ILP-BASED OPTIMAL CHECKPOINT PLACEMENT IN MULTICORE PROCESSORS

[1]ATIEH LOTFI, [2]SAEED SAFARI

[1]M.Sc., School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Iran
[2]Assistant Prof., School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Iran

E-mail: [1] a.lotfi@ece.ut.ac.ir , [2] 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 is one of the prevalent fault tolerant techniques reducing the execution time of long-running programs in presence of failures. 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. This paper presents a novel 0-1 integer linear programming (ILP) formulation for solving optimal checkpoint placement problem for parallel applications running on a multicore machine. Our experimental results demonstrate that our solution leads to a better execution time saving over existing methods.

Keywords: *Fault Tolerance; Optimal Checkpoint Placement; Multicore Architectures; Integer Linear Programming*

1. INTRODUCTION

Recent changes in the high performance parallel computing make fault tolerant system design important. The higher number of processors increases the overall performance, but it also increases the probability of failures. Moreover, there are many applications, like some optimization problems, that take days or even weeks to execute. As the execution time of a program becomes longer, the probability of failure during execution as well as 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 never gets 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 start the application from scratch in the event of failure; instead, 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. However, checkpoinitng comes with some overhead affecting the execution time of the program. Therefore, 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 taken 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 arise [2]. On the other hand, in coordinated checkpointing, all processes take checkpoints simultaneously, which is easy to implement but often causes significant overhead. In addition, 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 can take 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 can lead 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 system can fail any time and there is a failure detection mechanism that can detect 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 make an integer linear programming (ILP) formulation to find the optimal checkpoint placement for that parallel application. We also consider the system failure is a Poisson process.

The rest of this paper is organized as follows; in section 2 related works are studied. Section 3 introduces our checkpoint placement strategy. Experimental results are given in Section 4. Finally, conclusions are drawn in the last section.

2. RELATED WORKS

There is a trade-off between the overhead imposed by checkpointing and the amount of works 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 works use equidistance checkpoints, but it has been shown that for realistic system failures, periodic checkpointing is not the best choice [9].

In [7], an optimal solution has been proposed for periodic checkpoint placement problem in parallel applications. It considers the occurrence of a failure as a random distribution with constant failure rate, λ. The optimal checkpoint placement interval (T_c) approximated in this paper depends on the checkpoint overhead (T_s: time to save context files for each checkpoint). The exact formula is:

$$\mathrm{T_c} = \sqrt{\frac{2\mathrm{T_s}}{\lambda} - \mathrm{T_s}^2}. \qquad (1)$$

We compare our proposed method with this model.

Despite the importance of the communication induced protocols, there is no model to calculate the optimal checkpoint interval which minimizes the total execution time of a parallel program using these protocols. In this paper, we don't limit all the processors in our multicore architecture to checkpoint simultaneously and periodically. Instead, each core can take its checkpoints individually and provide coordination in case of dependency.

3. OPTIMAL CHECKPOINT PLACEMENT STRATEGY

The problem solved 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 execution time of the program in a faulty environment. 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 transmission delays are unpredictable but finite. 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 does not include messages received but not sent yet (orphan messages) [11]. It should be noted that a consistent global checkpoint can include sent but not received messages. With this assumption, all the messages that have been sent but not received must be restored in order to roll back the system correctly.

Our methodology for solving optimal checkpoint placement in a parallel program can be divided into five phases, which is summarized in the figure1.

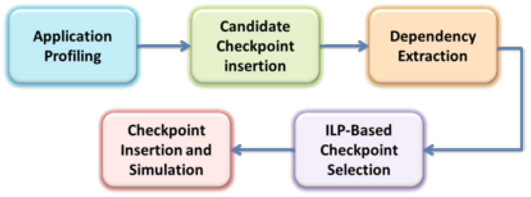
Application Profiling
Candidate Checkpoint insertion
Dependency Extraction
ILP-Based Checkpoint Selection
Checkpoint Insertion and Simulation

Figure 1. Overview Of The Optimal Checkpoint Placement Methodology

Step 1: Application Profiling

Firstly, we extract all the messages that are passed as inter-process communication. The send and receive messages and their relative order can simply be obtained by tracing the communication statements in a message passing application. For each communication statement we need to find the sender and receiver processes and its relative order to other messages passed in the system. In addition, we assign a logical time to each message.

