A Genetic-based Optimal Checkpoint Placement Strategy for Multicore Processors

Atieh Lotfi, Saeed Safari
School of Electrical and Computer Engineering
College of Engineering, University of Tehran, Iran
a.lotfi@ece.ut.ac.ir, saeed@ut.ac.ir

Abstract— **Nowadays multicore processors are increasingly being deployed in high performance computing systems. As the complexity of systems increases, the probability of failure increases substantially. Therefore, the system requires techniques for supporting fault tolerance. Checkpointing technique is widely used to reduce the execution time of long-running programs in the presence of failures and to enhance the reliability of such systems. Optimizing the number of checkpoints in a parallel application running on a multicore processor is a complicated and challenging task. Infrequent checkpointing results in long reprocessing time, while too short checkpointing intervals lead to high checkpointing overhead. Since this is a multi-objective optimization problem, trapping in local optimums is very plausible. On the other hand, bio-inspired algorithms are powerful function optimizers that are successfully used to solve problems in many different areas. In this paper, by applying genetic algorithm, which is a well-known bio-inspired computing algorithm, finding optimal checkpoint placement in parallel applications is exercised. Under certain fault conditions, this new checkpoint placement strategy outperforms the existing ones with a significant reduction in the total wasted times. Our experimental results show that our method, which is implementable on any message-passing multicore system, can optimally find the suitable points in which checkpoints should be taken**.

Keywords- Fault Tolerance; Optimal Checkpoint Placement; Multicore Architectures; Genetic Algorithm

I. Introduction

Recent changes in the high performance parallel computing make fault tolerant system design important. The higher number of processors leads to higher overall performance, but it also increases the probability of failures. Moreover, there are many noteworthy applications such as protein folding, N-body, or some optimization problems that takes days or even weeks to execute. As the execution time of a program becomes longer, both the probability of failure during execution and the overhead of such failures increase considerably. It is possible that the execution time of the program exceeds the mean time to failure of the underlying hardware. Therefore there is a risk that the application may never be finished.

One of the well-known techniques for making such applications resilient to failures is checkpointing [1]. In this technique, the system state is saved in some intermediate states, so there is no need to restart the application from scratch in the event of failure, and the system rollbacks to one of its recent checkpoints. Using this technique, the system can be recovered with hopefully the minimum loss of computation when a failure occurs. But, checkpoinitng comes with some overhead affecting the execution time of the program. So, it is necessary to avoid useless checkpoints and keep a balance between the overhead caused by taking checkpoints and the amount of work lost when the system fails.

Checkpointing in multicore message-passing systems poses several challenging issues. We consider such systems consisting of a number of processes communicating each other by means of messages. In fact, the main difference between sequential and parallel applications in case of failure recovery is the existence of dependencies imposed by inter-process communications. If checkpoints are placed without any coordination an inconsistency may occur upon recovery. Checkpoints on a recovery line should be consistent. Thus for every recorded received message, its corresponding sent message should be recorded to avoid creating orphan messages [1]. In order to create consistent recovery lines, coordination protocols between communicating processes are needed. In uncoordinated checkpointing in which each process takes checkpoints individually without any coordination, domino effect and possibility of creating useless checkpoints can be arisen [2]. On the other hand, in coordinated checkpointing, all processes take their checkpoints simultaneously, which is easy to implement but often leads to significant overhead. Also finding a way to synchronize all processes may not be always possible. The third existing way is to take checkpoints according to the messages passed between processes. In other words, each process takes its checkpoints individually and in case of dependency caused by sending and receiving messages, a checkpoint will be induced on the dependent process. This technique leads to a better performance. Since taking checkpoints causes overhead in the application, useless checkpoints (those checkpoints that do not belong to a consistent recovery line) should be avoided. A complete survey of checkpoint/recovery techniques can be found in [2] and [3].

Determining the optimal number of checkpoints in a message-passing program which are consistent and are not useless is a crucial issue. In this paper we propose a novel optimal checkpoint placement strategy which minimizes the execution time of the parallel program. We consider parallel programs that use message passing communication scheme. We assume the parallel programs run on a multicore machine with reliable message delivery system. The processors in this

978-1-4673-1482-4/12/$31.00 ©2012 IEEE

system can fail any time and there is a failure detection mechanism that detects failures immediately. Furthermore, we consider transient and intermittent faults, which have instantaneous duration. Given such system with a message passing application, and a given error probability, we propose a genetic-based algorithm to find the optimal checkpoint placement for that parallel application. We also assume Poisson process for system failure.

