On the Scheduling of Checkpoints in Desktop Grids

Mohamed Slim Bouguerra, Derrick Kondo
INRIA Rhone-Alpes Grenoble
ZIRST, 51, avenue Jean Kuntzmann
38330 MONBONNOT SAINT MARTIN, France
mohamed-slim.bouguerra@imag.fr; dkondo@imag.fr

Denis Trystram
Grenoble University and Institut Universitaire de France
ZIRST, 51, avenue Jean Kuntzmann
38330 MONBONNOT SAINT MARTIN, France
denis.trystram@imag.fr

***Abstract*—Frequent resources failures are a major challenge for the rapid completion of batch jobs. Checkpointing and migration is one approach to accelerate job completion avoiding deadlock. We study the problem of scheduling checkpoints of sequential jobs in the context of Desktop Grids, consisting of volunteered distributed resources. We craft a checkpoint scheduling algorithm that is provably optimal for discrete time when failures obey any general probability distribution. We show using simulations with parameters based on real-world systems that this optimal strategy scales and outperforms other strategies significantly in terms of checkpointing costs and batch completion times.**

***Keywords*-Fault tolerance; Checkpoint; Volunteer computing;**

I. Introduction

Batches of sequential jobs (often referred to as bags of tasks or BoTs) are one of the most common workloads for parallel and distributed systems. This is because BoTs scale efficiently, and because they are simple to program. The dominant workload in computational Grids are BoTs [12]. The same is true for desktop Grids, which use the idle cycles of Internet-distributed desktops for large computation.

BoTs are often submitted with soft deadlines in mind. These deadlines range from a few hours of days. In any case, timely completion of sequential jobs is often important for users. In the context of Desktop Grids, timely completion is also important to volunteers, who want to be granted virtual credit for their jobs as soon as possible.

Timely batch completion is especially important for a workflow of batches that must be completed. For several desktop Grids applications, such as FOLDING@home, one batch can be started only after the previous one has completed. Timely completion is also important for long-running sequential jobs on the order of days or even weeks of computation. This is the case for applications in climateprediction.net [4], for example.

A major challenge for fast batch completion times, especially in Desktop Grids, are failures and unavailability of the hosts. For instance, in SETI@home, the mean duration of unavailability is about 4.5 hours. These long periods of unavailability cause long latency in BoT completion. The median duration of availability is about 1 hour in length. So these are clearly volatile failure-prone systems.

Our approach to solve this problem is to use the checkpoint and migration of sequential jobs. Clients upload checkpoints periodically to the server. Process migration occurs when these checkpoints are downloaded by clients and restarted.

In this paper, we deal with the problem of scheduling the checkpoints uploaded by the clients. Clearly, non-optimal checkpointing scheduling can cause inefficiencies. Checkpointing too often can cause unnecessary bandwidth overheads and delay job execution. Checkpointing too infrequently can cause jobs to sit stagnantly on unavailable hosts, delaying completion time.

The checkpoint scheduling problem is usually formulated as a complex non-linear optimization based on a continuous objective function with unknown number of optimization variables. Unfortunately most of the time, such formulations lead to intractable problems.

In contrast, our main contribution is the design and evaluation of checkpointing algorithms that are provably optimal for sequential jobs in discrete time with any general failure law. Our formulation aims at minimizing the wasted time considering the variability on the checkpoint costs and the time to detect failures. To the best of our knowledge such a generic formulation does not exist. We provide optimal algorithms for any failure laws and for arbitrary checkpoint costs. Using simulation with models based on real-world desktop Grids, we show that our checkpointing algorithms perform well and outperform the state-of-the-art significantly.

II. System Model

A. Context

We focus on Desktop Grids, which use idle resources of desktops distributed over the Internet for massively parallel computation. Currently, these systems provide over 7 PetaFLOPS of computing power [2], [15], [19] for over 68 applications from a wide range of scientific domains (including climate prediction, protein folding, and gravitational physics). These projects have produced hundreds of scientific results [3] published in the world's most prestigious conferences and journals, such as Science and Nature.

Below we describe the system model of Desktop Grids used throughout this work. This model is based on our experience with BOINC [1], one of the most widely used middleware for Desktop Grids. BOINC serves as the basis of projects such as SETI@home, climateprediction.net, and EINSTEIN@home.

B. Client-Server Model

We consider a client-server model where a single centralized server oversees a set of clients. Clients are the entities that perform the distributed computation. The server is responsible for distributing jobs among the clients. Clients initiate all communication with the server. A client pulls jobs from the server, pulling a single job at each request. Upon request, the server decides which job is sent to a client. Once the job is completed, the client uploads the result to the server and requests a new job.

C. Network Model

Each client is connected to the server with its own single link. The server has a maximum number of client connections that it can maintain simultaneously. Typically, this maximum value is about 200-300 simultaneous connections. (This parameter can be specified in Desktop Grid server software such as BOINC.) Once connected, we assume each client has a maximum bandwidth equal to the bandwidth of its own network link. If the server already has the maximum number of connections, then we assume that the server will postpone any future requests to connect from new clients until a connection is ended. We assume that the delay for making the new connection is measurable and known.

