On Checkpoint Latency*

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: vaidya@cs.tamu.edu
Web: http://www.cs.tamu.edu/faculty/vaidya/

Abstract

Checkpointing and rollback is a technique for minimizing loss of computation in presence of failures. Two metrics can be used to characterize a checkpointing scheme: (i) checkpoint overhead *(increase in the execution time of the application because of a checkpoint), and (ii) checkpoint* latency *(duration of time required to save the checkpoint). For many checkpointing methods, checkpoint latency is larger than checkpoint overhead. This paper evaluates the expression for "average overhead" of the checkpointing scheme as a function of checkpoint latency and overhead. It is shown that the "average overhead" is much more sensitive to the changes in checkpoint overhead, as compared to checkpoint latency. Also, for equi-distant checkpoints, the optimal checkpoint interval is shown to be independent of the checkpoint latency.*

1 Introduction

Many applications (sequential and parallel) require large amount of time to complete. Such applications can encounter loss of a significant amount of computation if a failure occurs during the execution. *Checkpointing and rollback recovery* is a technique used to minimize the loss of computation in an environment subject to failures [1]. A *checkpoint* is a copy of the application's state stored on a *stable* storage – a *stable* storage is not subject to failures. The application periodically saves checkpoints; the application recovers from a failure by *rolling back* to a recent checkpoint. Checkpointing can be used for sequential as well as parallel (or distributed) applications. When the application consists of more than one process, a *consistent* checkpointing algorithm can be used to save a consistent state of the multi-process application.

Two metrics can be used to characterize a checkpointing scheme:

- **Checkpoint overhead** C is the increase in the execution time of the application because of a checkpoint.
- **Checkpoint latency** L is the duration of time required to save the checkpoint. In many implementations, checkpoint *latency* is larger than the checkpoint *overhead.* (Illustrated in Section 2.)

In the past, a large number of researchers have analyzed the checkpointing and rollback recovery scheme (e.g. [1, 2, 3, 11]). However, to our knowledge, the past work has not taken checkpoint *latency* into account. This paper evaluates the impact of checkpoint latency on the performance of a checkpointing scheme. This work is motivated by the schemes that attempt to reduce checkpoint overhead while causing an increase in the checkpoint latency (e.g., [5, 4, 8]).

Related work: Plank et al. [5, 4] present measurements of checkpoint latency and overhead for a few applications, however, they do not present any performance analysis. We measured checkpoint latency and overhead for a few uni-process applications, and briefly analyzed the impact of checkpoint latency on performance of "two-level" recovery schemes [7, 9].

2 Checkpoint Latency

We limit the discussion to uni-process applications. Due to lack of space, multi-process applications are not discussed here [10]. In this section, we illustrate the distinction between checkpoint *latency* and checkpoint *overhead* with two examples.

Sequential checkpointing is an approach for which checkpoint *overhead* is identical to checkpoint *latency.* In this approach, when an application process wants to take a checkpoint, it pauses and saves its state on the stable storage [5]. Therefore, the time required to save the checkpoint (i.e., checkpoint latency) is practically *identical* to the increase in the execution time of the process (i.e., checkpoint overhead). Figure 1 illustrates this approach. The horizontal line represents processor execution, time increasing from left to right. The shaded box represents the checkpointing operation. The sequential checkpointing approach achieves the *smallest* possible checkpoint *latency.* However, it results in a larger checkpoint *overhead* as compared to other approaches.

Forked checkpointing is an approach for which checkpoint *overhead* is usually much smaller than the checkpoint *latency.* In this approach, when a process wants to take a checkpoint, it *forks* a child process

*This work is supported in part by the National Science Foundation under CAREER Grant MIP-9502563.