Step 2: Potential Checkpoint Insertion

In the second phase, for every process, we find regions (bounds) that taking a local checkpoint might be suitable. For this purpose, we categorize applications by their granularity and suggest different candidate checkpoint insertion method 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 checkpoint should be placed as a candidate between two consecutive interaction of that process with outside (i.e. between each two consecutive send or receive messages). 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. For this category, we suppose that at least one checkpoint should be placed as a candidate between two consecutive messages. If the distance between two consecutive messages of a process is more than a limit, more than one candidate checkpoint is inserted in that region. We define this limit based on the failure distribution of the system. We are going to clarify this phase with an example. Assume that figure 2 shows a part of a sample program's communication behavior that should be run on multicore machine with two cores. Each line shows the timeline for a process. 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 put one checkpoint between each consecutive message. The candidate checkpoints for process P are {CP1, CP2, CP3, CP4}. Also, the candidate checkpoints for process Q denotes by {CQ1, CQ2, CQ3, CQ4}. According to the explanations of this phase, each checkpoint is bounded between its former and latter messages time. As an example, checkpoints CP1 and CP2 should satisfy the equations T1≤CP2_time ≤T3 and T4≤CQ3_time ≤T5, respectively.

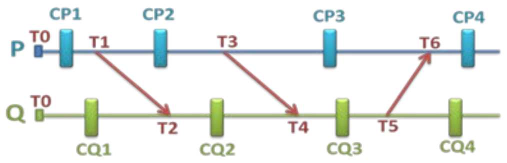

Figure 2. Example Of A Message Passing System

Step 3: Dependency Extraction

In order to find the best candidates in the parallel program for checkpoint placement, communication statements are analyzed to find the existing dependency between processes. So far, we have chosen suitable places in each process for inserting potential local checkpoints and here we find all the possible consistent recovery lines for each candidate checkpoint based on their communication pattern with other processes. When analyzing the program's communication pattern, it is important to note that a rollback to any arbitrary checkpoint should not create orphan messages. So, if a process rolls back to a checkpoint before its send message, the receiver process should roll back to a checkpoint before its receive time as well. After finding all the recovery lines, all the useless checkpoints (those that do not belong to any of the global consistent recovery lines) will be omitted from the potential candidate checkpoint list of that process. Therefore, given a parallel program with a number of processes, at the end of this phase we determine all the potential local checkpoints for each process and a list of consistent recovery lines for each checkpoint. So, for every checkpoint, a logical time interval in which it can be placed and the list of all dependent checkpoints available in other processes is extracted. List of *dependent checkpoints* for checkpoint *i* of process *P* are those checkpoints that other processes should rollback to them if process *P* rolls back to its checkpoint *i*. For example, in figure 2, CQ2 is in the list of dependent checkpoints for CP2.

Step 4: ILP-based Optimum Checkpoint Selection

In the fourth phase, an integer linear programming model selects the optimum number of checkpoints among all the potential candidate checkpoints to minimize the execution time in a faulty environment. The following two subsections explain the notations, definitions, and ILP equations.

1) Definitions and Notations

We consider a multicore architecture with *n* cores that runs a parallel application consisting of *n* processes (in general, there is no restriction that the number of processes and cores should be the same; here, we make this assumption for simplicity.) Suppose that the maximum number of potential checkpoints for a given process extracted from previous phases is *m*. For each checkpoint *j* in process *i*, we use a 0-1 integer variable $is\_taken_{i,j}$ to denote whether that checkpoint will be taken in the final optimum checkpoints or not. If $\text{is\_taken}_{i,j}$ is 1, the jth checkpoint of the ith process will be selected and taken; otherwise, it will be omitted. Another variable that is needed for every candidate checkpoint is $checkpoint\_time_{i,j}$, denoting the time of each checkpoint that previously bounded in step 2. For each checkpoint j in process i, there is an array of length n, $dependency\_list_{i,j}[n]$, that indicates all the dependent checkpoints for jth checkpoint of ith process and it contains the output of step 3. For example, $\text{dependency\_list}_{i,j}[k]$ holds the dependent checkpoint number of process k for jth checkpoint of the ith process. The overhead of storing each checkpoint extracted from simulation indicates by *CPoverhead* variable. We suppose failures can occur randomly everywhere in the execution of each process with a predetermined failure rate.