Implicitly, the network bottleneck is the network link of each individual client (versus the central network link of the server). The server is able to control the number of active client connections and bandwidth allocation for each connection. The BOINC server will force the clients to back-off exponentially when overloaded. So, it is reasonable to assume the network link of the end host is the bottleneck.

Note that we intentionally do not consider a peer-to-peer (P2P) storage network; clients can only communicate with the server directly. Despite being an area of active research, no real-world industrial Desktop Grid middleware employs peer-to-peer techniques over the Internet. In particular, in BOINC, P2P methods are avoided entirely because they add too much complexity in terms of security issues and configuration issues (such as ensuring certain ports are open). Also, data location, consistency, and versioning in P2P systems is still a complex and open issue.

D. Failure Model

We define a failure to be any event that causes the guest desktop application from running. Thus, failures are in sense from the perspective of the guest desktop application. Causes of failures include but are not limited to CPU load from other user applications, mouse movement, and powering off of the machine. We term the period during which a failure occurs to be an *unavailability* interval.

The clients exhibit two types of failures, namely, permanent failures and transient failures. Permanent failures occur when clients disappear from the system completely. The time until a permanent failure occurs is often referred to as client *lifetime*, and has been modeled in previous works [11].

Transient failures occur when the desktop be unavailable for the application intermittently. An application may be suspended or terminated for reasons such as user mouse activity, other active higher-priority processes, or machine shutdown.

In both cases, the time to failure is uncertain, and we model failures with probability distributions. Specifically, we use a probability distribution to model client lifetimes, i.e., the time between when the client first enters the system and it leaves the system permanently. In recent works, the authors show that client lifetime can be modeled accurately with a Weibull distribution.

For transient failures, we model continuous periods of availability and periods of unavailability for a single host, each with its own with a probability distribution. In recent work [14] , the authors devised a method to discover individual hosts whose distribution of availability and unavailability is truly random and stationary. The authors go on fit probability distributions to the availability durations of these hosts. They find that the Weibull and the hyper-exponential distributions are the best fitting for availability and unavailability respectively. In Section IV-A, we describe these model parameters in detail.

Note that the fraction of hosts with truly random availability was significant at roughly 20%. In desktop grid systems that contribute over 10 PetaFLOPS of computing power, that fraction of hosts still provides an enormous amount computational power.

E. Workload Model

The workload consists of a batch of independent and CPU-bound jobs. The number of jobs per batch ranges between 1000 to 10000 jobs. This reflects the number of jobs per batch found in real systems, such as those in the Grid Workload Archive [13] or BOINC [9]. We assume the runtime of job is generated from a uniform distribution between 10 to 240 CPU hours on a dedicated host of mean speed. The runtime corresponds to long running jobs of real desktop grid applications such as climate prediction [4]. In order to focus on the performance of our checkpointing methods, we assume there is only a single batch scheduled in the system at any given time.

III. Checkpoints Scheduling Policy

The checkpoint/restart protocol is a well-known technique that can be used to minimize the amount of lost computation

due to failures on workers by saving the intermediate state of computation on remote stable storage. After a failure or when a node disappears from the platform, jobs can be resumed from the last checkpoint and not from scratch. However, checkpointing too frequently can lead to expensive overheads in terms of on the system performance due to the huge amount of network I/O induced from checkpoint communication. On the other hand, checkpoints too infrequently can lead to a huge amount of lost computation due to the interruptions caused by host failures or disappearance. Hence, an efficient checkpointing policy is required to manage the tradeoff between the lost computation due to failures and network I/O overhead due to the checkpointing communication.

Consider the distributed property of a global scheduling policy on Desktop Grids where communication is initiated only by the worker. This implies that the checkpoint scheduling problem should be considered from the client's point of view. Each worker will need to compute an optimal local schedule for checkpoints with respect to its jobs and its hardware properties (such as CPU power and network speed). In what follows we present a sequential scheduling model for checkpoints that will be used by each client to manage the tradeoff between lost computation and network checkpointing overhead.

A. Sequential Checkpoint Model

In this section, we introduce the sequential checkpoint model that will be used to derive the performance model. Each worker pulls sequential jobs from the server. Jobs are composed from a set of atomic slices. This means that checkpoints are only permitted between slices. We assume the application scientist can define these slices as they have knowledge of when it is convenient and possible to save application state. This is currently done by scientists for their applications in the case of BOINC [1] .

More precisely, let us consider a given job where a checkpoint between slices represents the intermediate computation obtained up to that point. We assume that a job is composed from n slices. Each slice has a duration $p_j,\ 1 \leq j \leq n$. The checkpoint policy for this job is defined by a vector $\pi = (a_1, a_2 \cdots, a_n)$, $a_j \in \{0, 1\}$ where $a_j = 1$ when a checkpoint is scheduled after the j^{th} slice and $a_j = 0$ otherwise.

Without loss of generality, we suppose that a checkpoint should be scheduled just after the last slice ($a_n = 1$). This assumption is usually considered as an acknowledgment for the job completion.

We consider that the cost induced by a given checkpoint scheduled after the j^{th} slice depends only on the slice output data o_j to be stored on the remote stable storage. To send this data to the stable storage we assume that it takes c_j unit of time. (This cost depends on the amount of data to send o_j and network between the client and server). We consider a linear network model where the duration needed to send a certain amount of data is linear with respect to available bandwidth of the node plus a certain network latency.

