Optimum Checkpointing for Long-running Programs

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*
m.siavvas16@imperial.ac.uk, siavvasm@iti.gr

***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 propose a mathematical model to estimate the average execution time of a program that operates in the presence of dependability failures, without and with application level checkpointing, and use it to estimate the optimum interval in number of instructions executed between successive checkpoints. Specific emphasis is given on programs with loops, whereas the results are illustrated through simulation.**

***Keywords*—software reliability, roll-back recovery, application-level checkpoints, optimum checkpoints, program loops**

I. Introduction

Reliability is an important requirement for modern software applications, especially for long-running applications that have to run frequently and repeatedly. In such applications, a single failure may lead to the re-execution of a non-trivial number of operations, leading to significant performance overheads. This is more eminent in embedded software applications, which are often long-running, while they run on environments that are characterized by restricted resources (e.g., computational power). In such systems, failures may occur due to complex effects between various factors including system interactions, network connection, software and hardware defects, and security failures [1], [2]. Therefore, there is a need for mechanisms able to enhance the reliability of software applications, maintaining at same time their performance.

To address this issue, several fault tolerance mechanisms have been proposed over the years [3]–[6]. In the present paper, we specifically investigate the Application Level Checkpoint and Restart (ALCR), which is widely used to enhance the reliability of long-running programs [6]–[8] by periodically saving a checkpoint (i.e., a copy) of the current execution state of software. The most recent checkpoint 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]. Among the existing checkpointing strategies, ALCR [6], [19] is prefered since it uses a small memory footprint [7], [8], but it 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 judicious source code locations, which are normally long-running loops since computational loops constitute a significant source of failure-related re-executions [20], [21]. However, such tools do not provide a method to select the inter-checkpoint interval which has a significant impact 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 program after a failure, based on a mathematical model. We suggest that this approach can be implemented as an API within a broader platform (i.e., the SDK4ED platform presented in Section III), to help developers select the optimum checkpoint interval of program loops.

The rest of the paper is structured as follows. In Section II we review previous work. In Section III we give the overview of proposed method in the context of SDK4ED project. Section IV described mathematical model and numerical approach. The optimum checkpoint interval is discussed in Section IV-B. Section V presents numerical examples and Section VI presents conclusions and future research.

II. Related Work

In transaction-oriented systems, if no fault tolerance mechanism is adopted, all the successfully completed transactions will need to be re-processed 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). Moreover, in transaction-oriented systems, an "audit trail" is also kept, which contains the sequence of transactions that were executed since the most recent checkpoint. In case of a failure, only the transactions that were saved in the audit trail since the most recent checkpoint are re-executed [11]. Authors in [10], [16], [22] deal with the hierarchies failures by using multiple level checkpoints.

The system availability, which is defined as the fraction of time when the system is available for useful operations, is maximized when the optimum checkpoint interval between two successive checkpoints is selected [12]. A badly chosen checkpoint interval results in high system response times and long average execution times [23], [24]. Therefore, much research has focused on how system and failure rate parameters affect its value [9], [13]. Apart from transactions-oriented systems, in [25], [26] the impact of asynchronous checkpointing strategies on the performance of distributed systems has been studied.

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 [6], [14]. However, modern applications are considerably more complex than early transaction-oriented systems [3]. Therefore, a periodic copy of their overall execution state should be taken, in order to enhance their reliability [20].

Several rollback/recovery-based mechanisms for enhancing the reliability of long-running software applications exist, including: (i) Recovery Block schemes [3], (ii) N-Version Programming [4], [5], and (iii) the Checkpoint and Restart (CR) mechanism [6]. Despite their benefits, NVP and RB approaches are characterized by high development costs [30], which restricts their adoption to safety or reliability critical applications [31]. Despite attempts in addressing some drawbacks [28], [29], [31]–[33], their associated costs are still high.