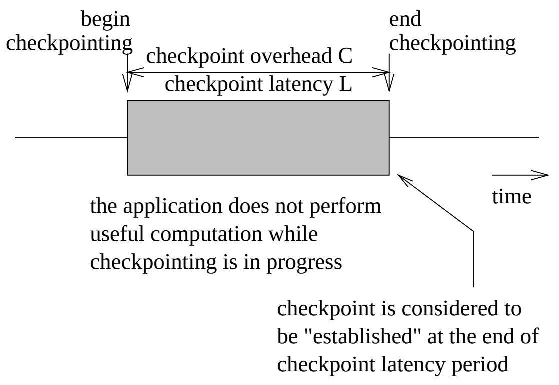
begin checkpointing
end checkpointing
checkpoint overhead C
checkpoint latency L
the application does not perform useful computation while checkpointing is in progress
time
checkpoint is considered to be "established" at the end of checkpoint latency period

Figure 1: Sequential Checkpointing

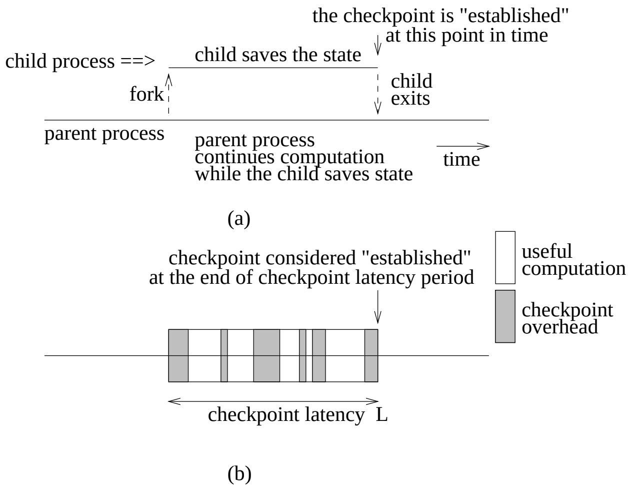
the checkpoint is "established" at this point in time
child process ==>
child saves the state
child exits
fork
parent process
parent process continues computation while the child saves state
time
(a)
checkpoint considered "established" at the end of checkpoint latency period
useful computation
checkpoint overhead
checkpoint latency L
(b)

Figure 2: Forked Checkpointing

[5]. The state of the child process is identical to that of the parent process when `fork` is performed. After the `fork`, the parent process continues computation, while the child process saves its state on the stable storage. Figure 2(a) illustrates this approach. In this approach, computation is overlapped with stable storage access (i.e., overlapped with state saving), therefore the checkpoint *overhead* is usually smaller than the *sequential* checkpointing approach. Also, as the parent and the child execute in parallel, checkpoint *latency* is larger than the checkpoint *overhead.* Figure 2(b) illustrates the interleaved execution of the child and parent processes on the same processor. As shown in the figure, *useful* computation performed by the parent process is interleaved with the checkpointing operation performed by the child process.

For future reference, note that, a checkpoint is said to have been *established* if a future failure can be tolerated by a rollback to this checkpoint. Thus, a checkpoint is not considered to be *established* until the end of the latency period. When the execution progresses past the end of the checkpoint latency period, the checkpoint is considered to have been established (refer Figures 1 and 2).

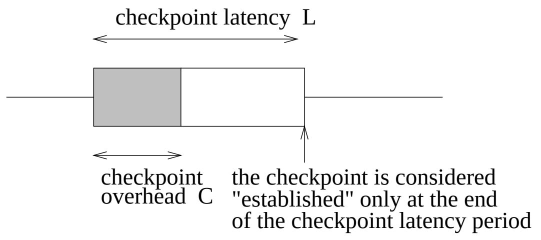
checkpoint latency L
checkpoint overhead C
the checkpoint is considered "established" only at the end of the checkpoint latency period

Figure 3: Modeling checkpoint latency and overhead

A checkpoint interval is defined as the duration between the *establishment* of two consecutive checkpoints. That is, a *checkpoint interval* begins when one checkpoint is established, and the interval ends when the next checkpoint is established. (For brevity, the term *interval* will be used to denote a *checkpoint interval.*) We assume that the checkpoints are equi-distant, i.e., the amount of *useful* computation performed during each interval is identical (denoted by T).