B. Performance Model

In this section we introduce the performance model that expresses the tradeoff between the amount of lost computation and the network I/O overhead due to checkpoints. In this work we focus on minimizing the wasted time during execution. The wasted time is the accumulation of two different types of useless work during the execution of a given job.

The first type of useless work is the *lost computation time* denoted by L. The lost computation represents the amount of lost work due to a failure. Thus, L is proportional to the elapsed time between the last checkpoint and a following failure. For instance, in transactional databases, the slices correspond to transactions where the time needed to re-execute the lost transactions due to failure is usually less than the actual running time of the slice. Formally consider that a failure occurs at a time t_{fault}. Suppose that this failure is in a given checkpoint interval denoted by $[\tau_{last}, \tau_{next}]$. Thus, we have $\tau_{last} < t_{fault} \leq \tau_{next}$ such that τ_{last} is the time where the last checkpoint was done before t_{fault}, and τ_{next} is the time of the next scheduled checkpoint. Thus, the lost computation time in given by $L = \alpha(t_{fault} - \tau_{last})$ where α $(0 < \alpha \leq 1)$ is the re-execution ratio.

The second type of wasted time is the *cumulative checkpoint overhead* (denoted by σ_t). This is the amount of time spent to write checkpoints on remote stable storage from the beginning of job execution until the time t. The expected wasted time with respect to a given checkpoint interval is expressed as the following. Let $f(t)$ denote the failure density function where $\int_{\tau_i}^{\tau_j} f(t)dt$ is the probability that a failure occurs in the interval $[\tau_i, \tau_j]$. By conditioning on the time t where the failure happens in a given checkpoint interval denoted by $[\tau_{last}, \tau_{next}]$, the expected wasted time with respect to this interval is given by :

$$\mathbb{E}_{wasted}(\tau_{last}, \tau_{next}) = \int_{\tau_{last}}^{\tau_{next}} [\sigma_t + \alpha(t - \tau_{last})] f(t)dt. \quad (1)$$

Notice that minimizing the expected wasted time maximizes directly the expected amount of the time that the worker spends doing useful computation. In this way, the worker would collect a maximal virtual credit.

Therefore, the performance model inputs are the following:

1) n the number of slices
2) each slices' duration $p_j,\ 1 \leq j \leq n$
3) each slices' checkpoint cost (in terms of time) c_j $1 \leq j \leq n$.

The policy output is a checkpoint schedule for a given job defined by a vector $(a_1, a_2 \cdots a_n)$.

We formulate the performance model using the Markov Decision Process method [17]. We briefly recall that a Markov Decision Process consists of five elements:

1) the set of decision epoch denoted by T
2) the set of states S that the system occupies
3) the set of possible actions A
4) the transition function $\phi()$ which describe the evolution of the system during the time
5) and the reward function $r()$ that describes the reward that the system receives during the time.

To identify these elements we introduce the following notation.

Let τ_l denotes the useful computing time from the beginning until the l^{th} slice such that $\tau_l = \sum_{j=1}^{l} p_j$. Hence, after each τ_j for ($j \leq n$) the system takes the decision to checkpoint or not after the j^{th} slice. In this work we consider the finite set $T = \{\tau_1, \tau_2, \cdots, \tau_j\}$, $j \leq n$ as the decision epoch where the system makes the decisions. For clarity we suppose that $T = \{1, 2, \cdots, n\}$.

The set of possible actions is $A = \{0, 1\}$. (1 when a checkpoint is performed else 0.)

We denote by $s^{j,\sigma}$ the state that the system occupies at a given epoch. Hence, the state is describe by the couple (j, σ) where j is the index of the last checkpoint until this epoch and σ denotes the accumulated checkpoint overhead until this epoch. Then, the set of states is defined by $S = \{s^{j,\sigma}\}$ with $0 \leq j \leq n,\ 0 \leq \sigma \leq nc_{max}$ where c_{max} is the maximal checkpoint cost. In this work we suppose that the checkpoint costs are integers ($c_j \in \mathbb{N}$). Therefore, the maximal accumulated checkpoint overhead is bound by $O(n \times c_{max})$. For the clarity, let $\Sigma = \{0, 1, \cdots, nc_{max}\}$ denotes the set of integers.

Let $\phi_i(s, a)$ denote the transition function at the i^{th} epoch which is a mapping from $S \times A$ to S. Considering that the system occupies the state $s^{j,\sigma}$ at the i^{th} epoch, then the transition function is given by (Expression 2): if the action at the i^{th} epoch is $a_i = 1$ then the system goes to the state where the last checkpoint is after i^{th} slice with an accumulated checkpoint overhead equal to $\sigma + c_i$. On the other hand, if the action $a_i = 0$ the system stays at the same state where the last checkpoint and accumulated overhead are the same.

$$\phi_i(s, a) = \begin{cases} \phi_i(s^{j,\sigma}, 1) = s^{i,\sigma+c_i} \\ \phi_i(s^{j,\sigma}, 0) = s^{j,\sigma} \end{cases} \quad (2)$$

In this work we suppose that failures on nodes follows a general failure density denoted by $f(t)$ (with a distribution $F(t)$). Then, by conditioning on the time t when a failure happens between two epochs (i^{th} and $i + 1^{th}$) the reward can be expressed as follows.