The rest of this paper is organized as follows; in Section II related works are studied. Section III introduces our checkpoint placement strategy. Experimental results are given in Section IV. And finally conclusions are drawn in the last section.

II. REALTED WORKS

Delving into the checkpointing researches so far, we can categorize existing works into two distinct classes. Indeed, existing researches on checkpoint/restart mainly focus on optimizing checkpoint or recovery operations. The issues related to optimizing checkpoint operation are reducing checkpoint overhead, checkpoint latency, and required storage space. Among mentioned issues, reducing checkpoint overhead is the most important one because it affects the execution time of application.

There is a trade-off between the overhead imposed by checkpointing and the amount of work lost due to the failure. Many different works have aimed at optimizing this trade-off. Generally either the expected completion time of a task ([4], [5]), or the availability of the system ([6]) is chosen as an optimization metric. The use of analytical and stochastical models for serial and parallel applications to determine suitable checkpoint intervals has been studied to a large extent (e.g. [4], [7], [8]). In all of these papers, coordinated checkpointing is considered, and the effect of communication and the dependency imposed by sending and receiving messages are not considered. They supposed systems without any dependency and solved their equations for just one processor and generalized it to the whole system, which is simple but not realistic. Most of the works use equidistance checkpoints, but it has been shown that for realistic system failures, periodic checkpointing is not the best choice [9]. In this paper, we do not limit all the processors in our multicore architecture to checkpoint simultaneously and periodically. Instead, each core takes its checkpoints individually and provides coordination in case of dependency.

In addition to these analytical and stochastical solutions that suggest optimal checkpoint placements, some other researches have been conducted to dynamically decide on the necessity of taking each checkpoint or changing the checkpoint interval at runtime. This technique called cooperative checkpointing introduced in [10], [11]. In [11], the authors proposed a solution in which the application, the compiler, and the system jointly decide where each checkpoint should be taken. In other words, checkpoints that are inserted at arbitrary locations in the application by programmer and optimized by compiler before execution can be skipped by system at runtime. In [12] an adaptive checkpointing technique has been proposed that changes the checkpoint frequency after taking each checkpoint regarding the grid system failure rate and remaining execution time of each job. However, it assumes the execution time of each job is known beforehand.

The other well-researched techniques to tackle the checkpointing overhead from the disk-storage point of view are copy-on-write buffering and incremental checkpointing [13], [14]. The simplest method to hide the latency of checkpointing is to store the checkpoint in main memory and continue the application execution while checkpoint is stored to the disk storage. The idea in incremental checkpointing is to avoid rewriting unchanged pages to disk which reduces the checkpoint latency. In other words, only the modified states since last checkpoint will be saved on a checkpoint.

Some other works have been carried out in order to reduce the restart latency. As a case in point, in [15] a mechanism for reducing the latency of recovery from failures has been proposed. It overlaps the process of recovery with retrieving data from the last checkpoint by dynamically studying the behavior of the application after each checkpoint.

III. METHODOLOGY

The problem solved in this paper is formally stated as follows: given a multicore system and a parallel program running on it; derive the minimum number of consistent checkpoints that minimizes the overall wasted time during the execution time of the program in a faulty environment. The overall wasted time is defined as sum of all lost processing times caused by failures and checkpointing overhead which is the time added to the execution time of the program due to taking checkpoints.

Unlike the conventional approaches on finding the optimal checkpoint locations using global coordinated checkpointing, our method uses a local checkpointing approach. Every process takes its checkpoints independently by the order of a controller that traces the communication messages. The controller asks processes to take their checkpoints according to the proposed optimal checkpoint placement strategy. The advantage of our approach is to reduce the amount of wasted times due to checkpointing and recovery processes. We find the dependency between different cores to omit taking the useless and unnecessary checkpoints. In the following, we explain how dependences affect checkpointing and recovery processes.

