Optimum Interval for Application-level Checkpoints

Miltiadis Siavvas*†, Erol Gelenbe‡ *Fellow, IEEE*
* Imperial College London, London, United Kingdom
†Centre for Research and Technology Hellas, Thessaloniki, Greece
‡Institute of Theoretical & Applied Informatics, Polish Academy of Sciences, Gliwice, Poland
Email: m.siavvas16@imperial.ac.uk, siavvasm@iti.gr

***Abstract*—Checkpointing is commonly adopted for enhancing the performance of software applications that operate in the presence of failures. Among the existing checkpointing strategies, Application-level Checkpoint and Restart (ALCR) is considered the most efficient, since it leaves smaller memory footprint, but it requires significant development effort. Although existing ALCR tools and libraries manage to reduce the effort required for implementing the checkpoints, they do not provide recommendations regarding their inter-checkpoint interval. To this end, in the present paper, we develop a mathematical model to estimate the optimum checkpoint interval, i.e., the interval between two successive checkpoints that minimises the average execution time of the application. The case of programs with loops and nested loops is also discussed. The results are illustrated with several numerical examples.**

***Index Terms*—Cloud Computing, Software Reliability, Roll-Back Recovery, Application Level Checkpoints, Optimum Checkpoints, Program Loops**

I. 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 network connections [2] and security failures [3]. A recent report [4] 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 [5]. 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.

Application Level Checkpoint and Restart (ALCR) is widely used to enhance the reliability of long-running programs [6]–[8] by periodically saving a copy or checkpoint of the current execution state of software. The most recent copy is then used to restart program execution in case of failure. Originally developed for transaction-oriented systems and databases [9]–[13], it has been widely adopted to improve the reliability of modern High Performance Computing (HPC) [14], [15] 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 interval must then be optimized so as to minimize a program's expected execution time in the presence of failures [16]–[18]. In [19], [20] the impact of asynchronous checkpointing strategies on the performance of distributed systems has been studied. Among the existing checkpointing strategies, ALCR [6], [21] uses a small memory footprint [7], [8], 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 [22], [23]. 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 loops 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 program 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 II reviews earlier work. Section II-A provides an example to help understand the ALCR mechanism and its associated costs. Section III describes the mathematical model and the numerical approach. The optimum checkpoint interval is discussed in Section III-C. Section IV presents numerical examples and Section V presents conclusions and future research.

II. Prior Work

If no fault tolerance scheme is adopted by a transaction-oriented system, 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) and in case of a failure, only the transactions since the most recent checkpoint are re-executed [11]. Multiple level checkpoints were introduced in [10], [16] to deal with hierarchies of failures, and are also discussed in [24].

The selection of the optimum checkpoint interval (OCI) between two successive checkpoints will maximize the overall system or program availability [12], 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 [25], [26]. Therefore, much research has focused on how system and failure rate parameters affect its value [9], [13].

Software applications are also often hampered by failure-provoking implementation issues [27]. Fault tolerance mechanisms are required to enhance their reliability [28], and checkpointing is a useful solution [6], [14]. However, since modern applications are considerably more complex than early transaction-oriented systems [29], a periodic copy of their overall execution state should be taken [22].

The problem of fault tolerance is more challenging in cloud-based systems since the cloud computing architecture is highly complex and dynamically growing [30], [31]. According to a recent survey [32], among the traditional fault tolerance mechanisms (e.g., [6], [29], [33]), Checkpoint and Restart (CR) [6] is commonly used for implementing fault tolerance in the Cloud. The CR mechanism is normally utilized in order to restart applications in case of failures (e.g., [34], [35]), while it is also used for migrating tasks and applications from one node of the cloud to another (e.g., [36]) when special circumstances impose it (e.g., node unavailability).

Mature CR tools and libraries exist both for single-process software programs [37], and for multi-process long-running applications (e.g., HPC applications) [23]. They are often divided into [7]: (i) system-level CR [37], (ii) library-level CR [38], and (iii) application-level CR (ALCR) [7]. ALCR [7], [21] is considered the most efficient, since it leaves the smallest memory footprint [7], [8], [22]; however it requires manual source code modifications for introducing checkpoints into the program.

