Frontiers in Signal Processing, Vol. 1, No. 1, July 2017
https://dx.doi.org/10.22606/fsp.2017.11004

Adaptive and Power-Aware Resilience for Extreme-scale Computing

Xiaolong Cui, Taieb Znati and Rami Melhem

Department of Computer Science, University of Pittsburgh, Pittsburgh, United States
Email: {mclarencui,znati,melhem}@cs.pitt.edu

Abstract With the concerted efforts from researchers in hardware, software, algorithm, and data management, HPC is moving towards extreme-scale, featuring a computing capability of quintillion (10^{18}) FLOPS. As we approach the new era of computing, however, several daunting scalability challenges remain to be conquered. Delivering extreme-scale performance will require a computing platform that supports billion-way parallelism, necessitating a dramatic increase in the number of computing, storage, and networking components. At such a large scale, failure would become a norm rather than an exception, driving the system to significantly lower efficiency with unprecedented amount of power consumption.

To tackle this challenge, we propose an adaptive and power-aware algorithm, referred to as Lazy Shadowing, as an efficient and scalable approach to achieve high-levels of resilience, through forward progress, in extreme-scale, failure-prone computing environments. Lazy Shadowing associates with each process a "shadow" (process) that executes at a reduced rate, and opportunistically rolls forward each shadow to catch up with the its leading process during failure recovery. Compared to existing fault tolerance methods, our approach can achieve 20% energy saving with potential reduction in solution time at scale.

Keywords: Lazy Shadowing, extreme-scale computing, forward progress, reliability.

1 Introduction

The system scale needed to address our future computing needs will come at the cost of increasing complexity. As a result, the behavior of future computing and information systems will be increasingly difficult to specify, predict and manage. This upward trend, in terms of scale and complexity, has a direct negative effect on the overall system reliability. Even with the expected improvement in the reliability of future computing technology, the rate of system level failures will dramatically increase with the number of computing and storage components, possibly by several orders of magnitude. This can be clearly seen in Figure 1. A system with 200,000 nodes is expected to experience hundreds of failures every day, even when the Mean Time Between Failures (MTBF) of an individual node is as large as 5 years. How to achieve fault tolerance for such high level of failures with acceptable overhead is a significant challenge.

Another direct consequence of the increase in system scale is the dramatic increase in power consumption. Recent studies show a steady rise in system power consumption to 1-3MW in 2008, followed by a sharp increase to 10-20MW in subsequent years, with the expectation that power consumption could surpass 50MW by 2016 [1]. The US Department of Energy has recognized this trend and established a power limit of 20MW, challenging the research community to provide a 1000x improvement in performance with only a 10x increase in power [1]. This huge imbalance makes system power a leading design constraint in future extreme-scale computing infrastructure [2,3].

Fault tolerance and power management have been studied extensively, although only recently have researchers begun to study the combination of these two competing goals. In today's systems the response to faults mainly consists of restarting the application, including all related software components that have been affected by the fault. To avoid full re-execution of the original task, systems often checkpoint the execution periodically. Upon the occurrence of a hardware or software failure, recovery is then achieved by restarting the computation from a known checkpoint [4].

Coordinated Checkpoint is a method to achieve a globally consistent state. Specifically, all processes coordinate with one another to produce individual states that satisfy the "happens before" communication

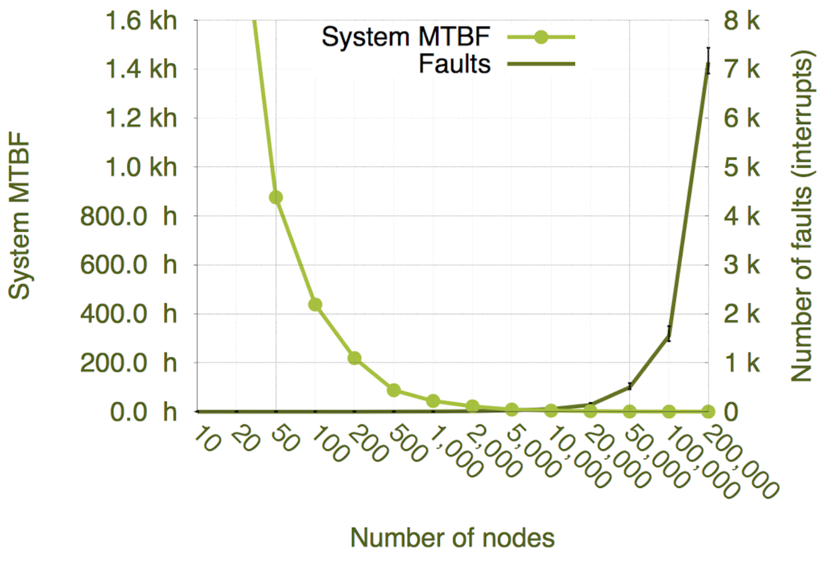


Figure 1. Impact of system size on system resilience.

relationship [5]. The major benefit of coordinated checkpointing stems from its simplicity and ease of implementation. Its major drawback, however, is the lack of scalability [6,7,8]. In uncoordinated checkpointing, processes record their states independently and postpone creating a globally consistent view until the recovery phase [9]. The major advantage is the reduced overhead during fault free operation. However, the scheme requires that each process maintains multiple checkpoints and message logs, necessary to construct a consistent state during recovery. It can also suffer the well-known domino effect, which may result in the re-execution of the entire application [10]. Although well-explored, uncoordinated checkpointing has not been widely adopted in HPC environments, due to its dependency on applications [11,12].

One of the largest overheads in any checkpointing process is the time necessary to write the checkpointing to stable storage, during which the application pauses its execution. Incremental checkpointing attempts to address this shortcoming by only writing the changes since the previous checkpoint [13,14]. This can be achieved using dirty-bit page flags [15,16]. Hash based incremental checkpointing, on the other hand, makes use of hashes to detect changes [13,14]. Another proposed scheme, known as copy-on-write, offloads the checkpointing process to a secondary task and only writes incremental checkpoints [17]. The main concern of these techniques is that some applications would require increase in their memory to support the simultaneous execution of the checkpointing and the application. It has been suggested that nodes in extreme-scale systems should be configured with fast local storage, which improves the performance of checkpointing [1]. Multi-level checkpointing, which consists of writing checkpoints to multiple storage targets, can benefit from such a strategy [18,19]. This, however, may lead to increased failure rates of individual nodes and increased per-node cost [20]. Furthermore, it may complicate the checkpoint writing process and requires that the system track the current location of all process's checkpoints.

Given the anticipated increase in system-level failure rates and the time required to checkpoint large-scale compute-intensive and data-intensive applications, it is predicted that, in extreme scale computing environments, the time required to periodically checkpoint an application and restart its execution will approach the system's MTBF. Consequently, applications will make little forward progress, thereby reducing considerably the overall performance of the system [7].

More recently, process replication, either fully or partially, has been proposed as an alternative to checkpointing, in order to scale to the resilience requirements of large distributed and mission-critical systems [21,7,22,23]. Based on this technique, each process is replicated across independent computing nodes. When the original process fails, one of the replicas takes over the computation task. Replication can be used to detect and correct system failures that are otherwise undetectable, such as silent data corruption [24] and Byzantine faults [21]. In addition, full and partial replication have also been used to augment existing checkpointing techniques, and to guard against silent data corruption [25,26]. There are several different implementations of replication in the widely used MPI library, each with their different tradeoffs and overheads. The overhead can be negligible or up to 70% depending upon the communication patterns of the application [27].