CR mechanisms are generally preferred [6]–[8] since they introduce significantly less overhead compared to other counterparts [3], [4]. Mature CR tools and libraries exist for single-process software programs [34]–[36], 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 [14], [37], OpenCL [15], OpenMP [20], [21], and CUDA [38]. They include [7]: (i) system-level CR [35], (ii) library-level CR [39], and (iii) application-level CR (ALCR) [7]. ALCR [7], [19] is considered the most efficient, since it leaves the smallest memory footprint [7], [8], [20], though it requires source code modifications to insert application-level checkpoints into the program, leading to higher development effort.

In [21] 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 [20] an application-level checkpointing solution for hybrid MPI-OpenMP applications is suggested as an extension of CPPC. In [8] a library (CRAFT) was proposed for incorporating the application-level CR mechanism into software implemented in C++. Similarly to CPPC [21], the proposed library reduces the development time associated with the ALCR mechanism, by identifying lengthy loops and the automatic insertion of application-level checkpoints [40], [41]. In [7] 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. 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. The proposed method can be used along with the existing ALCR libraries, in order to enhance the performance of software applications.

III. SDK4ED Optimum Checkpoint Recommendation

The work presented in the present paper is part of the EU-funded H2020 SDK4ED[1] project (Fig. III). The purpose of the SDK4ED project is to develop a platform that will enable the production of software products optimized with respect to important quality attributes with specific emphasis on Maintainability[2], Energy Consumption, and Dependability. In order to achieve this, the platform will focus on the identification of trade-offs between the aforementioned quality attributes and to the recommendation of source code transformations for quality enhancement. The envisaged platform, through its recommendations, is expected to facilitate developers optimize their code by achieving a satisfactory compromise between these often conflicting quality factors.

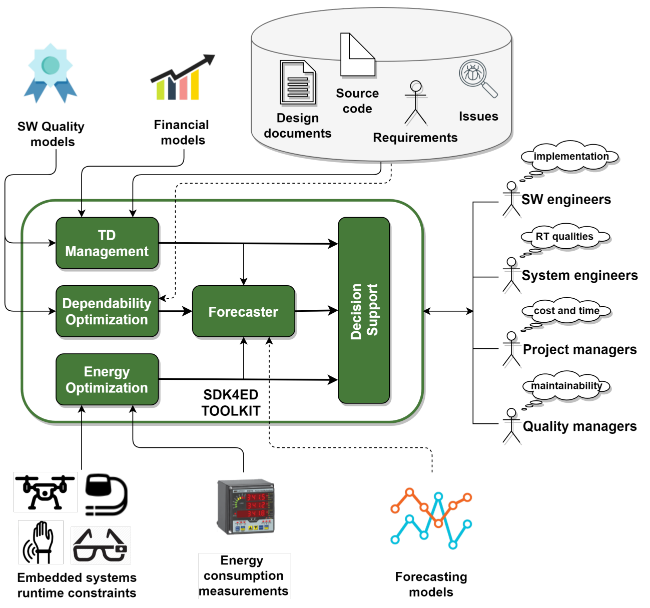

Fig. 1. The high-level overview of the envisaged SDK4ED Toolkit platform.

[1]https://sdk4ed.eu/

[2]It should be noted that Maintainability is quantified by the platform using the notion of Technical Debt (TD) [42], which is a widely-used measure of Software Quality and especially of Software Maintainability.

The purpose of the Dependability Optimization Toolbox of the SDK4ED platform (see Fig. III) is to provide recommendations that will help developers optimize the Dependability of software products. Emphasis is given on the attributes of Reliability and Security, which are fundamental facets of Dependability according to Avizenis [43]. With respect to Reliability, emphasis is given on enhancing the reliability of software applications using fault tolerance mechanisms, without affecting important runtime qualities like performance and energy consumption.

As already discussed, among the existing fault tolerance mechanisms, ALCR is the most efficient, but it requires significant development effort for the actual implementation of the checkpoints. It also requires expertise for the selection of the locations in which the checkpoints should be inserted and of their inter-checkpoint interval. To this end, the purpose of the Dependability Optimization Toolbox is to recommend: (i) the most suitable source code locations in which checkpoints should be inserted, and (ii) the optimum value of the inter-checkpoint interval that achieves a sufficient compromise between important quality attributes like performance and energy consumption. The work presented in the present paper contributes towards the latter, since it proposes a mathematical model that determines the optimum checkpoint interval, i.e., the interval between two successive application-level checkpoints that minimizes the average execution time of an application.