Existing ALCR tools and libraries (e.g., [7], [8], [23], [39]) manage to reduce the manual effort required by the developers, by (i) automatically identifying judicious locations in which checkpoints should be inserted (in fact, long loops), and by (ii) automating the insertion of the checkpoints into the identified locations. However, their major shortcoming is that they do not provide recommendations regarding the optimum checkpoint interval, which is selected by the developers usually in an arbitrary manner. Since the arbitrary selection of the checkpoint interval may affect the performance of software applications, in the present paper, we propose a mathematical model for the calculation of the checkpoint interval that minimizes the expected execution time of software applications.

A. Indicative Example

In this section, a real-world example is provided that demonstrates how an actual ALCR library is used for adding checkpoints in long-running loops of software applications. This example, which is illustrated in Listing 1, is based on a specific ALCR library called CRAFT [8]. Its main purpose is to familiarize the reader with the technical details of the ALCR mechanism and also to clarify why the arbitrary selection of the inter-checkpoint interval could potentially affect the execution time of a software application. As can be seen by Listing 1, an important number of statements (marked with red color) should be added to the program, in order to insert a checkpoint in a long-running loop. These additional statements are *methods* of a specific ALCR library, which may execute computationally expensive operations behind the scenes. For instance, the creation of a checkpoint (which corresponds to the *updateAndWrite()* method in the given example) usually requires multiple memory accesses, and therefore it is expensive in terms of execution time [7], [8]. Thus the execution time of checkpointing must be added to the overall execution time of the software application. If failures are rare occurrences 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. Hence, the checkpoint interval should be optimally selected, in order to avoid the introduction of execution time overhead.

```
#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;
}
```

```
#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);
    myCP.commit();
    myCP.restartIfNeeded(&iteration);
    for (; iteration <= 100; iteration++) {
        // Computation-communication loop
        modifyData(&dbl, dataArr);
        myCP.updateAndWrite(iteration, cpFreq);
    }
    return EXIT_SUCCESS;
}
```

Listing 1: The additional code (marked with red color) that should be inserted for adding an application-level checkpoint in a lengthy loop, using CRAFT (Adapted from [8]).

III. 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.

A. 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:

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

If a failure occurs, this only becomes known after δ time units, and 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

$$E\tau(P) = A + \delta + M, \tag{4}$$

since:

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

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

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

B. 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 $N \leftarrow N + 1$.

C. 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:

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

Therefore optimum checkpoint interval K^* is the value of K that minimizes $E_{cp}\tau(P)$, which can be computed numerically from (7).

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.

D. 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:

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

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 $I^* = \frac{K^*}{L}$.

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^*.

IV. Numerical Examples

In this section, a set of numerical examples illustrate the effect of the checkpoint interval K on the expected execution time of a software application. In Figure 1, the case of a software application with a relatively small loop having $M = 1000$ is presented. The upper part of Figure 1 compares the expected execution time of the application with and without the ALCR mechanism for different values of K, and the lower part shows the expected $Gain$ of Section III-C for different values of K. The values that correspond to the optimum checkpoint interval K^* are marked within a rectangle. Figure 1 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 2, 3 and 4. This emphasizes the importance of setting K to be close or at K^*.

The examples of Figures 1, 2, 3 and 4 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^*.

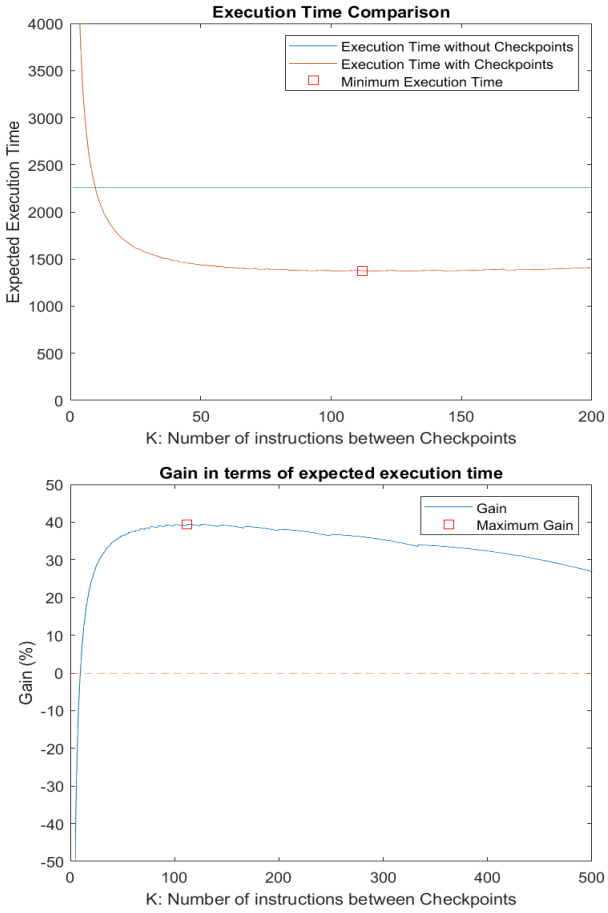

Fig. 1: Comparison of the expected execution time (above), and the Gain in the expected execution time (below), plotted versus the value of K, for a software application having a small computational loop with $M = 1000$.