Let $r_i(s^{j,\sigma}, a_i)$ denotes the reward function if the system chose the action a_i at the epoch i knowing that previous state is $s^{j,\sigma}$. For the first case, suppose that $a_i = 1$ and for $i < n$. Therefore, the expected wasted time is given by : $\sigma + c_i$ for the accumulated checkpoint overhead and $L = t - \tau_j - c_j$ as expected lost work where t is the failure time.

$$r_i(s^{j,\sigma}, 1) = \int_{\tau_j+\sigma}^{\tau_i+\sigma+c_i} [\sigma + c_i + \alpha(t - \tau_j - \sigma)] f(t) dt.$$

Then, if $a_i = 0$ for $i < n$, we suppose that the system does not get any reward while the last checkpoint is the same.

$$r_i(s^{j,\sigma}, 0) = 0.$$

Finally, if the system is at the final decision epoch where $i = n$, suppose the current state is $s^{j,\sigma}$. Therefore, the expected reward is given by $L = t - \tau_j - c_j$ and the accumulated checkpoint overhead is $\sigma + c_n$.

$$r_n(s^{j,\sigma}, a) = \int_{\tau_j+\sigma}^{\tau_n+\sigma+c_n} [\sigma + c_n + \alpha(t - \tau_j - \sigma)] f(t) dt.$$

C. Optimal Policy

We recall that the set of epochs is finite, and the choice of the action at the i^{th} epoch determines the subsequent state with certainty. The objective is to minimize the total reward (summed over all epochs). Consequently, the proposed Markov Decision process for the checkpoint scheduling problem is equivalent to the shortest path problem in a given graph. In what follow, we propose a deterministic dynamic programming schema to compute optimal policy of checkpoints. Consider a directed graph $G(E, V)$ where E is the set of edges and V is the set of vertices. Each vertex in G represents a given state at the Markov Decision Model. Vertices denoted by (v_j^{σ}) are labeled by the couple (j, σ) where $1 \leq j \leq n$ and $\sigma \in \Sigma$. Note that the number of vertices is bounded by $O(n^2 c_{max})$.

Suppose that the vertex v_0^0 represents the initial state of the system, and the vertex v_{sink}^* is the final state that the system occupies after the n^{th} epoch.

We note that edges between nodes denote the reward of traversing this edge. Thus, we propose these following rules to add edges between vertices:

1) For each vertex in G $v_j^{\sigma},\ (1 \leq j \leq n,\ \sigma \in \Sigma)$, we add an edge from this node to the node $v_i^{\sigma'}$ where i and σ' satisfy the following condition ($\sigma + c_i = \sigma',\ j < i \leq n$). The cost of traversing this edge is defined by Expression 3 which expresses the expected wasted time if two successive checkpoints are scheduled after slices j and i respectively and where the accumulated checkpoint overhead until the j^{th} slice is σ.

$$\int_{\tau_j+\sigma}^{\tau_i+\sigma+c_i} [\sigma + c_i + \alpha(t - \tau_j - \sigma)] f(t) dt. \quad (3)$$

2) An edge is added from the initial state v_0^0 to the vertex v_j^σ ,$\forall j$ if $\sigma = c_j$. We suppose that the reward function associated with these edges is given by the following expression:

$$\int_0^{\tau_j + c_j} [c_j + \alpha(t - \tau_j - c_j)] f(t) dt \qquad (4)$$

3) Finally an edge from v_n^σ $(\sigma \in \Sigma)$ to v_{sink}^* with a cost 0 is added to the graph.

Based on this construction it is clear that an optimal path with a minimal cost from v_0^0 to v_{sink}^* is equivalent to find an optimal schedule for checkpoints that minimizes the total expected reward over all epochs. Hence using this methodology, we can compute an optimal schedule in a reasonable amount of time. Without loss of generality we recall that the number of vertices in G is bounded by $O(n^2 c_{max})$ and from each vertex there are at most n edges (the first rule). Thus, the number of edges in G is bounded by $O(n^3 c_{max})$. Then, one may use the well-known Dijkstra algorithm to compute the optimal path where the computational complexity is bounded by the number of edges in the graph [7]. Therefore, the computational complexity of the dynamic programming methodology is bounded by $O(n^3 c_{max})$.

IV. Simulations and Experiments

In this part we present the simulation methodology used to assess the performance of the proposed fault tolerance policy. For our evaluation, we use the following metrics:

1) The maximum completion time. This is considered as the maximum completion time overall jobs in a given BoTs.

2) The overall wasted time due to checkpointing overheads and lost computation during the execution of the BoTs.

A. Experimental Methodology

In this section we describe the experimental methodology. First, we give a description for the simulation experiments. Then, we present the global scheduling policy used in conjunction with the proposed checkpointing policy to dispatch the BoTs over all available clients. Finally we report the different parameters used to run simulations.

