

Article

# Minimizing Energy and Computation in Long-Running Software

Erol Gelenbe <sup>1,\*</sup>  and Miltiadis Siavvas <sup>2</sup> 

<sup>1</sup> IITIS (Institute of Theoretical and Applied Informatics), Polish Academy of Sciences, ul. Baltycka 5, 44100 Gliwice, Poland

<sup>2</sup> Information Technologies Institute, Centre for Research and Technology Hellas, 6th km Harilaou-Thermi, 57001 Thessaloniki, Greece; siavvasm@iti.gr

\* Correspondence: gelenbe.erol@orange.fr

**Abstract:** Long-running software may operate on hardware platforms with limited energy resources such as batteries or photovoltaic, or on high-performance platforms that consume a large amount of energy. Since such systems may be subject to hardware failures, checkpointing is often used to assure the reliability of the application. Since checkpointing introduces additional computation time and energy consumption, we study how checkpoint intervals need to be selected so as to minimize a cost function that includes the execution time and the energy. Expressions for both the program's energy consumption and execution time are derived as a function of the failure probability per instruction. A first principle based analysis yields the checkpoint interval that minimizes a linear combination of the average energy consumption and execution time of the program, in terms of the classical "Lambert function". The sensitivity of the checkpoint to the importance attributed to energy consumption is also derived. The results are illustrated with numerical examples regarding programs of various lengths and showing the relation between the checkpoint interval that minimizes energy consumption and execution time, and the one that minimizes a weighted sum of the two. In addition, our results are applied to a popular software benchmark, and posted on a publicly accessible web site, together with the optimization software that we have developed.

**Keywords:** optimum energy savings; minimum execution time; checkpoint interval; analytical models



**Citation:** Gelenbe, E.; Siavvas, M. Minimizing Energy and Computation in Long-Running Software. *Appl. Sci.* **2021**, *11*, 1169. <https://doi.org/10.3390/app11031169>

Academic Editor: Roberto Rojas-Cessa

Received: 25 November 2020

Accepted: 21 January 2021

Published: 27 January 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Long-running programs include database systems, operating systems, and platforms that support sensor systems. Such software needs to be very reliable, but should also be efficient in execution time and energy consumption. Thus its reliability [1] is often assured via checkpoints, to avoid that each failure leads to excessive overhead in execution time [2–4] and energy consumption [5].

Indeed, among the mechanisms that restore or preserve system consistency after failures [6], Checkpointing and Recovery (CR) is used widely to periodically save an up-to-date copy of system or program state that is used to restart execution if a failure occurs. CR can also be found in high performance systems [7–10], operating systems including Linux [11–13], databases [14], and distributed systems [15–18].

Thus checkpoint intervals have been widely studied to maximize system availability and minimize program execution time for transaction-oriented systems [19–21], and imbedded multiple level checkpoints introduced in [22,23] were recently studied in [24]. CR [6,25] includes "Application-level Checkpoint and Restart" (ALCR) [26,27], that uses smaller memory space but requires significant programming skills to insert checkpoints in long-running loops [28,29]. Since longer inter-checkpoint intervals increase the required time and energy of system restart, and short intervals increase them due to frequent check-

points, the checkpoint interval should be chosen to minimize both energy consumption and execution time [30,31].

In recent years the importance of energy savings in information technology and software has been often emphasized [32–35], and research has addressed the efficient allocation of energy in computer systems [36–38] including the use of server or network node vacations to reduce energy consumption [39], techniques to select Cloud servers based on energy efficiency [40], and the use of renewable energy sources [41–44]. There has been less work on more detailed techniques such as checkpoints to reduce energy consumption [45,46], or on checkpoint optimization in modern software using ALCR [47,48]. In addition, commonly used tools such as ALCR do not offer assistance in selecting checkpoint intervals to optimize energy consumption or execution time, and a software tool was proposed recently to address this issue [49].

Thus in this paper we focus on analyzing the checkpoint intervals in a unified manner to effect savings in a weighted combination of execution time and energy, since energy consumption is of importance both in autonomously operating platforms and for software running in large scale Cloud data centers [50]. In the sequel, starting from first principles, we develop a mathematical model to estimate the average execution time as well as the energy consumption of a program with long loops that operates in the presence of failures, without and with ALCR. This allows us to compute the checkpoint interval that minimizes the program's energy consumption and average execution time, and its value that can minimize a cost function that is a weighted sum of both elements, expressed via the Lambert Function, with numerical examples that illustrate the results. In addition, we also apply these results to a well known software benchmark.

The rest of the paper is structured as follows. In Section 2, the mathematical model that estimates the average execution time and energy consumption of a software program that operates in the presence of failures with and without checkpoints is presented. In Section 3, based on this mathematical model, the closed-form expression of the optimum checkpoint interval is derived. In Section 4, we illustrate our results through a set of numerical examples. In addition, Section 5 is devoted to show how our model can be used to select checkpoints for the popular Rodinia Benchmark of real-world open-source software written in C and C++ programming languages, which is widely used for software performance evaluation and energy optimization, and in particular the *streamcluster* (<https://github.com/yuhc/gpu-rodinia/tree/master/openssl/streamcluster>) program. Finally, Section 6 concludes the paper and discusses directions for future work.

## 2. A Single Loop Program with Checkpoints

Consider a program  $P$  that executes  $y_n$  instructions between its  $(n - 1)$ -th and  $n$ -th checkpoint, without counting all possible failures and failure recoveries. Now consider the instant  $t_n > 0$  when the program creates its  $n$ -th checkpoint, and let  $Y_n$  denote the total number of instructions that the program has executed by time  $t_n$  since it started, where  $Y_n$  does not include all the repeated instructions that were executed due to checkpoints and failure recovery, and obviously:  $Y_n = \sum_{i=1}^n y_i$ .

Let  $B^c(Y_n)$  be the computation time needed to create the  $n$ -th checkpoint. This quantity will generally depend on the total memory space occupied by the program, but in certain cases it may depend on  $Y_n$ , since the program may generate new data as it is executing. Hence we will write  $B^c(Y_n) = B_0^c + B_1^c Y_n$  where  $B_0^c > 0$  and  $B_1^c \geq 0$  are constants for the given program.

On the other hand, suppose a failure occurs after the program has successfully executed  $y$  instructions after the  $n$ -th checkpoint, i.e., after the program has executed  $Y_n + y$  instructions. If  $b^c(Y_n, y)$  is the computation time needed to restart the program from the most recent checkpoint, when the program has successfully executed  $y \leq Y_{n+1} - Y_n$  instructions after the most recent checkpoint but before the  $(n + 1)$  checkpoint, then we will have:

$$b^c(Y_n, y) = b_0^c + b_1^c y, \text{ where } b_0^c > 0, \text{ and } b_1^c \geq 0 \text{ are constants.} \quad (1)$$

Therefore the time duration  $b^c(Y_n, y)$  depends on the number  $y$  of instructions that have been executed by the program since the last checkpoint was established. In summary, we are assuming that:

- The time  $B^c(Y_n)$  needed to establish the  $n$ -th checkpoint depends on the “age of the program” or the total number of instructions  $Y_n$  it has executed since the beginning, i.e.,  $B^c(Y_n) = B_0^c + B_1^c(Y_n)$ ,
- The time  $b^c(Y_n, y)$  needed to recover from a failure after the  $n$ -th checkpoint, including the time related to re-loading system state after the failure, only depends on  $y \leq Y_{n+1} - Y_n$ , the “computation time undertaken by the program since the last checkpoint”, i.e.,  $b^c(Y_n, y) \equiv b^c(y) = b_0^c + b_1^c y$ .

Similarly, we denote the energy consumption for creating the  $n$ -th checkpoint to be  $B^e(Y_n)$ , and  $b^e(y)$  is the energy used to recover from a failure after a failure that occurs when the total number of instructions executed is  $Y_n + y \leq Y_{n+1}$ . Also, we will have  $B^e(Y_n) = B_0^e + B_1^e Y_n$ , and  $b^e(y) = b_0^e + b_1^e y$  with  $B_0^e > 0$ ,  $B_1^e \geq 0$  and  $b_0^e > 0$ ,  $b_1^e \geq 0$ .

Let  $\alpha, \beta > 0$  be positive constants that represent the relative costs of computation time and energy consumption. We can then define the parameters:

$$\begin{aligned} B_0 &= \alpha B_0^c + \beta B_0^e, \\ B_1(Y_n) &= \alpha B_1^c(Y_n) + \beta B_1^e(Y_n), \\ b_0 &= \alpha b_0^c + \beta b_0^e, \\ b_1 &= \alpha b_1^c + \beta b_1^e, \end{aligned}$$

and the total cost of an instruction can be viewed as the weighted sum of its execution time and of its energy consumption  $c = \alpha K^c + \beta K^e$ .

### 2.1. Fixed Checkpoint Intervals

Earlier work has shown that “age dependent” checkpoints [51] can reduce the overall cost of checkpointing and failure recovery, when (for instance) the failure rate of a system increases with time. However, most practical checkpointing schemes use a simpler approach where checkpoints are carried out periodically each time the program has executed successfully a predetermined fixed number of instructions  $y_n = y$ . Thus, in the sequel we will make this assumption so that checkpoints are placed after  $Y_1 = y$ ,  $Y_2 = 2y$ , ..  $Y_n = ny$ , etc. instructions have been successfully executed, and we will proceed to compute the optimum value of  $y$ , assuming that  $n$  is fixed in advance.