The large increase in number of components significantly increases the propensity of extreme-scale computing systems to faults, while driving power consumption to unforeseen heights. Unfortunately, in so far as performance is concerned, resilience to failures and adherence to power budget constraints are two conflicting objectives, as achieving high performance may push the system's components past their thermal limit and increase their likelihood of failure. In addition, abrupt and unpredictable changes in system behavior may lead to unexpected fluctuations in performance, which can be detrimental to applications' QoS requirements. The inherent vulnerability of extreme-scale computing systems, in terms of the envisioned high-rate and diversity of their faults, together with the demanding power constraints under which these systems will be designed to operate, calls for a radical reconsideration of the fault tolerance problem.

To this end, we propose an adaptive and power-aware resilience algorithm, referred to as *Lazy Shadowing*, as an efficient and scalable alternative to achieve high-levels of resilience, through forward progress, in extreme-scale, failure-prone computing environments. In the proposed approach, each process (referred to as main) is associated with a lazy replica (referred to as shadow) to improve resilience. The shadow executes the same code as its associated main process, but at a reduced CPU rate to save power and energy. Upon failure of the main process, the shadow increases its execution rate to complete the task, thereby reducing the impact of such a failure on the progress of the remaining tasks. Successful completion of the main process, however, results in immediate termination of the shadow. Since the failure rate of an individual component is much lower than that of the whole system, it is very likely that, most of the main processes complete their execution successfully. Consequently, the high probability that shadows never have to complete the full task, coupled with the fact that they initially only consume a minimal amount of energy, dramatically increases a power-constrained system's tolerance to failure, at a significantly reduced energy consumption.

Conceptually, Lazy Shadowing can be viewed as a class of stochastically competitive algorithms, which achieve high level of resilience with a small relative performance penalty. One gets different algorithms depending upon the main and shadow processes' execution rates, which are adaptive to the desired balance among three important objectives, namely, the expected completion time of the supported application, the power constraints imposed by the underlying computing infrastructure, and the tolerance of failure.

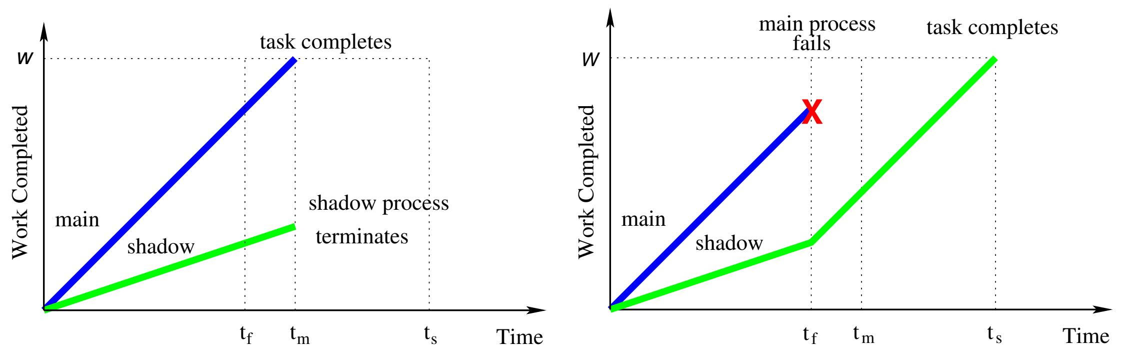
The main contributions of this paper are as follows:

– An adaptive, scalable, and power-aware algorithm, referred to as Lazy Shadowing, for fault tolerance in future extreme-scale, failure-prone computing systems.
– An evaluation framework composed of three analytical models to analyze the performance of Lazy Shadowing, compared to existing fault tolerance protocols.
– A thorough comparative study that shows the performance of Lazy Shadowing with different system characteristics and application requirements.

The rest of the paper is organized as follows. Section 2 introduces the Lazy Shadowing algorithm and Section 3 discusses how Lazy Shadowing can be used in extreme-scale computing environments for efficient fault tolerance. We then introduce three analytical models for performance assessment in Section 4, followed by experiments and evaluation in Section 5. Section 6 concludes this work and points out future research directions.

2 Lazy Shadowing

We use the term core to represent the computing resource allocation unit [28]. We further use $P(\sigma, w, t)$ to denote a process executing at rate σ to complete a workload w by time t. The basic tenet of Lazy Shadowing is the concept of shadowing, whereby each process is associated with a *lazy* replica. Assuming a single failure, Lazy Shadowing can be described as follows:



(a) Main process successful completion. (b) Main process failure.

Figure 2. Lazy Shadowing execution dynamics.

– A main process, $P_m(\sigma_m, w, t_m)$, that executes at the rate of σ_m to complete a task of size w at time t_m.
– A shadow process, $P_s(<\sigma_s^b\ , \sigma_s^a>, w, t_s)$, that executes at σ_s^b before the main process' failure, and σ_s^a after failure, to complete a task of size w at time t_s.

In order to deal with both permanent and temporary failures, the shadow process starts simultaneously with its associated main process, on a different node. Lazy Shadowing is able to tolerate any failure confined to a single node, including socket failure, CPU logical errors, bus errors, errors in the attached accelerators (e.g., GPUs), and even memory bit flips that exceed ECC's capacity.

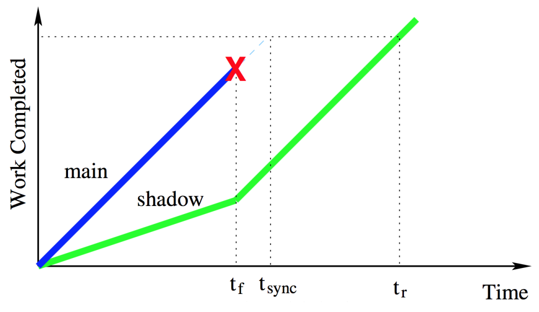
Initially, the main process executes at rate σ_m, while the shadow executes at rate $\sigma_s^b \le \sigma_m$. In the absence of failure, the main process completes execution at time $t_m = w/\sigma_m$, which immediately triggers the termination of the shadow. However, if at time $t_f < t_m$ the main process fails, the shadow, which has completed an amount of work $w_b = \sigma_s^b * t_f$, increases its execution rate to σ_s^a to complete the task by t_s. The execution dynamics of Lazy Shadowing are depicted in Figure 2.

The execution rates of the shadow, σ_s^b and σ_s^a, can be derived by balancing the trade-offs between completion time and energy consumption. For a delay-tolerant, energy-stringent application, σ_s^b is set to 0, and the shadow starts executing only upon failure of the main process. For a delay-stringent, energy-tolerant application, the shadow starts executing at $\sigma_s^b = \sigma_m$ to guarantee the completion of the task at the specified time t_m, regardless of when the failure occurs. For delay and energy tolerant applications, a broad spectrum of delay and energy tradeoffs can be explored either empirically or using optimization frameworks.

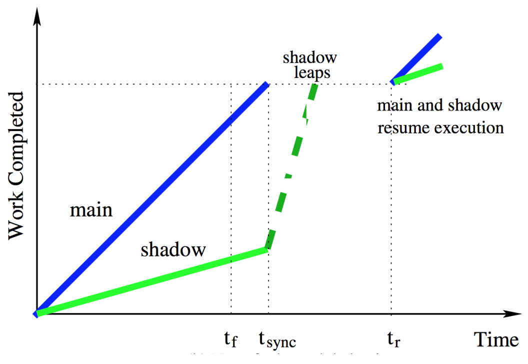
To control the shadow's execution rate, Dynamic Voltage and Frequency Scaling (DVFS), a commonly used power management technique, can be applied. The effectiveness of DVFS, however, may be markedly reduced in computational platforms that exhibit saturation of the processor clock frequencies, large static power consumption, or small power dynamic range. An alternative to DVFS is to collocate multiple processes on a single core, while keeping the core running at maximum rate. Time sharing can then be used to achieve the desired execution rates of the processes. Given our focus on extreme-scale, multi-core computing infrastructure, we will use collocation for execution rate control.

