An On-Line Algorithm for Checkpoint Placement*

Avi Ziv
IBM Israel, Science and Technology Center
MATAM - Advanced Technology Center
Haifa 31905, Israel
avi@haifa.vnat.ibm.com

Jehoshua Bruck
California Institute of Technology
Mail Code 136-93
Pasadena, CA 91125
bruck@paradise.caltech.edu

Abstract

Checkpointing is a common technique for reducing the time to recover from faults in computer systems. By saving intermediate states of programs in a reliable storage, checkpointing enables to reduce the lost processing time caused by faults. The length of the intervals between checkpoints affects the execution time of programs. Long intervals lead to long re-processing time, while too frequent checkpointing leads to high checkpointing overhead. In this paper we present an on-line algorithm for placement of checkpoints. The algorithm uses on-line knowledge of the current cost of a checkpoint when it decides whether or not to place a checkpoint. We show how the execution time of a program using this algorithm can be analyzed. The total overhead of the execution time when the proposed algorithm is used is smaller than the overhead when fixed intervals are used. Although the proposed algorithm uses only on-line knowledge about the cost of checkpointing, its behavior is close to the off-line optimal algorithm that uses a complete knowledge of checkpointing cost.

1 Introduction

Checkpointing is a common technique for reducing the execution time of programs in the presence of faults. Checkpointing consists of saving intermediate states of the task in a reliable storage, and upon a detection of a fault, restoring the previous stored state. Hence, checkpointing enables to reduce the time to recover from a fault, while minimizing the lost processing time.

The interval between checkpoints affects the execution time of a program. On one hand, inserting more checkpoints reduces the re-processing time after failures. On the other hand, inserting more checkpoints increases the checkpointing cost and the program execution time. This trade-off between the re-processing time and the checkpointing overhead leads to an optimal checkpoint placement strategy, that optimizes certain performance measures [3, 4, 5].

Considerable theoretical work has been devoted to analyzing checkpointing schemes and determining the optimal checkpoint placement strategy. Brock [1] and Duda [4] analyzed the execution time of a program with and without checkpoints. Gelenbe [5] showed that to maximize availability in transactions systems checkpoint intervals should be deterministic and of the same length. L'Ecuyer and Malenfant [7] derived a numerical approach for availability in dynamic checkpointing strategies when the fault rate is not constant. Nicola and van Spanje [9] compared analysis and optimization of several checkpointing models that differ in the checkpoint's placement and fault occurrence in transaction systems. Coffman and Gilbert [3] described optimal strategies for placement of checkpoints in a single program.

In all the work described above, it is assumed that the checkpointing overhead does not depend on the time the checkpoint is taken. Another approach for placing checkpoints, that takes into account the change of checkpointing overhead over time, can be found in [2]. In that paper Chandy and Ramamoorthy proposed an algorithm, based on a graph theoretic method, for a placement of checkpoints that allows the programmer to decide where to place checkpoints according to an a-priory knowledge about the cost of checkpointing. In [10], Toueg and Babaoğlu derive an optimal algorithm to place checkpoints when there is a small number of possible locations for the checkpoints and the cost of checkpointing and recovery at each such location is known. The CATCH tool [8] is a compiler assisted technique that helps to improve the placement of checkpoints using information about the cost of checkpointing that is gathered in previous executions of the program.

One of the operations that is performed at a checkpoint is saving the program state on a stable storage. Therefore,

*The research reported in this paper was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, and by a grant from the IBM Almaden Research Center, San Jose, California.

the size of the program state is one of the main factors that determine the checkpointing cost. During the execution of a program, the size of its state is dynamically changing due to allocation and deallocation of memory blocks. While the size of the program's state might not be known in advance, it is possible to keep track of the allocation and deallocation operations and to know the state size of the program at the current time. Therefore, an estimation of the checkpointing cost at the current time can be obtained.

In this paper we present a new on-line algorithm for placement of checkpoints. The algorithm keeps track of the state size of the program, and uses it to estimate the cost of checkpointing at the current point of execution. The knowledge about the cost of checkpointing is used when deciding at which points in the program to place checkpoints.

The main idea in the algorithm described here, is to look for points in the program in which placing a checkpoint is the most beneficiary. The algorithm tries to find points in the program in which the state size is small, and use these points for checkpoints. If such points are found, checkpoints are placed at these points with small intervals between the checkpoints, so that the re-processing time after a fault is small. If no such point is found after a period of time, a checkpoint is taken at a point with higher cost to avoid long re-processing time in case of a fault. In this case the interval between the checkpoints is longer, to reduce the checkpointing overhead. The main difference between the adaptive checkpointing presented in [8] and the algorithm presented in this paper is that our algorithm not only looks for points with low checkpointing cost, it also changes the interval between checkpoints to fit the current cost of checkpointing.

We analyze the performance of the new on-line placement algorithm for the simple case when the program have only two possible state sizes, and the state size changes according to a Markov chain. Comparison of the average execution time of a program when the proposed algorithm is used to the average execution time when the intervals between checkpoints are fixed shows that the overhead for the on-line algorithm is lower. Although the proposed algorithm uses only the past and present information about the cost of checkpointing when deciding whether or not to place a checkpoint, its performance is close to the optimal off-line algorithm given in [10], that knows the cost of all checkpoints ahead of time. Comparison of the decision on checkpoint placement done by the two algorithm shows that both algorithms can avoid long periods of high cost and efficiently use periods of low cost checkpointing.

While the program might not know ahead of time how its state size is going to change, it can detect changes in the state size just before they occur. This additional knowledge can be used to farther improve the placement algorithm. We show how the on-line algorithm can use this knowledge to place checkpoints just before the state size increases, and what benefits this knowledge can provide. Analysis of the modified algorithm shows that its performance is even closer to the optimal algorithm than the on-line algorithm.

