Adaptive and Power-Aware Resilience for Extreme-scale Computing

Xiaolong Cui, Taieb Znati, Rami Melhem
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
Email: {mclarencui, znati, melhem}@cs.pitt.edu

***Abstract*—With 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 these challenges, 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 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;**

I. 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 components, possibly by several orders of magnitude [1].

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, with the expectation of surpassing 50MW soon [2]. The US Department of Energy has recognized this trend and set a power limit of 20MW, challenging the research community to provide a 1000x improvement in performance with only a 10x increase in power [2]. This huge imbalance makes system power a leading design constraint in future extreme-scale computing infrastructure [3], [4].

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. To avoid full re-execution, 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. 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 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 efficiency of the system [1].

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 [5]. 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 requires at least doubling the number of computing resources, since each process must have at least one replica, thereby limiting the system efficiency to at most 50%.

There is a delicate interplay between fault tolerance and energy consumption. Checkpointing and replication require additional energy to achieve fault tolerance. Conversely, it has been shown that lowering supply voltages, a commonly used technique to conserve energy, increases the probability of transient faults [6]. The tradeoffs between fault free operation and optimal energy consumption has been explored in the literature. Limited insights have emerged, however, with respect to how adherence to application's desired QoS requirements affects and is affected by the fault tolerance and energy consumption dichotomy. For example, it has been shown that, at scale, coordinated and uncoordinated checkpointing may lead to cascading delays. 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 instability 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 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 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.

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 and power-aware algorithm for fault tolerance in future extreme-scale computing systems.
- An analytical model based evaluation framework for performance assessment.
- A thorough comparative study that explores the performance of Lazy Shadowing with different system characteristics and application requirements.

The rest of the paper is organized as follows. We begin with a survey on related work in Section II. Section III introduces the basic Lazy Shadowing algorithm and Section IV discusses how Lazy Shadowing can be applied to extreme-scale computing environments. We then introduce three analytical models for performance assessment in Section V, followed by experiments and evaluation in section VI. Section VII concludes this work.

II. RELATED WORK

Rollback and recovery is the predominate mechanism to achieve fault tolerance in current HPC environments. In the most general form, rollback and recovery involves the periodic saving of the current system state, with the anticipation that in the case of a failure, computation can be restarted from the most recently saved state [7]. Coordinated checkpointing is a popular approach for its ease of implementation. Its major drawback, however, is the lack of scalability, as it requires global coordination [8], [9].

In uncoordinated checkpointing, processes record their states independently and postpone creating a globally consistent view until the recovery phase. 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 [10]. One hybrid approach, known as communication induced checkpointing, aims at reducing coordination overhead [11]. The approach, however, may cause processes to store useless states. To address this shortcoming, "forced checkpoints" have been proposed [12]. This approach, however, may lead to unpredictable checkpointing rates. Although well-explored, uncoordinated checkpointing has not been widely adopted in HPC environments, due to its dependency on applications [13].

One of the largest overheads in any checkpointing process is the time necessary to write the checkpointing to stable storage. Incremental checkpointing attempts to address this by only writing the changes since previous checkpoint [14], [15], [16]. Another proposed scheme, known as in-memory checkpointing, minimizes the overhead of disk access [17]. The main concern of these techniques is the increase in memory requirement 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 [2]. Multi-level checkpointing, which consists of writing checkpoints to multiple storage targets, can benefit from such a strategy [18]. This, however, may lead to increased failure rates of individual nodes and complicate the checkpoint writing process.

Our work is based on process replication, or state machine replication, which has long been used to provide fault tolerance in distributed and mission critical systems[19]. Recently, replication has been proposed as a viable alternative to checkpointing in HPC [1], [20], [5]. In addition, full and partial replication have also been used to augment existing checkpointing techniques, and to guard against silent data corruption [21], [22].The most relevant works to ours is redundant multi-threading (RMT) whereby one leading thread of execution is running ahead of trailing threads [23], [24]. However, our approach is different in that it tunes the execution rates of the leading and trailing

threads in a finer grain, in order to achieve a "parameterized" trade-off between completion time and energy consumption. Further, we take advantage of the idle time during failure recovery and "leap" the trailing threads to achieve forward progress. This differs from RMT, of which the "leaping" of the trailing thread results in extra overhead.

