Optimum Checkpoints for Programs with Loops

Miltiadis Siavvas[a], Erol Gelenbe[b]

[a] *Imperial College London, SW7 2AZ, London, United Kingdom*
[b] *Institute of Theoretical & Applied Informatics, Polish Academy of Sciences, Gliwice, Poland*

Abstract

Checkpoints are widely used to improve the performance of computer systems and programs in the presence of failures, and significantly reduce the cost of restarting a program each time that it fails. Application level checkpointing has been proposed for programs which may execute on platforms which are prone to failures, and also to reduce the execution time of programs which are prone to internal failures. Thus we develop a mathematical model to estimate the average execution time of a program in the presence of failures, without and with application level checkpointing, and use it to estimate the optimum interval in number of instructions executed between successive checkpoints. The case of programs with loops and nested loops is also discussed. The results are illustrated with several numerical examples.

Keywords: Cloud Computing, Software Reliability, Roll-Back Recovery, Application Level Checkpoints, Optimum Checkpoints, Program Loops

1. Introduction

Cloud and Fog Computing allows diverse software applications to run on complex interconnected systems where reliability and security can be of significant concern. Major failures in such systems occur [1], due to complex effects between various factors including human decisions and systemic interactions in the architecture, the software systems, and the network connections [2]. Furthermore, a recent report [3] states that "The main problems affecting the cloud are insecure interface APIs, shared resources, data breaches, malicious insiders, and misconfiguration issues" including active adversarial mechanisms [4]. Clearly, Cloud providers will do their best to improve the security and reliability of their platforms. However, we also need methods that can limit the average execution time of applications that run on the Cloud and Fog despite the intermittent failures of the platforms. This is particularly of interest for long-running applications or those that are run frequently and repeatedly.

One such mechanism that we investigate in this paper is the Application Level Checkpoint and Restart (ALCR) that is widely used to enhance the reliability of long-running programs [5–7] by periodically saving a copy or checkpoint of the current execution state of software. The most recent copy is the used to restart program execution in case of failure. Originally developed for transaction-oriented systems and databases [8–12], it has been widely adopted to improve the reliability of modern High Performance Computing (HPC) [13, 14] software.

Long intervals of time between checkpoints will increase the overhead associated with system restart, while short intervals will increase the overhead caused by the checkpoints themselves. The checkpoint

*Corresponding Author.
Email addresses: m.siavvas16@imperial.ac.uk (Miltiadis Siavvas), e.gelenbe@imperial.ac.uk (Erol Gelenbe)

interval must then be optimized so as to minimize a program's expected execution time in the presence of failures [15–17]. In [18, 19] the impact of asynchronous checkpointing strategies on the performance of distributed systems has been studied. Among the existing checkpointing strategies, ALCR [5, 20] uses a small memory footprint [6, 7], but requires significant expertise for the selection of source code locations in which checkpoints should be inserted. Yet existing ALCR tools and libraries facilitate the insertion of checkpoints in long-running loops, since computational loops constitute a significant source of failure-related re-executions [21, 22]. However such tools do not provide a method to select the inter-checkpoint interval which has a significant influence on the average execution time of software.

In this paper we propose that the inter-checkpoint intervals in specific loop be selected optimally as a function of program failure rate, the execution cost for establishing a checkpoint, and the execution time related to restarting the rogram after a failure, based on a mathematical model. We suggest that this approach can be implemented as an API within an ALCR tool, to select the optimum checkpoint interval in program loops.

In the sequel, Section 2 reviews earlier work. Section 3 provides examples to help understand the ALCR mechanism and its associated costs. Section 4 describes the mathematical model and the numerical approach. The optimum checkpoint interval is discussed in Section 4.3. Section 5 presents numerical examples and Section 6 presents conclusions and future research.

2. Related Work

If no scheme is adopted to enhance the performance of a transaction oriented system in the presence of failures, all previously executed transactions would need to be re-executed in case of a failure. The Checkpoint and Rollback/Recovery mechanism saves a secure and faithful copy of the system state at predetermined instants (the checkpoints); in addition in the case of transaction oriented systems, it will save an "audit trail" of the sequence of transactions that were executed since the most recent checkpoint was established. In case of a failure, only the transactions that were saved in the audit trail since the most recent checkpoint are re-executed [10]. Multiple level checkpoints were introduced in [9, 15] to deal with hierarchies of failures, and are also discussed in [23].

The selection of the optimum checkpoint interval (OCI) between two successive checkpoints will maximize the overall system or program availability [11], defined as the fraction of time when the system is available for useful operations. A badly chosen checkpoint interval results in high system response times and long average execution times [24, 25]. Therefore, much research has focused on how system and failure rate parameters affect its value, as well as on providing formulas for the calculation of its optimum value [8, 12, 26]. More specifically, the first attempt for determining the OCI was made by Young [8], who provided a first-order approximation of the OCI, using analytical methods. Another notable attempt was made by Daly [26], who extended the work made by Young [8], by relaxing the first-order assumptions and consequently by providing higher order approximation of the OCI. In particular, Daly provided a perturbation solution for the optimization problem of selecting the inter-checkpoint interval and showed (through simulation) that the higher order model leads to more accurate estimations of the OCI compared to first-order models. Finally, significant contributions in the field of OCI calculation were made by Gelenbe et al. [9–12].

Software applications are also often hampered by failure-provoking implementation issues [27]. Fault tolerance mechanisms are required to enhance their reliability [28, 29], and checkpointing is a useful solution [5, 13]. However, modern applications are considerably more complex than early transaction-oriented systems [30]. Therefore, a periodic copy of their overall execution state should be taken, in order to enhance their reliability [21].