The rest of the paper is organized as follows. In Section 2, we describe the model of the program and environment we use in this paper and two existing placement strategies which we use to compare the new algorithm with. In Section 3, we describe the new on-line algorithm and show how to analyze the average execution time of a program when the algorithm is used. In Section 4, the performance of the new algorithm is compared with the fixed interval placement strategy and the optimal off-line algorithm. In Section 5, a modification to the algorithm, that enables to take advantage of detection of an increase in the state size before they occur is presented. In section 6, we discuss some practical issues regarding the implementation of the algorithm. Section 7 concludes the paper.

2 Background

In this paper we are interested in the average execution time of a program with checkpoints in a system that is vulnerable to faults, and the effects of different checkpointing placement strategies on the execution time. The faults in the system occur according to a Poisson process with rate λ. At some points during the execution of the program checkpoints are placed. At each checkpoint the state of the program is saved on a stable storage. After a fault is detected, the program is rolled back to the last saved state and execution is resumed from that point. We assume that the program is executed on a single processor, and that the processor has an internal fault detection mechanism that enables it to detect faults immediately. We also assume that the time to roll the program back after a fault is detected is zero and that faults cannot occur during checkpointing.

While the assumptions that we make are not necessary for the operation of the algorithm presented in this paper, they make the analysis simpler and help to illustrate the advantages of the on-line algorithm.

Using these assumptions, we can calculate the average execution time of a program with faults and checkpoints. The analysis given here is the same one that is used by Duda in [4]. A program of length t[1] is divided into n intervals of length $t_1, t_2, \ldots, t_n$, such that $\sum_{i=1}^{n} t_i = t$. At the end of each interval a checkpoint is placed. The cost of the checkpoint at the end of the i'th interval is c_i. Let T_i be the execution time of the i'th interval, including the checkpointing time at the end of it. The T_i's are random variables and their values depend on the number and locations of the faults

[1] Throughout the paper t, with and without subscripts, denotes productive time (*i.e.,* excludes time spent in checkpointing, repair, recovery and re-processing), while T denotes elapsed time.

that occur when the i'th interval is executed. Note that because of the memoryless property of the faults process and because faults cannot propagate over a checkpoint, the T_i's are independent of each other. The following proposition gives the average execution time of a single interval. The proof for the proposition can be found in [4].

Proposition 1 *Under the assumptions stated above, $\overline{T}_i$ the average execution time of the i'th interval and the overall execution time of the program are*

$$\overline{T}_i = \frac{e^{\lambda t_i} - 1}{\lambda} + c_i, \qquad \overline{T} = \sum_{i=1}^{n} \overline{T}_i. \tag{1}$$

A good metric to measure the performance of a checkpointing placement strategy is the average overhead ratio R, which is defined as the ratio between the average overhead, caused by the checkpointing and the faults, and the program length. In other words

$$R = \frac{\overline{T} - t}{t} = \frac{\overline{T}}{t} - 1.$$

When designing a placement strategy for checkpoints with a goal to minimize the average execution time of a program, or equivalently the overhead ratio R, two factors have to be considered. The first one is the overhead caused by the checkpoints themselves, and the second factor is the re-processing time that is needed after a fault is detected. If the checkpoints are placed close to each other, then the number of checkpoints in the program is large and so is the overhead caused by the checkpoints. However, the re-processing time after a fault has occurred is short. When the checkpoints are far from each other, the number of checkpoints and the checkpointing overhead are low, but long re-processing might be needed after a fault is detected.

This trade-off between the checkpointing overhead and the re-processing time leads to some optimal placement strategy, that minimizes the overhead ratio R. This optimal placement strategy depends on the fault rate in the system and the cost of checkpointing. Next, we describe two existing placement strategies which we use for comparison with the on-line placement algorithm we present in this paper.

Checkpointing with Fixed Intervals

When the cost of checkpointing does not change with time, or when only the average cost is known but not how it is changing with time, the optimal placement strategy is to place the checkpoints in fixed equi-distant intervals [1, 4]. Because the execution time of different intervals are independent of each other when the location of the checkpoints are known, and because placing a checkpoint at a specific point does not effect future intervals, minimizing the overhead ratio for each interval alone leads to the optimal placement strategy. Since the checkpointing cost and the fault rate are the same during the execution, the optimal lengths of all the intervals are identical, and the optimal placement strategy is fixed equi-distant intervals.

When fixed intervals are used, the overhead ratio of the whole program is the same as the overhead ratio for a single interval. Using Proposition 1, the overhead ratio for a single interval is

$$R(t) = \frac{\overline{T}(t)}{t} - 1 = \frac{e^{\lambda t} + \lambda \overline{c} - 1}{\lambda t} - 1\ . \tag{2}$$

where $\overline{c}$ is the average cost of a checkpoint. The optimal interval $\tilde{t}$, that minimizes the average execution time, is roughly equals to

$$\tilde{t} \simeq \sqrt{\frac{2\overline{c}}{\lambda}}.$$

Optimal Placement Algorithm

When checkpoints can be placed only in a finite number of locations and the cost of a checkpoint at each of these locations is known in advance, the optimal placement strategy can be found. In [10], Toueg and Babaoğlu describe an optimal algorithm for checkpoints placement. In this algorithm it is assumed that checkpoints can be placed only at a finite number of points in the program, and that the cost of checkpoints in each such point is known in advance. Using this assumptions, an $O(n^2)$ algorithm, based on dynamic programming technique, is given, where n is the number of points where checkpoints can be placed.

The algorithm assumes that there are n possible locations for checkpoints, numbered $1 \ldots n$, and that the cost of a checkpoint at point i is c_i. The algorithm iteratively finds the optimal placement of checkpoints and the average execution time when this placement is used, when no more than k checkpoints are used, for $k = 1, 2, \ldots, n$.