III. Lazy Shadowing

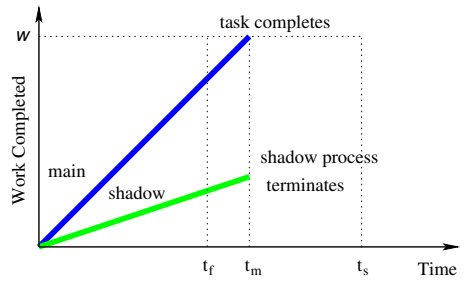
The basic tenet of Lazy Shadowing is the concept of shadowing, whereby each process is associated with a *lazy* replica that executes at a reduced rate. We only use one replica since the probability that failure occurs to both the original process and the replica is negligible [25]. If necessary, however, this can be easily extended to use a suit of replicas. Assuming a single failure, Lazy Shadowing can be described as follows:

- 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 initially executes at σ_s^b, and increases to σ_s^a if its main process fails, to complete a task of size w at time t_s.

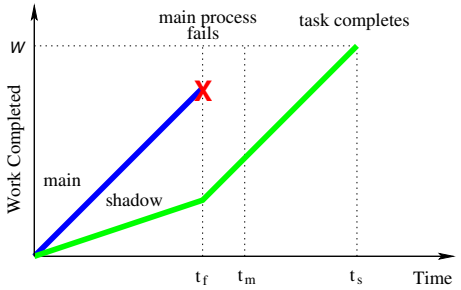
We use the term core to represent the resource allocation unit (e.g., a CPU core, a multi-core CPU, or a cluster node), so that our algorithm is agnostic to the granularity of the hardware platform [25]. Furthermore, we adopt the fail-stop failure model, where a core stops execution once a failure occurs and failure can be detected by other processes [26], [27]. 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 $\sigma_s^b \leq \sigma_m$. 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 are depicted in Figure 1.

In HPC, throughput consideration requires that the rate of the main process, σ_m, be set to the maximum. The execution rates of the shadow, σ_s^b and σ_s^a, however, can be derived by balancing the tradeoffs between delay and energy. 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. Therefore, Lazy Shadowing has the flexibility to converge to checkpointing/restart or process replication, if preferred. In addition, a broad spectrum of



(a) Main process successful completion.



(b) Main process failure.

Figure 1: Lazy Shadowing execution dynamics.

delay and energy tradeoffs in between can be explored either empirically or by using optimization frameworks for delay and energy tolerant applications.

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 is to collocate multiple processes on a single core, while keeping the core 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.

To execute an application with M tasks, $N = M + S$ cores are required, where M is a multiple of S. Each main process is allocated one core (referred to as main core), while $\alpha = M/S$ shadows are collocated on a core (shadow core). The N cores are grouped into S sets, which we call *shadowed sets*, each containing α main cores and 1 shadow core. This is illustrated in Figure 2. Collocation has an important ramification with respect to the resilience of the system. Specifically, one failure can be tolerated in each shadowed set. If a shadow core fails, the main processes can continue execution, but will have no shadows any more. On the other hand, 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

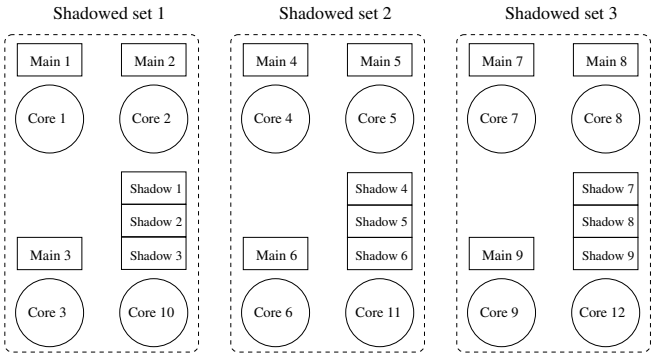


Figure 2: An example of collocation. $N = 12$, $M = 9$, $S = 3$.

mains in the shadowed set cannot be tolerated. After the first failure, a shadowed set becomes *vulnerable*[1]. As will be shown in Section V-A and Section VI, this should not be a concern as the provided reliability is more than enough.