2) Problem Formulation

The mathematical problem is to compute the optimum number of checkpoints that minimizes the completion time of the parallel computation under various failure assumptions. In the following, we will outline the ILP model to solve this problem.

The objective is to find the best checkpoint placement in order to minimize the total execution time of the parallel program (Equation 2):

$$Minimize\ \sum_{i=1}^{n} Execution\_Time(Process_i)\ . \qquad (2)$$

The first limiting equation is related to the output of step 2, i.e. the bounded interval of each checkpoint. As explained before, we extract an acceptable bound for each checkpoint; here, we denote them by $lower\_bound_{ij}$ and $upper\_bound_{ij}$ for the lower bound and upper bound of jth checkpoint of ith process, respectively. This can be formulated as the following equation:

$$\forall\, i \in \{1, \dots, n\}, j \in \{1, \dots, m\}:$$

$$lower\_bound_{ij} \le checkpoint\_time_{ij} \le upper\_bound_{ij}. \qquad (3)$$

If a checkpoint is chosen, all of its dependent checkpoints should be taken as well. This can be formulated using equation 4. In simple words, if jth checkpoint of ith process is selected, it forces all other checkpoints in its dependency list to be taken:

$$\forall\, i,l \in \{1, \dots, n\}, j,k \in \{1, \dots, m\}, (l,k) \in dependecy\_list_{i,j}\!:\ is\_taken_{ij} \le is\_taken_{lk} \qquad (4)$$

In order to minimize the total execution time of the parallel application, we compute the execution time of each process separately. The execution time of each process is calculated by adding the execution time of that process in a non-faulty environment and the wasted times due to each checkpoint and failures. We consider both the overhead caused by inserting each checkpoint which is the amount of time needed to store the checkpoint on the stable storage and the rollback time, which is the time lost between the fault occurrence time and the last system stable state that should be executed again. In other words, the execution time of each process is calculated as follows: (Please note that equation 5 is not an ILP equation and just mentioned here for more clarification, and its equivalent ILP formula is stated by equation 6)

$$\begin{aligned} &ExecutionTime(Process_i) \\ &= nonFaultyExecutionTime(Process_i) \\ &+ Number\ of\ Checkpoint \\ &\times checkpoint\ overhead \\ &+ rollback\ and\ recovery\ times \end{aligned} \qquad (5)$$

The last ILP equation computes the execution time of each process in the event of failure with its chosen checkpoints. Suppose F random faults are injected in our system. In this equation, the execution time of each process is calculated as explained in equation 5. The rollback time is the amount of total lost work in all processes caused by that fault, i.e. the sum of distance between the failure point and the nearest selected checkpoint (CP_x) in the faulty process and also non-faulty processes which contains *dependent* CP_x checkpoints:

$$\forall\, i \in \{1,\dots,n\}: \quad ExecutionTime(Process_i) = nonFaulty\_ExecTime(Process_i) + \sum_{j=1}^{m} is\_taken_{ij} * CPoverhead + \sum_{k=1}^{F} Min_{j=1}^{CP_{time(j)}<Fault\_Time_{(k)}} (is\_taken_{ij} * [fault\_Time_k - checkpoint\_time_{ij} + \sum_{l=0}^{N(dependency\_list(i,j))} (fault\_Time_k - checkpoint\_time_{p_l cp_l})]) \qquad (6)$$

The ILP formulation for optimal checkpoint placement problem is summarized in Figure 3.

$Minimize\ \sum_{i=1}^{n}(Execution\_Time(Process_i))$ *subject to:*

1. $\forall\, i \in \{1,\dots,n\}, j \in \{1,\dots,m\}: \ lower\_bound_{ij} \le checkpoint\_time_{ij} \le upper\_bound_{ij}$.

2. $\forall\, i,l \in \{1,\dots,n\}, j,k \in \{1,\dots,m\}, (l,k) \in dependecy\_list_{(i,j)}: \ is_{taken_{ij}} \le is_{taken_{lk}}$

3. $\forall\, i \in \{1,\dots,n\}: \ ExecutionTime(Process_i) = nonFaulty\_ExecTime(Process_i) + \sum_{j=1}^{m} is\_taken_{ij} * CPoverhead + \sum_{k=1}^{F} Min_{j=1}^{CP\_time(j)<Fault\_time(k)} (is\_taken_{ij} *[fault\_Time_k - checkpoint\_time_{ij} + \sum_{l=0}^{N(dependency\_list(i,j))}(fault\_Time_k - checkpoint\_time_{p_l cp_l})])$

Figure 3. ILP formulation for optimal checkpoint placement in a parallel application

Step 5: Checkpoint Insertion and Simulation

Finally, in order to evaluate our proposed method, we model our system and inject random faults. All the selected checkpoints are inserted in their determined places and then the execution time of the program is estimated.