The major difficulty for evaluating the proposed policy is to build a simulator that reproduces accurately the behavior of thousand of nodes. In such environments the platform is complex due to its large-scale (up to 20000 of clients), resource heterogeneity, network topology, and nodes volatility. For simulating the volunteers computing platform in this work we conduct all of our simulations below using the *SimGrid* simulation toolkit [5].

For modeling the volunteer computing platform we consider a centralized client-server model with M clients and one server. Using *SimGrid*, we implement the centralized scheduling policy used by the server to dispatch jobs, and the client policy used to pull jobs from the server.

The server scheduling policy is the FCFS policy based on the several rules. When the server receives a request, three kinds of responses are possible:

1) There are some unscheduled jobs from the BoTs. Thus the server chooses randomly a job from the remaining jobs to send.
2) All jobs have already been scheduled but have not yet been completed. Note that the server might not know the status of uncompleted jobs if they are still in execution mode or they are lost forever due the client's limited lifetime. Thus, the server chooses randomly one job to duplicate, and the execution of this copy will start from the most recent checkpoint. Note that there is no synchronization between copies. For duplicated jobs, both the outdated checkpoints and outdated job results are ignored.
3) When all work is done, then a halt message is sent to the client.

We assume that client pulls a single job at each request from the server. A client pull work from the server after two different events. The first event is the completion of the job. In this case, the client sends the result back to the server and requests a new job. The second event is the epoch of time when the client become again available after a failure. Thus, we assume that after a failure, clients should ask for a new job.

1) Simulation parameters: The considered platform is composed from M clients where $M = \{5000, 10000, 20000\}$ and one server. The server properties are the following: 2GFLOPS as the CPU speed and a 1Gb/s as network speed. In this work we assume that the server is failure-free. In these simulations we assume that the remote stable storage for checkpointing is located at the same server. We suppose that the server can have a maximum of 100 of simultaneous connections. Real desktop grids servers have similar limits. Once connected, each client has a maximum bandwidth equal to the bandwidth of its own network link.

Client properties (I) were randomly generated based on statistical studies on the well-know SETI@Home project [11], [14]. Client processor speeds (MIPS) are randomly generated based on the work described in [11]. There the authors shows that client speeds are well-fit by a normal distribution with the following parameters (mean =1771, standard deviation =669.5). Client network connection speed (Mb/s) is uniformly sampled between 1 and 2 (which is roughly is the range of ADSL connection speeds).

The availability and unavailability period are generated using the models proposed in [14] based on empirical

availability derived from SETI@home. In this work authors identify 6 different clusters of clients, each of which corresponds to a different distribution of availability. Clients in the same cluster have the same availability and unavailability distribution.

In our study we consider cluster number 3 which contains 20000 clients. In this cluster the availability period (hours) of clients follows a Weibull distribution with the following parameters $(shape = 0.431, scale = 1.682)$. For the unavailability period the best fitted distribution is the hyper-exponential with the following parameters $p_i \in \{0.398, 0.305, 0.298\}$ and $\mu_i \in \{0.031, 11.566, 1.322\}$.

The clients' lifetime is randomly generated according to the work in [11] where the lifetime follows a Weibull distribution according to the following parameters (the unit is days) $(scale = 137, shape = 0.58)$.

To conduct this simulation we investigate several BoTs's properties (II). We consider that the number of jobs in a given BoTs range in $\{1000, 3000, 5000, 7000, 9000, 10000\}$. Each job is composed from n slices, which is uniformly generated between $[1, 50]$. The amount of computation in each slice's, expressed in number of instructions, is randomly sampled between $[36^9, 36^{10}]$. This corresponds to an execution time range between $[1, 5]$ hours with respect to the dedicated execution time on a client with a mean CPU speed. This implies that the execution time range of one job is $[1, 250]$ hours. For instance, the overall execution time for a BoTs with 5000 jobs on one processor is [0.5,146] years. The checkpoint size (Mb) for each slice is also uniformly sampled between $[10, 512]$.

2) Fault Tolerance Policies and Metrics : To assess the performance of the proposed policy, we propose a comparison with a periodic checkpointing policy. In this policy the checkpoint schedule is computed using the optimal period obtained from Daly's work [8]. We notice that Daly's model inputs are the mean time between failures (MTTF) and the checkpoint cost (C). Hence, the MTTF computed using the client's failure density (MTTF=$\int_0^\infty t f(t) dt$). Then, for each job we consider the mean cost $C = \frac{1}{n}\sum_{i=1}^{n} c_i$. We note that the period obtained from this model denoted by ρ is a real value. Then the following mapping is used to compute the k^{th} checkpoint location with respect to ρ. Suppose that the $k-1^{th}$ checkpoint is scheduled after the j^{th} slice. Then, the k^{th} checkpoint is scheduled after the i^{th} where i minimize the following expression: $\min_i |\sum_{h=j+1}^{i} c_h - \rho|$.

We consider also the standard policy without checkpointing to illustrate the improvement of the proposed work.

To compute the optimal schedule with our model we consider that the quantum used to discretize the checkpoint cost is 1 minute. Then we consider that the re-execution factor $\alpha = 1$.

To compare policies we use two metrics. The first one is the maximum completion time over all the jobs in a given BoTs. The second one is the total wasted time (checkpoint overhead and lost computation) over the execution of the BoTs.