V. Conclusions and Future Work

In this paper, we proposed a method for determining the checkpoint intervals in ALCR for software applications that contain long-running loops, and which run on platforms 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 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. Our results were illustrated via a set of numerical examples.

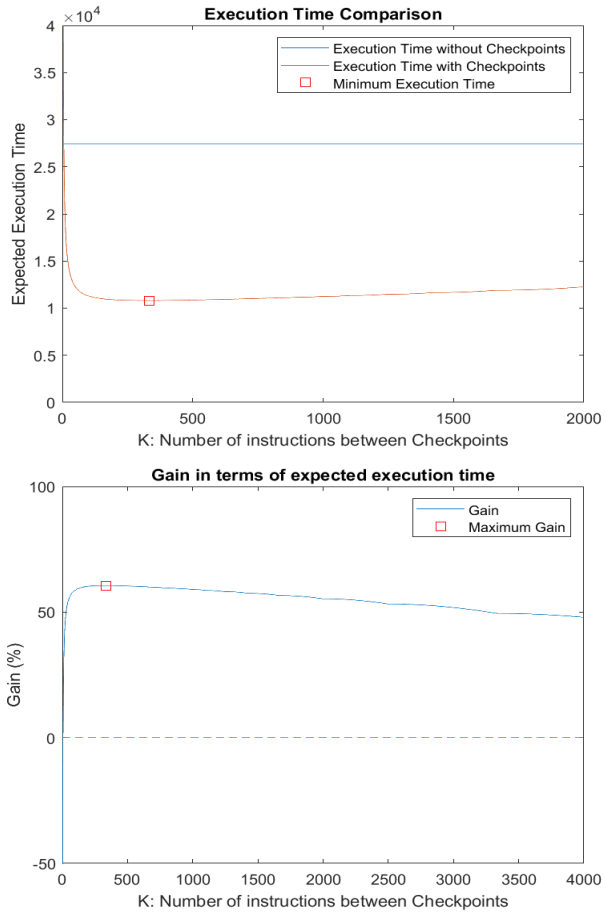

Fig. 2: Comparison of the expected execution time (above), and the Gain in the expected execution time (below), plotted versus the value of K, for a software application having a medium length computational loop with $M = 10^4$.

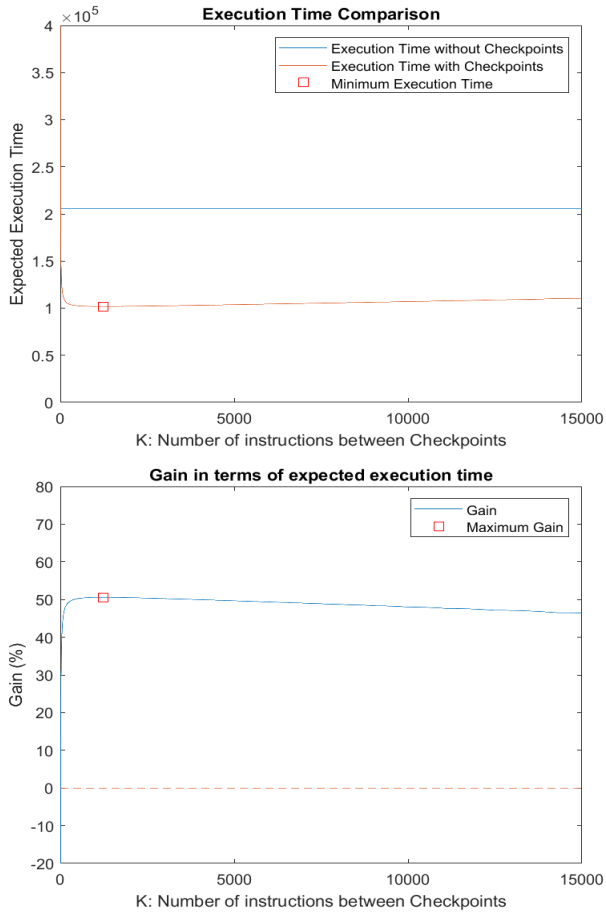

Fig. 3: Comparison of the expected execution time (above), and the Gain in the expected execution time (below), plotted versus the value of K, for a software application with a large computational loop having $M = 10^5$.