We assume the multicore system consists of a finite set of processes that communicate only by exchanging messages. Each process is executed on a core. We assume that communication channels are reliable and the order of messages will not change during the execution. Each message makes an inter-process dependency. In a parallel computing system, a local checkpoint is defined as the local state of one process and a global checkpoint is defined as a set of local checkpoints. A consistent global checkpoint is a global checkpoint that makes a consistent recovery line. It doesn't include messages received but not yet sent (orphan messages) [16]. It should be noted that a consistent global checkpoint can include sent but not received messages. Therefore, to construct a consistent recovery line to have a safe rollback in a message-passing system with local checkpointing approach, we consider the following rule. This rule, shown in Figure 1, is that, if the sender rolls back to a checkpoint before its send message, the receiver must roll back with it as well. The reason is that because the sender is faulty, the receiver may have used the wrong data and should reprocess the execution using the new data receiving from sender. Figure 1 shows the inter-process dependency created by passing a message between processes P1, P2, and P3. The mentioned rule is also illustrated which is the sender rollback leads to the receiver rollback with it. As it can be seen in this figure, a process rollback may result in more than one processes roll back.

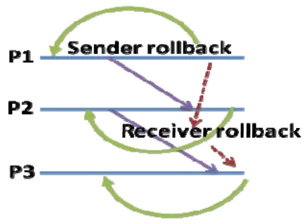

Figure 1. Rule to rollback under local checkpointing

Therefore, considering the mentioned rule, all the processes that communicate with one another in an interval should roll back together, and independently of other processes. If a fault occurs and a core needs to roll back, a controller forces its dependent processes to roll back as well.

Our methodology for solving optimal checkpoint placement problem in a parallel program can be divided into three phases which is summarized in Figure 2.

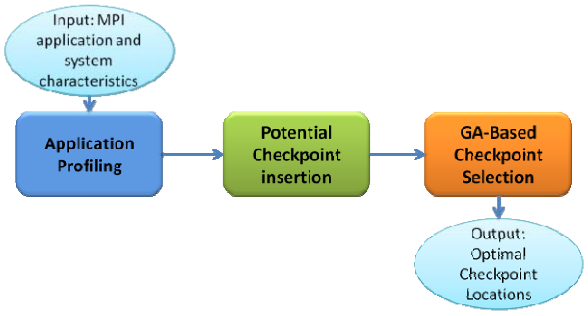
Input: MPI application and system characteristics
Application Profiling
Potential Checkpoint insertion
GA-Based Checkpoint Selection
Output: Optimal Checkpoint Locations

Figure 2. Overview of the optimal checkpoint placement methodology

A. Application Profiling

In our methodology to find optimal checkpoints in a parallel program, first we extract all the messages that are passed as inter-process communication between different cores of a multicore processor. The sent messages and their orders can simply be obtained by tracing the communication statements in a message passing application. This can be achieved because we assume the communication channels are reliable. Furthermore, in [17] it has been shown that for some family of parallel applications, namely, communication deterministic applications, the sequence of messages for any process is the same in any correct execution. There is also a category of send deterministic applications in which the sequence of message omission for any process is the same. In this paper, we focus on these kinds of applications. For each communication statement we need to find the sender and receiver processes and its relative order to the other messages passed in the system. In addition, we assign a logical time to each message and use it in our genetic-based solution.