3 Extreme-scale Fault Tolerance

Lazy shadowing provides the basis for the design of efficient energy- and power-aware fault-tolerance approaches, that takes the energy-delay tradeoffs of the underlying application into consideration. In this section, we show how Lazy Shadowing can be used as the basic building block for the design of a resilience model for extreme-scale computing environments. The resulting model, referred to as *Leaping Shadows*, takes into consideration the main characteristics of compute-intensive and highly-scalable applications to achieve high-tolerance to failure, while reducing energy consumption, in extreme-scale and failure-prone



(a) Faulty task behavior.



(b) Non-faulty task behavior.

Figure 3. The concept of Leaping Shadows.

computing environments. In the following, we first introduce the execution and communication model of the targeted applications. We then describe the dynamics of the *Leaping Shadows* resilience model. Lastly, we discuss important aspects of its implementation.

3.1 System Model

We consider the class of compute-intensive and strongly-scaled applications, executing on a large-scale multi-core computing infrastructure [1]. Communication between cores is achieved using low-latency, high-bandwidth interconnect networks, such as Infiniband.

We use W to denote the size of an application workload, and assume that the workload is split arbitrarily into a set of tasks, τ, which execute in parallel and are synchronized using barriers. Given the prominence of MPI in HPC environments, we assume message passing as the communication model between tasks. Based on this model, each pair of communicating tasks is associated with a logical first-in-first-out (FIFO) channel, which guarantees ordered delivery of messages.

Assuming the maximum speed of a core is $\sigma_{max} = 1$, the failure-free completion time of the application is $w = W/|\tau|$, when all the tasks execute in parallel. When a failure occurs, the non-failing tasks continue executing until they reach their synchronization barrier. These tasks remain idle until the failure recovery is complete. The main objective of the Leaping Shadows resilience model is to minimize the idle time induced by failures, and ensure forward progress by rolling-forward the shadows whose associated mains are still alive, during the recovery phase. In Section 4.2, we will show that the impact of failures is well bounded as a result of forward leaping, even for tightly-coupled applications.

3.2 Leaping Shadows

The execution of an application can be carried out by simultaneously running all main and shadow processes for all the tasks. Let m_i denote the main process executing the i^{th} task $task_i$ of the application, and s_i its associated shadow. The task execution is divided into a set of execution phases, each extending over a synchronization interval. In the absence of failure, the behavior of a main and its leaping shadow is identical to the behavior of a main and its lazy shadow, depicted in Figure 2. Figure 3 shows the behavior of the main processes and their associated leaping shadows, assuming a failure occurrence at time t_f.

Figure 3(a) depicts the execution dynamics of the failing main and its associated shadow. Upon failure of the main process, the associated shadow continues execution, but at a higher rate, to minimize the impact of failure recovery on the application's progress. Figure 3(b) illustrates the behavior of the remaining main processes and their associated shadows. Unaware of the failed process, the remaining main processes continue executing until they reach their synchronization barriers at time t_{sync}. While the shadow of the failed main process progresses toward its synchronization barrier, all remaining main processes become idle until time t_r. The Leaping Shadows model opportunistically takes advantage of the

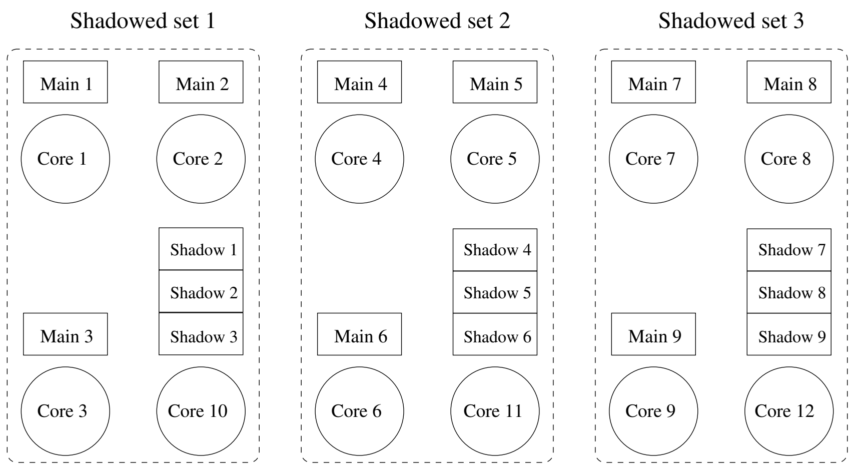


Figure 4. An example of collocation. $N = 12$, $M = 9$, $S = 3$, and $\alpha = 3$.

idle time, induced by failure recovery, to *leap forward* the shadows to their associated non-failing main processes. After the shadow of the failing main processes reaches its synchronization barrier, all processes resume execution from a consistent synchronization point. This process continues until the completion of all tasks. Forward leaping increases the shadow's rate of progress, at a minimal energy cost. Consequently, it reduces significantly the likelihood of a shadow falling excessively behind its associated main, thereby ensuring fast recovery, upon failure, while minimizing energy consumption.

Collocation is used to control the leaping shadows execution rates. To execute an application with M tasks, $M + S$ cores are required, where M is a multiple of S, all executing at the maximum rate. M of these cores are allocated to the main processes while the remaining S cores are shared among their associated shadows. Each main process is allocated one core, while $\alpha = M/S$ shadows are collocated on a single core. α is referred to as the main to shadow ratio. For example, if $M = 9$ and $S = 3$, then the 9 shadows share 3 cores, with every $\alpha = 3$ shadows collocated on each core, as shown in Figure 4.

Collocation of α shadows on a core has an important ramification with respect to the resilience of the system. Specifically, to speed up a shadow of a failed main to the maximum rate, all other collocated shadows must be terminated. Consequently, a second failure in any of the mains of the terminated shadows cannot be tolerated. In other words, the $M + S$ cores are grouped into S sets, which we call *shadowed sets*, each containing $\alpha + 1$ cores with α mains executing on α cores (referred to as main core) and their corresponding α shadows collocated on one core (referred to as shadow core). Each shadowed set can tolerate a failure in any of its cores, since failure of a main core would be tolerated by the shadow core and failure of the shadow core will not affect any main core. After the first failure in a shadowed set, the set is called *vulnerable* because it cannot tolerate another failure.