Several rollback/recovery-based mechanisms for enhancing the reliability of long-running software applications exist, including: (i) Recovery Block schemes [30], (ii) N-Version Programming [31, 32], and (iii) the Checkpoint and Restart (CR) mechanism [5]. Despite their benefits, NVP and RB approaches are characterized by high development costs [33], which restricts their adoption to safety or reliability critical applications [34]. Despite attempts in addressing some drawbacks [28, 29, 34–36], their associated costs are still high.

CR mechanisms are generally preferred [5–7] since they introduce significantly less overhead compared to other counterparts [30, 31]. Mature CR tools and libraries exist for single-process software programs [37–39], and current research focuses chiefly on how to incorporate the CR mechanism in long-running HPC applications by integrating the CR mechanism into libraries such as OpenMPI [13, 40], OpenCL [14], OpenMP [21, 22], and CUDA [41]. They include [6]: (i) system-level CR [38], (ii) library-level CR [42], and (iii) application-level CR (ALCR) [6]. ALCR [6, 20] is considered the most efficient, since it leaves the smallest memory footprint [6, 7, 21], though it requires source code modifications to insert application-level checkpoints into the program, leading to higher development effort.

In [22] CPPC, an ALCR tool is presented to reduce the manual effort required by the developers, by automatically identifying judicious locations in which checkpoints can be introduced (in fact, long-running loops) and inserting checkpoints in the identified locations. In [21] an application-level checkpointing solution for hybrid MPI-OpenMP applications is suggested as an extension of CPPC. In [7] a library (CRAFT) was proposed for incorporating the application-level CR mechanism into software implemented in C++. Similarly to CPPC [22], the proposed library reduces the development time associated with the ALCR mechanism, by identifying lengthy loops and the automatic insertion of application-level checkpoints [43, 44]. In [6] a tool named ITALC is proposed to assist developers to semi-automatically re-engineer software by introducing application-level checkpoints in automatically identified hotspots.

One shortcoming of existing ALCR tools and libraries is that they do not provide recommendations regarding the optimum checkpoint interval, which is manually selected by the developers usually in an arbitrary manner. However, not optimally selecting the checkpoint interval is a challenging problem for all CR mechanisms regardless of type since it may lead to important execution time overheads. In fact, several studies have examined the impact of checkpointing and especially of the checkpoint interval selection on the performance of software applications. For instance, in [45] Oldfield et al., using Daly's model [26] for the calculation of the OCI, examined the impact of application-directed checkpointing on next generation massively parallel processing (MPP) systems, which comprise hundreds of thousands of processors. The results of their study indicate that in the next generation systems traditional CR mechanisms will increasingly impact application performance, accounting for 50% of the total application execution time.

In [46], Stavrinides and Karatza investigated the impact of application-directed checkpointing on the performance of the Software-as-a-Service (SaaS) cloud. More specifically, they examined via simulation the impact of checkpoint interval selection on the scheduling performance of real-time fine-grained parallel applications with firm deadlines and approximate solutions that run on SaaS cloud, under different failure probabilities. The results of their study suggest that the checkpoint interval must be selected taking into account the failure probability, as well as the nature of the workload. In addition, the selected checkpoint interval should be above a specific threshold as unnecessarily frequent checkpointing may lead to performance degradation.

Based on the above analysis, the arbitrary selection of the checkpoint interval should be avoided, due to its potential impact on the application performance. To this end, we provide a numerical approach for the calculation of the optimum checkpoint interval of long-running loops, that minimizes the expected execution time of software applications that adopt the ALCR mechanism. The proposed method can be used along

with the existing ALCR libraries, in order to enhance the performance of software applications.

```
#include <mpi.h>

int main(int argc, char* argv[]){
  int n=5, iteration=1;
  double dbl = 0.0;
  int * dataArr    = new int[n];




  for(; iteration <= 100 ; iteration++){
    // Computation-communication loop
    modifyData(&dbl, dataArr);

  }
  return EXIT_SUCCESS;
}
```

Listing 1: A sample application without application-level checkpoints.

```
#include <mpi.h>
#include <craft.h>
int main(int argc, char* argv[]){
  int n=5, iteration=1, cpFreq=10;
  double dbl = 0.0;
  int * dataArr    = new int[n];
  // ===== DEFINE CHECKPOINT ===== //
  Checkpoint myCP("myCP", MPI_COMM_WORLD);
  myCP.add("dbl", &dbl);
  myCP.add("iteration", &iteration);
  myCP.add("dataArr", dataArr, n);
  myCP.commit();
  myCP.restartIfNeeded(&iteration);
  for(; iteration <= 100 ; iteration++){
    // Computation-communication loop
    modifyData(&dbl, dataArr);
    myCP.updateAndWrite(iteration, cpFreq);
  }
  return EXIT_SUCCESS;
}
```

Listing 2: Corresponding changes in Listing 1 for CRAFT enabled application-level checkpoints.

