2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress

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 resource management, HPC is moving towards extreme-scale, featuring a computing capability of exaFLOPS. 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].

In today's HPC systems the response to faults mainly consists of rolling back the execution. For long-running jobs, such as scientific applications, checkpoint of the execution is periodically saved to stable storage to avoid complete rollback. Upon the occurrence of a hardware or software failure, recovery is then achieved by restarting from a saved checkpoint. Given the anticipated increase in system-level failure rates and the time 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 Mean Time Between Failures (MTBF). Consequently, applications will make little forward progress, thereby reducing considerably the overall system efficiency [1].

More recently, process replication, either fully or partially, has been proposed to scale to the resilience requirements of future extreme-scale systems. 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 a tremendous increase in the number of computing resources, thereby constantly wasting a significant portion of system capacity.

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 [4]. The trade-off 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. 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.

978-1-5090-2771-2/16 $31.00 © 2016 IEEE
DOI 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.29

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. Lazy Shadowing extends the idea of shadowing from [5], [6], [7] with novel concepts of *shadow collocation* and *shadow leaping*. In the basic idea of shadowing, each process (referred to as a *main*) is associated with a replica (referred to as a *shadow*) to improve resilience. The shadows initially execute at a reduced rate. Upon failure of a main, its associated shadow increases its rate to complete the task, thereby reducing the impact of such a failure on the progress of other tasks. Instead of using Dynamic Voltage and Frequency Scaling (DVFS) for execution rate control as in [5], [6], [7], in this paper we explore the applicability of collocating shadows to simultaneously save power/energy and computing resources. Furthermore, we identify a unique opportunity that allows the lagging shadows to benefit from the faster execution of the mains, without incurring extra overhead. Specifically, when a failure occurs, Lazy Shadowing takes advantage of the recovery time and leaps forward the shadows by copying states from the mains. 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.

Lazy Shadowing is adaptive in that it enables a configurable trade-off between rollback (time redundancy) and replication (hardware redundancy) by dynamically adjusting the main and shadow processes' execution rates. Optimal execution rates can be derived either analytically or empirically, in order to achieve the desired balance between two important objectives, namely, the expected completion time of the supported application, and the power constraints imposed by the underlying computing infrastructure.

The main contributions of this paper are as follows:

- An efficient and scalable algorithm for fault tolerance in future extreme-scale computing systems.
- A performance modeling framework to quantify the expected completion time and energy consumption.
- A discussion on implementation details and potential runtime overhead.
- A thorough comparative study that explores the performance of Lazy Shadowing with different application requirements and system characteristics.

The rest of the paper is organized as follows. We begin with a survey on related work in Section II. Section III introduces Lazy Shadowing algorithm and Section IV discusses how it can be applied to extreme-scale systems. We then present a performance modeling framework in Section V, followed by experiments and evaluation in section VI. Section VII concludes this work.

II. RELATED WORK

Rollback recovery is the dominant mechanism to achieve fault tolerance in current HPC environments [8]. In the most general form, rollback recovery involves the periodic saving of the execution state (checkpoint), with the anticipation that in the case of a failure, computation can be restarted from a previously saved checkpoint. 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 [9].

In uncoordinated checkpointing, processes checkpoint 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 can also suffer the well-known domino effect [10], [11], [12]. Although well-explored, uncoordinated checkpointing has not been widely adopted in HPC environments for its complexities.

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 [13]. Another proposed scheme, known as in-memory checkpointing, minimizes the overhead of disk access [14], [15]. The main concern of these techniques is the increase in memory requirement. It has been suggested that nodes in extreme-scale systems should be configured with fast local storage [2]. Multi-level checkpointing can benefit from such a strategy [16]. This, however, may lead to increased failure rates of individual nodes and complicate the checkpoint writing process.

Process replication, or state machine replication, has long been used for reliability and availability in distributed and mission critical systems [17]. Although it is initially rejected in HPC communities, replication has recently been proposed to address the deficiencies of checkpointing for upcoming extreme-scale systems [18], [19]. Full and partial process replication have also been studied to augment existing checkpointing techniques, and to detect and correct silent data corruption [20], [21], [1], [22]. Our approach is largely different from classical process replication in that we dynamically configure the execution rates of main and shadow processes, so that less resource/energy is required while reliability is still assured.

Replication with dynamic execution rate is also explored in Simultaneous and Redundantly Threaded (SRT) processor whereby one leading thread is running ahead of trailing threads [23]. However, the focus of [23] is on transient faults within CPU while we aim at tolerating both permanent and transient faults across all system components. Also, this work differs from [5], [6], [7], where shadowing with DVFS is studied for single or loosely-coupled tasks.