B. Simulation Results

In this analysis we propose to assess the considered fault tolerance policies under 3 various configuration for the client number and 6 different sizes of BoTs. Thus, in the first scenario the number of jobs in the BoTs is two times more than the number of clients. In the second scenario, the number of jobs is at most equal to the number of clients. For the last configuration the number of jobs is at most two times less than the number of clients. This, will reveal how the policies will perform with different configurations of duplication.

In the first set of simulations, we consider that the number of available client is 5000. Therefore, we suppose that the number of jobs of the considered BoTs varies in $\{1000, 3000, 5000, 7000, 9000, 10000\}$.

The simulation result of this configuration is depicted in Figures 1(a) and 1(b). Figure 1(a) reports the variation of the maximum completion time with respect to the variation of BoTs size. As it can be seen in these figures, the proposed checkpoint policy (PCkP) and the periodic checkpoint policy (Periodic) outperforms up to a factor 40% of the completion time when there is no fault tolerance policy (Without FT). Hence, this result shows the improvement of fault tolerance policy in such a platform.

We note also that the completion time of the proposed model is slightly better than the periodic policy. In Figure 1(b) we depict the average wasted time. The average wasted time is the total amount of wasted time (due to checkpoint overhead and lost computation) over the entire execution divided by the number of clients. In each bar, we report the amount of checkpoint overhead with a solid bar and the lost computation time with a dashed bar with respect to a given size of BoTs. This outcomes shows that the proposed model reduces by a factor of 2 the average wasted time when the number of clients is two times less that the number of jobs (BoTs = 10000).

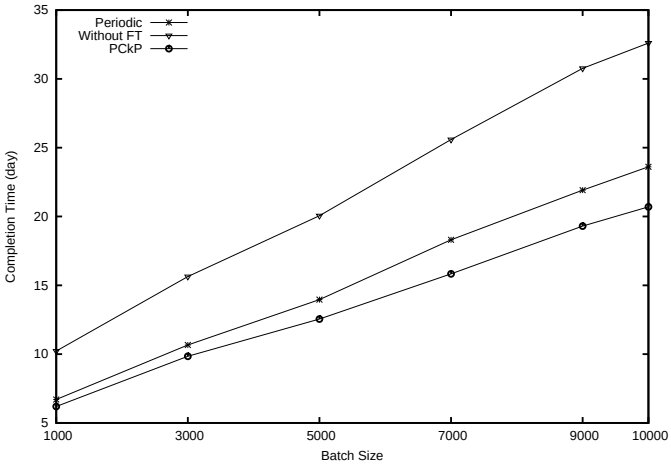
For the second configuration, we keep the same job number range and we increase the client number to 10000. The results of these experiments are shown in Figures 2(a),2(b). These results reveal also that the proposed policy outperforms both the the periodic and standard execution. The results in Figure 2(a) reveal that the improvement ratio of the checkpoint time is better than the previous one when the number of job is 10000. The results in Figure 2(b) reveal that the proposed model performs less checkpoints than the periodic policy at least by a factor 2. Thus, the proposed scheduling policy reduces the global completion time with less checkpoint overhead than the periodic policy. This issue could be explained by the fact that our model consider that

Table I
CLIENTS'S PROPERTIES

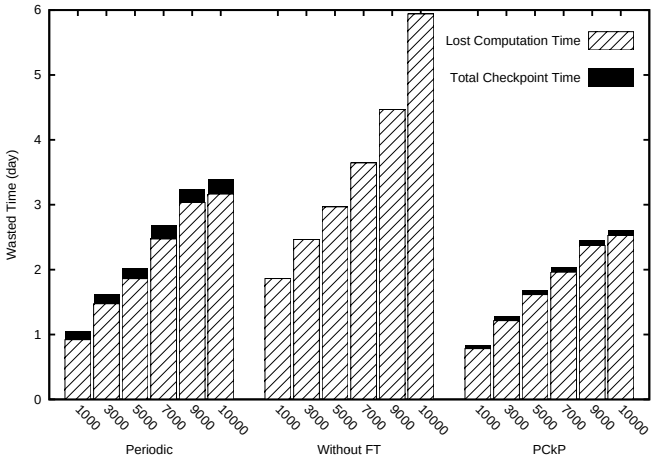
Total number of clients	Client's Cpu speed (MIPS)	Client's network speed (Mb/s)	Client's availability period (hours)	Client's unavailability period (hours)	Client's lifetime (days)
$\{5000, 10000, 20000\}$	Normal distribution $(mean = 1771, std = 669.5)$	Uniform distribution $[1, 2]$	Weibull distribution $(shape = 0.431, scale = 1.682)$	Hyper-exponential distribution $p_i \in \{0.398, 0.305, 0.298\}$, $\mu_i \in \{0.031, 11.566, 1.322\}$	Weibull distribution $(scale = 137,\ shape = 0.58)$

Table II
BOTS PROPERTIES

BoTs' size	Slices number per job	Instruction number per slice	Slices's checkpoint size (Mb)
$\{1000, 3000, 5000, 7000, 10000\}$	Uniform distribution $[1, 50]$	Uniform distribution $[36^9, 36^{10}]$	Uniform distribution $[10, 512]$