3 Latency and Overhead

As discussed in the previous section, the checkpoint latency period is divided into two types of execution: (1) useful computation, and (2) execution necessary for checkpointing. The two types are usually interleaved in time. However, for modeling purposes, we can assume that the two types of executions are separated in time, as shown in Figure 3. As shown in the figure, the first C units of time during the checkpoint latency period is assumed to be used for saving the checkpoint. The remaining $(L-C)$ units of time is assumed to be spent on useful computation. Although the C units of overhead is modeled as being incurred at the beginning of the checkpoint latency period, the checkpoint is considered to have been *established* only at the end of the checkpoint latency period. Although our representation of checkpoint latency and overhead is simplified, we now demonstrate that it will lead to accurate analysis. Two distinct situations may occur when a checkpoint interval is executed.

Situation 1: A failure does not occur while the interval is executed. In this case, the execution time from the beginning to the end of an interval is $T+C$. Of the $T+C$ units, T units are spent doing *useful* computation, while incurring an overhead of C time units. As shown in Figure 4(a), $(L-C)$ units of useful computation is performed during the checkpoint latency period. Now consider Figure 4(b). Similar to Figure 4(a), $L-C$ units of useful computation is performed during the latency period. Also, the execution time for the interval is $T+C$.

Situation 2: A failure occurs during the interval. When a failure occurs, the task must be rolled back to the previous checkpoint, incurring an overhead of R time units. In Figure 5(a), the task is rolled back to checkpoint CP1. After the rollback, $L-C$ units of useful computation performed during the latency period of CP1 must be performed again – this is necessary, because the state saved during checkpoint

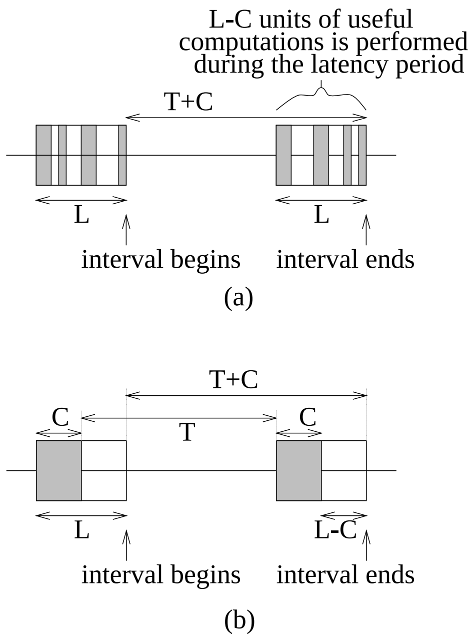

Figure 4: Situation 1

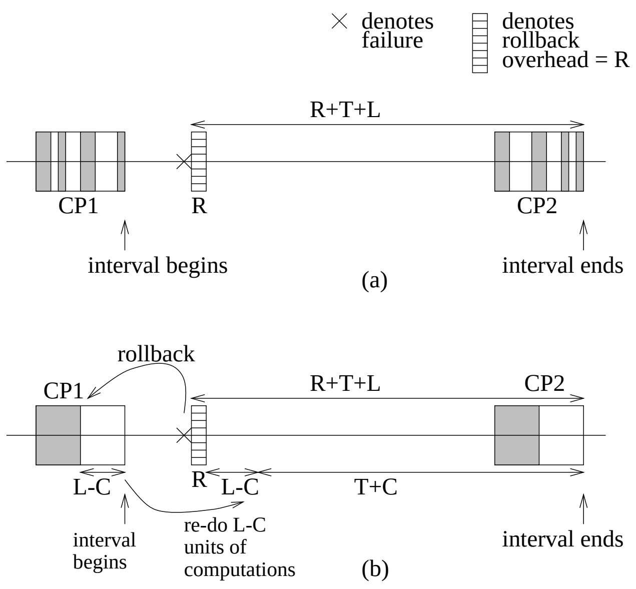

Figure 5: Situation 2

CP1 is the state at the *beginning* of the latency period for CP1. In the absence of a further failure, additional $T + C$ units of execution is required before the completion of the interval. Thus, after a failure, $R + (L - C) + (T + C) = R + T + L$ units of execution is required before the completion of the interval, *provided* additional failures do not occur.