3 On-Line Placement Algorithm for Checkpoints

The checkpointing cost depends on the point in the program at which the checkpoint is placed. More specifically, the checkpointing cost depends on the size of the program's state at that point. Since the state size of the program changes during the execution due to memory allocation and deallocation operations, the checkpointing cost is changing with time according to some random process. Therefore, the fixed intervals placement strategy is not optimal. On the other hand, the state size of the program is usually not known in advance, and therefore the optimal off-line algorithm for placement of checkpoints is not practical.

While the state size of the program is not known in advance, the program can keep track of its state size by monitoring memory allocation and deallocation operations. By monitoring these operations, the program knows its current state size. Therefore, it can estimate the current cost of checkpointing. In this section, we show how knowledge about the current cost of checkpointing can be used in placement of checkpoints.

The main idea in the algorithm described here, is to look for points in the program in which placing a checkpoint is the most beneficial. The algorithm finds points in the program in which the state size is small, and uses these points for checkpoints. If such points are found, checkpoints are placed at these points with small intervals between the checkpoints so that the re-processing time after a fault is small. If no such point is found after a period of time, a checkpoint is placed at a point with higher cost to avoid long re-processing time. In this case the interval between the checkpoints is longer to reduce the checkpointing overhead.

To demonstrate how a current knowledge about the checkpointing cost can improve the performance of checkpointing schemes, we use the following example. The program has two possible state sizes, s_1 and s_2, such that $s_1 < s_2$. The checkpointing cost when the state size is s_i is c_i ($c_1 < c_2$). The state size of the program changes according to a two state Markov chain with rate of leaving state s_i equal to μ_i.

The algorithm works in the following way. We define two points in time, t_1 and t_2, such that $t_1 \leq t_2$. The algorithm decides whether to place a checkpoint at t, where t is the time since the last checkpoint, according to the following rules:

1. If the state size at t_1 is s_1, then a checkpoint is placed at t_1. The cost of the checkpoint is c_1.

2. If the state size at t_1 is s_2, the system waits until the state size changes to s_1 and a checkpoint is placed at that time. The cost of the checkpoint is c_1.

3. If the state size at t_1 is s_2 and the state size does not change until t_2, then a checkpoint is placed at t_2. The cost of the checkpoint in this case is c_2.

Note that in order to avoid high checkpointing overhead, a checkpoint is never placed before t_1. Also, to avoid long re-processing time, a checkpoint is never placed after t_2. The values of t_1 and t_2 affect the performance of algorithm. By analyzing the overhead ratio of the algorithm, we can find the values of t_1 and t_2 that minimize the overhead ratio. In the following section, we calculate the overhead ratio of the on-line algorithm.

3.1 Analysis of the On-Line Algorithm

As we stated earlier, to focus on the benefits of the proposed algorithm, and simplify the analysis of the proposed algorithm, we assume that faults do not occur during checkpointing, and that the recovery time after a fault is 0. We also assume that the faults are detected immediately.

Lemma 2 *With the above assumptions, R the average overhead ratio when the on-line algorithm for checkpointing placement is used is given by*

$$R = \frac{e^{\lambda t_1} + \frac{\lambda p_2}{\lambda - \mu_2}\left(e^{\lambda t_2} - e^{\lambda t_1 + \mu_2 (t_2 - t_1)}\right) - 1}{\lambda\left(t_1 + \frac{e^{\mu_2 (t_2 - t_1)} - 1}{\mu_2} \cdot p_2\right)} + \frac{(1 - p_2) c_1 + p_2 c_2}{t_1 + \frac{e^{\mu_2 (t_2 - t_1)} - 1}{\mu_2} \cdot p_2} - 1 \quad (3)$$

where p_2 is the probability that the state size at a checkpoint is s_2.

Proof: The proof of the lemma consists of the following propositions that derive the probability of placing a checkpoint at t_2 and t_1, the average length of an interval between checkpoints and the average execution time of such interval.

Proposition 3 *In a steady-state, p_2 the probability that the state size at a checkpoint is s_2 is*

$$p_2 = \frac{\mu_1}{\mu_1 + \mu_2} \cdot \frac{e^{\mu_2 t_1} - e^{-\mu_1 t_1}}{e^{\mu_2 t_2} - e^{-\mu_1 t_1}}. \quad (4)$$

Proof: In a steady-state the probability that the state size at a checkpoint is s_2 satisfies the following equation

$$\begin{aligned} p_2 = {} & \Pr\{\text{state is } s_2 \mid \text{prev. state was } s_2\} \cdot p_2 + \\ & \Pr\{\text{state is } s_2 \mid \text{prev. state was } s_1\} \cdot (1 - p_2) \end{aligned} \quad (5)$$

The state size at a checkpoint is s_2 if, and only if, a checkpoint is placed at t_2, and a checkpoint is placed at t_2 if, and only if, the state size at t_1 is s_2 and the state size does not change in the interval $[t_1, t_2]$. Therefore,

$$\Pr\{\text{state is } s_2 \mid \text{prev. state was } s_2\} = P_{2,2}(t_1) \cdot e^{-\mu_2 (t_2 - t_1)},$$

and

$$\Pr\{\text{state is } s_2 \mid \text{prev. state was } s_1\} = P_{1,2}(t_1) \cdot e^{-\mu_2 (t_2 - t_1)},$$

where $P_{1,2}(t_1)$ and $P_{2,2}(t_1)$ are the transition probabilities from states s_1 and s_2 respectively to s_2 at time t_1 given by

$$\begin{aligned} P_{1,2}(t_1) &= \frac{\mu_1}{\mu_1 + \mu_2}(1 - e^{-(\mu_1 + \mu_2) t_1}), \\ P_{2,2}(t_1) &= \frac{\mu_1}{\mu_1 + \mu_2} + \frac{\mu_2}{\mu_1 + \mu_2} e^{-(\mu_1 + \mu_2) t_1}. \end{aligned}$$

Assigning these values to Eq. (5) and solving for p_2 yields Eq. (4). ∎

Note that p_2 is less than or equal to the steady-state probability of s_2, and it can be close to 0 for high μ_2. It means that the proposed algorithm uses the cheaper checkpoint more often than algorithms that do not consider the current checkpointing cost.