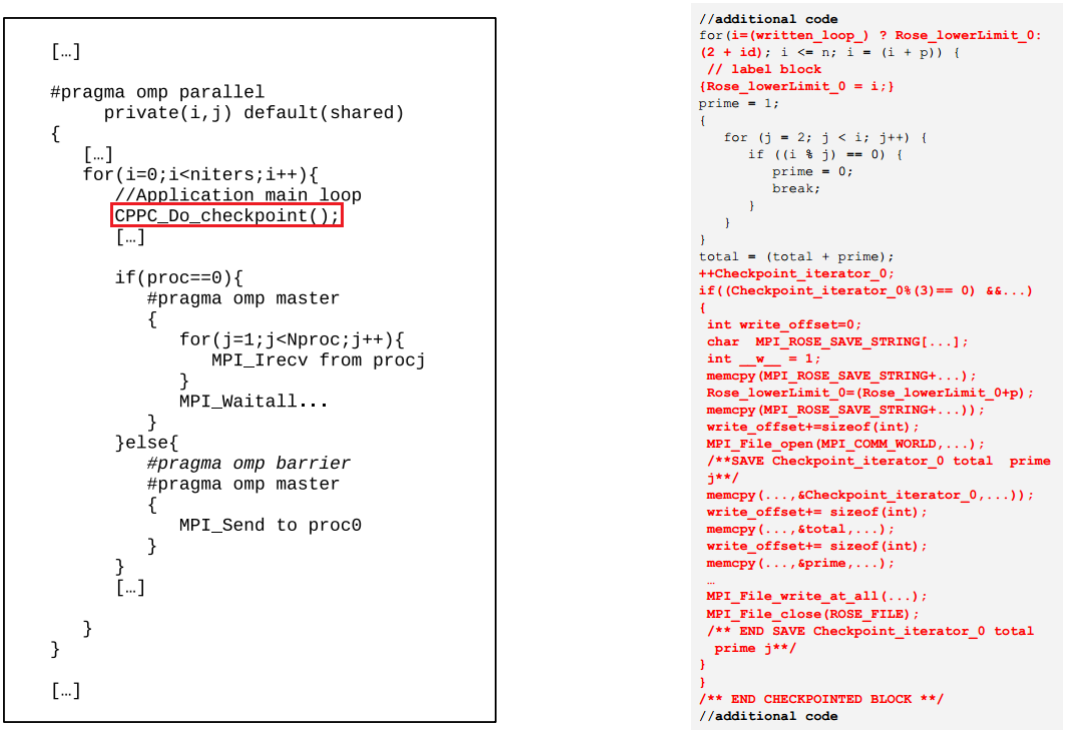
Figure 1: The additional code that should be inserted for adding an application-level checkpoint in a lengthy loop, using CRAFT (Adapted from [7]).

3. Indicative Examples

In this section, some examples are provided regarding the changes that should be performed to the source code of a software application, in order to add checkpoints into long-running loops, using actual ALCR libraries. Their purpose is to help the reader understand the overall concept of the ALCR mechanism, and also to explain why the arbitrary selection of the checkpoint interval may affect negatively the execution time of a software application. These examples are also expected to facilitate the understanding of the proposed mathematical model that is presented in Section 4, as well as the numerical examples that are provided in Section 5.

Figure 1 demonstrates how the source code of a software application should be modified, in order to insert an application-level checkpoint in a long-running loop. As can be seen by the given example, a non-trivial number of library-specific statements (marked with red color) should be added to the program, in order to insert a checkpoint in the selected loop. The example is based on a specific ALCR library called CRAFT [7]. However, similar modifications are also required by other ALCR libraries, as can be seen by Figure 2.

The examples in Figures 1 and 2 show that the insertion of checkpoints into long-running loops necessitates significant source code modifications. In fact, a number of statements should be inserted, in order to determine: (i) the location of the checkpoints, (ii) what data should be safely stored every time a checkpoint is generated, and (iii) how frequently a checkpoint should be generated (e.g. the value of cpFreq in Figure 1). These additional statements are *methods* of a specific library, and may require several operations with significant execution time. For instance, the *updateAndWrite()* method in Figure 1, is expensive in execution time, since the creation of a checkpoint usually requires multiple memory accesses [6, 7]. Thus the execution time of checkpointing must be added to the overall execution time of the software application. If failures are very infrequent and the cost of checkpointing is high, frequent checkpoints will result in an average execution time of the application that is higher than the same application which runs without checkpoints.



```
[…]

#pragma omp parallel
     private(i,j) default(shared)
{
   […]
   for(i=0;i<niters;i++){
      //Application main loop
      CPPC_Do_checkpoint();
      […]

      if(proc==0){
         #pragma omp master
         {
            for(j=1;j<Nproc;j++){
               MPI_Irecv from procj
            }
            MPI_Waitall...
         }
      }else{
         #pragma omp barrier
         #pragma omp master
         {
            MPI_Send to proc0
         }
      }
      […]

   }
}

[…]
```

(a)

```
//additional code
for(i=(written_loop_) ? Rose_lowerLimit_0:
(2 + id); i <= n; i = (i + p)) {
 // label block
{Rose_lowerLimit_0 = i;}
prime = 1;
{
   for (j = 2; j < i; j++) {
      if ((i % j) == 0) {
         prime = 0;
         break;
      }
   }
}
total = (total + prime);
++Checkpoint_iterator_0;
if((Checkpoint_iterator_0%(3)== 0) &&...)
{
 int write_offset=0;
 char  MPI_ROSE_SAVE_STRING[...];
 int __w__ = 1;
 memcpy(MPI_ROSE_SAVE_STRING+...);
 Rose_lowerLimit_0=(Rose_lowerLimit_0+p);
 memcpy(MPI_ROSE_SAVE_STRING+...));
 write_offset+=sizeof(int);
 MPI_File_open(MPI_COMM_WORLD,...);
 /**SAVE Checkpoint_iterator_0 total  prime
 j**/
 memcpy(...,&Checkpoint_iterator_0,...));
 write_offset+= sizeof(int);
 memcpy(...,&total,...);
 write_offset+= sizeof(int);
 memcpy(...,&prime,...);
 …
 MPI_File_write_at_all(...);
 MPI_File_close(ROSE_FILE);
 /** END SAVE Checkpoint_iterator_0 total
  prime j**/
}
}
/** END CHECKPOINTED BLOCK **/
//additional code
```

(b)

Figure 2: The additional source code (marked with red color) required for inserting an application-level checkpoint into a loop, using: (a) an extension of the CPPC tool proposed by Losada et al. [21], and (b) the ITALC ALCR library [6].