IV. Extreme-scale fault tolerance

Lazy Shadowing provides the basis for the design of efficient energy- and power-aware fault-tolerance solutions for extreme-scale computing environments. Taking into consideration the main characteristics of compute-intensive and highly-scalable applications, we design a novel technique, referred to as *leaping shadows*, in order to achieve high-tolerance to failure while minimizing delay and energy consumption. In the following, we first introduce the execution and communication model of the targeted applications. We then describe the dynamics of Lazy Shadowing when boosted with leaping shadows. Lastly, we discuss important aspects of its implementation.

A. Application model

We consider the class of compute-intensive and strongly-scaled applications, executing on a large-scale multi-core computing infrastructure [2]. 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 mechanism between tasks.

The execution of an application can be carried out by simultaneously running all tasks, each with a pair of main and shadow processes. Let m_i denote the main process executing the i^{th} task $task_i$, and s_i its associated shadow. The execution is divided into a set of phases by synchronization barriers. Assuming the maximum rate of a core is $\sigma_{max} = 1$, the failure-free completion time of the application is $W/|\tau|$. Failures, however, will incur delay as the shadows execute at a lower rate. This problem deteriorates sharply because synchronization barriers would propagate the delay of one task to other tasks of the application. The main objective of leaping shadows is to minimize the delay induced by failures, and ensure forward progress by opportunistically rolling-forward the shadows during failure recovery. In Section V-B, we will show that the delay is well bounded even for tightly-coupled applications.

[1]Rejuvenation techniques, such as restarting the lost shadows from the state of current mains on spare cores, can be used to eliminate vulnerability.

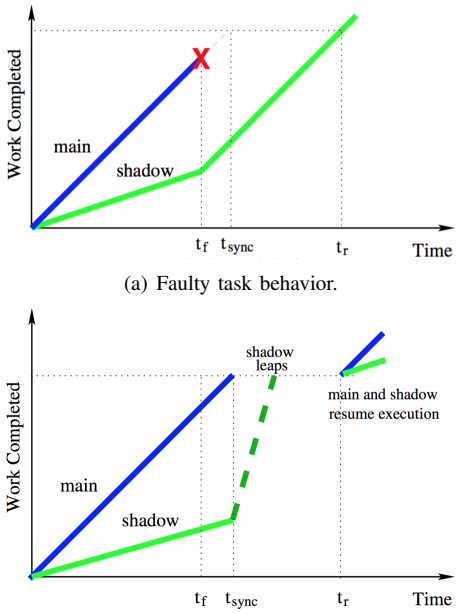


Figure 3: The illustration of shadow leaping.

B. Leaping shadows

In the absence of failure, the behavior of a main and its leaping shadow is identical to the behavior depicted in Figure 1. Assuming a failure occurrence at time t_f, Figure 3 shows the concept of shadow leaping. Upon failure of a main process, its associated shadow speeds up to minimize the impact of failure recovery on the application's progress, as illustrated in Figure 3(a). At the same time, as shown in Figure 3(b), the remaining main processes continue execution towards the barrier at time t_{sync}, and then become idle until t_r, when the shadow of the failed main process also reaches the barrier. Leaping shadows opportunistically takes advantage of this idle time to *leap forward* the shadows by copying state from their main processes, so that all processes, including shadows, can resume execution from a consistent synchronization point afterwards. 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, thereby ensuring fast recovery while minimizing energy consumption.