Now consider Figure 5(b). When the failure occurs, as shown in Figure 5(b), the system can be considered to have rolled back to the end of the "shaded portion" in the latency period for checkpoint CP1. (Note that no state change occurs during the "shaded portion".) Now it is apparent that, in the absence of further failure, $R + T + L$ units of execution is required to complete the interval. Thus, our representation of checkpoint latency and overhead yields the same conclusion as the more accurate representation in Figure 5(a).

The above discussion is also applicable if the failure occurs during the checkpoint latency period of checkpoint CP2. Such a failure will also require a rollback to checkpoint CP1, as checkpoint CP2 is *not* established when the failure occurred.[1] The above discussion can also be extended to multiple failures during a checkpoint interval.

The above two cases imply that the simplified representation of checkpoint latency and overhead will yield the same results as the accurate representation.

4 Evaluating the Overhead

We assume that C, L and R are constants for a given scheme. The fault model assumed for the analysis is as follows: Processor failures are governed by a Poisson process with rate λ. When a processor fails, its local state is corrupted. A processor can fail during normal operation, during checkpointing, as well as during rollback and recovery. The *stable* storage is not subject to failures. (The stable storage is used for storing checkpoints.)

Let $G(t)$ denote the expected (average) amount of execution time required to perform t units of *useful* computation. (*Useful* computation excludes the time spent on checkpointing and rollback recovery.) Then, we define *overhead ratio* (r) as:

$$\text{overhead ratio } r = \lim_{t\to\infty} \frac{G(t) - t}{t} = \lim_{t\to\infty} \frac{G(t)}{t} - 1.$$

We assume that the system is executing an infinite task that takes *equi-distant* checkpoints. The execution of the task can be considered to be a series of *intervals*, each interval *beginning* immediately after a checkpoint is established, and *ending* when the next checkpoint is established. Figure 6 illustrates intervals I1, I2 and I3. (Meaning of various *states* in the figure will be explained shortly.) As shown in the figure,

[1] Chandy et al. [1] present an analysis of checkpointing schemes that does not take checkpoint latency into account. However, for sequential checkpointing (with $L = C$), our analysis is similar to theirs with one exception. An assumption made by Chandy et al. [1] implies that a failure that occurs while checkpoint CP2 (in Figure 5) is being saved, only requires re-initiation of the checkpointing operation. As per their assumption, computation during the interval preceding checkpoint CP2 need not be performed again even if a failure occurs while checkpoint CP2 is being established (i.e. the failure occurs after checkpointing is initiated but before it is completed). For many environments this assumption is not realistic. Therefore, we make the realistic assumption that a failure that occurs while a checkpoint (say, CP2 in Figure 5) is being established requires a rollback to the previous checkpoint (checkpoint CP1).