Thus the total execution time of a program which contains checkpoints in its loops is presented in Section 4, and numerical examples are provided in Section 5.

4. Expected Execution Time of a Program Without and With Checkpoints

Consider a program P that executes a total of M instructions; it may contain loops so that M is the total number of instructions it executes. Assume that when the execution starts, there is an overhead associated with loading its data and code into memory, which consumes A time units. If the program is executed without any errors or failures, and if each instruction is executed in c time units, then the total execution time for P will be:

$$T(P) = A + cM. \tag{1}$$

Now suppose that no failures or errors occur during the initial and final durations A, B, however with probability g there may be a failure in any one of the instructions. We assume that the failure is detected after a delay which takes δ time units. It should be noted that the main notations and parameters that are used in this section are summarized in Table 1, in order to facilitate the readability and understandability of the mathematical equations presented in this section.

Table 1: Main notations and parameters of the mathematical model

Notation	Description
$E\tau(P)$	Expected execution time of a program P without checkpoints
$E_{cp}\tau(P)$	Expected execution time of a program P with checkpoints
A	Execution cost of initial program loading (code and data)
δ	Failure detection latency
c	Execution cost of a single instruction
g	Failure probability of a single instruction
M	Total number of instructions of the program
K	The checkpoint interval (expressed in number of instructions)
K^*	The optimum checkpoint interval
$B(K)$	Execution cost of checkpointing
$B(M,K)$	Number of successive blocks of instructions created due to checkpointing

4.1. Expected Execution Time Without Checkpoints

When a failure is detected, the program has to be re-executed, and if the failures occur during further executions, the execution may have to be repeated several times. Let $\tau(P)$ denote the total execution time of the program, and let $E\tau(P)$ be its expected value. Then:

$$\begin{aligned} &E\tau(P) = (1-g)^M(A + c.M + \delta) \\ &+\sum_{u=1}^{M}(1-g)^{u-1}g[A + c.u + \delta + E\tau(P)], \qquad (2) \\ &= A + (1-g)^M c.M + \delta + [1-(1-g)^M]E\tau(P) \\ &+c[\frac{1-(1-g)^M}{g} - M(1-g)^M],\ \textit{hence} \\ &E\tau(P) = \frac{A+\delta}{(1-g)^M} + c.\frac{1-(1-g)^M}{g(1-g)^M}. \qquad (3) \end{aligned}$$

If a failure occurs, this only becomes known after δ time units, the program has to be restarted and run again, so that the time $A + c.u + \delta$ has been wasted. When there are no failures we see from (3) that

$$E\tau(P) = A + \delta + M, \qquad (4)$$

since:

$$\lim_{g\to 0}\frac{1-(1-g)^M}{g(1-g)^M} = M. \qquad (5)$$

When g is very small so that $gM << 1$, we can use the following approximation directly from (3):

$$E\tau(P) \approx \frac{A+\delta+c.M}{1\ -\ g.M}. \qquad (6)$$

4.2. Estimating the Failure Probability g

In order to use the above expressions, we will need $(1-g)^M$ the probability that no error or failure will occur during the program's execution, and the probability that at least one failure occurs during the program's execution is $F = 1-(1-g)^M$. Note that the notion of a failure in this case is that of any event that stops the execution of the program and which arises from the program's execution environment, i.e. the platform. If $gM << 1$ then $F \approx gM$.

The value of g can be estimated as follows. Take a simple linear code that executes M instructions, and then repeats the execution, i.e. a single loop containing M sequential instructions. This code should not contain any ALCR or other checkpointing constructs.

1. Run the program repeatedly. Each time the program returns to the first instruction, increment the counter $N \leftarrow N+1$.
2. If the program execution stops, increment a counter $N_F \leftarrow N_F + 1$. Update $g \approx \frac{N_F}{N.M}$.
3. Then restart the program at its initial instruction and set $N \leftarrow N+1$.

4.3. Optimum Checkpoints

When the program must run for a long time, i.e. when M is large of failure Mg cannot be neglected, checkpoints can be placed at periodic intervals, say after K instructions are executed, but they result in a cost $B(K)$ in the amount of time needed to create the checkpoint, since the status of the program and all its data must be saved. $B(K)$ may be an increasing function of K when the data that the program has modified during the interval of execution of K instructions needs to be saved. Thus the program will now execute a total of M instructions in successive blocks of $b(M,K) = \lceil \frac{M}{K} \rceil$ instructions, all of which are of length K, except for the last one of length $K_o = M - K[\lceil \frac{M}{K} \rceil - 1]$.

Applying the previous analysis, we compute the total average execution time of the program with checkpoints:

$$
\begin{aligned}
E_{cp}\tau(P) &= \frac{A+\delta}{(1-g)^K} + c\frac{1-(1-g)^K}{g(1-g)^K} \\
&+ [b(M,K)-2][\frac{B(K)+\delta}{(1-g)^K} + c\frac{1-(1-g)^K}{g(1-g)^K}] \\
&+ \frac{B(K)+\delta}{(1-g)^{K_o}} + c\frac{1-(1-g)^{K_o}}{g(1-g)^{K_o}}, \\
&= \frac{A+\delta}{(1-g)^K} + \frac{[b(M,K)-2][B(K)+\delta]}{(1-g)^K} \\
&+ c\frac{[b(M,K)-1][1-(1-g)^K]}{g(1-g)^K} \\
&+ \frac{B(K)+\delta}{(1-g)^{K_o}} + c\frac{1-(1-g)^{K_o}}{g(1-g)^{K_o}}.
\end{aligned}
\tag{7}
$$

Therefore optimum checkpoint interval K^* is the value of K that minimizes $E_{cp}\tau(P)$, which can be computed numerically from (17).