Proposition 4 *In a steady-state, p_1 the probability that a checkpoint is placed at t_1 is*

$$p_1 = 1 - p_2 e^{\mu_2(t_2-t_1)}. \tag{6}$$

Proof: A checkpoint is placed at t_2 if, and only if, it was not placed at t_1 and the state size remained s_2 in the interval $[t_1, t_2]$. Therefore,

$$p_2 = (1-p_1) \cdot e^{-\mu_2(t_2-t_1)},$$

or

$$p_1 = 1 - p_2 e^{\mu_2(t_2-t_1)}. \qquad \blacksquare$$

Corollary 5 *The probability density function* (pdf) *of the interval length $f(t)$ is*

$$\begin{aligned} f(t) &= (1-p_1)\mu_2 e^{-\mu_2(t-t_1)} \cdot (U_{t_1}(t) - U_{t_2}(t)) + \\ &\quad p_1 \cdot \delta_{t_1}(t) + p_2 \cdot \delta_{t_2}(t), \end{aligned} \tag{7}$$

where $\delta_\tau(\cdot)$ and $U_\tau(\cdot)$ are the impulse and step functions at τ respectively.

Proposition 6 *The average length of an interval between checkpoints is*

$$\overline{t_i} = t_1 + \frac{e^{\mu_2(t_2-t_1)} - 1}{\mu_2} \cdot p_2. \tag{8}$$

Proof: Let $f(t)$ be the probability density function *(pdf)* of the interval length, then

$$\begin{aligned} \overline{t_i} &= \int_{t_1}^{t_2} t\, f(t)\, dt \\ &= p_1 \cdot t_1 + p_2 \cdot t_2 + (1-p_1) \int_{t_1}^{t_2} t \mu_2 e^{-\mu_2(t-t_1)}\, dt \\ &= t_1 + \frac{e^{\mu_2(t_2-t_1)} - 1}{\mu_2} \cdot p_2. \qquad \blacksquare \end{aligned}$$

Proposition 7 *The average execution time of an interval between checkpoints is*

$$\begin{aligned} \overline{T_i} &= \frac{e^{\lambda t_1} + \frac{\lambda p_2}{\lambda - \mu_2}\left(e^{\lambda t_2} - e^{\lambda t_1 + \mu_2(t_2-t_1)}\right) - 1}{\lambda} + \\ &\quad (1-p_2)c_1 + p_2 c_2. \end{aligned} \tag{9}$$

Proof: Let $T(t, c(t))$ be the average execution time of an interval of length t with checkpoint of cost $c(t)$ at the end of it. From Proposition 1 we know that

$$T(t, c(t)) = \frac{e^{\lambda t} - 1}{\lambda} + c(t).$$

and

$$\begin{aligned} \overline{T_i} &= \int_{t_1}^{t_2} T(t, c(t))\, f(t)\, dt \\ &= (1-p_1) \int_{t_1}^{t_2} T(t, c_1) \mu_2 e^{-\mu_2(t-t_1)}\, dt + \\ &\quad p_1 \cdot T(t_1, c_1) + p_2 \cdot T(t_2, c_2) \\ &= \frac{e^{\lambda t_1} + \frac{\lambda p_2}{\lambda - \mu_2}\left(e^{\lambda t_2} - e^{\lambda t_1 + \mu_2(t_2-t_1)}\right) - 1}{\lambda} + \\ &\quad (1-p_2)c_1 + p_2 c_2. \qquad \blacksquare \end{aligned}$$

Proposition 8 *The average overhead ratio of a program is*

$$R = \frac{\overline{T_i}}{\overline{t_i}} - 1. \tag{10}$$

Proof: To calculate the overhead ratio of a program it is not enough to calculate the average overhead of an interval. We need to consider also the length of the intervals, since longer intervals occupy more of the program, and thus they have bigger influence on the overhead ratio. Therefore, using similar arguments to those used when considering the current life of a random point in time in renewal theory [6], the average overhead ratio of a program is given by

$$\begin{aligned} R &= \frac{1}{\overline{t_i}} \int_{t_1}^{t_2} \frac{T(t, c(t)) - t}{t}\, t\, f(t)\, dt \\ &= \frac{\int_{t_1}^{t_2} (T(t, c(t)) - t)\, f(t)\, dt}{\overline{t_i}} \\ &= \frac{\overline{T_i}}{\overline{t_i}} - 1. \qquad \blacksquare \end{aligned}$$

Assigning the values of $\overline{t_i}$ from Eq. (8) and $\overline{T_i}$ from Eq. (9) into the expression of the overhead ratio of a program given in Eq. (10) yields the expression in Eq. (3) and completes the proof of Lemma 2. $\blacksquare$

Given λ, μ_1, μ_2, c_1 and c_2, we can numerically find the values of t_1 and t_2 that minimize the overhead ratio R. More on the selection of t_1 and t_2 can be found in Section 4.

3.2 On-line Algorithm with More Than Two State Sizes

The on-line algorithm we have described in this section is designed for the case when the program has two possible state sizes. The algorithm can be extended to the case when there are more than two state sizes in the following way.