B. Potential Checkpoint Insertion

In the second phase, for every process, we find places that might be suitable for taking a local checkpoint. For this purpose, we categorize applications by their granularity and propose different potential checkpoint insertion methods for them. The first category includes those applications that are parallelized in a fine-grained way. Recall that a parallel application is fine-grained if its subtasks communicate frequently. For this category, we assume that one potential checkpoint could be placed as a candidate after each sent message. The second category contains those applications that are parallelized in a coarse-grained or embarrassing way. The inter-process communications in this category are usually rare. Most of the applications in this category just communicate at the beginning and end of the program execution. For this category, we suppose that at least one checkpoint should be placed as a candidate after each sent message. If the distance between two consecutive sent messages of a process is more than a predefined limit, more than one potential checkpoint is inserted with equivalent distances in that region. We define this limit based on the failure distribution of the system. In both cases, not only the sender process but also all other processes may take a checkpoint at each potential checkpoint time. This can be handled using a controller that force the processes to checkpoint in particular places by monitoring the communication channels. We work under this assumption to represent our individuals in a simple way. We are going to clarify this phase with an example. Assume that Figure 3 shows a part of a sample program's communication behavior that should be run on a multicore machine with two cores. Each line shows the timeline for a process. We distinguish processes by letters P and Q. We assign a logical time to each interaction denoted by Ti. Assuming this program is a kind of fine-grained parallel program, so we just consider one potential checkpoint after each send message. In this figure, every potential checkpoint is shown by a vertical rectangle. It is important to note that all the processes can put a checkpoint in the time after a sent message and this is not restricted to the sender process only. The potential checkpoints for process P are {CP1, CP2, CP3}. Also the potential checkpoints for process Q denotes by {CQ1, CQ2, CQ3} in this figure.

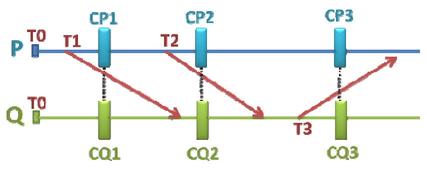

Figure 3. Example of a message passing system

Therefore, after finding all the potential checkpoint places, we have a large set of candidate checkpoints. Of course, it is not necessary to store all of these potential checkpoints. In the next phase, we select the best combination of them to have the minimal wasted time in the event of failure. As we will explain in the next subsection, every individual in our genetic checkpoint placement algorithm is a subset of this large set.

In order to explain our genetic solution better, we define the term global potential checkpoint which stands for all the potential checkpoints with the same logical time, one from each core. For example, in Figure 3 {CP1, CQ1} is the first global potential checkpoint of the system. The second and third global potential checkpoints are {CP2, CQ2} and {CP3, CQ3}, respectively. It should be noted that it doesn't mean that we take checkpoints globally; rather each checkpoint in a global potential checkpoint may or may not be taken.

C. *Genetic-based Optimal Checkpoint Placement Strategy*

In the third phase, a genetic-based algorithm selects the optimum number of checkpoints among all the potential candidate checkpoints found in the previous phase to minimize the total wasted time in a faulty environment. One of the most popular and powerful approaches for search and optimization problems are evolutionary algorithms (EAs) [18]. In this field, several algorithms, e.g. genetic algorithm, are discussed. Genetic algorithm maintains a pool of potential solutions called Chromosomes and produces optimal solutions through combining the good features of existing solutions. In fact, it tries to simulate the process of gradual evolution in nature. In the following we will outline the proposed genetic model to solve this problem. Every genetic algorithm must take into account the following components:

1) A genetic representation of solutions to the problem,
2) An evaluation function which gives the fitness of each individual,
3) Genetic operators that alter the genetic composition of children during reproduction, and
4) A way to create an initial population of solutions, conditions to terminate the algorithm, values for the parameter that the genetic algorithm uses, such as population size, probabilities of applying genetic operators, etc.

Each of the mentioned components greatly affects the solution obtained and the performance of the genetic algorithm.

In the following subsections, we examine each of them for the problem of finding optimal checkpoint placement for parallel applications. The general procedure of our genetic algorithm is summarized in Figure 4.

```
Input : A set of potential checkpoints times, messages transmit in the
multicore system, and the system failure distribution
Output : The optimal checkpoint placement set
Begin
    Generate random initial population of checkpoint
    placement strings $Pop$;
    For each $ind \in Pop$, compute $Fitness(ind)$;
    While (termination condition is not met) do
        Crossover ($Pop$);
        Mutation ($Pop$);
        For each $ind \in Pop$, compute $Fitness(ind)$;
        Selection ($Pop$) and Mutate again if needed;
    End
End
```