In order to better illustrate the benefit of the ALCR we also define the percentage *Gain*:

$$
Gain = \frac{E\tau(P) - E_{cp}\tau(P)}{E\tau(P)} \times 100, \tag{8}
$$

where $E\tau(P)$ is the expected execution time of the program (or software application) P when ALCR is not used.

4.4. Program with a Long Loop

Suppose that a program contains a single loop with L instructions that is executed repeatedly n times so that the program executes $M = n.L$ instructions. If a checkpoint is inserted for each I loops so that the block of executed instructions between checkpoints is of length $K = I.L$, then a total of $b(nL, IL) = \lceil \frac{n}{I} \rceil - 1$ checkpoints are placed, since the start of the loop will in itself require a checkpoint. From equation (9) with $M = n.L$ and $K = I.L$ we have:

$$\begin{aligned} E_{cp}\tau(P) &= \frac{[b(M,K)-1][B(K)+\delta]}{(1-g)^K} \\ &+c\frac{[b(M,K)-1][1-(1-g)^K]}{g(1-g)^K} + \frac{B(K)+\delta}{(1-g)^{K_o}} \\ &+c\frac{1-(1-g)^{K_o}}{g(1-g)^{K_o}}. \end{aligned} \tag{9}$$

If the number of instructions that are executed during a single loop iteration is L, the optimum number of iterations between two successive checkpoints is $I^* = \frac{K^*}{L}$.

4.5. Nested Loops

Suppose that we identify, either manually or using an ALCR library, that the best location for adding checkpoints is a loop that contains one or more internal loops. These internal loops can be treated in a black-box manner as normal statements (e.g. method calls), which require the execution of a number of instructions. The number of instructions executed in the internal loops can be used to calculate the values of L and M of the selected outer loop yielding the optimum number of loop iterations between checkpoints I^*.

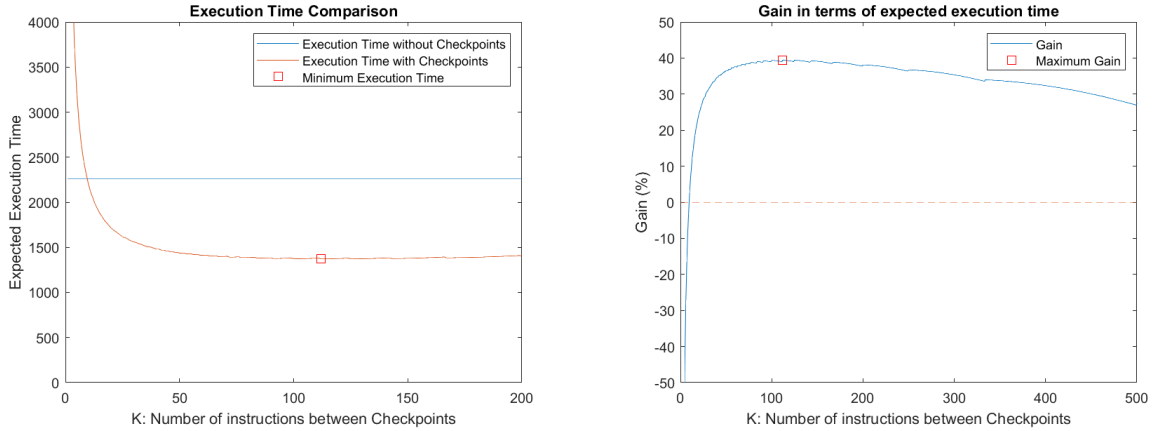
Another possible approach, especially in the case of nested loops with multiple levels of nesting, is to treat the nested loops individually. In this case, checkpoints could be generated in the innermost loop. However the latter approach would require additional development effort, in order to map iterations to actual instructions, since ALCR libraries usually focus on iterations.

5. Numerical Examples

In this section, the effect of the checkpoint interval K on the expected execution time of a software application is illustrated through a set of numerical examples. More specifically, the case of a software application with a single loop is considered, and the analysis is repeated for different loop sizes (i.e., different values of M). For each one of these cases, the expected execution time of the same program with and without the adoption of the ALCR mechanism is calculated, while the optimum checkpoint interval is computed based on the mathematical model proposed in Section 4. In particular, the results presented in this section were obtained through simulation, by implementing the mathematical equations presented in Section 4 in MATLAB, and by solving the corresponding optimization problem numerically.

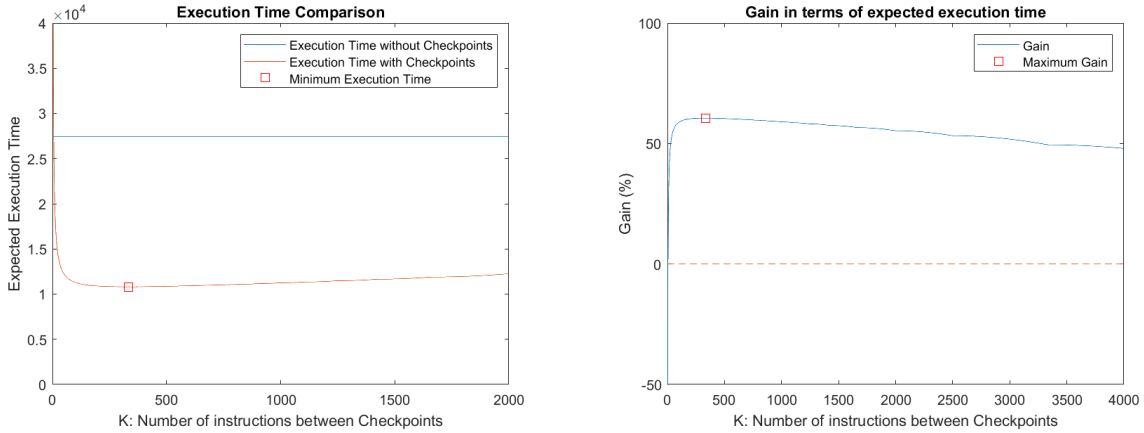
For the purposes of the present experiment, four cases of loop sizes were considered. In particular, we considered the cases of a small, medium, large, and very large loop, comprising 10^3, 10^4, 10^5, and 10^7 iterations respectively. We selected these values of M so that there would be at least one order of magnitude of difference between the loop sizes of the different numerical examples. The reasoning behind this option is that we wanted to examine whether the same behavior is observed and the same observations hold for significantly different loop sizes.