When the program ends after  $Y = Ny$  instructions are executed, a further  $(N + 1)$ -th checkpoint is not needed, while the first checkpoint is obviously installed before the first instruction is executed.

We can then formulate our problem as that of a program that executes a total fixed number of instructions  $Y$ , where we want to choose the constant value  $y$  of the number of instructions between checkpoints, or equivalently we can choose  $N$ , the number of checkpoints so that  $Y = Ny$  so that the total overhead in additional work and energy consumption due to failures and due to checkpoints is minimized.

For a given  $y$ , let us compute  $K^c(y)$ , which is the corresponding total expected execution time including all restarts due to failures, starting from the most recent checkpoint. When the average execution time per instruction is  $c$ , and the failure probability per instruction is  $(1 - a)$ , the total average time elapsed time for the execution of  $y$  instructions is:

$$\begin{aligned} K^c(y) &= cy a^y + (b_0 + K^c(y))(1 - a^y), \\ &+ (K^c + b_1^c) \sum_{x=1}^y x a^{x-1} (1 - a), \end{aligned} \tag{2}$$

because with probability  $a^y$  a failure does not occur during the  $y$  instructions, leading to an execution time of  $K^c y$  time units, while with probability  $(1 - a)^y$  at least one failure does occur among the  $y$  instructions, and the first of those requires a program re-start time of  $b_0^c$ , to which we should add  $K^c(y)$  representing the effect of all future failures after the program has been re-initialised from the checkpoint.

Also, we have to include the execution time plus the amount of additional work needed per executed instruction, until the failure occurs—hence the term  $(K^c + b_1^c)$ —multiplied by  $x$  and the probability that the failure occurs at instruction  $x$  which is  $a^{x-1} \cdot a$ , summed over  $x$  running from 1 to  $y$ . Since

$$\sum_{x=0}^y a^x = \frac{1 - a^{y+1}}{1 - a},$$

$$\text{and } \frac{d}{da} \frac{1 - a^{y+1}}{1 - a} = \frac{1 - ya^y(1 - a) - a^y}{(1 - a)^2},$$

we obtain:

$$K^c(y) = b_0^c[a^{-y} - 1] + \frac{K^c + b_1^c}{1 - a}[a^{-y} - 1] - b_1^c y. \tag{3}$$

the total expected energy consumption  $K^e(y)$  for a number of instructions  $y$  after the most recent checkpoint, we similarly obtain the quantity:

$$K^e(y) = b_0^e[a^{-y} - 1] + \frac{K^e + b_1^e}{1 - a}[a^{-y} - 1] - b_1^e y, \tag{4}$$

where  $K^e$  denotes the average energy consumption per instruction, so that

$$\begin{aligned} C(y) &= \alpha K^c(y) + \beta K^e(y), \\ &= b_0[a^{-y} - 1] + \frac{c + b_1}{1 - a}[a^{-y} - 1] - b_1 y. \end{aligned} \tag{5}$$

Interestingly enough, we can show using l'Hôpital's Rule, for all  $y \geq 1$ , that:

$$\lim_{a \rightarrow 1} C(y) = c^y, \tag{6}$$

as would be expected.

Treating  $y$  as if it were a real number, we can compute the derivative of  $C(y)$ . We first note that for a differentiable function  $f(y)$  of the real variable  $y$ , we can write:

$$\frac{df}{dy} = f \cdot \frac{d \ln f}{dy}, \text{ hence } \frac{d}{dy} a^{-y} = -a^{-y} \cdot \ln a, \tag{7}$$

and therefore

$$\frac{dC(y)}{dy} = -\ln a \cdot \left[ \frac{b_0}{a^y} + \frac{c + b_1}{a^y(1 - a)} \right] - b_1. \tag{8}$$

Because  $a \leq 1$ , the quantity  $-\ln a \geq 0$ , and since  $y$  is large,  $\frac{1}{a^y}$  is very large and  $\frac{dC(y)}{dy} > 0$ .

### 3. Minimizing Computation Time and Energy

When we include both the time and energy needed to create each checkpoint, and assuming a fixed number of instructions  $y$  executed between successive checkpoints, we

can obtain the total cost of the program up to and including the last instruction executed at  $Y = yN$  as:

$$G_N(y) = NB_0 + \sum_{i=1}^N iyB_1 + NC(y), \quad (9)$$

$$= NB_0 + C(y) + \frac{N(N+1)}{2}yB_1. \quad (10)$$

The optimum checkpoint interval  $y^*$  is then the value of  $y$  that minimizes  $\kappa_N(y)$ , the overall cost per unit work that is accomplished, i.e.,  $G_N(y)$  divided by  $Y = Ny$  which is the total number of useful instructions executed over this time:

$$\begin{aligned} \kappa_N(y) &\equiv \frac{G_N(y)}{Ny} \\ &= \frac{B_0 + C(y)}{y} + \left(\frac{Y}{y} + 1\right) \frac{B_1}{2} \\ &= \frac{B_0 + \frac{B_1Y}{2} + C(y)}{y} + \frac{B_1}{2}. \end{aligned} \quad (11)$$

Therefore, to seek the optimum value of  $y$ , we compute the following derivative and set it to zero:

$$\frac{d\kappa_N}{dy} = \frac{y \frac{dC(y)}{dy} - (B_0 + \frac{B_1Y}{2} + C(y))}{y^2}, \quad (12)$$

so that the optimum value of  $y$  is:

$$\begin{aligned} y^* &= \frac{B_0 + \frac{B_1Y}{2} + C(y^*)}{\frac{dC(y)}{dy} \Big|_{y=y^*}} = \\ &= \frac{B_0 + \frac{B_1Y}{2} + b_0[a^{-y^*} - 1] + \frac{c+b_1}{1-a}[a^{-y^*} - 1] - b_1y^*}{-\ln a \cdot [\frac{b_0}{ay^*} + \frac{c+b_1}{ay^*(1-a)}] - b_1}, \\ \text{or } -\frac{y^* \ln a + 1}{ay^*} &= \frac{B_0 + \frac{B_1Y}{2}}{b_0 + \frac{c+b_1}{1-a}} - 1. \end{aligned} \quad (13)$$

Defining  $B = B_0 + \frac{B_1Y}{2}$  and

$$A = b_0 + \frac{c+b_1}{1-a}, \quad (14)$$

we have:

$$\begin{aligned} -\frac{y^* \ln a + 1}{ay^*} &= \frac{B_0 + \frac{B_1Y}{2}}{b_0 + \frac{c+b_1}{1-a}} - 1, \\ \text{or } \ln(a^{-y^*} \cdot e^{-1})[e^{-1}a^{-y^*}] &= \\ -(y^* \ln a + 1)e^{-(y^* \ln a + 1)} &= \frac{B - A}{e \cdot A}, \end{aligned} \quad (15)$$

To verify that  $y^*$  is the minimum value, we compute:

$$\frac{d^2\kappa_N(y)}{dy^2} = \frac{y^3C''(y) - 2y(yC'(y) - B - C(y))}{y^4}, \quad (16)$$

where  $C'(y)$ ,  $C''(y)$  denote the first and second derivatives of  $C(y)$  with respect to  $y$ , and  $B = B_0 + B_1Y$ . Since at  $y^*$  we have  $y^*C'(y^*) = B + C(y^*)$ , we can write:

$$\frac{d^2\kappa_N(y)}{dy^2}\Big|_{y=y^*} = \frac{C''(y)}{y}\Big|_{y=y^*}, \quad (17)$$

and we need to examine the sign of  $C''(y^*)$ . Starting from (8) we have:

$$C''(y) = a^{-y}(\ln a)^2\left[b_0 + \frac{c + b_1}{1 - a}\right] - a^{-y} \ln a \frac{c + b_1}{(1 - a)^2}, \quad (18)$$

which is positive, so that  $y^*$  is indeed the value of  $y$  at the minimum.

### 3.1. The Optimum Checkpoint Using the Lambert Function

Let us first recall the definition of the *Lambert Function*  $W(z)$  [52–55]. Consider any two numbers  $z$ ,  $w$ , which have the following relation:

$$z = w \exp w; \iff w = W(z). \quad (19)$$

Thus if we can write  $z = we^w$ , then  $w = W(z)$ , and similarly if  $w = W(z)$ , then  $z = we^w$ .

Applying (19) to Equation (15), we can write the expression for  $y^*$  as:

$$y^* = -\frac{1}{\ln a} \left[ W\left(\frac{B - A}{e \cdot A}\right) + 1 \right], \quad (20)$$

which provides an explicit solution for the value of the optimum checkpoint interval  $y^*$ . Clearly, if we set  $\alpha = 1$  and  $\beta = 0$ , we obtain the optimum checkpoint that simply minimizes the overall execution time, without consideration for the energy consumption.

Also, if in the system under consideration the creation of a checkpoint does not depend on the amount of successful computation that the program has accomplished until the time of the checkpoint, then we simply set  $B_1^c = B_1^e = 0$  in the expression for  $B$ , so that  $B = B_0$  which is the case that is usually discussed in the literature.