The steps of applying Lazy Shadowing with leaping

```
input  : W, M, S
output: Application execution status
1  split W into M tasks;
2  assign M tasks to S shadowed sets;
3  start a pair of main and shadow for each task;
4  while execution not done do
5      if failure detected in a shadowed set then
6          if the shadowed set is vulnerable then
7              notify "Application failure";
8              terminate all mains and shadows;
9              repair all failures;
10             restart execution;
11         else
12             mark the shadowed set as vulnerable;
13             if failure happened to a main then
14                 promote its shadow to new main;
15                 perform shadow leaping;
16             end
17         end
18     end
19 end
20 output "Application completes";
```

Algorithm 1: Lazy Shadowing

shadows are depicted in Algorithm 1. To use $M+S$ cores to execute an 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, the system runs a failure monitor (out of scope of this work) that triggers corresponding actions when a failure is detected (line 5 to 18). A failure occurring in a vulnerable shadowed set (e.g., ss_j) results in an application failure and forces a re-execution (line 7 to 10). On the other hand, failure in a non-vulnerable shadowed set does not translate into an application failure, but would mark the shadowed set in question as vulnerable (line 12). In this case, failure of a main process has different impact from that of a shadow process. While a shadow failure does not impact the normal execution and thus can be ignored, failure of a main process (e.g., m_k) triggers promotion of its shadow process, s_k, to a new main process (line 14). Simultaneously, a shadow leaping is undertaken by all remaining shadows to align their states with those of their associated mains (line 15). This process continues until all tasks of the application are successfully completed.

C. Implementation issues

We implemented an Open MPI based prototype of Lazy Shadowing, which can be used to execute existing HPC workloads without any change of user code. Since the focus of this paper is to introduce algorithmic perspectives of the Lazy Shadowing paradigm by discussing novel concepts of shadow collocation and forward leaping, we only give a brief discussion of the implementation issues.

State consistency is required both during normal execution and following a failure of a main process to roll-forward the 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 this end, a message-logging protocol is used, which typically uses a minimum amount of meta-information to store and replicate the non-deterministic decisions [28].

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 process, 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 allows to transfer data directly from the main process to its shadow. The zero-copy feature of RDMA considerably reduces latency, thereby enabling fast transfer of data between the main and its shadow.

V. ANALYTICAL MODELS

Three important metrics for assessing the quality of an application's execution are 1) reliability; 2) completion time; and 3) energy consumption. In the following we develop mathematical models to analyze the expected performance of Lazy Shadowing, as well as prove the bound on performance compared to non-failure case, with the understanding that process replication is a special case of Lazy Shadowing where $\alpha = 1$. All the analysis below is under the assumption that there are a total of N cores, and W is the application workload. M of the N cores are allocated for main processes, each having a workload of $w = \frac{W}{M}$, and the rest S cores are for collocated shadow processes. For process replication, $M = S = \frac{N}{2}$, and $w = \frac{2W}{N}$.

A. Application failure probability

An application has to start over when one of its tasks has permanently lost its work. We call this an application failure,

which is inevitable even when every process is replicated. Lazy Shadowing is able to tolerate one failure in each shadowed set. 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.

Since each process is replicated with a shadow, Lazy Shadowing has the potential to significantly increase the Mean Number of Failures To Interrupt (MNFTI). The impact of process replication on MNFTI has been studied in [25]. Applying the same methodology, we can derive the new MNFTI with Lazy Shadowing for different number of shadowed sets (S), as shown in Table I. 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. It is projected that the Mean Time To Interrupt (MTTI) of an extreme-scale application can be increased to tens of hours from minutes assuming each core's MTBF is 25 years.

The level of reliability can be quantified by calculating the probability of application failure. Let $f(t)$ denotes the failure probability density function of each core, then $F(t) = \int_0^t f(\tau)d\tau$ 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.,

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

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.,

$$P_a = 1-(P_g)^S \quad (2)$$

where $S = \frac{N}{\alpha+1}$ is the number of shadowed sets. The application failure probability can then be calculated by using t equal to the expected completion time of the application, which will be modeled in the next subsection.

B. Expected completion time

An analytical model is developed for the expected completion time. For simplicity, we assume that no subsequent failure happens during 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 4.

Table I: Application's MNFTI when Lazy Shadowing is used. Results are independent of $\alpha = \frac{M}{S}$.