Figure 4. Genetic checkpoint placement algorithm

1) *Representing the Search Space*

The problem representation has a major role in the genetic algorithm. Each chromosome represents a potential checkpoint placement solution for the system. In this paper, chromosome length is equal to the number of potential checkpoints that have been extracted from profiling the application. Each gene in chromosome is an integer value that represents which cores take their potential checkpoints in a specific global potential checkpoint location. Recall that every potential checkpoint can be placed after each send message or after a predefined limit in all active processes. Since we assigned a time to each global potential checkpoint, we have an ordered set of potential checkpoints for all cores from the beginning to the end of the program execution. There is a one to one correspondence between each gene in the chromosome and each global potential checkpoint in the system. For each global potential checkpoint, the content of its corresponding gene represents which potential checkpoints are taken and which are not taken. For example, if the first gene is 0, it shows that none of the potential checkpoints of the first global potential checkpoint are taken. As another example, if the third gene is 1, it means that just the corresponding potential checkpoint in the third global potential checkpoint location for the first core is taken and other cores don't take any checkpoint in this place. To clearly explain how we represent chromosomes, first suppose that there is just one core in the system. The chromosome of this core is a binary string with the length equal to the number of global potential checkpoints. The value of each gene for a core is equal to 0 or 1. If a gene is 1 then its corresponding checkpoint is taken and if it is 0 the corresponding checkpoint is not taken. Thus, each chromosome for a core indicates which potential checkpoints are taken. For example, if we have 5 potential checkpoints (or 4 messages) in a fine-grained program, then the chromosome {10011} means CP1 is taken, CP2 and CP3 are not taken, and CP4 and CP5 are taken. But, in a multicore system, each chromosome is the result of merging core's chromosomes. In other words, we concatenate core's genes in every global potential checkpoint position with an arbitrary order and convert it to integer which is used as genes in our multicore system. Here is an example that makes the explanation more clear. Consider Core 1, Core 2, and Core 3 chromosomes are equal to {1 0 0 1 1}, {1 1 0 0 1}, and {0 1 1 1 1}, respectively. Merging these chromosomes, the corresponding chromosome for the whole system is {011 110 100 101 111}. In order to reduce the chromosome length we convert it to integer which is equal to {3 6 4 5 7}.

2) *Fitness Function*

The main objective of this problem is to minimize the total wasted time in the program execution in a faulty environment. The total wasted time is defined as sum of all reprocessing times caused by faults and checkpointing overhead which is the time to store checkpoints on the stable storage. After injecting faults in the system, the fitness value is calculated for each individual considering the rule mentioned in Figure 1. The less the fitness value, the more rewards the individual received.

3) *Genetic Operators*

Genetic operators modify individuals within a population to produce new individuals for another combination of checkpoint placement. Historically, selection, crossover, and mutation have been the most important genetic operators. Selection forms a new generation by choosing the suitable individuals from the new and old populations. We implement two kinds of selection namely proportionate selection and elitism. Individuals with less fitness values are selected as the next generation population in every iteration using one of the selection mechanisms. Crossover takes two individuals and produces new individuals, by swapping portions of their bits. Two types of crossover have been implemented in this paper, which are one-point and two-point crossover. Mutation just changes some random bits within selected individuals. We use both substitution and inversion operations to mutate individuals. The probability of each operation is chosen in such a way to have a balance between exploration and exploitation.

4) *Initial Population and Termination Condition*

Initial population is randomly generated. Each gene is equal to a random integer value between 0 and $2^{number\ of\ cores} - 1$.

The algorithm runs as long as the user defined number of iterations has not been elapsed yet or when the best fitness values do not grow any further. In each generation, the best old and new generation fitness values are compared. If the best individual fitness value doesn't vary for N generations, the algorithm execution will be terminated. N is found for each application separately based on the problem size.

IV. EXPERIMENTAL RESULTS

In our experiments, we focus on MPI applications because of its popularity (although our method can easily be mapped on any other message passing programs). All of these applications are either communication deterministic or send deterministic. Generally, an MPI application is decomposed and run among many computing nodes, where message passing mechanism is used for subtasks communications. In this section, we use Fortran/MPI version of NAS parallel benchmarks (NPB3.3) [19] to evaluate the performance of our optimal checkpoint placement strategy and compare it with optimal coordinated checkpointing. These programs have been executed on a multicore machine with 4 cores running MS Windows 7 operating system.