### 3.2. Sensitivity of the Optimum to Energy Consumption and Computation Time

An important question concerns how  $y^*$  varies with changes in the relative importance of the energy expenditure with respect to computation time. To address this issue as a single parameter problem, we will set  $\alpha = 1$ , and consider the derivative of  $y^*$  with respect to  $\beta$ . Noting that we can now write  $B = B^c + \beta B^e$  and  $A = A^c + \beta A^e$ , we have:

$$\begin{aligned} \frac{\partial y^*}{\partial \beta} &= -\frac{1}{\ln a} W'\left(\frac{B - A}{e \cdot A}\right) \cdot \frac{B^e A^c - A^e B^c}{(eA)^2}, \\ &= -\frac{1}{\ln a} \frac{W\left(\frac{B - A}{e \cdot A}\right) (B^e A^c - A^e B^c)}{\frac{B - A}{e \cdot A} (1 + W\left(\frac{B - A}{e \cdot A}\right)) ((eA)^2)}, \\ &= -\frac{(y^* \ln a + 1) (B^e A^c - A^e B^c)}{y^* (\ln a)^2 (B - A) e A}, \end{aligned} \quad (21)$$

where we have used the identity:

$$\frac{dW(x)}{dx} = \frac{W(x)}{x(1 + W(x))}, \quad (22)$$

when  $x \neq 0$  and  $x \neq -\frac{1}{e}$ . These two conditions will be satisfied because it is unlikely in practice that the system parameters be such that  $B = A$ , furthermore it is impossible that  $B - A = -A$  because  $B > 0$ .

Thus we can use the expression (21) to determine how fast  $y^*$  will vary as a function of  $\beta$ . In particular we have the following very interesting result.

**Result:** When  $B^e A^c = A^e B^c$ , then  $y^*$  does not depend on the relative weight of the execution time and energy consumption, so that a single value of  $y^*$  will minimize the overall cost for  $\alpha = 1$  and any value of  $\beta$  that represents the relative importance of energy consumption to computation time.

#### 4. A Program with a Single Long Loop

In this section, we will apply the previous results to a program with a single long loop of length  $L$  instructions which is executed some number, say  $T$  times, so that  $Y = LT$ . For this program, we may be constrained to place checkpoints either at the start of a loop so that  $y = m.L$  with one checkpoint for each  $m > 1$  loops, or  $n$  checkpoints may be placed within the loop with  $L = ny$  where  $n > 1$ , or we set  $n = 1$ . We first apply the previous results to compute  $y^*$ :

$$y^* = -\frac{1}{\ln a} [W(\frac{B-A}{e.A}) + 1], \tag{23}$$

where:

$$B = B_0 + B_1 L.T, A = b_0 + \frac{c + b_1}{1 - a}, \tag{24}$$

so that

$$y^* = -\frac{1}{\ln a} [W(\frac{(1-a)(B_0 + B_1 L.T)}{e[b_0(1-a) + c + b_1]} - \frac{1}{e}) + 1]. \tag{25}$$

Let us denote by  $I(x)$  the integer that is closest to the real number  $x$ . Then we compute  $r = \frac{L}{y^*}$ , and:

- If  $r \geq 1$  we set  $n = I(r)$ ,
- If  $r < 1$ , we set  $n = I(\frac{1}{r})$ .

To illustrate these results, numerical examples are provided in order to show the effect of the checkpoint interval  $n$  (expressed in terms of the number of loop repetitions between checkpoints) on the expected execution time and the total energy consumption of a software application that operates in the presence of failures. In order to differentiate the effect of computation time and energy consumption, we use  $n^o$  to represent the checkpoint interval that minimizes the total computation time, while  $n^+$  refers to the optimum checkpoint interval that minimizes the total energy consumption. Note that in the preceding analysis,  $n^o$  can be obtained by setting  $\alpha = 1, \beta = 0$ , while  $n^+$  is obtained by setting  $\alpha = 0, \beta = 1$ .

These examples consider the case of a program with a single loop in which checkpoints are established at the beginning (or at the end) of the loop. We consider a small, medium, large, and very large program, comprised of  $Y = 10^4, 10^5, 10^6, 10^7$  instructions, respectively. The expected execution time of the same program with and without the adoption of the ALCR mechanism is calculated and the corresponding optimization problem is shown numerically. The parameter values that we use are:

$$\begin{aligned} B_0^e &= 500, B_1^e = 0, B_0^c = 10^5, B_1^c = 0, K^c = 1 \\ K^e &= 10^{-5}, b_0^e = 100, b_1^e = 10, b_0^c = 100, b_1^c = 10 \\ g &= 5 \times 10^{-6}, L = 100. \end{aligned}$$

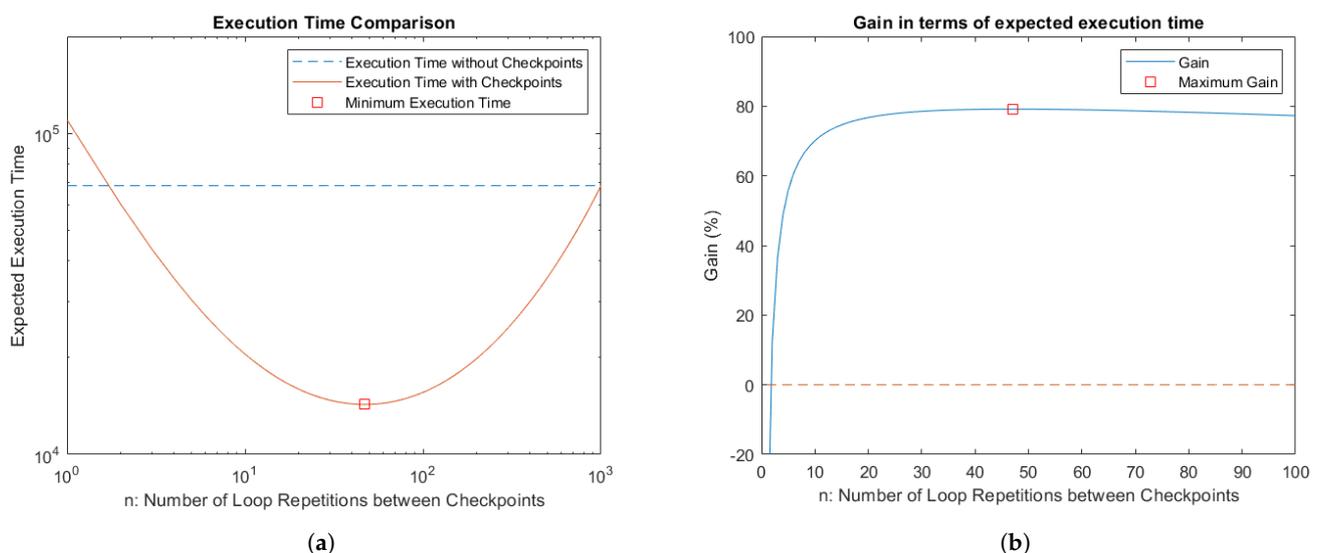
In Figure 1, the example of a small software program (i.e.,  $Y = 10^4$ ) is considered. Figure 1a compares the expected execution time of the application with and without the ALCR mechanism for different values of  $n$ , while Figure 1b shows the expected Gain in terms of expected execution time for different values of  $n$ . The values that correspond to the optimum checkpoint interval  $n^o$  are marked within a rectangle.

Figure 1 illustrates the fact that the optimum checkpoint interval  $n^o$  minimizes the overall execution time of the application and maximizes the overall expected Gain. From

Figure 1 it is clear that 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  $n$ , the expected execution time of the application with checkpointing is higher than the expected execution time of the same application without checkpoints. For instance in this example, choosing a very small checkpoint interval (i.e., below 5) will actually lead to an increase in the expected execution time of the software program, compared to the execution time of the same program when the checkpointing mechanism is not adopted. This suggests that frequent checkpointing which enhances the reliability of the software program, may result in increases of execution time due to the cost of checkpointing itself.

Similar observations can be made for software with longer loops in Figures 2–4. This emphasizes the importance of setting  $n$  to be close or at  $n^0$ , when there is a need for minimizing the execution time of the program.

The examples of Figures 1–4 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  $n^0$ . In these examples, the Gain ranges from 64% to 80%. 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  $n^0$ .

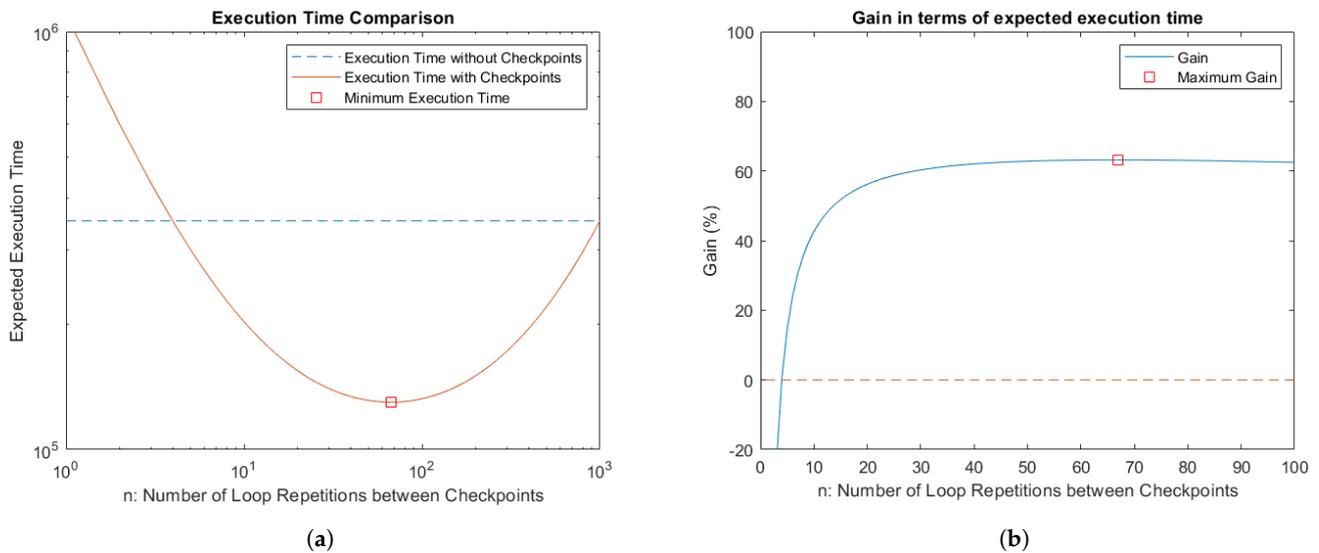


**Figure 1.** The case of a small software program (i.e.,  $\gamma = 10^4$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected execution time gain.