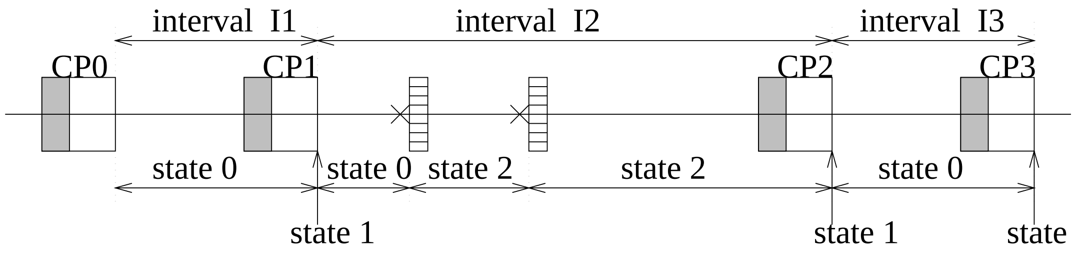

Figure 6: Intervals

interval I1 begins after checkpoint CP0 is established, and ends when checkpoint CP1 is established. Interval I2 begins after checkpoint CP1 is established. Before checkpoint CP2 is established, two failures occur, each requiring a rollback to checkpoint CP1. Subsequently, the computation progresses without failure and checkpoint CP2 is established. Interval I2 is completed when checkpoint CP2 is established.

Observe that T units of *useful* computation is performed during each checkpoint interval. Provided no failures occur during the interval, the total time required to execute an interval is $T + C$. If one or more failure occurs while executing an interval, then the execution time is longer than $T + C$. Let Γ denote the expected (average) execution time of an interval. Then, it is easy to see that,

$$\text{overhead ratio} \quad r = \lim_{t\to\infty} \frac{G(t)}{t} - 1 = \frac{\Gamma}{T} - 1$$

Expected execution time Γ of a single checkpoint interval can be evaluated using the Markov chain [6, 12] in Figure 7. State 0 is the initial state, when an interval starts execution. A transition from state 0 to state 1 occurs when the interval is completed without a failure. If a failure occurs while executing the checkpoint interval, then a transition is made from state 0 to state 2. After state 2 is entered, a transition occurs to state 1 if no further failure occurs before the next checkpoint is established. If, however, another failure occurs after entering state 2 and before the next checkpoint is established, then a transition is made from state 2 back to state 2. When state 1 is entered, the interval has completed execution. Therefore, state 1 is a sink state – there are no transitions out of state 1. Figure 6 illustrates the various states for an example execution. As shown in Figure 6, during interval I1, the task is in state 0, and enters state 1 when the interval completes without a failure. During interval I2, the task is initially in state 0, and enters state 2 when a failure occurs. When another failure occurs, the task re-enters state 2. Subsequent to the second failure, interval I2 completes without any further failures. State 1 is entered at the end of the interval.

Each transition (X, Y), from state X to state Y in the Markov chain, has an associated *transition probability* P_{XY} and a *cost* K_{XY}. *Cost* K_{XY} of a transition (X, Y) is the expected (average) time spent in state X before making the transition to state Y.

It is easy to see that, $P_{01} = e^{-\lambda(T+C)}$ and $K_{01} =$

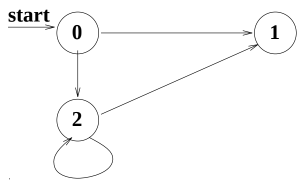

Figure 7: Markov chain

$T + C$. Also, $P_{02} = 1 - P_{01} = 1 - e^{-\lambda(T+C)}$. Now, the cost K_{02} of transition (0,2) is the expected duration, from the beginning of the interval until the time when the failure occurred, given that a failure occurs before the end of the interval. Therefore,

$$\begin{aligned} K_{02} &= \int_0^{T+C} (t)\, \frac{\lambda\, e^{-\lambda t}}{1 - e^{-\lambda(T+C)}} dt \\ &= \lambda^{-1} - \frac{(T+C)e^{-\lambda(T+C)}}{1 - e^{-\lambda(T+C)}} \end{aligned}$$