The basic steps describing the execution of an application, when using Leaping Shadows based on shadow collocation, are depicted in Algorithm 1. To use $M + S$ cores to execute the application, the total workload is split into M parallel tasks (line 1), which are then assigned to S shadowed sets, each with $\alpha = M/S$ cores for α main processes and 1 core for all the associated shadow processes (line 2). The execution starts by simultaneously running all the main and shadow processes (line 3). During the execution of the application, the system runs a failure monitor that triggers corresponding actions when a failure is detected (line 5 to 18). A failure may trigger different actions, depending on its type and precedence with respect to other failures. A shadowed set becomes *vulnerable* after the occurrence of the first failure in the set. A failure occurring in a vulnerable shadowed set (ss_j), results in an application failure. In response, the system terminates all running processes, initiates a recovery phase, either by rebooting or replacing failing cores, and restarts execution (line 8 to 10). On the other hand, failure in a non-vulnerable shadowed set can be tolerated and does not translate into an application failure. In this case, the shadowed set in question (ss_j) is marked as vulnerable (line 12). Note, however, that a failure of a main process has different impact from that of a shadow process. While a shadow failure does not impact the normal execution of the main processes, and thus can be ignored, failure of the main process forces the remaining main processes to suspend execution after they reach their synchronization point. The shadow process, s_k, associated with the failing process, m_k, becomes the primary process of the associated task and increases its execution to the maximum rate (line 14). This, in turn, forces the termination of all other collocated processes. Simultaneously, a forward leaping is undertaken by all remaining shadows to align their states with those of their associated mains (line 15). This process continues until the application is successfully completed.

```
input  : $W, M, S$
output : Application execution status
1  split $W$ into $M$ tasks;
2  assign $M$ tasks to $S$ shadowed sets;
3  start $m_i$ and $s_i$ for each $task_i$;
4  while execution not done do
5      if failure detected in $ss_j$ then
6          if $ss_j$ is vulnerable then
7              notify "Application failure";
8              terminate all mains and shadows;
9              repair all failures;
10             restart execution;
11         else
12             mark $ss_j$ as vulnerable;
13             if failure happened to $m_k$ then
14                 promote $s_k$ to $m_k$;
15                 perform forward leaping;
16             end
17         end
18     end
19 end
20 output "Application completes";
```
Algorithm 1: Leaping Shadows

3.3 Leaping Shadows Implementation Issues

State consistency is required both during normal execution and following a failure of a main process to roll-forward the state of the leaping shadows. During normal execution, shadows remain mute, in the sense that all outgoing messages from shadows are suppressed. A shadow process, however, will typically lag behind its main process during execution. Therefore, it is necessary to ensure that the shadow's state is consistent with that of its associated main, to successfully complete its associated task in case of failure. To this end, a message-logging protocol is used to ensure consistency [29]. These protocols typically use a minimum amount of meta-information to store and replicate the non-deterministic decisions in the execution of an application. These meta-data, also called determinants, are exchanged through system-level messages. To provide correct recovery after failure, a mechanism is required to guarantee that every shadow process follows the same computation and communication steps as its main process. After a main process m_i fails, s_i will take over m_i's role to recover from this failure. If there are other shadows sharing the same core with s_i, they will be terminated and s_i will start consuming the messages in its receiver-side message log at a faster speed. The message logging protocol will ensure that shadow s_i reaches a consistent state with the rest of the system.

Upon failure of a main processes, shadow processes will update their address space to "catch up" with their associated non-failing main processes. A technology, such as remote direct memory access (RDMA), can be used to roll-forward the state of the shadow to be consistent with that of its associated main. Rather than copying data to the buffers of the operating system, RDMA enables the network adapter to transfer data directly from the main process to its shadow. The zero-copy networking feature of RDMA considerably reduces latency, thereby enabling fast transfer of data between the main and its shadow.

4 Analytical Models

Three important metrics for assessing the quality of an application's execution, when Lazy Shadowing is used, are 1) the application failure probability, which measures the level of reliability; 2) expected completion time; and 3) expected energy consumption. In the following we develop mathematical models to analyze the expected performance of Lazy Shadowing, as well as proving the bound on the performance compared to non-failure case, with the understanding that process replication is a special case of Lazy Shadowing where $\alpha = 1$.

4.1 Application Failure Probability

Application failure, which forces the execution to start over, is inevitable even when every process is replicated. Lazy Shadowing is able to tolerate one failure in each shadowed set, and the second failure in any shadowed set implies the need to restarting the execution from the very beginning. However, Lazy Shadowing is orthogonal to checkpointing in the sense that we can combine the two, to avoid rolling the execution back to the very beginning when application failure occurs.

Since each process is replicated with a shadow, Lazy Shadowing has the potential to significantly increase the Mean Number of Failures To Interrupt (MNFTI), i.e., the average number of core failures until application failure occurs, and Mean Time To Interrupt (MTTI), i.e., the average time elapsed until application failure occurs. Therefore, the checkpointing interval should be increased to a large extent when checkpointing is combined with Lazy Shadowing. Furthermore, if the resulted checkpointing interval is larger than the completion time of the application, then checkpointing may not be used at all. In the following, the first question to study is the new MNFTI and MTTI when Lazy Shadowing is used.

The impact of process replication on MNFTI has been studied in [28]. Our problem is equivalent to that, with the difference that our work can tolerate one failure in each shadowed set while [28] can tolerate one failure in each replica-group of size 2. Therefore, we can apply the methodology in [28] to our case, and the MNFTI with Lazy Shadowing for different number of shadowed sets (S) is shown in Table 1. Note that when processes are not replicated, every failure would interrupt the application, i.e., MNFTI=1, so MNFTI can be significantly increased by Lazy Shadowing. At the same time, it is projected that an extreme-scale application's MTTI can be increased to tens of hours from minutes assuming each core's MTBF is 25 years.

Even though the above results imply that checkpointing may not be necessary when Lazy Shadowing is used, it is important to quantify the probability that an application failure would occur during the application's execution, defined as "application failure probability". Let $f(t)$ denote the failure probability density function of each core, and $F(t)$ be the corresponding cumulative distribution function, i.e., $F(t) = \int_0^t f(\tau)d\tau$ is the probability that a core fails in the next t time. Since each shadowed set can tolerate one failure, then the probability that a shadowed set with α main cores and 1 shadow core does not fail by time t is the probability of no failure plus the probability of one failure, i.e.,

$$P_g = \Big(1 - F(t)\Big)^{\alpha+1} + \binom{\alpha+1}{1} F(t) \times \Big(1 - F(t)\Big)^{\alpha} \tag{1}$$

and the probability that the application using N cores fails within t time is the complement of the probability that none of the shadowed sets fails, i.e.,

$$P_a = 1 - (P_g)^S \tag{2}$$

Table 1. Application's MNFTI when Lazy Shadowing is used. Results are independent of α.

S	2^0	2^1	2^2	2^3	2^4	2^5	2^6
MNFTI	3.0	3.7	4.7	6.1	8.1	11.1	15.2
S	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
MNFTI	21.1	29.4	41.1	57.7	81.2	114.4	161.4
S	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MNFTI	227.9	321.8	454.7	642.7	908.5	1284.4	1816.0

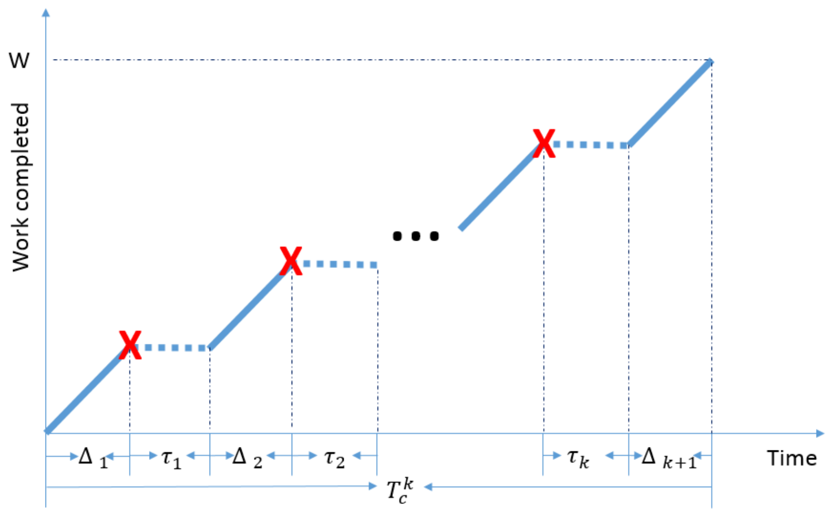


Figure 5. Illustration of application's progress with failure incurred delays.

where $S = \frac{N}{\alpha+1}$ is the number of shadowed sets.