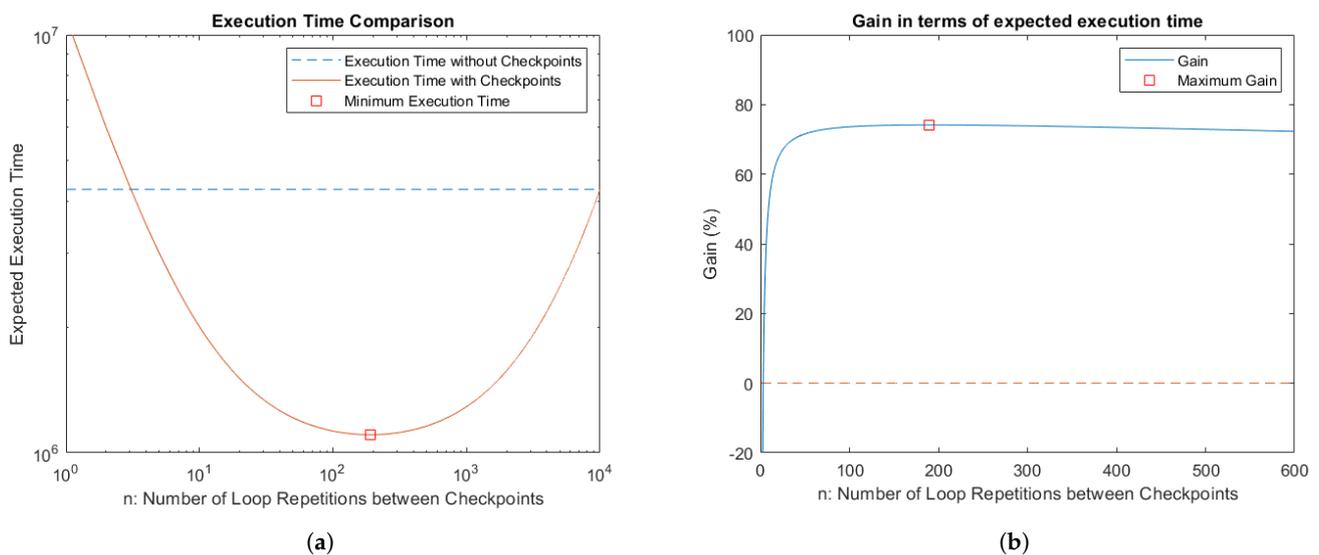
Still, there is a relationship between calculations for  $n^0$  and  $n^+$ . However, we must have in mind that the optimum checkpoint interval will be different regarding energy consumption and execution time. Figure 5 shows how they correspond to each other. More specifically, Figure 5a shows how execution time changes when we want to use optimal checkpoint interval calculated for energy consumption. Similarly, Figure 5b shows how energy consumption changes when we want to use the checkpoint interval that optimizes execution time.

The numerical example presented in Figure 5 shows that the checkpoint interval that minimizes the energy consumption does not necessarily minimize the execution time as well and vice versa. In particular, in the given example, setting the value of  $n$  to  $n^0$  will minimize the expected execution time of the software program, but will lead to around half the maximum achievable energy savings. Similarly, setting the value of  $n$  to  $n^+$  will minimize the expected energy consumption of the software program, but will lead to lower than the maximum achievable savings in execution time. Hence, the type of the application

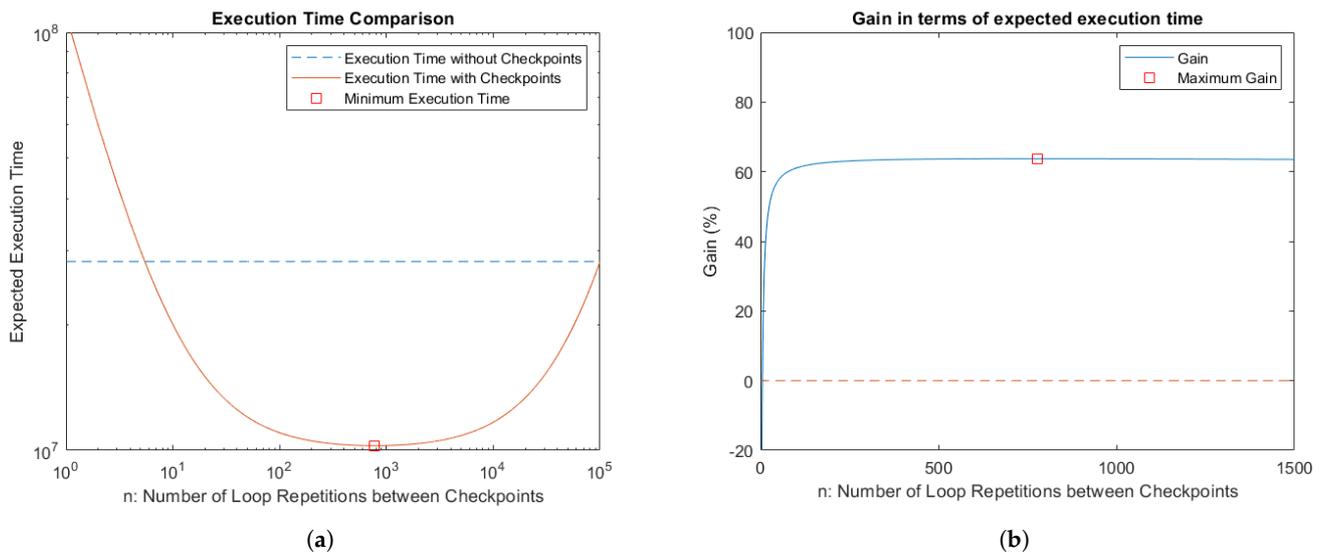
should be also taken into account in order to decide, whether to prioritize the execution time or the energy consumption of a given program. It should be noted that the model is highly configurable, which means that the user can define the relative importance of the quality attributes of execution time and energy consumption for a given software program, by properly setting the  $\alpha$  and  $\beta$  parameters of the model (see Section 3). This enables the calculation of the checkpoint interval that strikes a desired balance between these two quality attributes.



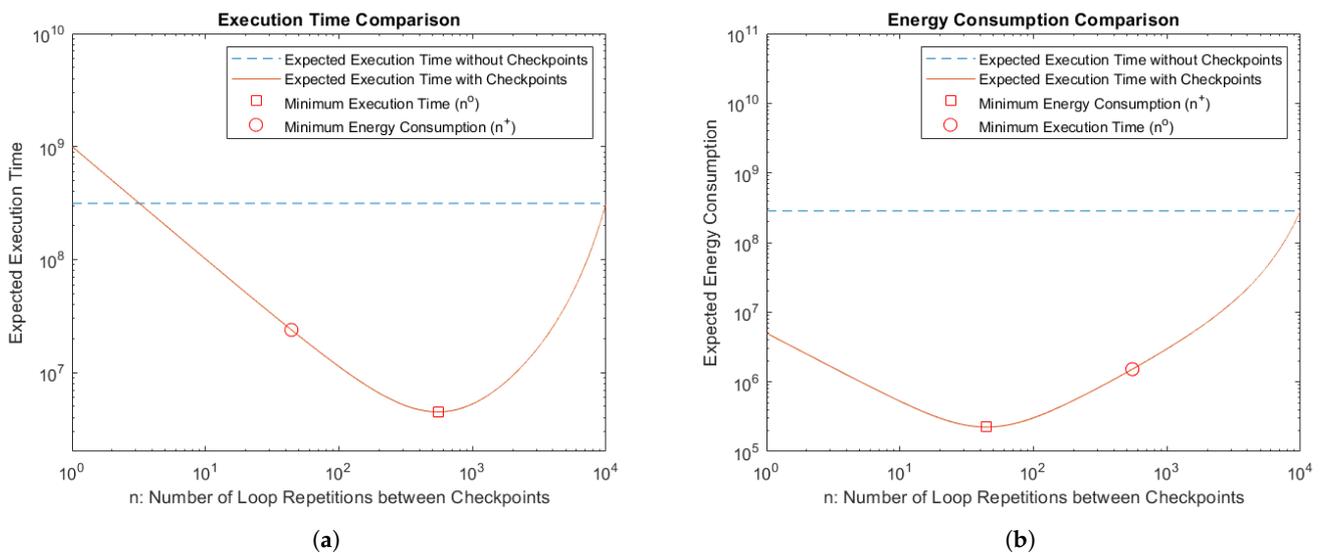
**Figure 2.** The case of a software program of medium size (i.e.,  $Y = 10^5$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected execution time gain.



**Figure 3.** The case of a large software program (i.e.,  $Y = 10^6$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected execution time gain.



**Figure 4.** The case of a very large software program (i.e.,  $Y = 10^7$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected execution time gain.