S	2^2	2^4	2^6	2^8	2^{10}
MNFTI	4.7	8.1	15.2	29.4	57.7
S	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
MNFTI	114.4	227.9	454.7	908.5	1816.0

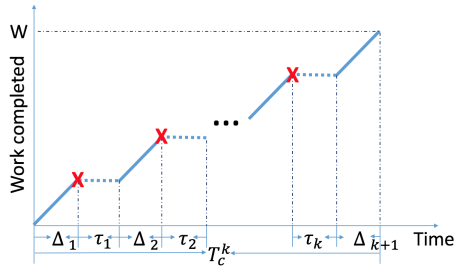


Figure 4: Application's progress with failure incurred delays.

The following theorem expresses the completion time, T_c^k, as a function of k.

Theorem 1: Assuming that no failure occurs during the recovery of a 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. Lazy Shadowing guarantees 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 VI). The total delay induced by all k failures is $\sum_{i=1}^{k}\tau_i$. Finally, according to Figure 4, 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 growing as the number of failures increases, it turns out to be well bounded, as a benefit of forward leaping:

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, $\sum_{i=1}^{k}\Delta_i = w - \Delta_{k+1} \le w$. Therefore, $(1-\sigma_s^b)\sum_{i=1}^{k}\Delta_i \le (1-\sigma_s^b)w$. ■

Typically, the number of failures to be encountered will not be known. Given a failure distribution, however, we can calculate 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 is

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

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

Theorem 2: Assuming that no failure occurs during recovery of a 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 using the same amount of cores:

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, 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 regardless of the number of failures, i.e., $T_c = T_c^k = w = 2W/N$. Finally, the expected completion time considering the possibility of re-execution is $T_{total} = T_c/(1-P_a) = 2W/N/(1-P_a)$. ■

C. 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 static power.

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}:

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

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 consume 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 first perform shadow leaping and then become idle. 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 the expected total time spent on leaping.

VI. EVALUATION

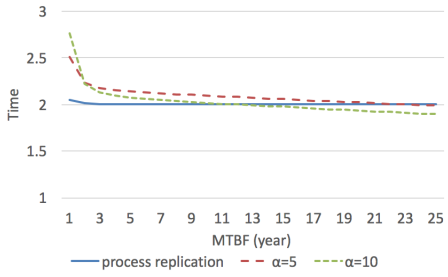
Careful analysis of the mathematical models above leads us to identify several important factors that influence the quality of an application's execution using Lazy Shadowing. These factors can be classified into three categories, i.e., system category, application category, and Lazy Shadowing category. 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 shadowing 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.

A. Comparison to checkpointing and process replication

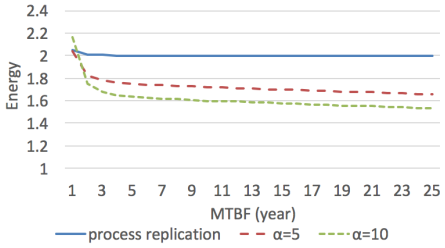
To study the performance of Lazy Shadowing, we compare with process replication using the analytical models in Section V. We also compare to checkpointing, of which the completion time is calculated with Daly's model [29] and the energy consumption is then derived using Equation 4. It is clear from THEOREM 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 [29]. 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 $\sigma_s^b = \frac{1}{\alpha}$. To calculate the expected energy consumption for Lazy Shadowing with Equation 5, we assume that the dynamic power during shadow leaping is twice of that during normal execution, i.e., $p_l = 2 * p_d$, and the time for shadow leaping through RDMA is half of the recovery time, i.e., $T_l = 0.5 * (T_{total} - w)$. The static power ratio ρ is fixed at 0.5 for our first study, other values are also studied in this section.

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.