In Figure 3, the case of a software application with a relatively small loop having $M = 1000$ is presented. Figure 3a compares the expected execution time of the application with and without the ALCR mechanism for different values of K, while Figure 3b shows the expected *Gain* of Section 4.3 for different values of K. The values that correspond to the optimum checkpoint interval K^* are marked within a rectangle. Figure 3 illustrates the fact that the optimum checkpoint interval K^* minimizes the overall execution time of the application and maximizes the overall expected Gain. Therefore, the ALCR mechanism will not reduce the expected execution time of a given software application unless the checkpoint interval is optimally selected. Indeed, for some poorly chosen values of K, the expected execution time of the application with checkpointing is higher than the expected execution time of the same application without checkpoints. Similar observations can be made for software with longer loops in Figures 4, 5 and 6. This emphasizes the importance of setting K to be close or at K^*.



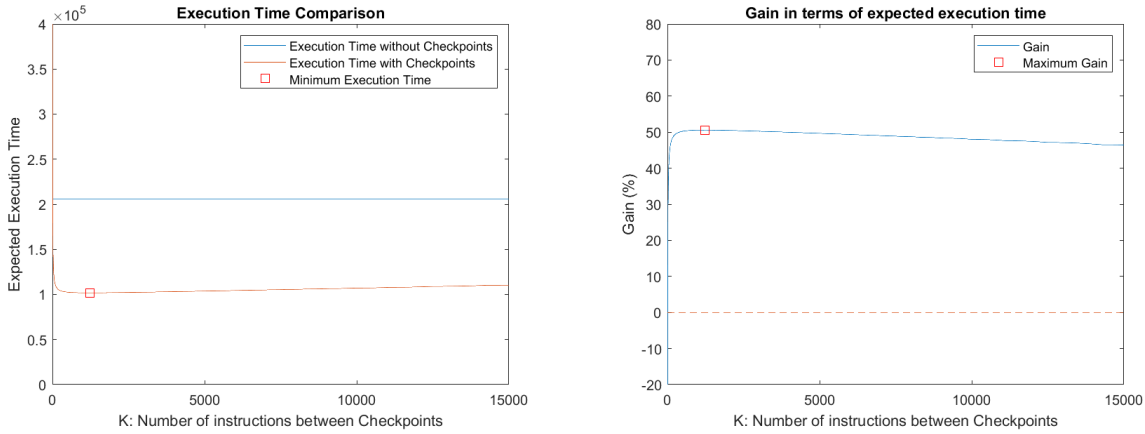
(a) (b)

Figure 3: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K, for a software application having a small computational loop with $M = 1000$.



(a) (b)

Figure 4: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K, for a software application having a medium length computational loop with $M = 10^4$.



(a) (b)

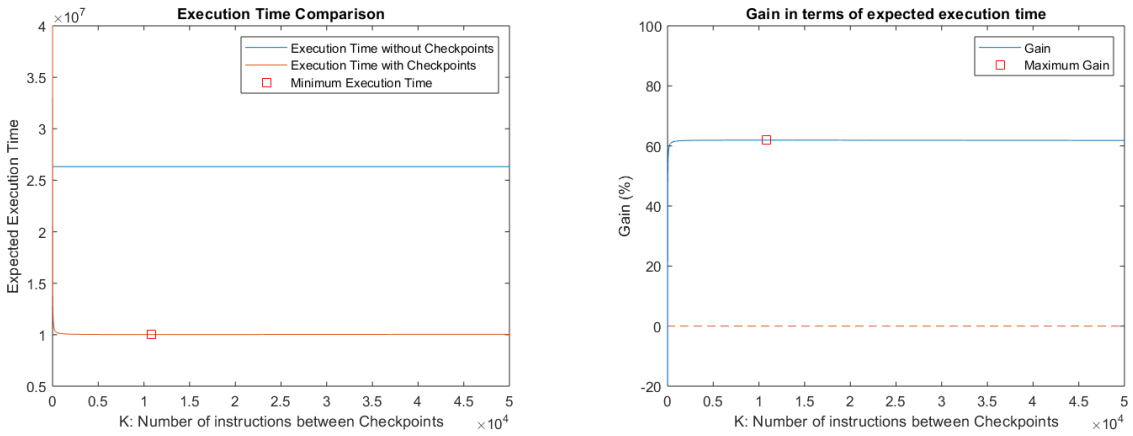
Figure 5: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K, for a software application with a large computational loop with $M = 10^5$.