**Figure 5.** The case of a large software program (i.e.,  $M = 10^6$ ): (a) Expected execution time with highlighted  $n^+$ . (b) Expected energy consumption with highlighted  $n^0$ .

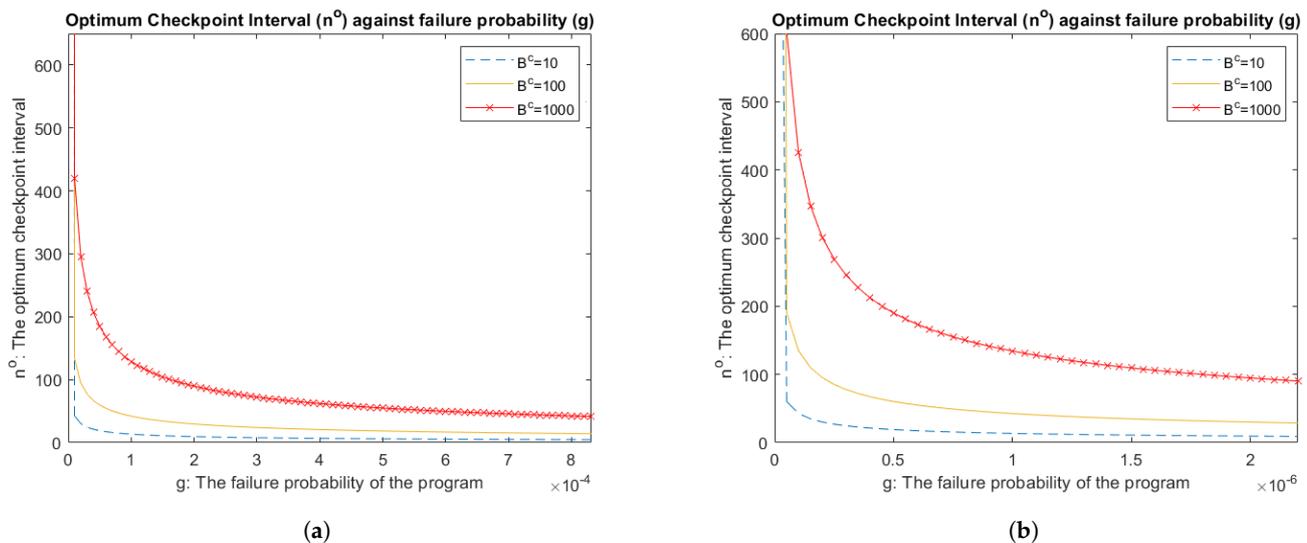
*Impact of  $g$  and  $B$  on the Optimum Checkpoint Interval*

The optimum checkpoint interval  $n^0$  is expected to be influenced both by the probability of failure  $g = 1 - a$ , and by the cost of checkpointing  $B^c = B_0^c$ . In Figure 6, the optimum checkpoint interval  $n^0$  is plotted against the probability of failure  $g$ , for three different cases of checkpointing cost  $B^c$ . Four different examples are provided, corresponding to a sample software program of small, medium, large, and very large size. In fact, the same cases of programs that were investigated in Section 4 were considered in this section.

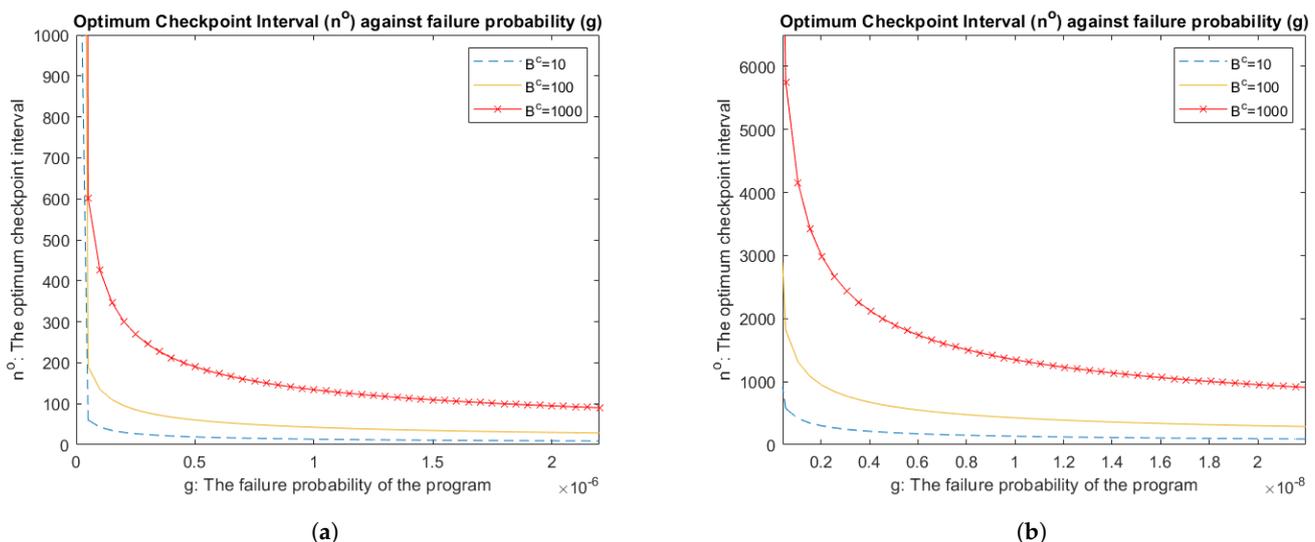
From the different graphs in Figures 6 and 7, we notice that the same behavior is observed regarding the impact that the values of  $B^c$  and  $g$  have on the optimum checkpoint interval, regardless of program size. Indeed for a given checkpointing cost  $B^c$ , the higher the probability of failure  $g$ , the lower the optimum checkpoint interval  $n^0$ . This means that for a given checkpointing cost, the higher the probability of failure the more frequently the checkpoints should be generated. This is reasonable since the more frequent the failures are the more frequent the checkpointing should be, in order to reduce the cost incurred by

the failure-related re-executions. Conversely, for a specific probability of failure  $g$ , a higher cost of a single checkpoint  $B^c$  leads to a larger optimum checkpoint interval  $n^o$ . This is also reasonable, since the higher the checkpointing cost (given that the frequency of failures is constant) the less frequent the checkpointing, since frequent checkpointing may incur checkpoint-related costs.

These observations are highly intuitive since frequent checkpointing should be applied when the probability of failure is high, while checkpoints should be generated less frequently when the checkpointing cost is high. The same observations hold for the case of the optimum checkpoint interval  $n^+$  that minimizes the total expected energy consumption of the program.



**Figure 6.** The optimum checkpoint interval  $n^o$  against the probability of failure  $g$  for different cases of checkpointing cost  $B^c$ , for a program of small (a) size with  $Y = 10^4$ , and a medium sized program in (b) with  $Y = 10^5$ .



**Figure 7.** The optimum checkpoint interval  $n^o$  against the probability of failure  $g$  for different cases of checkpointing cost  $B^c$ , for a large program (a) with  $Y = 10^6$ , and a very large program (b) with  $Y = 10^7$ .

### 5. Demonstration through a Real-World Example

In Section 4, we illustrated the effect of the checkpoint interval  $n$  (i.e., the number of loop repetitions between consecutive checkpoints) on the expected execution time and energy consumption of a software program that operates in a failure-prone environment

through a set of numerical examples. The results of the simulation led us to the observation that the checkpoint interval should be chosen to be at (or close to) its optimum value (computed by our mathematical model) in order to achieve significant gains with respect to execution time or energy consumption and to avoid potential costs that may be caused by assigning arbitrary values to  $n$ .

To enhance the completeness of the present work, we also illustrate the effect of the checkpoint interval selection on the computation time and energy consumption of a real-world software program. More specifically, instead of being based on simulated values, we selected a real-world open-source software program with a configurable computational loop and we determined the required model parameters through actual measurements. Then we used our model in order to compute the optimum checkpoint intervals that optimize the execution time and energy consumption of the selected program for different cases of program size (in fact, loop length). We focused on the execution time and energy savings that could be achieved through the selection of the checkpointing interval using the proposed model.

For the purposes of the present experiment, we used the Rodinia Benchmark (<https://github.com/yuhc/gpu-rodinia>) [56] as the basis of our analysis. The Rodinia Benchmark is a popular benchmark of real-world open-source software programs written in C and C++ programming languages, which is widely used for benchmarking techniques and mechanisms for software performance and energy optimization. From the different programs that Rodinia contains, we used the *streamcluster* (<https://github.com/yuhc/gpu-rodinia/tree/master/opencl/streamcluster>) program as the basis of our example. The reasoning behind the selection of this program is that it contains a computational loop that is also highly configurable, making it suitable for the purposes of our analysis. In fact, by providing the correct input, the loop can be as lengthy as we wish, allowing us to take different cases of loop length.