After state 2 is entered, a transition is made to state 1 if no further failure occurs before the next checkpoint is established. As discussed earlier, the execution time required for the next checkpoint to be established after a failure is $R + T + L$. Therefore, $P_{21} = e^{-\lambda(R+T+L)}$ and $K_{21} = R + T + L$. If, however, another failure occurs after entering state 2 and before the next checkpoint is established, then a transition is made from state 2 back to state 2. It follows that, $P_{22} = 1 - P_{21} = 1 - e^{-\lambda(R+T+L)}$, and

$$\begin{aligned} K_{22} &= \int_0^{R+T+L} (t)\, \frac{\lambda\, e^{-\lambda t}}{1 - e^{-\lambda(R+T+L)}} dt \\ &= \lambda^{-1} - \frac{(R+T+L)e^{-\lambda(R+T+L)}}{1 - e^{-\lambda(R+T+L)}} \end{aligned}$$

The expected execution time Γ is the expected *cost* of a path from state 0 to state 1. It follows that,

$$\Gamma = P_{01}K_{01} + P_{02}\left(K_{02} + \frac{P_{22}}{1 - P_{22}}K_{22} + K_{21}\right)$$

Substituting the expressions for various costs and transition probabilities, and simplifying, the following expression is obtained [10].

$$\Gamma = \lambda^{-1} e^{\lambda(L-C+R)}(e^{\lambda(T+C)} - 1) \qquad (1)$$

It follows that, the overhead ratio r is given by

$$r = \frac{\Gamma}{T} - 1 = \frac{\lambda^{-1} e^{\lambda(L-C+R)}(e^{\lambda(T+C)} - 1)}{T} - 1 \quad (2)$$

4.1 Minimizing the Overhead Ratio

Consider a checkpointing scheme that achieves a certain overhead C and checkpoint latency L. For this checkpointing scheme, the objective now is to choose an appropriate value of T so as to minimize the overhead ratio r. The optimal value of T must satisfy the following equation:

$$\frac{\partial r}{\partial T} = \frac{\partial}{\partial T}\left[\frac{\lambda^{-1} e^{\lambda(L-C+R)}(e^{\lambda(T+C)} - 1)}{T} - 1\right] = 0$$

$$\Longrightarrow \quad e^{\lambda(T+C)}(1 - \lambda T) = 1 \text{ for } T \neq 0 \quad (3)$$

As the above equation does not include L or R, the optimal value of T is *not* dependent on L and R – the optimal T, however, depends on C. Thus, to evaluate the optimal checkpoint interval for a given checkpointing scheme, it is adequate to know the value of C. However, to evaluate the overhead ratio with the optimal T, L and R must also be known.

5 Inter-Dependence Between L and C

Checkpoint latency and checkpoint overhead are dependent on each other. An attempt to reduce the checkpoint overhead typically causes an increase in the checkpoint latency. Now, from Equation 2,

$$\frac{\partial r}{\partial L} = \frac{e^{\lambda(L-C+R)}(e^{\lambda(T+C)} - 1)}{T} > 0 \quad (4)$$

$$\frac{\partial r}{\partial C} = \frac{e^{\lambda(L-C+R)}}{T} > 0 \quad (5)$$

From Equations 4 and 5, observe that

$$\frac{\partial r}{\partial L} = (e^{\lambda(T+C)} - 1)\, \frac{\partial r}{\partial C}.$$

Also observe that $(e^{\lambda(T+C)} - 1)$ is likely to be much smaller than 1 for realistic values of λ, T and C. Thus, $\frac{\partial r}{\partial L}$ will typically be much smaller than $\frac{\partial r}{\partial C}$ – this implies that r is more sensitive to the changes in C, as compared to changes in L. This is intuitive: Observe that L does not affect the failure-free execution time, while C does. L only affects the execution time when a failure occurs. As failures do not occur too frequently, r is less sensitive to L, as compared to C.

In practice, if a checkpointing scheme increases L and also results in an increase in C, then one will not use that checkpointing scheme. Therefore, in practice, an increase in L is accompanied by a decrease in C.