The examples of Figures 3, 4, 5 and 6 show that a significant reduction in the execution time of a software application can be achieved by the ALCR mechanism, if the checkpoint interval is selected to be at, or close to, the optimum K^*. In these examples the Gain, ranges from 40% to 60%. However, suboptimal values of the checkpoint interval will lead to a smaller Gain or even to an average execution time which is larger than when ALCR is not used. Indeed, the checkpoint interval should not be selected arbitrarily and must be tuned to a value at, or close to, the optimum K^*.

6. Conclusions and Future Work

This paper has proposed a method for setting the checkpoint intervals in ALCR for software applications which contain long-running loops, and which run on platforms that are subject to failures. We have shown that the optimum checkpoint interval, which minimizes the expected execution time of the program, depends on various parameters which can be incorporated into a single numerical expression. The expression can then be used as part of an ALCR tool to compute the optimum checkpoint interval for each individual loop in the program. The approach can be used through a set of MATLAB scripts that calculate the optimum checkpoint interval of different computational loops. We have illustrated our results via different numerical examples.

Several directions for future work can be considered. Energy consumption is a critical property that will be affected by checkpointing, as well as by the re-execution of a program in case of failure. Thus in recent work [47], checkpointing was considered from the perspective of its effect on energy consumption. Further research is needed to see how the checkpoint interval can be selected to achieve a compromise between energy consumption and execution times. The issue can become quite intricate if we consider the effect of the secondary memory medium. Since checkpointing will generally increase the use of secondary memory, and secondary memory related failures may increase with the amount of usage, a platform where many applications use ALCR, may have a failure rate which increases with usage and age. Thus a time dependent value of the failure probability g will need to be considered in this case. Furthermore, with rotating secondary memories and some other devices, the use of ALCR will also increase energy consumption. These are all questions that require further research.



(a) (b)

Figure 6: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K, for a software application with a very large computational loop with $M = 10^7$.

In addition, an important parameter of time-critical software applications, apart from their overall performance and energy consumption, is the satisfaction of strict deadlines. More specifically, time-critical programs that run on systems like SaaS and HPC clouds, often require their execution to be completed before specific deadlines. Failure to meet these time restrictions often lead to the rejection of their results. Hence, meeting these deadlines is critical, whereas it can be affected (either positively or negatively) by the adoption of checkpointing. Therefore, in the future, we are planning to take into account this parameter for the calculation of the optimum checkpoint interval, along with the performance and energy consumption of the applications.

Finally, another interesting area of investigation is the use of ALCR to restart applications after attacks. Indeed, ALCR could perhaps be used to disrupt the attacker, but in turn attackers may exploit ALCR to create an increase in workload in a system, leading to a form of Denial of Service through workload saturation. Thus the interaction of ALCR and checkpointing in general, and security, is also a worthwhile subject of investigation.

Acknowledgments

This research was supported by the European Commission through the Horizon 2020 SDK4ED Project under Grant Agreement No. 780572, but does not engage the responsibility of the European Commission.

References

[1] Summary of the Amazon s3 service disruption in the northern Virginia (us-east-1) region, 2018.
[2] Top 20 high profile cloud failures of all time, TechFlier (2016).
[3] C. Wueest, M. B. Barcena, L. O'Brien, Mistakes in the IaaS cloud could put your data at risk, Symantec (2015).
[4] E. Gelenbe, Dealing with software viruses: a biological paradigm, Information Security Technical Report 12 (2007) 242–250.
[5] I. P. Egwutuoha, D. Levy, B. Selic, S. Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, Journal of Supercomputing 65 (2013) 1302–1326.
[6] R. Arora, ITALC : Interactive Tool for Application - Level Checkpointing, Proceedings of the Fourth International Workshop on HPC User Support Tools (2017).