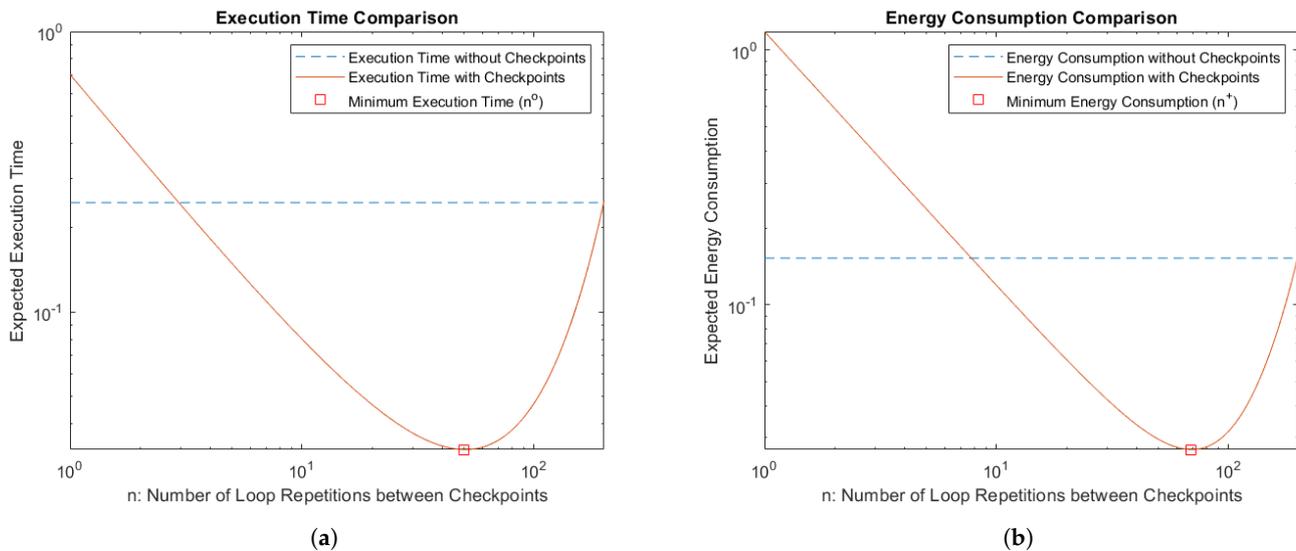
To compute the actual parameters that are necessary for the execution of our mathematical model, the *Energy Toolbox* of the SDK4ED Project was utilized [57,58]. The *Energy Toolbox* provides measurements of the execution time and energy consumption of a software program at the loop-level of granularity, being mainly based on popular profiling tools like Linux Perf (<https://perf.wiki.kernel.org/>) and Valgrind (<http://www.valgrind.org/>), as well as on static estimations [57,59]. The provision of loop-level performance and energy measurements made it highly suitable for our case, which actually constitutes the main reason for its selection. After executing the *Energy Toolbox* for the selected software program the following parameters were determined (It should be noted that all the measurements were made on an ARM Cortex A57 (Nvidia Jetson TX1) processor.):

$$\begin{aligned} B_0^e &= 0.0059, B_1^e = 0, B_0^c = 0.00347, B_1^c = 0, K^c = 0.097 \times 10^{-7}, \\ K^e &= 0.03345 \times 10^{-9}, b_0^e = 0.752 \times 10^{-6}, b_1^e = 6.51 \times 10^{-9}, \\ b_0^c &= 0.031 \times 10^{-6}, b_1^c = 0.45 \times 10^{-9}, g = 5 \times 10^{-6}, L = 4280. \end{aligned}$$

As already mentioned, since the benchmark is highly configurable, we considered three cases of loop length (in fact, of program size). In particular, we considered the case of a small, medium, and large loop comprising  $Y = 5 \times 10^5$ ,  $Y = 5 \times 10^6$ , and  $Y = 10^7$  instructions respectively. It should be noted that this characterization is based exclusively on the relative size of the loops that the program contains and it is used to better facilitate the description of the present experiment.

In Figure 8, the example of the program with the small loop is illustrated (i.e.,  $Y = 5 \times 10^5$ ). Figure 8a compares the expected execution time of the software program with and without checkpointing. Similarly, Figure 8b compares the expected energy consumption of the selected software program with and without the adoption of the ALCR mechanism. The checkpoint interval that minimizes the expected execution time ( $n^o$ ) and the checkpoint interval that minimizes the expected energy consumption ( $n^+$ ) are marked within a rectangle in Figure 8a,b respectively.

Figure 8 shows that important savings in both the expected execution time and energy consumption are achieved for software program, if the checkpoint interval is selected to be at (or close to) the values of  $n^o$  or  $n^+$  respectively computed by the mathematical model. More specifically, if  $n$  is selected to be equal to  $n^o$ , a 74.8% gain in execution time, and a gain of 67.3% in energy consumption is obtained when  $n$  is chosen equal to  $n^+$ . It is very clear that selecting arbitrary values for the checkpoint interval should be avoided, as this may lead to excessive increase in the execution time and energy consumption: i.e. no gain but even additional costs.

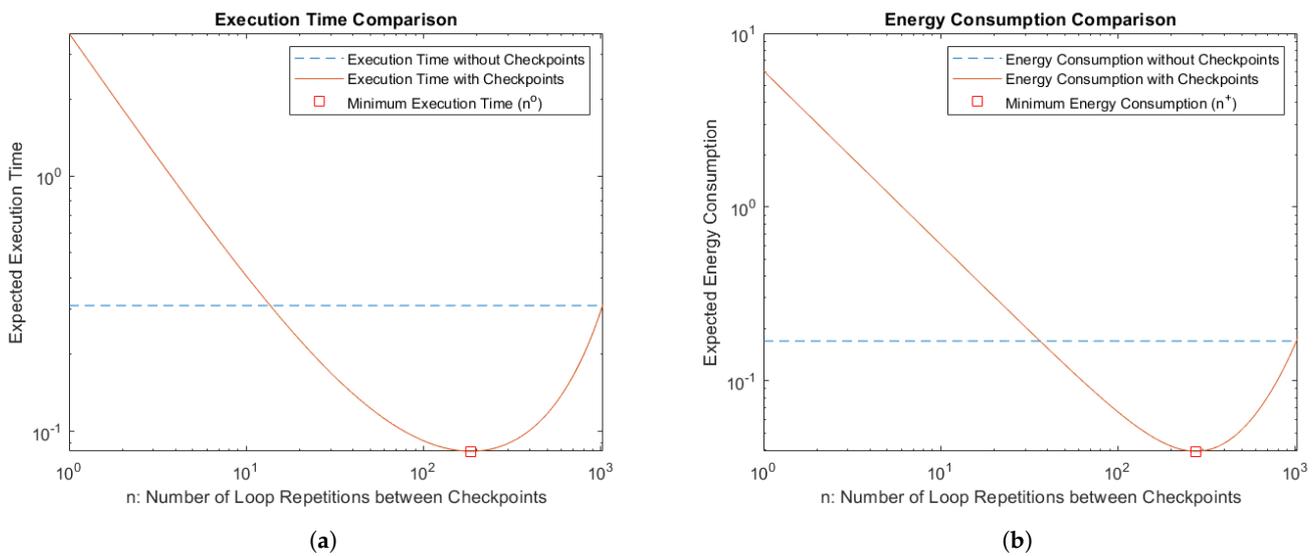


**Figure 8.** The case of a software program with a relatively small loop (i.e.,  $Y = 5 \times 10^5$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected energy consumption (logarithmic axes).

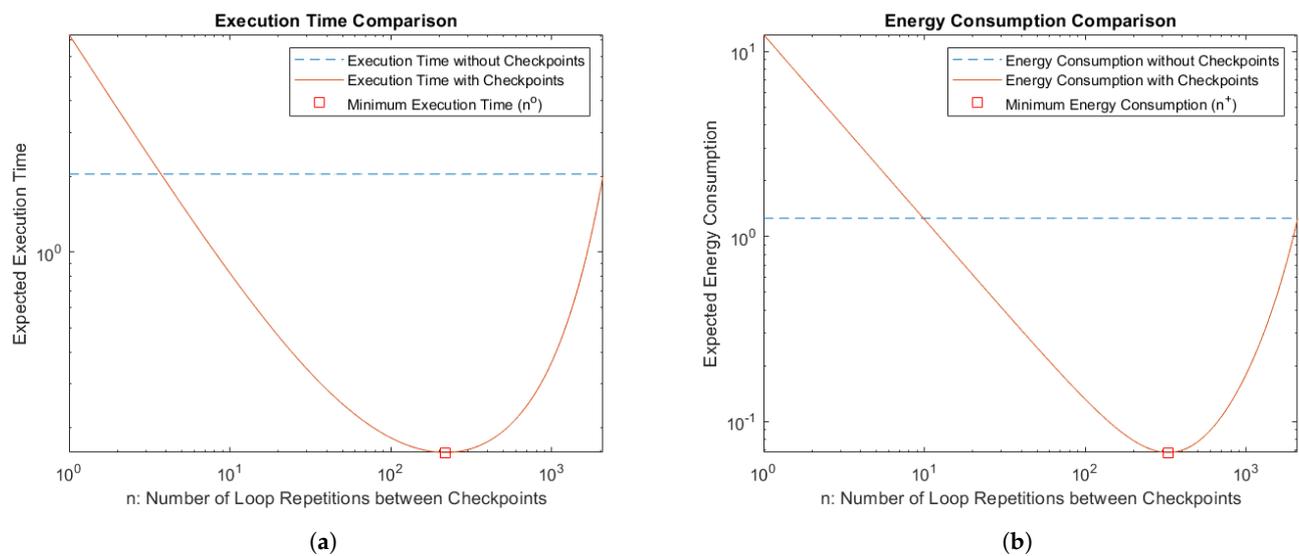
As can be seen by the given example, if  $n$  is set to be less than 3 in Figure 8a, the expected execution time of the program will be higher than its expected execution time when checkpointing is not adopted. Similarly, if  $n$  is set to a value lower than 8 in Figure 8b, the expected energy consumption of the program will be higher than the expected energy consumption of the same program when checkpointing is not adopted. This indicates that frequent checkpointing may lead to the introduction of additional costs with respect to execution time and energy consumption. In addition to this, in both cases, if  $n$  is set to a value different (lower or higher) than the optimum values  $n^o$  and  $n^+$  that are computed by our model, lower than the maximum achievable gains in terms of execution time and energy consumption are achieved, leading to omission of important savings. Hence, this suggests that the arbitrary selection of the checkpoint interval should be avoided, as it may lead to omission of important savings or even introduction of additional costs, and, in turn, it verifies that there is a need for a mechanism (model) for recommending the optimum checkpoint interval.