4. EXPERIMENTAL RESULT

In our experiments, we focus on MPI applications because of its popularity (albeit our method can be easily mapped on any other message passing programs). 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) [12] and three C/MPI benchmarks 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. The system runs windows 7. We used a free ILP solver, called lpsolve [10], to solve the ILP equations for each program.

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 step 1 of our method. We run different NPB benchmarks on a multicore machine in a fault-free environment and use the MPI timer, namely MPI_Wtime() routine to extract the time of each communication. Using these values in hand, we model our system and trace the dependency of different processes. Then, having all the necessary inputs of ILP formulation, we find the optimum points for checkpoint placement. Table 1 summarized the characteristics of benchmarks that have been run on a fault-free quad-core machine. In this paper, we use BT, CG, MG, and FT benchmarks of MPI NPB 3.3. The number of messages passed in each benchmark is also shown in this table.

Table 1. . BENCHMARK CHARACTERISRICS

Benchmark	*Language*	*Number of Messages*
BT	Fortran	192
CG	Fortran	117
MG	Fortran	123
FT	Fortran	120
Matrix Multiplication	C	20
PI Calculation	C	12
Matrix Addition	C	9

In order to evaluate our proposed model, we compare our checkpoint placement method with the model proposed in [7] which solves the optimal checkpoint placement problem for MPI applications using a coordinated checkpoinitng protocol (Equation 1).

Table 2 shows the comparison of the average program execution times. The programs have been executed under different fault injection scenarios and the average execution times (in seconds) have been reported. In each simulation, the failure rate is changed and faults are injected randomly during the programs' execution. As it can be seen, our method mostly reduces the programs' execution time. This is due to the omission of taking unnecessary checkpoints.

Table 2. Average Execution Time Of Test Programs Using Two Different Checkpointing Methods (Checkpoint Overhead = 5s)

Benchmark	*Execution time using Periodic Checkpointing([7]) (Second)*	*Execution time using our proposed Method (Second)*
FT	12665	11896
CG	12770	11975
MG	13946	13668
BT	8517	9031
Matrix Multiplication	4260	1550
PI Calculation	3347	1730
Matrix Addition	6057	4903

Figure 4 shows the average execution time of the chosen benchmarks using the two different checkpointing placement techniques. As it can be seen, our proposed checkpointing placement strategy mostly provides better performance comparing periodic checkpointing [7]. The only exception is BT benchmark. This is due to this fact that when the number of messages increases, the number of variables and equations in our ILP formulation grows and consequently the time needed to solve the ILP formulas grows exponentially. And, since we set a time limit on the execution time of ILP solver, it may not find the global optimum point.

5. 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 integer linear programming formulation. The second contribution of this paper is that the solution is not restricted to coordinated checkpointing rather each core can take its local checkpoints independently and force other cores to take a checkpoint in case of dependency. This is possible due to process' dependency extraction phase. Experimental results show that this method leads to better performance than the other existing models. As the problem size becomes larger the number of variables and equations in our ILP formulation grows and consequently the time needed to solve the ILP formulas grows exponentially. To overcome this limitation, we are going to use evolutionary algorithms in our future works to solve the optimal checkpoint placement problem.

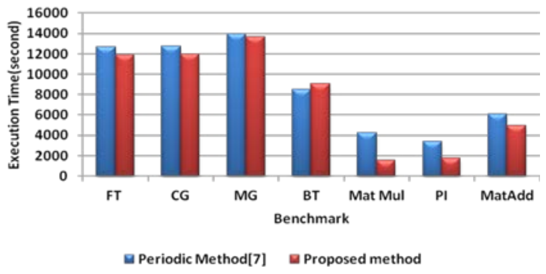

Figure 4. The Average Execution Time Comparison For Two Checkpointing Methods

REFRENCES:

[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] 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.

[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]lpsolve,http://lpsolve.sourceforge.net/5.5/index.htm

[11] 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.

[12] NAS Parallel Benchmarks:

http://www.nas.nasa.gov/Resources/Software/npb.html.