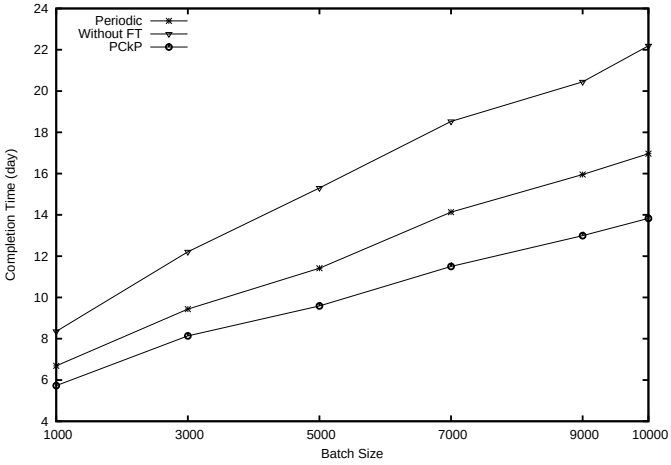


(a) BoTs Completion Time

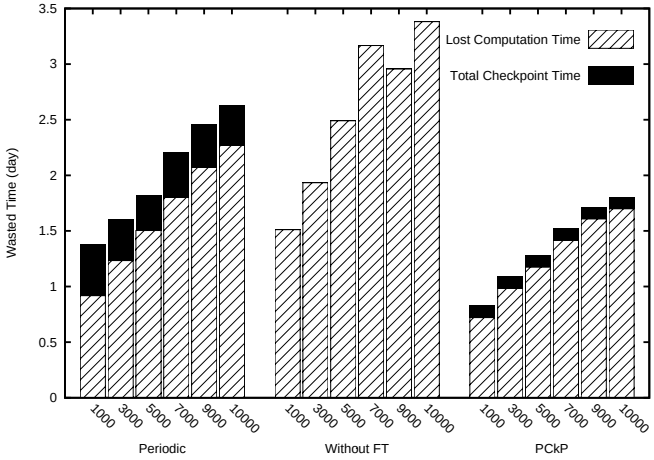


(b) Overall Wasted Time

Figure 1. The maximum completion time (a) and overall execution wasted time (b) with 5000 clients

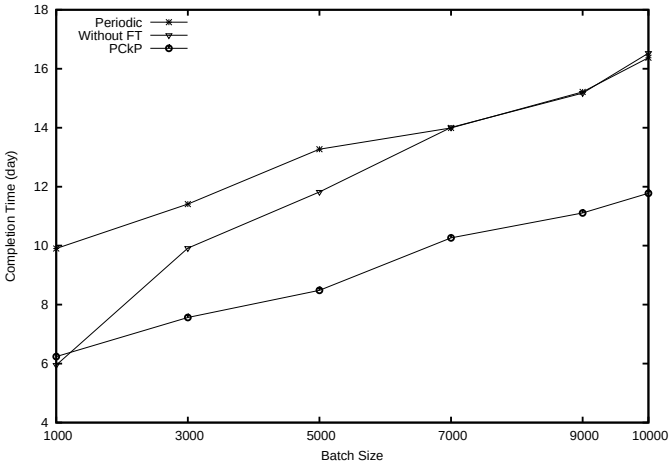


(a) BoTs Completion Time

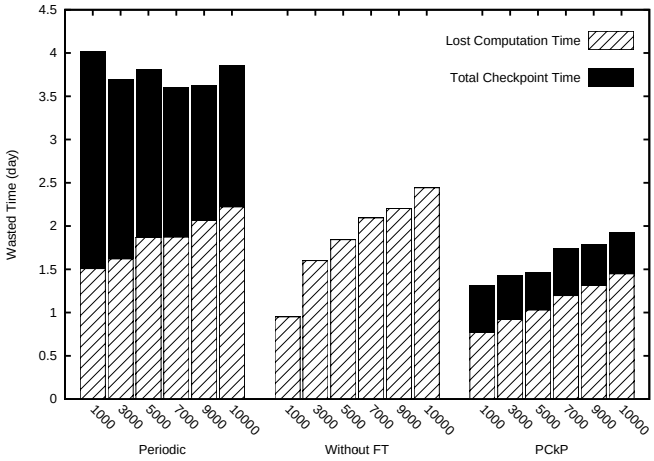


(b) Overall Wasted Time

Figure 2. The maximum completion time (a) and overall execution wasted time (b) with 10000 clients



(a) BoTs Completion Time



(b) Overall Wasted Time

Figure 3. The maximum completion time (a) and overall execution wasted time (b) with 20000 clients

slices could have different durations. Therefore the proposed schedule is more accurate than the periodic policy which ignore jobs' properties.