Similar observations can be made for longer loops in programs as can be seen by Figures 9 and 10. As for the previous case, these examples show that important savings in terms of execution time and energy consumption can be achieved, provided that the checkpoint interval is properly set. Here the maximum execution time savings are 73.12% and 92.21%, whereas the maximum energy savings are 77% and 94.6%, for both medium length and long loops, respectively. These examples also show that a poorly chosen value for the checkpoint interval may lead to the introduction of additional overhead with respect to the execution time and energy consumption of the software program, highlighting the importance of the choice of an optimum checkpoint interval. These results for a real program example also agree with the “theoretical” conclusions drawn from the numerical examples of Section 4.

Although in our examples it appears that the optimum values for computation time and energy, namely  $n^0$  and  $n^+$ , are relatively close to each other, this will not be generally the case, and depending on various parameters these values can differ significantly. Hence, the end user can decide whether execution time or energy consumption should be prioritized by using the parameters  $\alpha$  and  $\beta$ . As mentioned in Section 3, by carefully setting these parameters, the model can be used in order to compute the optimum checkpoint interval that optimizes the execution time ( $\alpha = 1$  and  $\beta = 0$ ), energy consumption ( $\alpha = 0$  and  $\beta = 1$ ), or a weighted combination of those two requirements ( $\alpha \neq 0$  and  $\beta \neq 0$ ). Hence, the mathematical model presented in this paper can be used in practice to satisfy different user needs with respect to energy consumption and execution time of software programs with loops.



**Figure 9.** The case of a software program with a relatively medium loop (i.e.,  $Y = 5 \times 10^6$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected energy consumption (logarithmic axes).



**Figure 10.** The case of a software program with a relatively large loop (i.e.,  $Y = 10^7$ ): (a) Expected execution time comparison (logarithmic axes) (b) Expected energy consumption (logarithmic axes).

## 6. Conclusions

Checkpoints are widely used to allow a system to recover from failures without having to restart a program's execution from scratch every time a failure occurs. However, checkpointing may add costs in additional time and energy, even when no failures occur. Thus, we have analyzed the choice of optimum checkpoint intervals in a unified manner from the perspective of energy consumption and execution time. Starting from first principles we have derived the optimum checkpoint for programs with a long running outer loop. Explicit analytic results have been derived and illustrated with numerical examples. The model was also demonstrated using a real-world software program retrieved from a popular benchmark.

More specifically, in this paper, we have focused on the importance of energy consumption on the appropriate choice of checkpoint intervals for long-running programs that require highly reliable operations. To this effect, we have developed a mathematical model that details the manner in which program execution time and energy consumption interact in a system that is subject to the establishment of regularly spaced checkpoint intervals.

The analysis has been used to determine the optimum number of checkpoints that either minimizes total average energy consumption, or total average execution time, or a linear combination of both. The solution to this optimization problem has been shown to relate directly to an expression that includes the classical Lambert function. The sensitivity of the optimum checkpoint interval to variations in all systems and checkpointing parameters has also been computed analytically.

The results were then used to derive the optimum checkpointing interval for a program with a long loop, so that checkpoints are installed either within each loop, or at the beginning of some of the loops. Several numerical examples were presented to illustrate the manner in which this approach could be used in a practical setting, for instance, to guide the choices that need to be made with application-level checkpointing and recovery (ALCR). A real-world example using an actual software program retrieved from the Rodinia Benchmark was also presented.

Both the numerical examples and the example that was based on the real-world software program led to some interesting observations. Firstly, in order to achieve important savings (i.e., gains) in terms of execution time and energy consumption, the checkpoint interval should be chosen to be at (or, at least, close to) its optimum value, as reported by our mathematical model. In addition to this, the arbitrary selection of the checkpoint interval should be avoided, as it may lead to lower than the maximum achievable gains in terms of execution time and energy consumption or even to the introduction of additional overheads. This further supports the need for a mechanism (i.e., a model) able to compute the optimum checkpoint interval. Finally, the results of these examples also highlighted the ability of the proposed model to be used in practice for satisfying different user and application needs with respect to execution time and energy consumption through properly setting its parameters. In fact, the proposed model can be used to compute the optimum checkpoint interval that minimizes its execution time, energy consumption, or a combination of those requirements.

The programs that provide the numerical solutions we have discussed have been made publicly available at the GitHub repository with Matlab scripts of our mathematical model at <https://github.com/siavvasm/optimum-checkpoint-interval>.

Future work will consider nested program structures, and ways of linking checkpointing and program structure in a useful manner, similar to what is done in this paper for programs with a large single loop. The impact of multiple programs running on the same platform also needs to be considered. Indeed the ALCR approach deals with each program singly, while the checkpoint for each program dilates the execution time and energy consumption of each individual program, and by extension of the collection of programs, which share the same platform.

**Author Contributions:** E.G. developed and wrote the mathematical model, wrote the introductory material including the literature survey, edited the section on numerical results and the Rodinia benchmark, and the figure captions, and wrote the conclusions. M.S. pointed to the use of the Lambert function, computed the numerical examples, wrote the description of the numerical results, developed the link with ALCR, and developed the application to the Rodinia Benchmark and wrote the corresponding section. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was supported by the European Commission through the Horizon 2020 SDK4ED Project under Grant Agreement No. 780572. The contents of this paper represent the opinions of the authors, and do not engage the responsibility of the European Commission.

**Institutional Review Board Statement:** Not applicable for studies not involving humans or animals.

**Informed Consent Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Randell, B. System Structure for Software Fault Tolerance. *IEEE Trans. Softw. Eng.* **1975**, *2*, 220–232. [\[CrossRef\]](#)
2. Kale, L.V.; Krishnan, S. CHARM++: A portable concurrent object oriented system based on C++. *Parallel Process. Lett.* **1993**, *28*, 91–108.
3. Zheng, G.; Shi, L.; Kale, L.V. FTC-Charm++: An In-Memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In Proceedings of the 2004 IEEE international Conference on Cluster Computing, San Diego, CA, USA, 20–23 September 2004; pp. 93–103.
4. Stavrinides, G.L.; Karatza, H.D. The impact of checkpointing interval selection on the scheduling performance of real-time fine-grained parallel applications in SaaS clouds under various failure probabilities. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4288. [\[CrossRef\]](#)
5. Dauwe, D.; Jhaveri, R.; Pasricha, S.; Maciejewski, A.A.; Siegel, H.J. Optimizing checkpoint intervals for reduced energy use in exascale systems. In Proceedings of the 2017 Eighth International Green and Sustainable Computing Conference (IGSC), Orlando, FL, USA, 23–27 October 2017; pp. 1–8.
6. Egwutuoha, I.P.; Levy, D.; Selic, B.; Chen, S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.* **2013**, *65*, 1302–1326. [\[CrossRef\]](#)
7. Vadhiyar, S.; Dongarra, J. SRS—A framework for developing malleable and migratable parallel software. *Parallel Process. Lett.* **2003**, *13*, 291–312. [\[CrossRef\]](#)
8. Mehnert-Spahn, J.; Ropars, T.; Schoettner, M.; Morin, C. The architecture of the xtreemos grid checkpointing service. In Proceedings of the European Conference on Parallel Processing, Delft, The Netherlands, 25–28 August 2009; pp. 429–441.
9. Agrawal, S.; Garg, R.; Gupta, M.S.; Moreira, J.E. Adaptive incremental checkpointing for massively parallel systems. In Proceedings of the ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing, ACM, Saint-Malo, France, 26 June–1 July 2004; p. 277–286.
10. Moody, A.; Bronevetsky, G.; Mohror, K.; De Supinski, B.R. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In Proceedings of the SC'10: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 13–19 November 2010; pp. 1–11.
11. Plank, J.S.; Beck, M.; Kingsley, G.; Li, K. *Libckpt: Transparent Checkpointing under UNIX*; Technical Report UT-CS-94-242; Department of Computer Science, University of Tennessee: Knoxville, TN, USA, 1994.
12. Duell, J. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*; Lawrence Berkeley National Laboratory: Berkeley, CA, USA, 2005.
13. Litzkow, J.B.M.; Tannenbaum, T.; Livny, M. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*; Technical Report; University of Wisconsin: Madison, WI, USA, 1997.
14. Gelenbe, E.; Hebrail, G. A probability model of uncertainty in data bases. In Proceedings of the 1986 IEEE Second International Conference on Data Engineering, Los Angeles, CA, USA, 5–7 February 1986; pp. 328–333.
15. Chandy, M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **1985**, *3*, 63–75. [\[CrossRef\]](#)
16. Chesnais, A.; Gelenbe, E.; Mitrani, I. On the modeling of parallel access to shared data. *Commun. ACM* **1983**, *26*, 196–202. [\[CrossRef\]](#)
17. Wang, Y.M.; Fuchs, W.K. Optimistic message logging for independent checkpointing in message-passing systems. In Proceedings of the 11th Symposium on Reliable Distributed Systems, Houston, TX, USA, 5–7 October 1992; pp. 147–154.
18. Sancho, J.C.; Petrini, F.; Johnson, G.; Frachtenberg, E. On the feasibility of incremental checkpointing for scientific computing. In Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, 26–30 April 2004; p. 58.
19. Young, J.W. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* **1974**, *17*, 530–531. [\[CrossRef\]](#)