We assume that the possible state sizes are $s_1, s_2, \ldots, s_n$, and that the cost of a checkpoint for a state size s_i is c_i, where $c_1 \leq c_2 \leq \cdots \leq c_n$. Each state size s_i has an interval t_i associated with it. The algorithm decides whether to place a checkpoint at time t where t is the time since the last checkpoint according to the following rules:

- A checkpoint is never placed at the interval $[0, t_1)$.
- If at some time in the interval $[t_i, t_{i+1})$ the state size is $s_1, s_2, \ldots, s_i$ a checkpoint is placed at that time.
- At time t_n a checkpoint is placed, regardless of the state size at that time.

The analysis of the algorithm is essentially the same as the analysis for the two state sizes case. We calculate the distribution function of the interval length and the distribution of the checkpointing cost at each point. Using these functions, we can calculate the average interval length, the average execution time of an interval, and the overhead ratio.

4 Comparison with Existing Algorithms

To illustrate how the current knowledge about the cost of checkpointing and the proposed on-line algorithm can be used in reducing the execution time of a program, we compare the overhead ratio of a program using the on-line algorithm to the overhead ratio when the two strategies described in Section 2 are used, namely the fixed intervals strategy and the optimal placement. The comparison to the fixed interval placement helps to understand how the current knowledge about the cost of checkpointing helps to reduce the average execution time of a program. It also provides insight to the optimal values of t_1 and t_2 that minimize the overhead ratio. The comparison to the optimal algorithm shows how much the performance of the on-line algorithm can be improved when the cost of checkpoints in all possible locations is known in advance and how the on-line and optimal algorithms differ in the placement of checkpoints.

The comparison of the new on-line algorithm with the fixed intervals placement strategy is done by comparing the overhead ratio of the on-line algorithm, given in Lemma 2, with the overhead ratio of the fixed intervals placement strategy, given in Eq. (2). The values of t_1 and t_2 for the on-line algorithm and the interval length t for the fixed intervals placement strategy are those that minimize the overhead ratio.

Since we cannot analytically find the overhead ratio of the optimal algorithm, We used experimental results to compare the on-line placement algorithm with the optimal placement. We generated a large number of instances of the program's state size according to the two states Markov chain. For each such instance, we found the placement of the checkpoints when the optimal algorithm and the on-line algorithm are used. After the checkpoints were placed, we calculated the overhead ratio of the instance when both algorithms are used. Finally, we calculated the average overhead ratio over all instances that used the same parameters (λ, μ_1, μ_2). The experimental values of the overhead ratio for the on-line algorithm are identical to the analytical values obtained using Lemma 2.

Checkpointing with Fixed Intervals

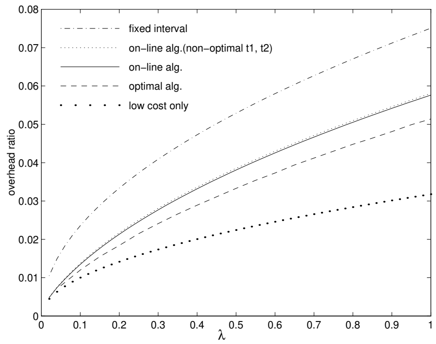
In Figure 1 the overhead ratio of a program as a function of the fault rate λ is shown. The figure shows the execution time when the checkpointing costs are $c_1 = 0.0005$ and $c_2 = 0.005$. The figure shows the execution time for two cases of μ_1 and μ_2. In Figure 1a $\mu_1 = \mu_2 = 10$, and in Figure 1b $\mu_1 = \mu_2 = 1$. The figure compares the execution time of a program when fixed equi-distant intervals are used to the execution time when the on-line algorithm for placement of checkpoints is used. As a reference, the figure also shows the overhead ratio if the cost of the checkpoints is only c_1. It can be seen in the figure that the on-line algorithm has a lower overhead ratio.

To understand why the on-line algorithm has a lower overhead ratio than the fixed interval placement, lets consider two extreme cases; the first is when the rate of changes in the state size is very low, and the second when the rate of changes is very high.

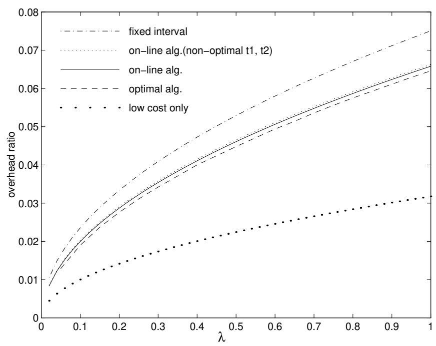
When the rate of changes in the state size is very low, the probability of a change in the state size between t_1 and t_2 is practically 0, and checkpoints are placed only at t_1 and t_2. In this case, by using the optimal checkpointing intervals when the cost of a checkpoint is only c_1 or c_2 as t_1 and t_2 respectively, the on-line algorithm adapts to the current checkpoint cost, and uses the optimal interval for that cost. Therefore, for low rate of changes in the state size, the optimal values of t_1 and t_2 are $t_{1,\text{opt}} = \tilde{t}_1$ and $t_{2,\text{opt}} = \tilde{t}_2$, where $\tilde{t}_1$ and $\tilde{t}_2$ are the optimal checkpointing intervals when the the cost of checkpoints are the constants c_1 and c_2 respectively.

When the rate of changes in the state size is high, the on-line algorithm uses this fact to locate a point with a low cost near $\tilde{t}_1$ and place a checkpoint at that point. The result is that the cost of a checkpoint is always the low cost, and the interval between the checkpoints is close to the optimal interval for that cost. In this case its is always better to wait for a point with a low cost, and the optimal value for t_2 is very high.