Since the application only fails during its execution, we can calculate the application failure probability using t equal to the expected completion time of the application. We will develop the model for the expected completion time in the next subsection.

4.2 Expected Completion Time

One of the major performance metrics of interest to end-users, is the application's completion time. To evaluate this, we develop an analytical model for the expected completion time of Lazy Shadowing, with all probabilities of failures considered. For simplicity, we assume that no subsequent failure happens before the recovery of the previous failure.

First we discuss the case of k failures, which separate the execution into $k+1$ intervals. Denote by Δ_i $(1 \le i \le k+1)$ the i^{th} continuous execution interval, and τ_i $(1 \le i \le k)$ the recovery time after Δ_i. The application's progress with delay incurred by failures is illustrated in Figure 5.

Since Lazy Shadowing uses M cores for executing main processes and S cores for shadowing ($M + S = N$), the total workload W will be split into M tasks, each of which will be assigned a pair of main and shadow processes. Therefore, the workload of each process is $w = W/M$. The following theorem gives the expression for the completion time, T_c^k, when there are k failures.

Theorem 1. *If no subsequent failure happens before the recovery of the previous failure, then using Lazy Shadowing,*

$$T_c^k = w + (1 - \sigma_s^b) \sum_{i=1}^{k} \Delta_i$$

Proof. The recovery time τ_i is the time needed for the lazy shadow of the failed main to catch up. Under Leaping Shadows, it is guaranteed that all the shadows reach the same execution point as the mains (See Figure 3) after the previous recovery, so every recovery time is proportional to its previous continuous execution length, which is Δ_i. That is, $\tau_i = \Delta_i \times (1 - \sigma_s^b)$ (The value of Δ_i can be obtained given a failure probability distribution, as will be demonstrated in Section 5). The total delay induced by all k failures is $\sum_{i=1}^{k} \tau_i$. Since we assume there are k failures, then Δ_{k+1} is the failure free execution interval until the workload is complete, i.e., $\Delta_{k+1} = w - \sum_{i=1}^{k} \Delta_i$. Finally, according to Figure 5, the completion time with k failures is $T_c^k = \sum_{i=1}^{k+1} \Delta_i + \sum_{i=1}^{k} \tau_i = w + (1 - \sigma_s^b) \sum_{i=1}^{k} \Delta_i$. □

Although it may seem that the delay would keep deteriorating as the number of failures increases, it turns out to be well bounded, as a benefit of Leaping shadows:

Corollary 1.1. *The delay induced by failures is bounded by* $(1-\sigma_s^b)w$.

Proof. From above theorem we can see the delay from k failures is $(1-\sigma_s^b)\sum_{i=1}^{k}\Delta_i$. It is straightforward that, for any non-negative integer of k, we have the equation $\sum_{i=1}^{k+1}\Delta_i = w$. As a result, $\sum_{i=1}^{k}\Delta_i = w-\Delta_{k+1} \leq w$. Therefore, $(1-\sigma_s^b)\sum_{i=1}^{k}\Delta_i \leq (1-\sigma_s^b)w$, which means the delay is bounded by $(1-\sigma_s^b)w$. □

Typically, the number of failures to be encountered will not be known before the execution. Given a failure distribution, however, we can estimate the probability for a specific value of k. We assume that failures do not occur during recovery, so the failure probability of a core during the execution can be estimated as $P_c = F(w)$. Then the probability that there are k failures among the N cores during the execution is

$$P_s^k = \binom{N}{k} {P_c}^k (1-P_c)^{N-k} \tag{3}$$

The following theorem gives an expression for the expected completion time, T_{total}, considering all possible cases of failures.

Theorem 2. *If no subsequent failure happens before the recovery of the previous failure, then using Lazy Shadowing,* $T_{total} = T_c/(1-P_a)$, *where* $T_c = \sum_i T_c^i \cdot P_s^i$.

Proof. If application failure does not happen, the completion time considering all possible failures can be averaged as $T_c = \sum_i T_c^i \cdot P_s^i$. If application failure occurs, however, the application needs to restart from the beginning. Considering the possibility of re-execution, the total expected completion time is $T_{total} = T_c/(1-P_a)$. □

Process replication is a special case of Lazy Shadowing where $\alpha = 1$, so we can use the above theorem to derive the expected completion time for process replication:

Corollary 2.1. *The expected completion time for process replication is* $T_{total} = 2W/N/(1-P_a)$.

Proof. Using process replication, half of the available cores are dedicated to replicas so that the workload assigned to each task is significantly increased given the fixed number of cores available, i.e., $w = 2W/N$. Different from the case of $\alpha \geq 2$, failures do not incur any delay unless application failure occurs, since the replicas are executing at the same rate as the main processes. As a result, the completion time of process replication without application failure is constant with respect to the number of failures, i.e., $T_c^k = w = 2W/N$. Therefore, the completion time, T_c, considering all values of k, is also $2W/N$. Finally, the expected completion time considering the possibility of re-execution is $T_{total} = T_c/(1-P_a) = 2W/N/(1-P_a)$. □

A closer look at the above analysis one can realize that Lazy Shadowing has both advantage and disadvantage compared to traditional process replication. When collocating multiple shadow processes on each core, more than half of the available cores will be dedicated to main processes, leading to less workload per process. At the same time, collocation slows down the shadow processes, which incurs delays when failures occur. We will discuss more about the conflicting effects in Section 5.

4.3 Expected Energy Consumption

The power consumption of one core consists of two parts, dynamic power, p_d, which exists only when the core is executing, and static power, p_s, which is constant as long as the machine is on. This can be modeled as $p = p_d + p_s$. Note that in addition to CPU leakage, other components, such as memory and disk, also contribute to the static power consumption.

For process replication, all cores are running all the time until the application is complete. Therefore, the expected energy consumption, En, is proportional to the expected execution time T_{total}:

$$En = N * p * T_{total} \tag{4}$$

Although the failed components should not consume any power, we ignore this since the number of failures is negligible compared to the total number of cores.

Lazy Shadowing has the potential to save power compared to process replication, since main cores are idle during the recovery time after each failure, and the shadows can achieve forward progress through shadow leaping. During the normal execution time, all the cores are consuming static power as well as dynamic power. During recovery time, however, the main cores are idle and consume only static power, while the shadow cores perform shadow leaping, which may lead to higher dynamic power due to memory access and communication. After the leaping, the shadow cores become idle with no dynamic power consumption, until the failure recovery is completed. Again, we include the power consumption of the failed components. Altogether, the expected energy consumption for Lazy Shadowing can be modeled as

$$En = N * p_s * T_{total} + N * p_d * w + S * p_l * T_l. \tag{5}$$

with p_l denoting the dynamic power consumption of each core during shadow leaping and T_l denoting the expected total time spent on leaping during the execution of the application. Based on Equation 5 and Corollary 1.1, we can also establish an upper bound on the expected energy consumption for Lazy Shadowing:

Theorem 3. *If no subsequent failure happens before the recovery of the previous failure, then using Lazy Shadowing, the upper bound on expected energy consumption is* $(2N * p_s + N * p_d + S * p_l) * w$.