Finally, in the last scenario we consider that the number of clients is 20000 while we keep the same range for the BoTs size. For this simulation the maximum completion time is depicted in Figure 3(a). As shown, the proposed model still is the best among all the candidate policies. For the case where the number of jobs is 1000, the policy without checkpointing outperforms slightly the proposed model because of the congestion on the checkpoint server. In fact when the amount of work is small each job will be duplicated on many clients. Therefore, the congestion on the network link will be more important while we have many client that try to write the checkpoint for the same job which is useless. This simulation points out that when the number of clients increases, the standard policy without checkpointing outperforms the periodic policy in terms of completion time and wasted time. One possible explanation for this issue is the important amount of duplication. In conclusion, we note that a blind duplication policy can deteriorate system performances. Figure 3(b) confirms the previous remarks where the proposed model achieves better performances with less checkpoint overhead.

In summary, the proposed model outperforms both the periodic policy and the standard execution policy without fault tolerance in terms of completion time. Our model reduces the accumulated checkpoint overhead by considering the job duration variability. This set of simulations shows that a blind fault tolerance policy (checkpointing or duplication) leads to dramatic deterioration of the system performance.

V. Related Work

We discuss and contrast other works related to the scheduling of checkpoints. Young proposed a first order approximation to determine the optimal interval between checkpoints that minimizes the expected lost time before failure [21]. This result was established considering that checkpoints are periodically scheduled and failures arrivals follow a Poisson process. Daly [8] extended Young's result and proposed a higher order approximation under the same hypothesis.

Ling et al. [20] introduced a new variational technique that gives a first order approximation of the checkpoint frequency for minimizing the expected lost time before a failure. This method is based on the first order truncation of the Taylor expansion of the failure distribution under the hypothesis that checkpoints do not alternate the probability of failure of the application.

Notice that all these previous works consider that the checkpoint cost is constant. They provide different approximation solutions for determining the intervals between checkpoints under some restricted assumptions: infinite execution time, preemption, limited failure laws. Moreover, most of the proposed algorithms do not provide any guarantee on the time needed to reach a good solution.

In terms of checkpointing in real-world Desktop Grids such as BOINC [1] and XtremWeb [10], checkpointing is done only *locally* on the same disk as the executing client. The cost of local checkpoints, which are relatively small (less than 500 MB), is negligible. Checkpointing is done at the application-level (versus with system-level checkpointing libraries or with virtual machines). Applications checkpoint when it is convenient, often around every 1-10 minutes [4]

While checkpointing is not currently done remotely to the Desktop Grid server, several real projects (such as climateprediction.net [6] and Rechenkraft.net [18], and PrimeGrid [16]) have long-running jobs (on the order of days) that need remote checkpointing and migration. BOINC in particular does have the mechanisms to enable remote

checkpointing. For instance, the BOINC client has the ability to send "trickle" messages to the server during the execution of a job. Through these trickle messages, a checkpoint can be transmitted to the server.

VI. Conclusion and Future Work

A major impediment for batch job completion are resource failures. Checkpoint and migration can mitigate the affects of failures. We devise a checkpoint and migration strategy in the context of Desktop Grids, where checkpoints are sent to remote storage on a server. We prove that our strategy is optimal with a reasonable computational complexity. We show with simulations that our checkpointing strategy outperforms others in terms of batch completion time and waste.

For future work, we will investigate the case where congestion can occur on the network. One approach would be for the client to record its bandwidth for each upload or download. Then we could use this history to predict actual bandwidth during periods of possible congestion.

Acknowledgment

This work was carried out in part under the the ANR project Clouds@home (ANR-09-JCJC-0056-01). We thank Eric Heien for useful discussions.

References

[1] D. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[2] D. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *CCGRID*, pages 73–80, 2006.

[3] BOINC Papers. http://boinc.berkeley.edu/trac/wiki/BoincPapers.

[4] Catalog of boinc projects. http://www.boinc-wiki.info/Catalog_of_BOINC_Powered_Projects.

[5] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[6] C. Christensen, T. Aina, and D. Stainforth. The challenge of volunteer computing with lengthy climate model simulations. In *eScience*, 2007.

[7] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[8] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[9] T. Estrada, K. Reed, and M. Taufer. Modeling job lifespan delays in volunteer computing projects. In *Proceedings of to the 9th IEEE International Symposium on Cluster Computing and Grid (CCGrid)*, 2009.

[10] G. Fedak, C. Germain, V. N'eri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*, May 2001.

[11] E. Heien, D. Kondo, and D. Anderson. Correlated resource models of internet end hosts. In *31st International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[12] A. Iosup, C. Dumitrescu, D. H. J. Epema, H. Li, and L. Wolters. How are real grids used? the analysis of four grid traces and its implications. In *GRID*, pages 262–269, 2006.

[13] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Generation Comp. Syst.*, 24(7):672–686, 2008.

[14] B. Javadi, D. Kondo, JM. Vincent, and D.P. Anderson. Discovering statistical models of availability in large distributed systems: An empirical study of seti@home. *to appear in IEEE Trans. Parallel Distrib. Syst.*, 2010.

[15] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2003.

[16] PrimeGrid. http://primegrid.com.

[17] M.L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[18] Rechenkraft.net. http://rechenkraft.net.

[19] Boinc stats for seti. http://boincstats.com/stats/project_graph.php?pr=sah\&view=hosts.

[20] X.Lin Y.Ling, J.Mi. A variational calculus approach to optimal checkpoint placement. *IEEE Trans. Comput.*, 50(07):699–708, 2001.

[21] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.