In the medium range, when $\frac{1}{\mu_2}$ has the same order of magnitude as $\tilde{t}_1$, the on-line algorithm can take advantage of the points with low checkpointing cost that are near $\tilde{t}_1$.



(a) $\mu_1 = \mu_2 = 10$



(b) $\mu_1 = \mu_2 = 1$

Figure 1. Overhead ratio as a function of λ

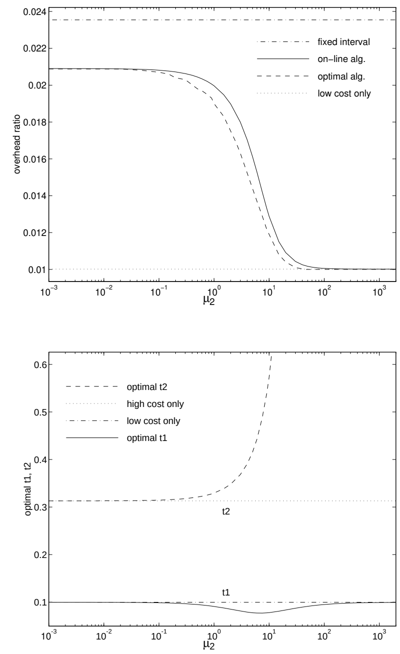


Figure 2. Overhead ratio as a function of μ_2

To be sure that such points are not missed, the algorithm starts to look for it before the optimal interval $\tilde{t}_1$. Therefore, for this range of μ_2, $t_{1,\text{opt}} < \tilde{t}_1$. In this range there is a good chance that the state size is going to change from high to low when $\tilde{t}_2$ is reached and that this change will occur fast enough so that it is better to wait for that change, and therefore in that range $t_{2,\text{opt}} > \tilde{t}_2$.

Figure 2 shows the overhead ratio and the optimal t_1 and t_2 as a function of μ_2. The figure shows this values when $\lambda = 0.1$, and the possible costs of a checkpoint are $c_1 = 0.0005$ and $c_2 = 0.005$ and $\mu_1 = \mu_2$. The figure shows that for low μ_2 the overhead ratio is somewhat lower than the execution time when fixed intervals are used. As μ_2 increases, the overhead ratio of the on-line algorithm drops, and for $\mu_2 > 100$ the overhead ratio is as if the cost of checkpointing was c_1 everywhere. The plot of the optimal t_1 and t_2 shows that for low μ_2 the optimal values equals to $\tilde{t}_1$ and $\tilde{t}_2$. When μ_2 increases, the optimal value of t_1 decreases so that a point of low cost near $\tilde{t}_1$ is not missed, while the optimal value for t_2 increases to enable the algorithm to catch points with low cost at that area. Further increasing μ_2 causes the optimal t_1 to increase and be closer to $\tilde{t}_1$ because for these values of μ_2 the chance of finding a point with low cost is getting higher. The behavior of the overhead ratio and the optimal t_1 and t_2 for different ratios of μ_2/μ_1 are identical to the behavior shown in Figure 2.

Optimal Placement Algorithm

Figure 1 shows the overhead ratio of a program as a function of the fault rate λ for the on-line and optimal algorithms. The figure shows that the optimal algorithm performs better than the on-line algorithm, but the difference between the algorithm is not large, and the on-line algorithm is closer to the optimal algorithm than the fixed intervals strategy.

In Figure 2a the overhead ratio of the program as a function of the rate of changes in the state size is show for both algorithms. The figure shows that both algorithms are affected in the same way by μ_2. When μ_2 is low, both algorithms adapt to the current state size and use the optimal interval for that state size. When μ_2 is high, both algorithms can find points with small state size close to the optimal interval for that state size and place checkpoints there. Therefore, the overhead ratio is the same as if only the low state size exists. In the medium range for μ_2, the optimal algorithm can use its knowledge about the cost of future possible checkpoints to achieve lower overhead ratio.

To understand the difference and similarities in checkpointing placement between the two algorithms, we examined few of the instances of the random state sizes we generated, and looked where each of the algorithms placed

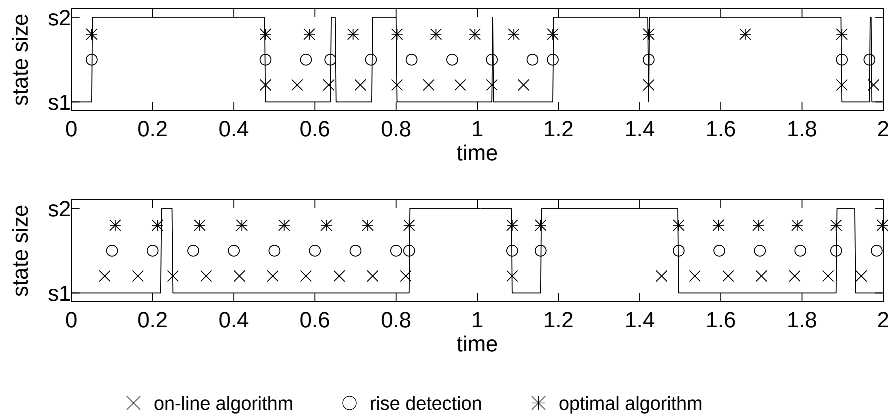


Figure 3. Placement of checkpoints by the optimal off-line algorithm and the on-line algorithm with and without rise detection

its checkpoints. Figure 3 shows two such instances. In both cases the fault rate is $\lambda = 0.1$ and the checkpointing costs are $c_1 = 0.0005$ and $c_2 = 0.005$. In the top plot $\mu_1 = \mu_2 = 10$, and in the bottom plot $\mu_1 = \mu_2 = 3$. The plots show the state size of the program as a function of the time t, and the points where each of the algorithms places the checkpoints. The figure also shows the checkpointing placement of a modified version of the on-line algorithm that is described later in the paper, in Section 5.