20. Gelenbe, E.; Derochette, D. Performance of Rollback Recovery Systems Under Intermittent Failures. *Commun. ACM* **1978**, *21*, 493–499. [[CrossRef](#)]
21. Gelenbe, E. On the Optimum Checkpoint Interval. *J. ACM* **1979**, *26*, 259–270. [[CrossRef](#)]
22. Gelenbe, E. A Model of Roll-back Recovery with Multiple Checkpoints. In Proceedings of the 2nd International Conference on Software Engineering, Los Alamitos, CA, USA, 13–15 October 1976; pp. 251–255.
23. Gelenbe, E. A model on information renewal by the method of multiple test points. *Avtom. Telemekhanika* **1979**, *4*, 142–151.
24. Benoit, A.; Cavelan, A.; Le Fèvre, V.; Robert, Y.; Sun, H. Towards optimal multi-level checkpointing. *IEEE Trans. Comput.* **2016**, *66*, 1212–1226. [[CrossRef](#)]
25. Siavvas, M.; Gelenbe, E.; Kehagias, D.; Tzovaras, D. Static Analysis-Based Approaches for Secure Software Development. In *Security in Computer and Information Sciences. Communications in Computer and Information Science*; Gelenbe, E.; Gelenbe, E., Campegiani, P., Czachorski, T., Katsikas, S., Komnios, I., Romano, L., Tzovaras, D., Eds.; Springer: Cham, Switzerland, 2018; Volume 821, pp. 142–157. [[CrossRef](#)]
26. Arora, R. ITALC : Interactive Tool for Application—Level Checkpointing. In Proceedings of the Fourth International Workshop on HPC User Support Tools, Denver, CO, USA, 12 November 2017.
27. Shahzad, F.; Thies, J.; Wellein, G. CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *30*, 501–514. [[CrossRef](#)]
28. Losada, N.; Martín, M.J.; Rodríguez, G.; Gonzalez, P. Portable application-level checkpointing for hybrid MPI-OpenMP applications. *Procedia Comput. Sci.* **2016**, *80*, 19–29. [[CrossRef](#)]
29. Rodríguez, G.; Martín, M.J.; González, P.; Tourino, J.; Doallo, R. CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. *Concurr. Comput. Pract. Exp.* **2010**, *22*, 749–766. [[CrossRef](#)]
30. Tripathi, S.K.; Finkel, D.; Gelenbe, E. Load sharing in distributed systems with failures. *Acta Inform.* **1988**, *25*, 677–689. [[CrossRef](#)]
31. Gelenbe, E.; Finkel, D.; Tripathi, S.K. Availability of a distributed computer system with failures. *Acta Inform.* **1986**, *23*, 643–655. [[CrossRef](#)]
32. Pernici, B.; Aiello, M.; Vom Brocke, J.; Donnellan, B.; Gelenbe, E.; Kretsis, M. What IS can do for environmental sustainability: A report from CAiSE'11 panel on Green and sustainable IS. *Commun. Assoc. Inf. Syst.* **2012**, *30*, 18. [[CrossRef](#)]
33. Gelenbe, E.; Caseau, Y. The impact of information technology on energy consumption and carbon emissions. *Ubiquity* **2015**, *2015*, 1–15. [[CrossRef](#)]
34. Pinto, G.; Castor, F. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* **2017**, *60*, 68–75. [[CrossRef](#)]
35. Anser, M.K.; Ahmad, M.; Khan, M.A.; Zaman, K.; Nassani, A.A.; Askar, S.E.; Abro, M.M.Q.; Kabbani, A. The role of information and communication technologies in mitigating carbon emissions: evidence from panel quantile regression. *Environ. Sci. Pollut. Res.* **2021**, 1–20. [[CrossRef](#)]
36. Gelenbe, E. Energy packet networks: ICT based energy allocation and storage. In Proceedings of the International Conference on Green Communications and Networking, Colmar, France, 5–7 October 2011; pp. 186–195.
37. Stavrinides, G.L.; Karatza, H.D. The impact of workload variability on the energy efficiency of large-scale heterogeneous distributed systems. *Simul. Model. Pract. Theory* **2018**, *89*, 135–143. [[CrossRef](#)]
38. Stavrinides, G.L.; Karatza, H.D. An energy-efficient, QoS-aware and cost-effective scheduling approach for real-time workflow applications in cloud computing systems utilizing DVFS and approximate computations. *Future Gener. Comput. Syst.* **2019**, *96*, 216–226. [[CrossRef](#)]
39. Gelenbe, E.; Iasnogorodski, R. A queue with server of walking type (Autonomous Service). *Ann. Institut Henri Poincaré, Probabilités et Statistiques* **1980**, *16*, 63–73.
40. Gelenbe, E.; Lent, R.; Douratsos, M. Choosing a local or remote Cloud. In Proceedings of the Second Symposium on Network Cloud Computing and Applications, IEEE, London, UK, 3–5 December 2012; pp. 25–30. [[CrossRef](#)]
41. Gelenbe, E. Energy packet networks: smart electricity storage to meet surges in demand. In Proceedings of the SIMUTOOLS'12: International ICST Conference on Simulation Tools and Techniques, Desenzano del Garda, Italy, 19–23 March 2012; pp. 1–7.
42. Gelenbe, E. Energy packet networks: adaptive energy management for the cloud. In Proceedings of the CloudCP'12: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, Bern, Switzerland, 10–13 April 2012; pp. 1–5. [[CrossRef](#)]
43. Gelenbe, E.; Ceran, E.T. Energy packet networks with energy harvesting. *IEEE Access* **2016**, *4*, 1321–1331. [[CrossRef](#)]
44. Gelenbe, E.; Zhang, Y. Performance optimization with energy packets. *IEEE Syst. J.* **2019**, *13*, 3770–3780. [[CrossRef](#)]
45. Aupy, G.; Benoit, A.; Renaud-Goud, P.; Robert, Y. Energy-Aware Algorithms for Task Graph Scheduling, Replica Placement and Checkpoint Strategies. In *Handbook on Data Centers*; Khan, S.U., Zomaya, A.Y., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; pp. 37–80.
46. Morán, M.; Ballardini, J.; Rexachs, D.; Luque, E. Checkpoint and Restart: An Energy Consumption Characterization in Clusters. In Proceedings of the Argentine Congress of Computer Science, Tandil, Argentina, 8–12 October 2018; pp. 19–33.
47. Siavvas, M.; Gelenbe, E. Optimum interval for application-level checkpoints. In Proceedings of the 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), Paris, France, 21–23 June 2019; pp. 145–150.
48. Siavvas, M.; Gelenbe, E. Optimum checkpoints for programs with loops. *Simul. Model. Pract. Theory* **2019**, *97*, 101951. [[CrossRef](#)]

49. Gelenbe, E.; Boryszko, P.; Siavvas, M.; Domanska, J. Optimum Checkpoints for Time and Energy. In Proceedings of the 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Nice, France, 17–19 November 2020; pp. 1–8. [\[CrossRef\]](#)
50. Berl, A.; Gelenbe, E.; Di Girolamo, M.; Giuliani, G.; De Meer, H.; Dang, M.Q.; Pentikousis, K. Energy-efficient cloud computing. *Comput. J.* **2010**, *53*, 1045–1051. [\[CrossRef\]](#)
51. Gelenbe, E.; Hernández, M. Optimum checkpoints with age dependent failures. *Acta Inform.* **1990**, *27*, 519–531. [\[CrossRef\]](#)
52. Lambert, J.H. Observationes variae in mathesis puram. *Acta Helv.* **1758**, *III*, 128–168.
53. Euler, L. De serie Lambertina Plurimisque eius insignibus proprietatibus. *Acta Acad. Sci. Petropol.* **1783**, *2*, 29–51.
54. Pólya, G.; Szegő, G. *Aufgaben und Lehrsätze der Analysis*; Springer: Berlin/Heidelberg, Germany, 1925.
55. Daly, J. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.* **2006**, *22*, 303–312. [\[CrossRef\]](#)
56. Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J.W.; Lee, S.; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 4–6 October 2009; pp. 44–54. [\[CrossRef\]](#)
57. Siavvas, M.; Tsoukalas, D.; Marantos, C.; Tsintzira, A.A.; Jankovic, M.; Soudris, D.; Chatzigeorgiou, A.; Kehagias, D. The SDK4ED Platform for Embedded Software Quality Improvement-Preliminary Overview. In Proceedings of the International Conference on Computational Science and Its Applications, Cagliari, Italy, 1–4 July 2020; pp. 1035–1050.
58. Kehagias, D.; Jankovic, M.; Siavvas, M.; Gelenbe, E. Investigating the Interaction between Energy Consumption, Quality of Service, Reliability, Security, and Maintainability of Computer Systems and Networks. *SN Comput. Sci.* **2021**, *2*. [\[CrossRef\]](#) [\[PubMed\]](#)
59. Marantos, C.; Salapas, K.; Papadopoulos, L.; Soudris, D. A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis. *SN Comput. Sci.* **2021**, *2*. [\[CrossRef\]](#)