Several directions for future work can be considered. Firstly, 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. Hence, further research is needed to see how the checkpoint interval can be selected to achieve a compromise between energy consumption and execution times. Secondly, an interesting topic would be to consider the effect of the secondary 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. Finally, as far as security is concerned, ALCR can be used to disrupt attackers (e.g., [40]), while it can be also exploited by malicious individuals to increase the workload of the system, potentially 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.

ACKNOWLEDGEMENTS

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

REFERENCES

[1] "Summary of the Amazon s3 service disruption in the northern Virginia (us-east-1) region," February 2018.
[2] "Top 20 high profile cloud failures of all time," *TechFlier*.
[3] E. Gelenbe, P. Campegiani, T. Czachòrski, S. K. Katsikas, I. Komnios, L. Romano, and D. Tzovaras, "Security in computer and information sciences," in *First International ISCIS Security Workshop 2018, Euro-CYBERSEC 2018*, vol. 821, Springer, 2018.
[4] C. Wueest, M. B. Barcena, and L. O'Brien, "Mistakes in the IaaS cloud could put your data at risk," *Symantec*.
[5] E. Gelenbe, "Dealing with software viruses: a biological paradigm," *Information Security Technical Report*, vol. 12, no. 4, pp. 242–250, 2007.
[6] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.
[7] R. Arora, "ITALC : Interactive Tool for Application - Level Checkpointing," *Proceedings of the Fourth Internat'l.Workshop on HPC User Support Tools*, 2017.

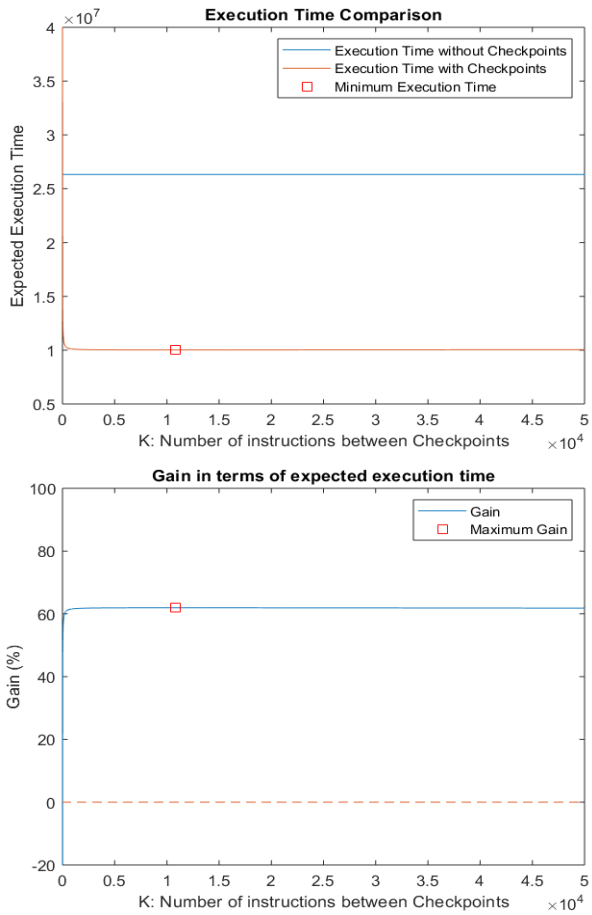

Fig. 4: Comparison of the expected execution time (above), and the Gain in the expected execution time (below), plotted versus the value of K for a software application with a very large computational loop having $M = 10^7$.

[8] F. Shahzad, J. Thies, and G. Wellein, "CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[9] J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[10] E. Gelenbe, "A Model of Roll-back Recovery with Multiple Checkpoints," in *Proceedings of the 2nd Internat'l.Conf. on Software Engineering*, ICSE '76, (Los Alamitos, CA, USA), pp. 251–255, IEEE Computer Society Press, 1976.

[11] E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems Under Intermittent Failures," *Commun. ACM*, vol. 21, no. 6, pp. 493–499, 1978.

[12] E. Gelenbe, "On the Optimum Checkpoint Interval," *Journal of the ACM*, vol. 26, no. 2, pp. 259–270, 1979.