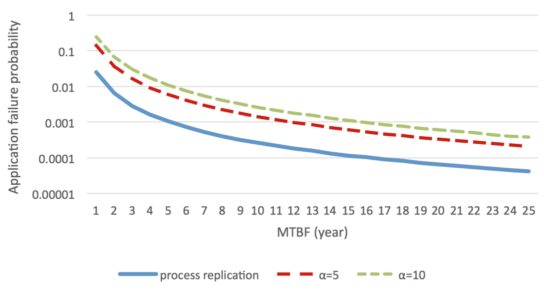
Proof. From Corollary 1.1 we know that the delay is at most $(1 - \sigma_s^b)w \leq w$, so $T_{total} \leq 2w$. Also, since the leaping time overlaps with the recovery time (delay), $T_l \leq (1 - \sigma_s^b)w \leq w$. Therefore, $En = N*p_s*T_{total}+N*p_d*w+S*p_l*T_l \leq N*p_s*(2w)+N*p_d*w+S*p_l*w = (2N*p_s+N*p_d+S*p_l)*w$. □

5 Evaluation

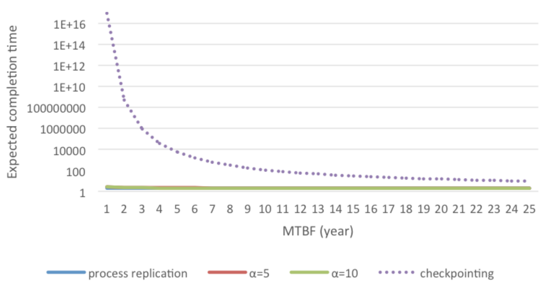
Careful analysis of the mathematical models above leads us to identify several important factors that influence the quality of an application's execution using Lazy Shadowing. These factors can be classified into three categories, i.e., system category, application category, and Lazy Shadowing category. The system category includes static power ratio, ρ ($\rho = p_s/p$), and MTBF of each core; the application category is mainly the total workload, W; and Lazy Shadowing category involves the number of cores to use, N, and main to shadow ratio, α, which together determine the number of main cores and shadow cores ($N = M + S$ and $\alpha = M/S$). In this section, we evaluate each performance metric of Lazy Shadowing, with the influence of each of the factors considered.

5.1 Comparison to Checkpointing and Process Replication

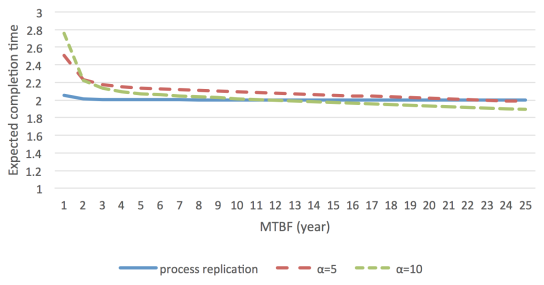
To study the efficiency of Lazy Shadowing in terms of reliability, completion time, and energy consumption, we compare with traditional process replication using the analytical models developed in Section 4. We also compare to checkpointing, of which the completion time is calculated with Daly's model [30] assuming the recovery time and checkpointing time are both 10 minutes, and then the energy consumption is derived using Equation 4. It is clear from THEOREM 1 that the total recovery delay $\sum_{i=1}^{k} \tau_i$ is determined by the execution time $\sum_{i=1}^{k} \Delta_i$, independent of the distribution of failures which determines the individual value of Δ_i. Therefore, our models are generic with no assumption about failure probability distribution, and the expectation of the total delay from all failures is the same as the failures are uniformly distributed [30]. Specifically, $\Delta_i = w/(k+1)$, and T_c^k can be rewritten as $w + w * (1 - \sigma_s^b) * \frac{k}{k+1}$ for Lazy Shadowing. Further, we assume that each shadow process gets a fair share of its shadow core's execution rate so that $\sigma_s^b = \frac{1}{\alpha}$. One may argue that the execution rate of the shadows should be degraded because of increased memory pressure or communication overhead. To take that into account, we have also studied the slowing down effect using a penalty model, to be discussed later. To calculate the expected energy consumption for Lazy Shadowing with Equation 5, we further assume that the dynamic power during shadow leaping is twice that during normal execution, i.e., $p_l = 2 * p_d$, and the time for shadow leaping through RDMA is half of the time required for recovery, i.e., $T_l = 0.5 * (T_{total} - w)$. The static power ratio ρ is fixed at 0.5 for now, other values will be explored later in this section.



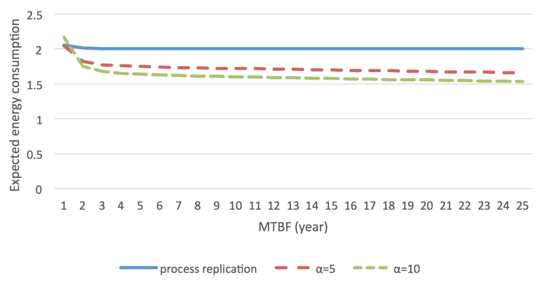
(a) Application failure probability



(b) Expected completion time



(c) Expected completion time with checkpointing removed



(d) Expected energy consumption with checkpointing removed

Figure 6. Comparison to process replication and checkpointing. $W = 10^6$ hours, $N = 10^6$, $\rho = 0.5$.

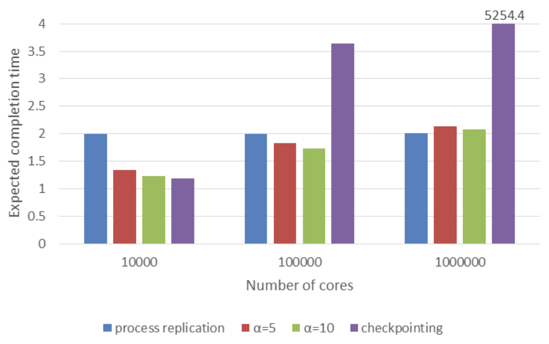
The first study uses $N = 1$ million cores, effectively simulating the future extreme-scale computing environment, and assumes that $W = 1$ million hours. For Lazy Shadowing we varied α from 1 to 10, with the understanding that it is unrealistic to collocate too many processes on a core. Besides $\alpha = 1$, which is equivalent to process replication, we only show $\alpha = 5$ and $\alpha = 10$ as others can be easily inferred from the figures.

By definition, the application failure probability for checkpointing is 0, as it periodically saves the execution states, from which the computation can be resumed upon failure. Figure 6(a) shows that due to collocation, the application failure probability increases for Lazy Shadowing. However, even with the lowest MTBF (1 year), Lazy Shadowing is still able to complete the application without re-execution with greater than 0.75 probability.

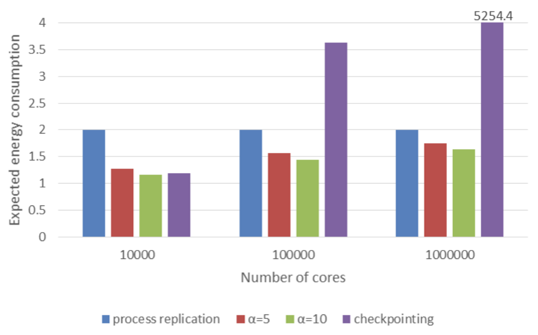
It is clear from Figure 6(b) that checkpointing incurs significant delay, indicating that it may not be a viable fault tolerance approach for future extreme-scale computing. The completion time and energy consumption of checkpointing are so large that the comparison between process replication and Lazy Shadowing is invisible. Therefore, we re-plot the comparison between Lazy Shadowing and process replication in Figure 6(c) and 6(d), with checkpointing removed. Figure 6(c) reveals that the most time efficient choice depends on MTBF. More specifically, process replication is more suited when MTBF is low while otherwise Lazy Shadowing is better. In terms of energy consumption, the advantage of Lazy Shadowing is much more obvious. For MTBF from 2 to 25 years, Lazy Shadowing with $\alpha = 5$ can achieve 9.6-17.1% energy saving, while the saving increases to 13.1- 23.3% for $\alpha = 10$. The only exception is when MTBF is extremely low (1 year), Lazy Shadowing with $\alpha = 10$ consumes more energy than process replication because of its higher probability of re-execution.