In MPI applications routine calls may belong to one of the following classes:

- Routine calls used to initialize, terminate, manage, and synchronization.
- Routine calls to create data types.
- Routine calls used to communicate between exactly two processes, one sender and one receiver (Pair communication)
- Routine calls used to communicate among groups of processors (Collective communication)

We mainly focus on communication calls to extract the communication model of the system as explained in Section III. We run different NPB benchmarks on a multicore machine in a fault-free environment and extract the order of communications. We use this system communication model as input to our genetic-based solution and find the optimum points for checkpoint placement. Table 1 summarizes the characteristics of benchmarks that have been run on a fault-free quad-core machine. The number of messages passed in each benchmark as well as their execution time in a fault free environment is reported in this table. Also, the third row indicates the type of each benchmark. In this row *F.G.* is short for fine-grained programs and *E.P.* is an abbreviation for Embarrassingly Parallel program. As it can be seen in this table, among all benchmarks, the only embarrassingly parallel code is the EP benchmark which runs without significant communication.

TABLE 1. BENCHMARK CHARACTERISRICS

Benchmark	*BT*	*CG*	*EP*	*FT*	*MG*	*SP*
Number of Messages	1992	1523	48	120	1437	4064
Execution Time(Second)	941	319	336	502	89	1250
Type	F.G.	F.G.	E.P.	F.G.	F.G	F.G

All the proposed static optimal checkpoint placement methods for multiprocessor systems thus far use a global periodic checkpointing approach. Therefore, in order to evaluate our proposed algorithm, we compare it with the best periodic checkpoint placement method. We find the best periodic interval by simulation, and calculate the wasted time using this optimum interval. In our simulation for finding the best periodic interval, we vary checkpointing interval in an acceptable region, which is not less that checkpoint overhead and not more than one third of program execution time, and then calculate the wasted time for each case. The interval with least wasted time is the best checkpointing interval for global periodic method.

Benchmarks have been executed under different fault injection scenarios and the average wasted times (in seconds) have been reported. In each simulation, the failure rate is changed and faults are injected randomly. We change the number of faults from 1 to 20. For each number of injected faults, the scenario is repeated 20 times. The average wasted time for these scenarios is calculated and compared to the best periodic checkpoint placement method in Table 2. As it can be seen, our method greatly reduces the programs' wasted time. This is due to the omission of taking unnecessary checkpoints. As it can be observed in Table 2, our proposed method reduces the average wasted times by 64% on average compared with the best periodic method.

TABLE 2. AVERAGE WASTED TIME OF TEST PROGRAMS USING TWO DIFFERENT CHECKPOINTING METHODS (CHECKPOINT OVERHEAD = 3S)

Benchmark	*Avg wasted time using our proposed Method (Second)*	*Avg wasted time using Periodic Checkpointing (Second)*	***Improvement (%)***
BT	84.6	154.9	**45%**
CG	71.8	432.8	**83%**
EP	58.8	212.9	**72%**
FT	108	359.4	**69%**
MG	48	113	**57%**
SP	106	235	**54%**

As it can be seen in Table 1, our selected benchmarks are not run for a very long time and transient faults are rarely occur in these periods of time in practice. So, injecting the mentioned number of faults in their execution time periods may not lead to a real situation. The reason behind injecting high number of faults is to evaluate our proposed method and to show how it outperforms the conventional methods in both high and low failure rate conditions.

Figure 5 shows the comparison between average number of checkpoints for the chosen benchmarks using the two different checkpoint placement strategies. As it can be observed, our strategy reduces the number of checkpoints considerably. Our proposed checkpoint placement method decreases the number of checkpoints 47% compared with the best periodic checkpoint placement method.

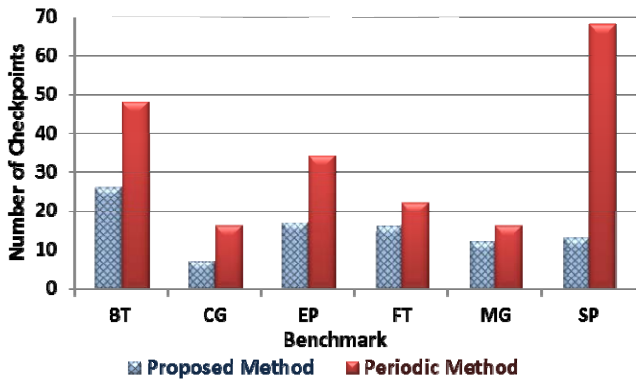

Figure 5. Comparison between number of checkpoints for two different checkpoint placement strategies

From experimental results, we can conclude that our proposed checkpoint placement strategy provides better performance comparing optimal periodic checkpointing under different fault injection scenarios. The execution time of our method depends on the number of messages passed in each application. As the number of messages gets larger, the execution time of our proposed algorithm gets slightly slower. However, this is not a major concern since this is an offline method that doesn't affect the application execution time.