[13] E. Gelenbe and M. Hernández, "Optimum Checkpoints With Age-Dependent Failures," *Acta Informatica*, vol. 531, pp. 519–531, 1990.

[14] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-recovery Protocols in Message-passing Systems," *ACM Comput. Surveys*, vol. 34, pp. 375–408, sep 2002.

[15] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "CheCL: Transparent checkpointing and process migration of OpenCL applications," *Proceedings - 25th IEEE Internat'l.Parallel and Distributed Processing Symposium, IPDPS 2011*, pp. 864–876, 2011.

[16] E. Gelenbe, "Model of information recovery using the method of multiple checkpoints (russian translation)," *Automation and Remote Control*, vol. 40, no. 4, pp. 598–605, 1979.

[17] E. Gelenbe and G. Pujolle, *Introduction aux Réseaux de Files d'Attente*. Editions Hommes et Techniques, Eyrolles, 1982.

[18] E. Gelenbe and I. Mitrani, *Analysis and synthesis of computer systems*. World Scientific, 2010.

[19] E. Gelenbe, D. Finkel, and S. K. Tripathi, "Availability of a distributed computer system with failures," *Acta Informatica*, vol. 23, p. 643, 1986.

[20] S. K. Tripathi, D. Finkel, and E. Gelenbe, "Load sharing in distributed systems with failures," *Acta Inf.*, vol. 25, no. 6, pp. 677–689, 1988.

[21] J. Walters and V. Chaudhary, "Application-Level Checkpointing Techniques for Parallel Programs," *Distributed Computing and Internet Technology*, pp. 221–234, 2006.

[22] N. Losada, M. J. Martín, G. Rodríguez, and P. Gonzalez, "Portable application-level checkpointing for hybrid MPI-OpenMP applications," *Procedia Computer Science*, vol. 80, pp. 19–29, 2016.

[23] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[24] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of multilevel checkpoint performance models," in *2018 IEEE Internat'l.Parallel and Distributed Processing Symposium Workshops*, pp. 783–792, 2018.

[25] E. Gelenbe and M. Hernández, "Enhanced availability of transaction oriented systems using failure tests," *Software Reliability Engineering, 1992. Proceedings., Third Internat'l.Symposium on*, pp. 342–350, 1992.

[26] E. Gelenbe and M. Hernández, "Virus tests to maximize availability of software systems," *Theoretical Computer Science*, vol. 125, no. 1, pp. 131–147, 1994.

[27] E. Dijkstra, J. Buxton, and B. Randell, "Software Engineering Techniques," *NATO Science Committee*, 1969.

[28] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds: Exploiting the intrinsic redundancy of web applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, pp. 16:1–16:42, May 2015.

[29] B. Randell, "System Structure for Software Fault Tolerance," *Science*, no. 2, pp. 1–18, 1975.

[30] P. Mell and T. Grance, "The NIST definition of cloud computing." URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf, 2011.

[31] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[32] M. Hasan and M. S. Goraya, "Fault tolerance in cloud computing environment: A systematic survey," *Computers in Industry*, 2018.

[33] L. Chen and 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)*, vol. 1, pp. 3–9, 1978.

[34] B. Nicolae and F. Cappello, "Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots," in *Proceedings of 2011 Internat'l.Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 34:1–34:12, ACM, 2011.

[35] D. Sun, G. Chang, C. Miao, and X. Wang, "Modelling and evaluating a high serviceability fault tolerance strategy in cloud computing environments," *Internat'l.Journal of Security and Networks*, vol. 7, no. 4, pp. 196–210, 2012.

[36] J. Cao, M. Simonin, G. Cooperman, and C. Morin, "Checkpointing as a service in heterogeneous cloud environments," in *2015 15th IEEE/ACM Internat'l.Symposium on Cluster, Cloud and Grid Computing*, pp. 61–70, May 2015.

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

[38] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," *IPDPS 2009 - Proceedings of the 2009 IEEE Internat'l.Parallel and Distributed Processing Symposium*, 2009.

[39] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *2010 ACM/IEEE Internat'l.Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, nov 2010.

[40] F. Kong, M. Xu, J. Weimer, O. Sokolsky, and I. Lee, "Cyber-physical system checkpointing and recovery," in *Proceedings of the 9th ACM/IEEE Internat'l.Conf. on Cyber-Physical Systems*, ICCPS '18, (Piscataway, NJ, USA), pp. 22–31, IEEE Press, 2018.