5.2 Impact of the Number of Cores

The system scale, measured in number of cores, has a direct impact on the failure rate seen by the application. To study its impact, we vary N from 10,000 to 1,000,000 with W scaled proportionally, i.e., $W = N$. When MTBF is 5 years, the results are shown in Figure 7. Please note that the time and

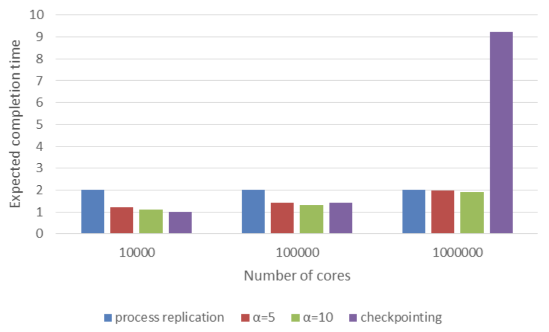


(a) Expected completion time

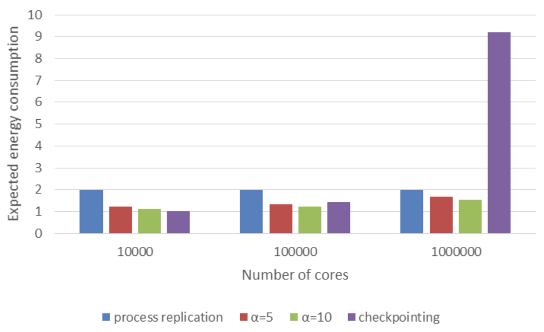


(b) Expected energy consumption

Figure 7. Sensitivity to number of cores. $W = N$, MTBF=5 years, $\rho = 0.5$.



(a) Expected completion time



(b) Expected energy consumption

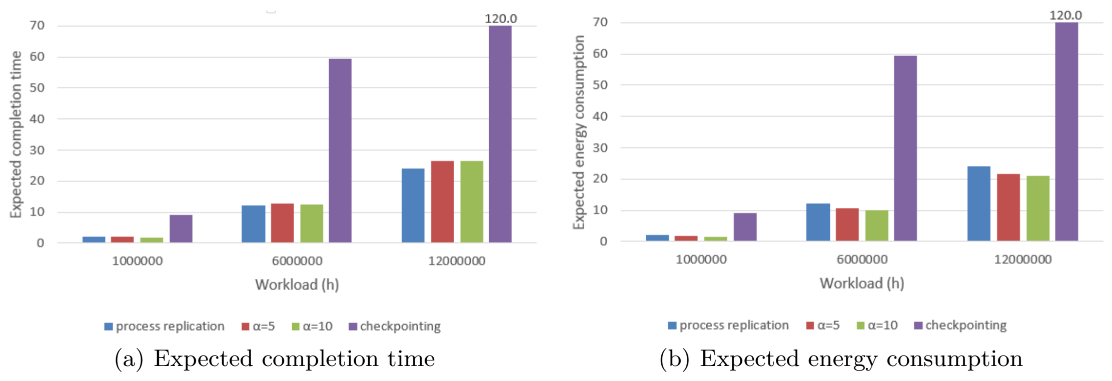
Figure 8. Sensitivity to number of cores. $W = N$, MTBF=25years, $\rho = 0.5$.

energy for checkpointing when $N = 1,000,000$ are beyond the scope of the figures, so we mark their values on top of their columns. When completion time is considered, Figure 7(a) clearly shows that each of the three fault tolerance alternatives has its own advantage. Specifically, checkpointing is the best choice for small systems with around 10,000 cores, Lazy Shadowing outperforms others for systems with around 100,000 cores, while process replication has slight advantage over Lazy Shadowing for larger systems. On the other hand, Lazy Shadowing wins for all system sizes covered in Figure 7(b), when energy consumption is considered. The comparison is different when MTBF is 25 years, as shown in Figure 8. Though improved to a large extent, the performance of checkpointing is still much worse than that of the other two approaches when N is 1,000,000. Lazy Shadowing also benefits from the increased MTBF, and further reduces its completion time and energy consumption for system sizes. In addition, Lazy Shadowing is able to achieve shorter completion time than process replication when N reaches 1,000,000.

5.3 Impact of Workload

To a large extent, workload determines the time exposed to failure. With other factors being the same, an application with a larger workload is likely to encounter more failures during its execution. Hence, it is intuitive that workload would impact the performance comparison among the three fault tolerance approaches.

Fixing N at 1,000,000, we increase W from 1,000,000 hours to 12,000,000 hours. Figure 9 assumes a MTBF of 25 years and shows both the time and energy. Checkpointing has the worst performance in both time and energy in all cases. In terms of completion time, process replication is more efficient than Lazy Shadowing when workload reaches 6,000,000 hours. Considering energy consumption, however,



(a) Expected completion time (b) Expected energy consumption

Figure 9. Sensitivity to workload. $N = 10^6$, MTBF=25 years, $\rho = 0.5$.

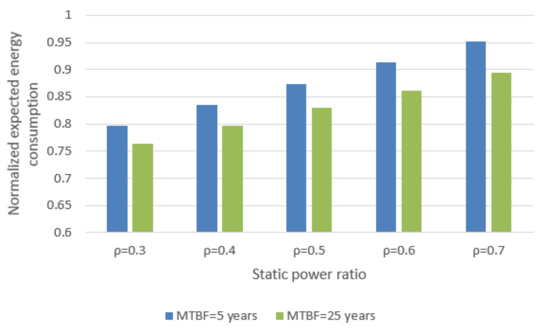


Figure 10. Impact of static power ratio on energy consumption normalized to process replication. $W = 10^6$ hours, $N = 10^6$, α=5.

Lazy Shadowing is able to achieve the most energy saving in all cases. When MTBF of 5 years is used, the difference is that process replication consumes less energy than Lazy Shadowing when W reaches 6,000,000.

5.4 Impact of Static Power Ratio

With various architectures and organizations, servers deployed at different supercomputers have different characteristics in terms of power consumption. The static power ratio ρ is used to abstract the amount of static power consumed versus dynamic power. In our models, ρ does not impact the completion time, but power and energy consumption. Considering modern systems, we vary ρ from 0.3 to 0.7 and study its effect on the expected energy consumption. The results for Lazy Shadowing with $\alpha = 5$ are normalized to that of process replication and shown in Figure 10. The results for other values of α have similar behavior and thus are not shown. Lazy Shadowing achieves more energy saving when static power ratio is low, since it saves dynamic power but not static power. When static power ratio is low ($\rho = 0.3$), Lazy Shadowing is able to save 20%-24% energy for the MTBF of 5 to 25 years. The saving decreases to 5%-11% when ρ reaches 0.7.

5.5 Adding Collocation Overhead

The last study is conducted to capture the impact on the performance of Lazy Shadowing brought by collocation overhead. We re-model the speed of shadows as $\sigma_s^b = \frac{1}{\alpha^{1.5}}$ to simulate the effect of memory thrashing and context switch. Figure 11 shows the impact of collocation overhead on expected energy consumption for Lazy Shadowing with $\alpha = 5$, with all the values normalized to that of process replication.