We had a comprehensive study of the application failure probabilities using models in Section V-A. Results show that application failure probability increases slightly for Lazy Shadowing compared to process replication, as a result of shadow collocation (discussed in Section IV-B). However, even with unrealistically low MTBF of 1 year, Lazy Shadowing is still able to complete without re-execution with

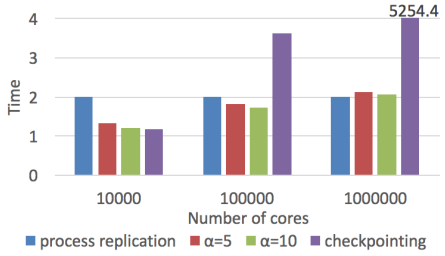


(a) Expected completion time

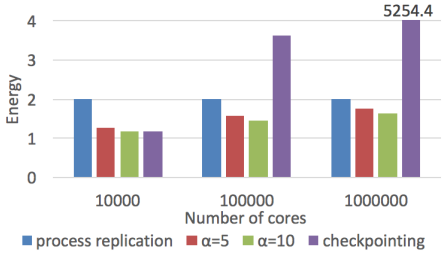


(b) Expected energy consumption

Figure 5: Comparison of completion time and energy consumption. $W = 10^6$ hours, $N = 10^6$, $\rho = 0.5$.



(a) Expected completion time



(b) Expected energy consumption

Figure 6: Sensitivity to number of cores. $W = N$, MTBF=5 years, $\rho = 0.5$.

greater than 0.75 probability. Therefore, we will not further discuss application failure probability in this paper.

Our results show that at extreme-scale, the completion time and energy consumption of checkpointing are orders of magnitude larger than those of Lazy Shadowing and process replication. Thus, we choose not to plot a separate graph for checkpointing in the interest of space. Figure 5(a) reveals that the most time efficient choice largely depends on MTBF. More specifically, process replication consumes less time when MTBF is low while otherwise Lazy Shadowing is more efficient. This is because of the increased probability of re-execution for Lazy Shadowing when failure rate is high. In terms of energy consumption, Lazy Shadowing has much more advantage over process replication. 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 because of extended execution time.

B. Impact of the number of cores

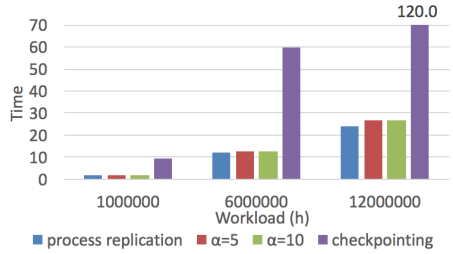
The system scale, measured in number of cores, has a direct impact on the failure rate seen by the application. To study its impact, we varies N from 10,000 to 1,000,000 with W scaled proportionally, i.e., $W = N$. When MTBF is 5 years, the results are shown in Figure 6. Please note that the time and 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 6(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 when energy consumption is the objective.

When MTBF is changed to 25 years, the performance of checkpointing improves a lot, but is still much worse than that of the other two approaches. Lazy Shadowing also benefits from the increased MTBF, and further reduces its completion time and energy consumption. Specifically, Lazy Shadowing is able to achieve shorter completion time than process replication when N reaches 1,000,000.

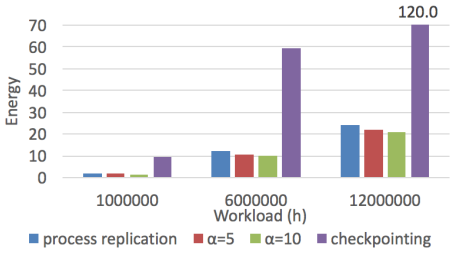
C. Impact of workload

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.

Fixing N at 1,000,000, we increase W from 1,000,000 hours to 12,000,000 hours. Figure 7 assumes a MTBF of 25 years and shows both the time and energy. Checkpointing has the worst performance 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, Lazy Shadowing is able to achieve the most energy saving in all cases.



(a) Expected completion time



(b) Expected energy consumption

Figure 7: Sensitivity to workload. $N = 10^6$, MTBF=25 years, $\rho = 0.5$.

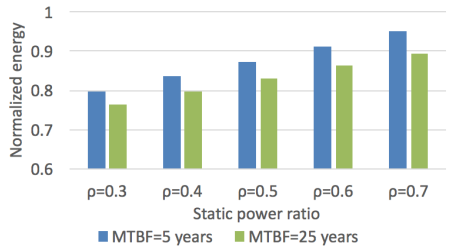


Figure 8: Impact of static power ratio on energy consumption. $W = 10^6$ hours, $N = 10^6$, α=5.