[7] F. Shahzad, J. Thies, G. Wellein, CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance, IEEE Transactions on Parallel and Distributed Systems (2018).
[8] J. W. Young, A First Order Approximation to the Optimum Checkpoint Interval, Commun. ACM 17 (1974) 530–531.
[9] E. Gelenbe, A Model of Roll-back Recovery with Multiple Checkpoints, in: Proceedings of the 2nd International Conference on Software Engineering, ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, pp. 251–255. URL: http://dl.acm.org/citation.cfm?id=800253.807684.
[10] E. Gelenbe, D. Derochette, Performance of Rollback Recovery Systems Under Intermittent Failures, Commun. ACM 21 (1978) 493–499.
[11] E. Gelenbe, On the Optimum Checkpoint Interval, Journal of the ACM 26 (1979) 259–270.
[12] E. Gelenbe, M. Hernández, Optimum Checkpoints With Age-Dependent Failures, Acta Informatica 531 (1990) 519–531.
[13] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A Survey of Rollback-recovery Protocols in Message-passing Systems, ACM Comput. Surveys 34 (2002) 375–408.
[14] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, H. Kobayashi, CheCL: Transparent checkpointing and process migration of OpenCL applications, Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011 (2011) 864–876.
[15] E. Gelenbe, Model on information renewal by the method of multiple test-points (english translation of *avtomat. i telemekh.*, 1979:4, pp. 142-151, Automation and Remote Control 40 (1979) 598–605.
[16] E. Gelenbe, G. Pujolle, Introduction aux Réseaux de Files d'Attente, Editions Hommes et Techniques, Eyrolles, 1982.
[17] E. Gelenbe, I. Mitrani, Analysis and synthesis of computer systems, World Scientific, 2010.
[18] E. Gelenbe, D. Finkel, S. K. Tripathi, Availability of a distributed computer system with failures, Acta Informatica 23 (1986) 643.
[19] S. K. Tripathi, D. Finkel, E. Gelenbe, Load sharing in distributed systems with failures, Acta Inf. 25 (1988) 677–689.
[20] J. Walters, V. Chaudhary, Application-Level Checkpointing Techniques for Parallel Programs, Distributed Computing and Internet Technology (2006) 221–234.
[21] N. Losada, M. J. Martín, G. Rodríguez, P. Gonzalez, Portable application-level checkpointing for hybrid MPI-OpenMP applications, Procedia Computer Science 80 (2016) 19–29.
[22] G. Rodríguez, M. J. Martín, P. González, J. Touriño, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurrency and Computation: Practice and Experience 22 (2010) 749–766.
[23] D. Dauwe, S. Pasricha, A. A. Maciejewski, H. J. Siegel, An analysis of multilevel checkpoint performance models, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, 2018, pp. 783–792. doi:10.1109/IPDPSW.2018.00125.
[24] E. Gelenbe, M. Hernández, Enhanced availability of transaction oriented systems using failure tests, Software Reliability Engineering, 1992. Proceedings., Third International Symposium on (1992) 342–350.
[25] E. Gelenbe, M. Hernández, Virus tests to maximize availability of software systems, Theoretical Computer Science 125 (1994) 131–147.
[26] J. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Generation Computer Systems 22 (2006) 303 – 312.
[27] E. Dijkstra, J. Buxton, B. Randell, Software Engineering Techniques, NATO Science Committee (1969).
[28] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, M. Pezze, Automatic recovery from runtime failures, Proceedings - International Conference on Software Engineering (2013) 782–791.
[29] A. Carzaniga, A. Gorla, N. Perino, M. Pezzè, Automatic workarounds: Exploiting the intrinsic redundancy of web applications, ACM Trans. Softw. Eng. Methodol. 24 (2015) 16:1–16:42.
[30] B. Randell, System Structure for Software Fault Tolerance, Science (1975) 1–18.
[31] L. Chen, A. Avizienis, N-version programming: A fault-tolerance approach to reliability of software operation, in: Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8), volume 1, 1978, pp. 3–9. URL: http://asusrl.eas.asu.edu/share/xyz/Research/Research/Dissertation/papers/IndexedPapers/nversionprogramming.pdf. doi:10.1109/FTCSH.1995.532621.
[32] A. Avižienis, Fault-tolerance and Fault-intolerance: Complementary Approaches to Reliable Computing, SIGPLAN Not. 10 (1975) 458–464.
[33] E. Gelenbe, I. Mitrani, Modelling the behaviour of block structured processes with hardware and software failures, in: e. a. Giuseppe Iazeolla (Ed.), Mathematical Computer Performance and Reliability, Elsevier Science Publishers, 1984, pp. 329–339.
[34] A. Armoush, F. Salewski, S. Kowalewski, A hybrid fault tolerance method for recovery block with a weak acceptance test, Proceedings of The 5th International Conference on Embedded and Ubiquitous Computing, EUC 2008 1 (2008) 484–491.
[35] A. Carzaniga, A. Gorla, N. Perino, M. Pezzè, Automatic Workarounds for Web Applications, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, ACM, New York, NY, USA, 2010, pp. 237–246. URL: http://doi.acm.org/10.1145/1882291.1882327. doi:10.1145/1882291.1882327.

[36] M. Uva, P. Ponzio, G. Regis, N. Aguirre, M. F. Frias, Automated Workarounds from Java Program Specifications Based on SAT Solving, in: M. Huisman, J. Rubin (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 356–373.

[37] J. S. Plank, M. Beck, G. Kingsley, K. Li, Libckpt: Transparent Checkpointing Under Unix, in: Proceedings of the USENIX 1995 Technical Conference Proceedings, TCON'95, USENIX Association, Berkeley, CA, USA, 1995, p. 18. URL: http://dl.acm.org/citation.cfm?id=1267411.1267429.

[38] J. Duell, P. Hangrove, E. Roman, The design and implementation of berkeley lab's linux checkpoint/restart, Berkeley Lab Technical Report (publication LBNL-54941) (2002).

[39] M. Litzkow, T. Tannenbaum, M. Linvy, Checkpoint and migration of UNIX processes in the Condor distributed processing system, Technical Report CS-TR- 199701346, University of Wisconsin, Madison (1997).

[40] J. Hursey, J. M. Squyres, T. I. Mattox, A. Lumsdaine, The Design and Implementation of Checkpoint / Restart Process Fault Tolerance for Open MPI, Architecture (2007).

[41] H. Takizawa, K. Sato, K. Komatsut, H. Kobayashit, CheCUDA: A checkpoint/restart tool for CUDA applications, Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings (2009) 408–413.

[42] J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent checkpointing for cluster computations and the desktop, IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (2009).

[43] F. Shahzad, M. Wittmann, T. Zeiser, G. Wellein, Asynchronous Checkpointing by Dedicated Checkpoint Threads, in: J. L. Träff, S. Benkner, J. J. Dongarra (Eds.), Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 289–290.

[44] A. Moody, G. Bronevetsky, K. Mohror, B. R. d. Supinski, Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, in: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–11. doi:10.1109/SC.2010.18.

[45] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, P. C. Roth, Modeling the impact of checkpoints on next-generation systems, in: 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007), IEEE, 2007, pp. 30–46.

[46] G. L. Stavrinides, H. D. Karatza, The impact of checkpointing interval selection on the scheduling performance of real-time fine-grained parallel applications in SaaS clouds under various failure probabilities, Concurrency and Computation: Practice and Experience 30 (2018) e4288.

[47] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, H. J. Siegel, Optimizing checkpoint intervals for reduced energy use in exascale systems, in: Eighth International Green and Sustainable Computing Conference, IGSC 2017, Orlando, FL, USA, October 23-25, 2017, 2017, pp. 1–8. URL: https://doi.org/10.1109/IGCC.2017.8323598.