For sequential checkpointing, checkpoint overhead and latency are identical, say C_{max}. A recovery scheme that attempts to achieve a smaller checkpoint overhead (C) than sequential checkpointing will achieve a latency (L) larger than sequential checkpointing. One would not use such a scheme, unless it resulted in a lower overhead ratio r as compared to sequential checkpointing. Equations 4 and 5 imply that a recovery scheme that achieves a smaller checkpoint overhead and larger latency, as compared to sequential checkpointing, can achieve a smaller overhead ratio than sequential checkpointing, *provided that* the latency is not "too much" larger than sequential checkpointing.

It is our objective here to determine when the latency is not "too much" larger than sequential checkpointing. More precisely, the objective is to determine a function g of C such that, for any $C < C_{max}$, the overhead ratio r is smaller than the sequential checkpointing scheme if $L < g(C)$. Derivation of function $g(C)$ is omitted here [10]. It can be shown that,

$$g(C) = C + \lambda^{-1} \ln \frac{1 - \lambda T_c}{1 - \lambda T_m}$$

where T_c is the value of T that satisfies Equation 3, and T_m is the solution of Equation 3 with $C = C_{max}$.[2]. Clearly, as one would expect, $g(C_{max}) = C_{max}$. (Note that, when $C = C_{max}$, $T_c = T_m$.)

The $g(C)$ expression derived above can be used to determine when a checkpointing scheme will perform better than the sequential checkpointing scheme. Figure 8 plots $g(C)$ for $\lambda = 10^{-6}$ and 10^{-4}. ($\lambda = 10^{-6}$ for curves (1)-(4) and $\lambda = 10^{-4}$ for curves (5)-(8).) Consider the $g(C)$ curve for $C_{max} = 25$ and $\lambda = 10^{-6}$. The definition of $g(C)$ implies that, if a checkpointing scheme achieves overhead and latency corresponding to a point "below" the $g(C)$ curve for $C_{max} = 25$, then this scheme achieves a smaller overhead ratio than the corresponding sequential checkpointing scheme (with checkpoint overhead 25). For instance, if some scheme reduces C from 25 to 10, then it can achieve a smaller overhead ratio r than the sequential checkpointing scheme, even if it increases the latency from 25 to as large as 2000.

Comparison of curves for $\lambda = 10^{-6}$ and 10^{-4} indicates that, for the same C_{max}, as λ increases, $g(C)$ decreases. This is intuitive, because with larger λ, it is necessary to keep checkpoint latency smaller (to avoid an increase in the overhead ratio).

The "measured L" curve in Figure 9 plots checkpoint overhead and latency measured for a merge sort program using four different checkpointing schemes – the data is borrowed from Li et al. [4]. (Although the data in [4] corresponds to a parallel implementation on a shared memory machine, our analysis is applicable to this implementation.) One of the four schemes in the "measured L" curve is sequential checkpointing with overhead $C_{max} = 31$ seconds. For comparison, Figure 9 also plots $g(C)$ for three different values of λ. Observe that, even when λ is as large as 10^{-4} per second, the measured checkpoint latency is well below the $g(C)$ curve. This indicates that, the checkpointing

[2] T_c is approximately $\sqrt{2 * C/\lambda}$ when $C << 1/\lambda$ (similarly, $T_m \approx \sqrt{2 * C_{max}/\lambda}$). Young [11] previously obtained this expression by a somewhat different analysis

