaxTask
13.        {
14.            TasksCP = null;
15.            for k=1 to Groups[i].length //processors in the group
16.            {
17.                if Groups[i].Processors[k].Task[j] ≠ null
18.                    CP = Groups[i].Processors[k].Task[j].CP
19.                    TasksCP = XOR(TasksCP , CP)
20.            }
21.            Groups[NextGroup].Processors[1].Add(TasksCP);
22.            Groups[NextGroup].Processors[2].Add(TasksCP);
23.        }
24.    }
25. }
```

Figure 3 – calculating parities of groups checkpoints

When a failure occurs, it is recovered by *ini_recover()* function. This function which is depicted in Figure 4 by using checkpointing processors, recovers a faulty processor and all of its assigned tasks. As this function shows, if the faulty processor is a checkpointing processor, addition to restoring its tasks' checkpoints, it should get a copy of the coded checkpoints of the previous group from the alternate checkpointing processor.

In the recovery process, at first the group of the faulty processor and then the group which stores checkpoints of the faulty processor are identified. Afterward an XOR is taken from corresponding tasks in the group. However it should be noted that faulty processors should not be involved in the calculation. A faulty processor can be recovered by calculating Parity of the corresponding tasks in the group and the coded and stored checkpoints in the checkpointing processors in the next group.

B. Evaluation

1) The number of processors

As mentioned before, it is supposed the number of processors is limited and hence resources should be used efficiently. Therefore in contrast to the work presented by Chiu et al [3] which assumes each processor has only one task, our presented method supposes each processor has more than one task and tries to recover all of the tasks assigned to a given processor when the processor fails.

```
1. // NextGroup: index of the next group to store checkpoints of current group
2. // CP, TaskCP: Tasks Checkpoint
3. // CID: ID of the checkpointing processor
4. ini_recover()
5. {
6.     for i=1 to group_count
7.     if Groups[i] has a faulty processor
8.     {
9.         if i < group_count
10.            NextGroup = i + 1;
11.        else
12.            NextGroup = 1;
13.        PID = index of faulty processor
14.        if PID = 1
15.            Groups[i].Processors[1].TasksCP = Groups[i].Processors[2].TasksCP
16.        else if PID = 2
17.            Groups[i].Processors[2].TasksCP = Groups[i].Processors[1].TasksCP
18.        MaxTask = Maximum Task Count in Processors of Groups[i]
19.        for j=1 to MaxTask
20.        {
21.            TasksCP = null;
22.            for k=1 to Groups[i].length //processors in the group
23.            {
24.                if Groups[i].Processors[k].Task[j] ≠ null and k ≠ PID
25.                    CP = Groups[i].Processors[k].Task[j].CP
26.                    TasksCP = XOR(TasksCP , CP)
27.            }
28.            if Groups[NextGroup].Processors[1] is faulty
29.                CID = 2;
30.            else
31.                CID = 1;
32.            CP = Groups[NextGroup].Processors[CID].TasksCP[j]
33.            TasksCP = XOR(TasksCP , CP)
34.            Groups[i].Processors[PID].Task[j].CP = TasksCP
35.        }
36.    }
37. }
```

Figure 4 - The pseocode of faulty processors recovery

$$P(A_i) = \frac{\binom{i}{1}\binom{2k-i}{1} * \binom{i-1}{1}\binom{2k-i-1}{1} * \dots * \binom{1}{1}\binom{2k-2i+1}{1} * \binom{2k-2i}{2} * \dots * \binom{2}{2} * \binom{k}{i}}{\binom{2k}{2}\binom{2k-2}{2} \dots \binom{4}{2}\binom{2}{2}} \quad (1)$$

With the assumption of having only one task per processor, Figure 5 shows that at least how many processors is required to recover different number of simultaneous processor failure. By increasing the number of simultaneous failures k, the minimum number of required processors to recover faulty processors in the basic paper [3] increases dramatically. For example to tolerate 10 simultaneous failures, the work presented by [3] requires 167 processors whereas our proposed method only requires 20 processors.

As Figure 5 shows there is a significant difference between the number of required processors. This is due to the special condition that is considered for error recovery in [3]. For example, [3] indicates that if each processor has a set of checkpointing processors to store its checkpoints, any two given checkpointing processors set cannot have more than one common processor. By considering such conditions determination of checkpointing processors become hard which leads to use more processors to recover system from failure.