V. CONCLUSION

Homogeneous and heterogeneous multicore processors are widely deployed in the current and also the next generation of supercomputers. In this paper, to the best of our knowledge, for the first time the problem of optimal checkpoint placement in multicore processors has been solved using genetic-based solution. The second contribution of this paper is that the solution is not restricted to coordinated periodic checkpointing rather each core takes its local checkpoints independently. This is possible due to process' dependency extraction phase. Experimental results show that this method leads to better performance comparing the existing models.

It should be noted that the method proposed in this paper is implementable on any message passing multicore environments. It is enough to know the message pattern in a message passing application as well as the failure distribution of the system and the proposed method finds the optimal checkpoint placement efficiently.

REFERENCES

[1] I. Koren, C. Krishna, Fault-Tolerant Systems. Morgan Kaufmann, San Francisco, 2007.

[2] E. Elnozahy, L. Alvisi, Y. Wang, D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, vol. 34, no. 3, pp. 375–408 , 2002.

[3] S. Kalaiselvi, V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," Sadhana, vol. 25, no. 5, pp. 489-510, 2000.

[4] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, S. Scott, "A Reliability-aware Approach for an Optimal Checkpoint/Restart Model in HPC Environments," IEEE International Conference on Cluster Computing, 2007.

[5] J.T. Daly, "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps, " ICCS , 2003.

[6] J.S. Plank, M.A. Thomason, "The Average Availability of Parallel Checkpointing Systems and Its Importance in Selecting Runtime Parameters," IEEE Proc. Int'l Symp. On Fault-Tolerant Computing, 1999.

[7] Y. Liu, C. Leangsuksun, H. Song, S.L. Scott, "Reliability-aware Checkpoint /Restart Scheme: A Performability Trade-off," IEEE International Conference on Cluster Computing , 2005.

[8] Y. Ling, J. Mi, X. Lin, "A Variational Calculus Approach to Optimal Checkpoint Placement," IEEE Trans. Computers, vol. 50, no. 7, 699–707 , 2001.

[9] A.J. Oliner, L. Rudolph, R. Sahoo, "Cooperative Checkpointing Theory," In Proceedings of the Parallel and Distributed Processing Symposium , 2006.

[10] A. J. Oliner, "Cooperative Checkpointing for Supercomputing Systems," Master's thesis, Massachusetts Institute of Technology, 2005.

[11] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," Proc. 20th Annual Int'l Conf. Supercomputing (SC '06), 2006.

[12] M. Chtepen, F. Claeys, B. Dhoedt, F. DeTurck, P. Demeester, P. Vanrolleghem, "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolarant Grids," IEEE Transactions on Parallel and Distributed Systems, 2009.

[13] S. Agarwal, R. Garg, M. Gupta, J. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," Proc. 18th Ann. Int'l Conf. Supercomputing (SC '04), 2004.

[14] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, S. L. Scott, "Reliability-aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," Cluster Computing and the Grid, 2008.

[15] Y. Li, Z. Lan, "FREM: A Fast Restart Mechanism for General Checkpoint/Restart, " IEEE Transactions on Computers, Vol. 60, No. 5, 2011.

[16] J-M. Hélary, R.H.B. Netzer, M. Raynal, "Consistency Issues in Distributed Checkpoints," IEEE Transactions on Software Engineering, vol. 25, no. 2, pp. 274-281, 1999.

[17] F. Cappello, A. Guermouche, M. Snir, "On Communication Determinism in Parallel HPC Applications," 19th International Conference on Computer Communications and Networks, 2010.

[18] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing. Berlin Heildberg, Springer-Verlag, 2003.

[19] NAS Parallel Benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.htm