IV. 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:

$$E\tau(P) = \frac{A+\delta}{(1-g)^M} + c.\frac{1-(1-g)^M}{g(1-g)^M}. \tag{2}$$

If a failure occurs at instruction u, 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 (2) that

$$E\tau(P) = A + \delta + M, \tag{3}$$

since:

$$\lim_{g\to 0} \frac{1-(1-g)^M}{g(1-g)^M} = M. \tag{4}$$

When g is very small so that $gM << 1$, we can use the following approximation directly from (2):

$$E\tau(P) \approx \frac{A+\delta+c.M}{1 \ - \ g.M}. \tag{5}$$

B. Optimum Checkpoints

When the program must run for a long time, i.e. M is large, and the probability of failure g 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:

$$\begin{aligned} E_{cp}\tau(P) = & \frac{A+\delta}{(1-g)^K} + \frac{[b(M,K)-2][B(K)+\delta]}{(1-g)^K} \\ & + c\frac{[b(M,K)-1][1-(1-g)^K]}{g(1-g)^K} \\ & + \frac{B(K)+\delta}{(1-g)^{K_o}} + c\frac{1-(1-g)^{K_o}}{g(1-g)^{K_o}}. \end{aligned} \tag{6}$$

Therefore optimum checkpoint interval K^* is the value of K that minimizes $E_{cp}\tau(P)$, which can be computed numerically from (6).

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

$$Gain = \frac{E\tau(P) - E_{cp}\tau(P)}{E\tau(P)} \times 100, \tag{7}$$

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

C. 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 (8) with $M = n.L$ and $K = I.L$ we have:

$$E_{cp}\tau(P) = \frac{[b(M,K)-1][B(K)+\delta]}{(1-g)^K} \quad (8)$$
$$+c\frac{[b(M,K)-1][1-(1-g)^K]}{g(1-g)^K} + \frac{B(K)+\delta}{(1-g)^{K_o}}$$
$$+c\frac{1-(1-g)^{K_o}}{g(1-g)^{K_o}}.$$

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^*.

V. Numerical Example

In this section, a numerical example is used to illustrate the effect of the checkpoint interval K on the expected execution time of a software application. In Figure 2, the case of a software application with a loop having $M = 1000$ is presented. This value was selected for demonstration purposes since it led to more intuitive results. For the failure rate, for the purposes of the present experiment we considered $g = 10^{-3}$, which is a relatively high failure probability. The upper part of Figure 2 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 IV-B for different values of K. The values that correspond to the optimum checkpoint interval K^* are marked within a rectangle. Figure 2 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, suboptimal values of the checkpoint interval K may lead to a suboptimal reduction in the expected execution time of the application, whereas there are also values of K that lead to an execution time higher than the expected execution time of the same application that does not adopt checkpointing. This emphasizes the importance of setting K to be close or at K^*.

The example of Figure 2 suggests 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 this example, the Gain is approximately 40%. 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