The top plot shows that both algorithms avoid placing checkpoints when the cost is high, even when there are long intervals of high cost. The difference in the algorithms in this plot is the interval between the checkpoints. The optimal algorithm knows exactly the intervals of low and high cost so it can use them to place the checkpoints with the optimal interval between them. On the other hand, the on-line algorithm does not know when the cost is going to change from low to high, and so it prefers to use intervals which are shorter than the optimal interval when the cost is low, instead of losing the possibility to place a checkpoint with a low cost.

The top plot also shows an example where the optimal algorithm places a checkpoint at a point with a high cost, while the on-line algorithm avoids it. In this example the on-line algorithm anticipates a fast change in the state size, and therefore it decides to wait for the small state size and place the checkpoint there. On the other hand, the optimal algorithm knows that the interval is going to be long, and therefore it is better to place a checkpoint in it.

The second plot gives an example where the on-line algorithm places a checkpoint with high cost while the optimal algorithm avoids the high cost interval. The optimal algorithm knows the length of the high cost interval, and that it is better not to place a checkpoint in it. On the other hand, the on-line algorithm anticipates that the interval is going to be much longer (because of the value of μ_2), and therefore it concludes that it is better to place a checkpoint in it.

5 Detection of Increase in the State Size

So far we have assumed that the program does not have any knowledge about future changes in its state size. While this assumption is generally true, there are some cases when a partial knowledge about the future behavior exists. This partial knowledge can be used to improve the placement strategy. The simplest example about future knowledge is knowledge about changes in the state size just before they occur. When the memory allocation or deallocation functions are called, the program knows that state size is going to change before the change actually occur.

Detection of changes in the state size before they occur is important when the state size increases. In this case, it might be beneficial to place a checkpoint with lower cost just before the state size increases. The ability to place a checkpoint just before the state size increases can contribute to the performance of the placement strategy in two ways. When the algorithm can place a checkpoint before the state size increases, it does not have to be 'over-eager' when looking for points with low cost (the drop down in the value of $t_{1,\mathrm{opt}}$ in Figure 2). Instead, it can wait until $\tilde{t}_1$ is reached, or the state size is about to change, and place the checkpoint at that time. Also, when a checkpoint is placed before the state size increases, the probability of placing a checkpoint with a large state size gets lower, and thus the checkpointing overhead is smaller.

In this section we show how to modify the on-line algorithm we presented in Section 3 to include the case of detection of an increase in the state size before they occur. We also compare the modified algorithm performance to the original on-line algorithm and optimal off-line placement.

5.1 The Modified Algorithm

In the modified algorithm we add another point in time t_0, such that $t_0 \leq t_1$. A checkpoint is placed at time t, $t_0 \leq t < t_1$, if the state size at t is s_1 and the state size at t^+ is s_2. In other words, if the state size is changing from s_1 to s_2 during the interval $[t_0, t_1)$, then a checkpoint is placed just before the change. If a checkpoint is not placed in the interval $[t_0, t_1)$, then the algorithm continues as the algorithm in Section 3.

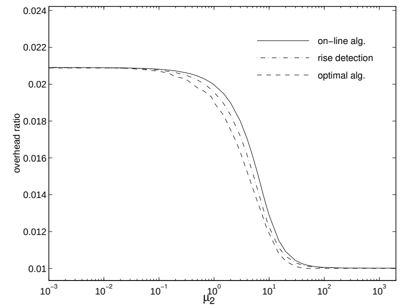
The analysis of the modified algorithm is essentially the same as the analysis of the original on-line algorithm that was shown in Lemma 2. Due to space limitation, we omit the analysis of the modified algorithm. The analysis can be found in [11].

Figure 4 shows the overhead ratio and the optimal t_0 and t_1 as a function of μ_2. The figure shows this values when $\lambda = 0.1$, the possible costs of a checkpoint are $c_1 = 0.0005$ and $c_2 = 0.005$, and $\mu_1 = \mu_2$.

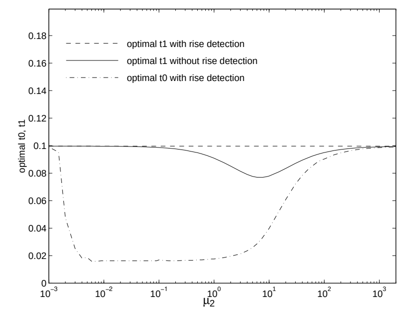
Figure 4a shows the overhead ratio as a function of μ_2 for the modified algorithm, the original on-line algorithm and the optimal off-line algorithm. The figure shows that the modified algorithm has a lower overhead ratio then the original on-line algorithm, and its behavior is closer to the optimal algorithm.

One of the reasons the modified algorithm performs better than the original on-line algorithm, is that it does not have to be 'over eager' when looking for points with a small state size. The original algorithm does not know when the state size is going to increase, therefore, in order not to lose the small state size, it places checkpoints before the optimal interval for the small state size $\tilde{t}_1$ is reached. On the other hand, the modified algorithm can wait until just before the state size changes or $\tilde{t}_1$ is reached before it places a checkpoint, because it knows about the change in the state size before it occurs. Also, when the algorithm knows that the state size is going to increase, it is sometimes better to place a checkpoint after a short interval, specially when μ_2 is low. Therefore, the optimal value for t_0 for the modified algorithm is lower then the optimal value for t_1 in the original algorithm.

Figure 4b shows the optimal values of t_0 and t_1 for the modified algorithm as a function of μ_2, and for comparison the optimal value of t_1 for the original algorithm. The figure confirms that the optimal value of t_1 for the modified algorithm is equal to $\tilde{t}_1$, and the drop in the value of $t_{1,\mathrm{opt}}$ that occur in the original algorithm to avoid losing points with a small state size is not needed in the modified algorithm.