D. Impact of static power ratio

With various architectures and organizations, servers vary in terms of power consumption. The static power ratio ρ is used to abstract the amount of static power consumed versus dynamic power. 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 8. 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.

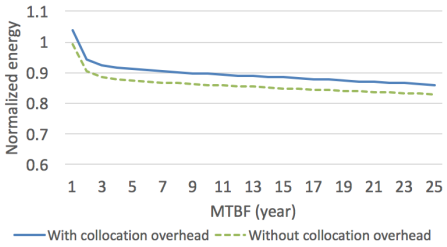


Figure 9: Impact of collocation overhead on energy consumption. $W = 10^6$ hours, $N = 10^6$, ρ=0.5, α=5.

E. Adding collocation overhead

Lazy Shadowing increases memory requirement[2] when multiple shadows are collocated. Moreover, this may have an impact on the execution rate of the shadows due to cache contention and context switch. To capture this effect, we re-model the rate of shadows as $\sigma_s^b = \frac{1}{\alpha^{1.5}}$. Figure 9 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. 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 quite small, 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%.

VII. CONCLUSION

As the scale and complexity of HPC systems 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 efficiently. Existing fault tolerance methods rely on either time or hardware redundancy. Neither of them appeals 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.

Lazy Shadowing is a novel algorithm that can serve 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

[2]Note that this problem is not intrinsic to Lazy Shadowing, as in-memory checkpointing also requires extra memory.

of the alternatives should be chosen. Specially, checkpointing consumes the least time and energy for small scale systems, while Lazy Shadowing is the choice as system scale and complexity keep growing. In addition, Lazy Shadowing has the ability to converge to process replication when failure rate is extremely high.

In the future, we will generalize our approach to multiple jobs. Initially we will assume that each job has an arrival time, a workload, and a deadline by which it must complete, and will seek minimally adaptive stochastically competitive strategies. Another future direction is to apply our approach to servers with inter-processor power heterogeneity. 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.

REFERENCES

[1] K. Ferreira and et. al., "Evaluating the viability of process replication reliability for exascale systems," ser. SC '11. New York, NY, USA: ACM.

[2] 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.

[3] O. Sarood and et. al., "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," ser. SC '14, Piscataway, NJ, USA, pp. 807–818.

[4] O. Villa and et. al., "Scaling the power wall: A path to exascale," ser. SC '14, Piscataway, NJ, USA.

[5] 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.

[6] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS," in *Defect and Fault Tolerance of VLSI Systems*, 2008.

[7] 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.

[8] 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.

[9] 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.

[10] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975.

[11] L. Alvisi and et. al., "An analysis of communication induced checkpointing," in *Fault-Tolerant Computing*, 1999.

[12] J. Helary and et. al., "Preventing useless checkpoints in distributed computations," in *RDS*, 1997.

[13] A. Guermouche and et. al., "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, May 2011, pp. 989–1000.

[14] S. Agarwal and et. al., "Adaptive incremental checkpointing for massively parallel systems," in *ICS 04*, St. Malo, France.

[15] 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.

[16] 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.

[17] G. Zheng and et. al., "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Cluster Computing*, 2004, pp. 93–103.

[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.

[19] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[20] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *IJHPCA*, vol. 23, no. 3, pp. 212–226, 2009.

[21] J. Stearley and et. al., "Does partial replication pay off?" in *DSN-W*, June 2012, pp. 1–6.

[22] J. Elliott and et. al., "Combining partial redundancy and checkpointing for HPC," in *ICDCS '12*, Washington, DC, US.

[23] S. K. Reinhardt and et. al., *Transient fault detection via simultaneous multithreading*. ACM, 2000, vol. 28, no. 2.

[24] J. Wadden and et. al., "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *ISCA '14*. Piscataway, NJ, USA: IEEE Press.

[25] H. Casanova and et. al., "Combining Process Replication and Checkpointing for Resilience on Exascale Systems," INRIA, Rapport de recherche RR-7951, May 2012.

[26] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 1–26, Mar. 1999.

[27] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[28] L. Alvisi and K. Marzullp, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 149–159, 1998.

[29] 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.