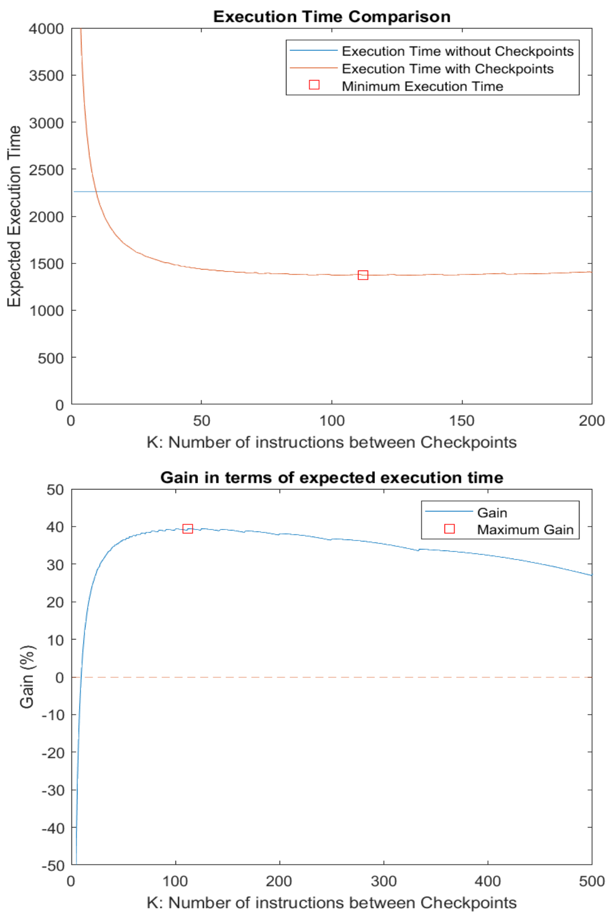

Fig. 2. Comparison of the expected execution time (above), and the Gain in the expected execution time (below), plotted versus the value of K, for a software application having a computational loop.

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^*.

VI. 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, i.e., the interval between two successive checkpoints that minimizes the expected execution time of the program, depends on various parameters which can be incorporated into a single numerical expression. These parameters include: (i) the program failure rate, (ii) the execution cost for establishing a checkpoint, and (iii) the execution cost for restarting a program after a failure. The produced 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.

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 [44], 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.

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.

The majority of the above mentioned points will be examined in the next steps of the SDK4ED project. More specifically, emphasis will be given on the applicability and the actual implementation of the theoretical approach that is presented in this paper. The platform will potentially manage to identify long loops that require checkpointing and to recommend the optimum inter-checkpoint interval that optimizes important quality attributes like performance (i.e., execution time) and energy consumption.

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] E. Gelenbe, "Dealing with software viruses: a biological paradigm," *Information Security Technical Report*, vol. 12, no. 4, pp. 242–250, 2007.

[2] 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.

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

[4] 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.

[5] A. Avižienis, "Fault-tolerance and Fault-intolerance: Complementary Approaches to Reliable Computing," *SIGPLAN Not.*, vol. 10, no. 6, pp. 458–464, 1975.

[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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, "Automatic recovery from runtime failures," *Proceedings - Internat'l.Conf. on Software Engineering*, pp. 782–791, 2013.

[29] 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.

[30] E. Gelenbe and I. Mitrani, "Modelling the behaviour of block structured processes with hardware and software failures," in *Mathematical Computer Performance and Reliability* (e. a. Giuseppe Iazeolla, ed.), pp. 329–339, Elsevier Science Publishers, 1984.

[31] A. Armoush, F. Salewski, and S. Kowalewski, "A hybrid fault tolerance method for recovery block with a weak acceptance test," *Proceedings of The 5th Internat'l.Conf. on Embedded and Ubiquitous Computing, EUC 2008*, vol. 1, pp. 484–491, 2008.

[32] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, “Automatic Workarounds for Web Applications,” in *Proceedings of the Eighteenth ACM SIGSOFT Internat'l.Symposium on Foundations of Software Engineering*, FSE '10, (New York, NY, USA), pp. 237–246, ACM, 2010.

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

[34] J. S. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent Checkpointing Under Unix,” in *Proceedings of the USENIX 1995 Technical Conf. Proceedings*, TCON'95, (Berkeley, CA, USA), p. 18, USENIX Association, 1995.

[35] 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.

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

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

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

[39] 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.

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

[41] 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.

[42] W. Cunningham, “The wycash portfolio management system,” *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[43] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.

[44] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, “Optimizing checkpoint intervals for reduced energy use in exascale systems,” in *Eighth Internat'l.Green and Sustainable Computing Conference, IGSC 2017, Orlando, FL, USA, October 23-25, 2017*, pp. 1–8, 2017.