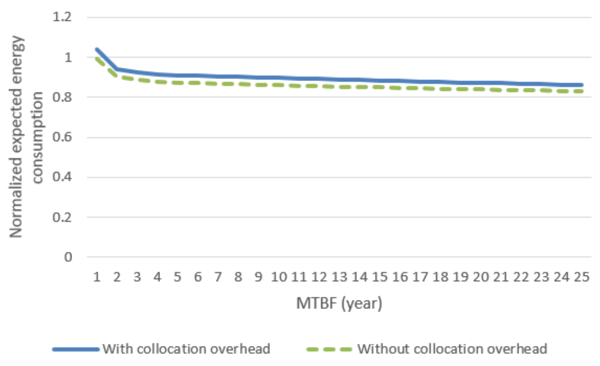


Figure 11. Impact of collocation overhead on normalized energy consumption. $W = 10^6$ hours, $N = 10^6$, ρ=0.5, α=5.

The results for other values of α have similar behavior and thus are not shown. As expected, energy consumption is penalized because of slowing down of the shadows. It is surprising, however, that the impact is negligible, with the largest difference being 4.4%. The reason is that Lazy Shadowing can take advantage of the recovery time after each failure and achieve forward progress for shadow processes that fall behind. When $\alpha = 10$, the largest difference further decreases to 2.5%.

6 Conclusion and Future Work

As the scale and complexity of HPC continue to increase, both the failure rate and energy consumption are expected to increase dramatically, making it extremely challenging to deliver extreme-scale computing performance with satisfactory efficiency and reliability. Existing fault tolerance methods rely on either time or hardware redundancy. Neither of them appeal to the next generation of supercomputing, as the first approach may incur significant delay while the second one constantly wastes over 50% of the system resources. The need for an efficient and reliable solution in extreme-scale, failure-prone computing environments calls for a reconsideration of the fault tolerance problem.

To this end, we propose an adaptive and power-aware algorithm, referred to as Lazy Shadowing, as an efficient and scalable alternative to achieve high-levels of fault tolerance for future extreme-scale computing. In this paper, we present a comprehensive discussion of the techniques that enable Lazy Shadowing. In addition, we develop a series of mathematical models to assess its performance in terms of reliability, completion time, and energy consumption. Through comparison with existing fault tolerance approaches, we identify the scenarios where each of the alternatives should be chosen. Specially, checkpointing consumes the least time and energy when system size is small; process replication should be used to minimize completion time when system size is extremely large and failure rate is extremely high; and Lazy Shadowing is the choice for all other cases, for the consideration of both completion time and energy consumption.

In the future, we will generalize our approach to multiple jobs. Initially we will assume that each job has an arrival time a_j, a workload w_j, and a deadline d_j by which it must complete, and will seek minimally adaptive stochastically competitive strategies. The strategies in the case of no failure in [31] are a good staring point. Another future direction is to apply our approach to servers with inter-processor power heterogeneity. In particular, it is likely that a chip will contain a few high-rate power-hungry processors, many low-rate power-efficient processors, and possibly an intermediate number of processors with intermediate rate and power efficiency. Generally speaking, inter-processor power heterogeneity is much harder to handle theoretically than intra-processor power heterogeneity. The main reason is that the number of jobs that can be run at high speed is fixed, and thus some jobs cannot be guaranteed to finish by their deadlines. A solution would be to allow a job to miss its deadline with some penalty, and to consider the objective as energy plus penalty. Again we seek to find stochastically competitive algorithms, and a good starting point may be the algorithms that are known to be competitive without faults [32].

References

1. S. Ahern and et. al., "Scientific discovery at the exascale, a report from the doe ascr 2011 workshop on exascale data management, analysis, and visualization," 2011.
2. O. Sarood and et. al., "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," ser. SC '14, Piscataway, NJ, USA, 2014, pp. 807–818. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.71
3. O. Villa and et. al., "Scaling the power wall: A path to exascale," ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 830–841. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.73
4. E. Elnozahy and et. al., "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.
5. K. Chandy and C. Ramamoorthy, "Rollback and recovery strategies for computer programs," *Computers, IEEE Transactions on*, vol. C-21, no. 6, pp. 546–556, June 1972.
6. E. Elnozahy and J. Plank, "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery," *DSC*, vol. 1, no. 2, pp. 97 – 108, april-june 2004.
7. R. Riesen, K. Ferreira, J. R. Stearley, R. Oldfield, J. H. L. III, K. T. Pedretti, and R. Brightwell, "Redundant computing for exascale systems," December 2010.
8. P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.
9. J. Plank and M. Thomason, "The average availability of parallel checkpointing systems and its importance in selecting runtime parameters," in *Fault-Tolerant Computing*, 1999, pp. 250 –257.
10. B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449. [Online]. Available: http://doi.acm.org/10.1145/800027.808467
11. G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Cluster Computing, 2004 IEEE International Conference on*, Sept 2004, pp. 93–103.
12. A. Guermouche and et. al., "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, May 2011, pp. 989–1000.
13. H. chang Nam, J. Kim, S. Lee, and S. Lee, "Probabilistic checkpointing," in *In Proceedings of Intl. Symposium on Fault-Tolerant Computing*, 1997, pp. 153–160.
14. S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *ICS*, St. Malo, France, 2004.
15. J. Plank and K. Li, "Faster checkpointing with n+1 parity," in *Fault-Tolerant Computing*, June 1994, pp. 288–297.
16. E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *TC*, vol. 41, pp. 526–531, 1992.
17. K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 8, pp. 874–879, Aug. 1994. [Online]. Available: http://dx.doi.org/10.1109/71.298215
18. A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18
19. D. Hakkarinen and Z. Chen, "Multilevel diskless checkpointing," *Computers, IEEE Transactions on*, vol. 62, no. 4, pp. 772–783, April 2013.
20. F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," ser. ICS, New York, USA, 2011, pp. 22–32. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995902
21. D. Fiala and et. al., "Detection and correction of silent data corruption for large-scale high-performance computing," ser. SC, Los Alamitos, CA, USA, 2012, pp. 78:1–78:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389102
22. A. Lefray, T. Ropars, and A. Schiper, "Replication for send-deterministic MPI HPC applications," ser. FTXS '13. New York, NY, USA: ACM, 2013, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/2465813.2465819
23. F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *IJHPCA*, vol. 23, no. 3, pp. 212–226, 2009.
24. X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," ser. SC. New York, NY, USA: ACM, 2013, pp. 7:1–7:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503266
25. J. Stearley and et. al., "Does partial replication pay off?" in *DSN-W*, June 2012, pp. 1–6.
26. J. Elliott and et. al., "Combining partial redundancy and checkpointing for HPC," ser. ICDCS. Washington, DC, USA: IEEE Computer Society, 2012, pp. 615–626. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2012.56

27. C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *PDCN*, 2011, pp. 15–17.
28. H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Combining Process Replication and Checkpointing for Resilience on Exascale Systems," INRIA, Rapport de recherche RR-7951, May 2012. [Online]. Available: http://hal.inria.fr/hal-00697180
29. L. Alvisi and K. Marzullp, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 149–159, 1998.
30. J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2004.11.016
31. S. Albers, A. Antoniadis, and G. Greiner, "On multi-processor speed scaling with migration: Extended abstract," ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 279–288. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989539
32. P. Kling and P. Pietrzyk, "Profitable scheduling on multiple speed-scalable processors," ser. SPAA '13. New York, NY, USA: ACM, 2013, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/2486159.2486183