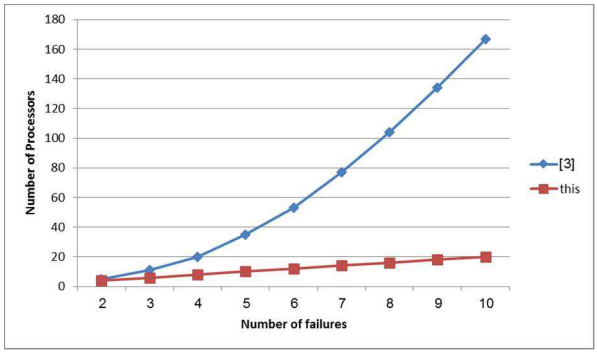

Figure 5 - Comparision of the number of processors for different number of simultaneous failures in the proposed method and the method proposed by [3]

2) Optimal Groups

It is clear that by having a fixed number of tasks, when the number of processors is decreased, the number of tasks on each processor is increased as well. Therefore tasks Makespan highly depends on the number of processors and the number of tasks on each processor. The less the number of processors, the more the Makespan of tasks is achieved. On the other hand running only one task on each processor would decrease processors utilization dramatically.

As result, in order to group processors optimally, some criteria such as the number of processors, the time required to calculate Parity when coding checkpoints, the time required to decode checkpoints when recovering, and finally the number of simultaneous tolerable failures should be taken into account.

As pointed out before, processors are grouped based on the maximum tolerable failures. The ideal situation occurs when failure of half of processors is tolerable and recoverable. In the proposed method, an optimal grouping happens when each group has only two processors and each processor has only two tasks. By this grouping strategy, compared to basic paper [3], less processors is required to run tasks and also maximum tolerable processor failure is attainable. That is in this method the system can be recovered even when up to half of processors are failed simultaneously which results an acceptable system reliability.

Based on the optimal grouping if it is possible to have two faulty processors in one group and there are k groups coupled with n processors, while $k = n/2$, the probability of recovering i faulty processors ($1 \le i \le k$) is denoted by $P(A_i)$ and can be obtained from (1). This equation is derived by dividing all favorable states by all possible states. A state is favorable for a group if there is at most one faulty processor in the group. As there are i faulty processors and $2k - i$ healthy processors, there exist $\binom{i}{1}\binom{2k-i}{1}$ favorable states for group 1; similarly, there will be $\binom{i-1}{1}\binom{2k-i-1}{1}$ favorable states for group 2 and $\binom{1}{1}\binom{2k-2i+1}{1}$ favorable states for group i. Afterwards, as for groups $i+1$ to k all processors are healthy, there are actually $\binom{2k-2i}{2} * \dots * \binom{2}{2}$ favorable states for these groups. This selection can be done in $\binom{k}{i}$ different ways.

After simplification, (1) can be easily seen as (2):

$$P(A_i) = \frac{i! * (2k-i)! * \binom{k}{i} * 2^i}{(2k)!} \quad (2)$$

Therefore, by optimal grouping strategy, the probability of recovering half of faulty processors is obtained by (3):

$$P(A_k) = \frac{(k!)^2 * 2^k}{(2k)!} \quad (3)$$

V. Conclusions

Error recovery is one of the most important parts of fault tolerance which leads to increase systems dependability. Backward recovery techniques store fault-free system states (checkpoints) in some points and resume the system operation from the last checkpoint when an error occurs. As the time required to store and read-back checkpoints from low speed nonvolatile storage devices (such as hard disks) would become a bottleneck, diskless checkpointing would be a good alternative strategy. In this paper a diskless checkpointing strategy for backward recovery in multiprocessor safety-critical systems was proposed.

In contrast to previous work [3] that each processor can run only one task, in the proposed method, the number of tasks in each processor is not limited. Therefore to tolerate a given number of simultaneous failures, compared to [3], the proposed method needs much less processors.

To achieve such fault tolerance our proposed method combines neighbor-based and coding-based strategies.

The method groups tasks based on the intended maximum number of tolerable faults. Then tasks of processor in each group are corresponded and the Parity of corresponding tasks in each group is stored in two processors of the next group. When a processor fails, its checkpoint is recovered by using checkpoints of other processors of the same group and the checkpoint which is coded, compressed and stored in the next group. This strategy in addition to recover multiple errors, needs less processors and also does not required any dedicated processor for only saving checkpoints.

If processors and tasks are grouped optimally, that is if each group has only two processors and each processor has only two tasks, then the system can be recovered even when up to half of processors are failed simultaneously.

For future work it is suggested to consider the cost of taking and saving checkpoints in order to specify the optimal number of groups in a way that by having minimum computation cost, the maximum passible simultaneous failures are tolerated.