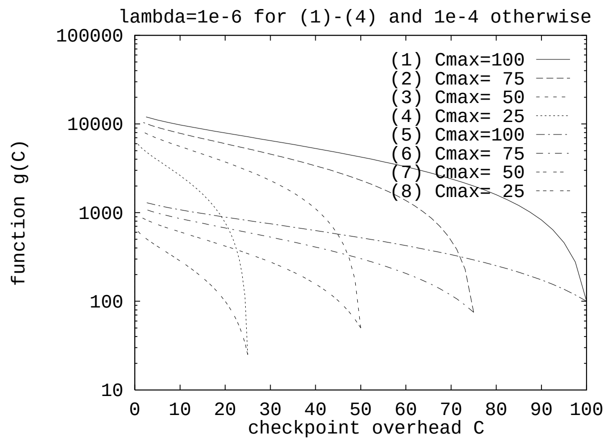

Figure 8: $g(C)$ for various values of C_{max}

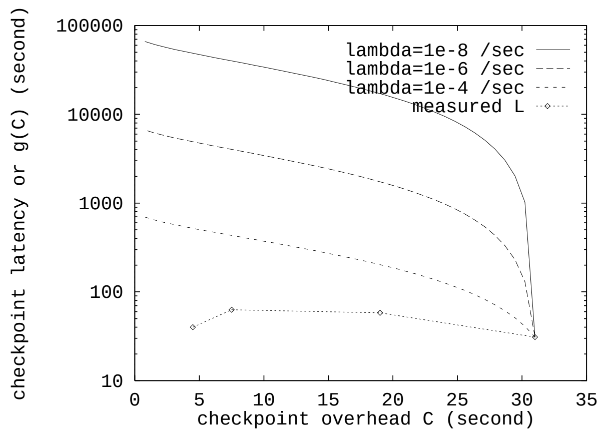

Figure 9: Comparison of measured checkpoint latency and $g(C)$ for $\lambda = 10^{-4}, 10^{-6}, 10^{-8}$ /sec

techniques used in practice can achieve a significantly smaller overhead ratio as compared to the sequential checkpointing scheme.

6 Conclusions

This paper evaluates an expression for the *overhead ratio* of a checkpointing scheme, as a function of checkpoint *latency* (L) and checkpoint *overhead* (C). Our analysis shows that, for an equi-distant checkpointing strategy, the optimal checkpoint interval is not dependent on the value of L – though it depends on the value of C. It is also observed that the *overhead ratio* is much more sensitive to the changes in C, as compared to changes in L. The paper uses a simple analytical model – if a different model is used, the previous observation will remain valid, however, the optimal checkpoint interval may not remain independent of L (although it should be less sensitive to L than C).

The paper considers only uni-process applications; the results can potentially be extended to multi-process applications as well [10].

References

[1] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, “Analytic models for rollback and recovery strategies in data base systems,” *IEEE Trans. Softw. Eng.*, vol. 1, March 1975.

[2] E. Gelenbe, “On the optimum checkpointing interval,” *J. ACM*, vol. 2, pp. 259–270, April 1979.

[3] P. L’Ecuyer and J. Malenfant, “Computing optimal checkpointing strategies for rollback and recovery systems,” *IEEE Trans. Computers*, vol. 37, pp. 491–496, April 1988.

[4] K. Li, J. F. Naughton, and J. S. Plank, “Low-latency, concurrent checkpointing for parallel programs,” *IEEE Trans. Par. Distr. Syst.*, vol. 5, pp. 874–879, August 1994.

[5] J. S. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent checkpointing under Unix,” in *Usenix Winter 1995 Technical Conference, New Orleans*, January 1995.

[6] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Prentice-Hall, 1988.

[7] N. H. Vaidya, “Another *two-level* failure recovery scheme: Performance impact of checkpoint placement and checkpoint latency,” Tech. Rep. 94-068, Computer Science Dept., Texas A&M University, College Station, December 1994.

[8] N. H. Vaidya, “Consistent logical checkpointing,” Tech. Rep. 94-051, Computer Science Department, Texas A&M University, College Station, July 1994.

[9] N. H. Vaidya, “A case for two-level distributed recovery schemes,” in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[10] N. H. Vaidya, “On checkpoint latency,” Tech. Rep. 95-015, Computer Science Department, Texas A&M University, College Station, March 1995.

[11] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Comm. ACM*, vol. 17, pp. 530–531, September 1974.

[12] A. Ziv and J. Bruck, “Analysis of checkpointing schemes for multiprocessor systems,” Tech. Rep. RJ 9593, IBM Almaden Res. Center, Nov. 1993.