(a) Overhead ratio



(b) Optimal Intervals

Figure 4. Overhead ratio and optimal t_0 and t_1 for the modified algorithm

The figure also shows that for low values of μ_2, when the average time before the state size changes from s_2 to s_1 is high, it is beneficial to place checkpoints with a very short interval between them to use the small state size. As the value of μ_2 gets higher, the value of t_0 is also getting higher, until it reaches t_1 for very high values of μ_2.

The placement examples that are shown in Figure 3 also help to illustrate the advantages of the modified algorithm over the original algorithm. The plots in the figure show two instances of changes in the state size, and the points where the on-line algorithm, the modified on-line algorithm and the optimal algorithm placed their checkpoints. The figure shows that during long periods of small state size, the modified algorithm places its checkpoints with the same intervals as the optimal algorithm, while the original algorithm uses smaller intervals. Another advantage that the modified algorithm has on the original algorithm is that it can sometime avoid checkpoints with large state size, as can be seen in the bottom plot of Figure 3. Because the modified algorithm places checkpoints just before the state size increases, the probability that the state size will not change to s_1 before t_2 is smaller than the same probability in the original algorithm that places the checkpoint some time before the state size increases.

6 Implementation Issues

The on-line algorithm can be easily implemented in systems where the system hardware is used to determine the place of checkpoints. For example, if an interrupt by a timer is used to determine the time of the next checkpoint, this timer can be updated every time the state size changes. After a checkpoint is placed, the timer is initialized to t_i, where s_i is the state size when the checkpoint was placed, and the timer starts counting downward. After each memory allocation or deallocation operation that causes a change in the state size of the program from s_i to s_j, the timer value is increased by $t_j - t_i$. A checkpoint is placed when the timer value is less than or equal to 0.

The algorithm presented in this paper assumes that the program has only a finite set of state sizes (two in the analyzed example) and that the state size of the program is changing according to a Markov process with known parameters. In practice, both assumptions are not valid. The state size of a program is a continuous random process whose parameters are hard to estimate. To overcome the continuous state size problem, we can quantize it, for example to the nearest K-byte. If the quantization error is not big, the affects of the quantization on the performance of the algorithm are minimal. The parameters of random process that controls the state size of program are used to calculate the optimal values for the t_i's. Without knowledge about these parameters, the optimal values have to be estimated. A good estimation for $t_{i,\mathrm{opt}}$ are optimal intervals when the cost of checkpointing is a constant $\tilde{t}_i$. The dotted line in Figure 1 shows the overhead ratio when $\tilde{t}_1$ and $\tilde{t}_2$ are used instead of $t_{1,\mathrm{opt}}$ and $t_{2,\mathrm{opt}}$. As can be seen in the figure, the overhead ratio is almost identical (about 5% difference). Since $\tilde{t}_i$ are independent of the parameters of the Markov process, they can be used even if these parameters are not known.

7 Conclusions

In this paper we showed that knowledge about the current state size of the program can be used in placement of checkpoints in a program, and that using this knowledge can lead to a significant reduction in the overhead ratio. To illustrate how this knowledge can be used, we presented a new on-line algorithm for placement of checkpoints. The algorithm first tries to place a checkpoint in places where the cost of the checkpoint is small. Only if no such point was found, a checkpoint is placed at a point with higher checkpointing cost.

We analyzed the overhead ratio of a program using this algorithm, and compared the performance of the proposed algorithm to a simple algorithm that places the checkpoints at fixed intervals, and to the optimal algorithm that uses a perfect a-priori knowledge on the cost of checkpoints at all possible locations. The comparison results show that the proposed algorithm performs better than the fixed intervals algorithm, and a significant reduction of up to 66% in the overhead ratio can be obtained. Although the proposed algorithm uses only the cost of a checkpoint at the current location, its behavior is close to the optimal algorithm that uses an a-priory knowledge of the checkpointing cost in all possible locations.

The same on-line placement strategy can be combined with other placement algorithms and improve their performance when the fault rate in the system is not a constant or when the changes in the state size do not occur according to a Markov process.

An interesting problem is to combine the on-line algorithm with some partial knowledge about the state size of the program in the future, like the information collected by the CATCH tool [8]. This additional knowledge about the state size can be used to improve the decision about the placement of checkpoints, and bring the algorithm closer to the optimal algorithm.

References

[1] A. Brock. An analysis of checkpointing. *ICL Technical Journal*, 1, 1979.

[2] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21:546–556, June 1972.

[3] E. G. Coffman and E. N. Gilbert. Optimal strategies for scheduling checkpoints and preventive maintenance. *IEEE Transactions on Reliability*, 39:9–18, April 1990.

[4] A. Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, June 1983.

[5] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26:259–270, April 1979.

[6] S. Karlin and H. M. Taylor. *A First Course in Stochastic Processes*. Academic Press, 1975.

[7] P. L'Ecuyer and J. Malenfant. Computing optimal checkpointing for rollback and recovery systems. *IEEE Transactions on Computers*, 37:491–496, April 1988.

[8] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-assisted full checkpointing. *Software — Practice and Experience*, 24:871–886, October 1994.

[9] V. F. Nicola and J. M. van Spanje. Comparative analysis of different models of checkpointing and recovery. *IEEE Transactions on Software Engineering*, 16:807–821, August 1990.

[10] S. Toueg and O. Babaoğlu. On the optimum checkpoint selection problem. *SIAM Journal on Computing*, 13:630–649, August 1984.

[11] A. Ziv. *Analysis and Performance Optimization of Checkpointing Schemes with Task Duplication*. PhD thesis, Stanford University, 1995.