ACKNOWLEDGEMENT

The authors wish to acknowledge the advices and supports that they have received from outstanding members of Dependable Distributed Embedded Systems (*DDEmS*) Laboratory[1] of Ferdowsi University of Mashhad.

VI. REFERENCES

[1] M. Short, "Development guidelines for dependable real-time embedded systems," *IEEE/ACS International Conference on Computer Systems and Applications*, 31 March-4 April 2008, pp. 1032-1039.

[2] T. Li, R. Ragel, and S. Parameswaran, "Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors," *in Design, Automation & Test in Europe Conference & Exhibition (DATE),* 12-16 March 2012, pp. 875-880.

[3] Ge-Ming. Chiu, J.F. Chiu, "A New Diskless Checkpointing Approach for Multiple Processor Failures," *IEEE Transaction on Dependable and Secure Computing, Vol. 8, Issue 4,* JULY/AUGUST 2011, pp. 481-493.

[4] D. Hakkarinen, Z. Chen, "Multilevel Diskless Checkpointing," *IEEE Transaction on Computers, Vol. 62, Issue 4,* 2013, pp. 772-783.

[5] Ramezani, Reza, and Yasser Sedaghat. "An Overview of Fault Tolerance Techniques for Real-Time Operating Systems." In *The 3rd International Conference on Computer and Knowledge Engineering-ICCKE 2013*. 2013.

[6] J.S. Plank, K. Li, and M.A. Puening, "Diskless checkpointing," *IEEE Trans. on PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 9, Issue. 10,* OCTOBER 1998, pp. 972-986.

[7] K. Naruse, S. Umemura, and S. Nakagawa, "Optimal checkpointing interval for two-level recovery schemes," *Computers & Mathematics with Applications, VOL. 51, Issue. 2,* 2006. pp. 371-376.

[8] C.H. Chen, Y. Ting, and J.S. Heh, "Low Overhead Incremental Checkpointing and Rollback Recovery Scheme on Windows Operating System," *Third International Conference on Knowledge Discovery and Data Mining,* 9-10 Jan 2010, pp. 268-271.

[9] T. Yeh, W. Cheng, "Improving Fault Tolerance through Crash Recovery," *2012 International Symposium on Biometrics and Security Technologies (ISBAST), IEEE,* 2012, pp. 15-22.

[10] S. Feng, S. Gupta, et al, "Encore: Low-cost, fine-grained transient fault recovery," *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM,* 2011, pp. 398-409.

[11] H. Tabkhi, S.G. Miremadi, and A. Ejlali, "An Asymmetric Checkpointing and Rollback Error Recovery Scheme for Embedded Systems," *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems,* 2008.

[12] S.I. Feldman, and C.B. Brown. "Igor: A system for program debugging via reversible execution," *in ACM SIGPLAN Notices, VOL. 24, Issue. 1,* 1989, pp. 112-123.

[13] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," *Proc. of 11th IEEE Symposium on Reliable Distributed Systems,* 5-7 Oct 1992, pp. 39 - 47.

[14] C.C.J. Li, and W.K. Fuchs. "Catch-compiler-assisted techniques for checkpointing," *20th International Symposium* in *Fault-Tolerant Computing, IEEE,* 26-28 Jun 1990, pp. 74-81.

[15] J.S. Plank, M. Beck, G. Kingsley, K.Li, "Libckpt: Transparent checkpointing under unix," Computer Science Department, 1994.

[16] D.A. Patterson, and J.L. Hennessy, *Computer organization and design: the hardware/software interface*, Morgan Kaufmann, 2008.

[17] J.S. Plank, and L. Kai. "Faster checkpointing with N+1 Parity," *Twenty-Fourth International Symposium on Fault-Tolerant Computing, IEEE,* 15-17 June 1994, pp. 288-297.

[18] J.F. Chiu, "Double Mutual-Aid Checkpointing for Fast Recovery," *14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE,* 25-27 June 2012, pp. 1015-1020.

[19] N.A. Kofahi, S. Al-Bokhitan, and A. Al-Nazer, "On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis," *Information Technology Journal, VOL. 4, Issue. 4,* 2005, pp. 367-376.

[20] J.F. Chiu, and W.H. Hao, "Mutual-Aid: Diskless Checkpointing Scheme for Tolerating Double Faults," *10th IEEE International Conference on High Performance Computing and Communications,* 25-27 Sept. 2008, pp. 540-547.

[1] http://ddems.um.ac.ir