III. Lazy shadowing

By carefully analyzing the characteristics of HPC applications, we devise novel ideas of shadow collocation and shadow leaping, and integrate them with the basic concept of shadowing, which jointly form a complete algorithm that we call Lazy Shadowing. To make it easy to follow, we first re-emphasize important details of shadowing. Assuming the fail-stop failure model [24], [25], the concept of 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.

Under fail-stop model, Lazy Shadowing is able to tolerate failures in hardware, such as power supply, CPU, attached accelerators (e.g., GPU), memory bit flips that exceed ECC's capacity, as well as software, such as OS and runtime.

Initially, the main executes at rate σ_m, while the shadow executes at $\sigma_s^b \leq \sigma_m$. In the absence of failure, the main completes 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 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.

In HPC, throughput consideration requires that the rate of the main, σ_m, and the rate of the shadow after failure, σ_s^a, be set to the maximum. The initial execution rate of the shadow, σ_s^b, however, can be derived by balancing the trade-offs 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. In addition, a broad spectrum of delay and energy trade-offs in between can be explored either empirically or by using optimization frameworks for delay and energy tolerant applications.

IV. Extreme-scale fault tolerance

Enabling Lazy Shadowing for resiliency in extreme-scale computing brings about a number of challenges and design decisions that need to be addressed, including the applicability of this concept to a large number of tasks executing in parallel, the effective way to control shadows' execution rates, and the runtime mechanisms and communications support to ensure efficient coordination between a main and its shadow. Taking into consideration the main characteristics of compute-intensive and highly-scalable applications, we design two novel techniques, referred to as *shadow collocation* and *shadow leaping*, in order to achieve high-tolerance to failure while minimizing delay and energy consumption.

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 into a set of tasks, T, which execute in parallel. Assuming the maximum execution rate is $\sigma_{max} = 1$, the failure-free completion time of the application is $W/|T|$. Given the prominence of MPI in HPC environments, we assume message passing as the communication mechanism between tasks. The execution is composed of a set of iterations separated by synchronization barriers.

B. Shadow collocation

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 [26]. Each main process executes on one core exclusively to achieve maximum throughput. On the other hand, we collocate multiple shadows on a single core and use time sharing to achieve the desired execution rates. To execute an application of M tasks, $N = M + S$ cores are required, where M is a multiple of S. Each main is allocated one core (referred to as *main core*), while $\alpha = M/S$ (referred to as *collocation ratio*) shadows are collocated on a core (*shadow core*). The N cores are grouped into S sets, each of which we call a *shadowed set*. Each shadowed set contains α main cores and 1 shadow core. This is illustrated in Figure 1.

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, all the shadows in the shadowed set will be lost without interrupting the execution of the mains. On the other hand, if a main core fails, the associated shadow will be promoted to a new main, and all the other collocated shadows will be terminated to speed up the new main. Consequently, a failure, either in main or shadow core, will result in losing all the shadows in the shadowed set, thereby losing the tolerance to any other failures. Quantitative study of this effect using probability theory is presented in Section V-A.

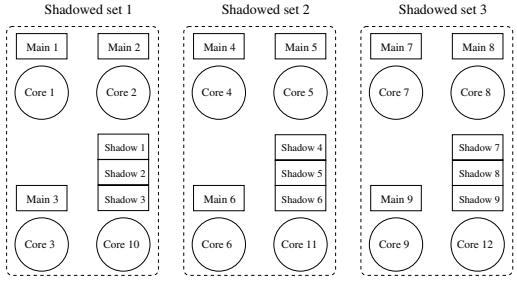


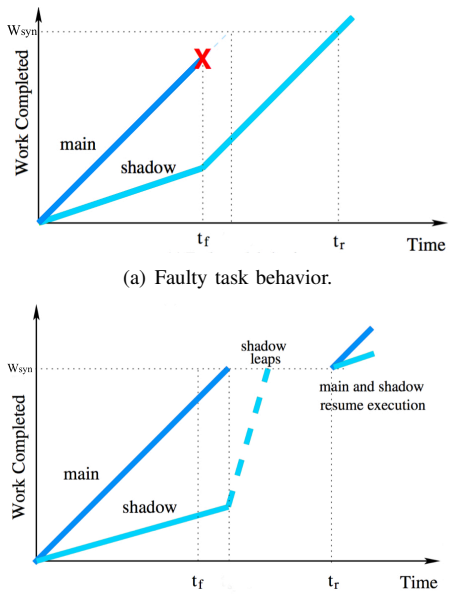
Figure 1: An example of collocation. $N = 12$, $M = 9$, $S = 3$.

C. Shadow leaping

As the shadows execute at a lower rate, failures will incur delay for recovery. This problem deteriorates as dependencies incurred by messages and synchronization barriers would propagate the delay of one task to others. Fortunately, slowing down the shadows provides an opportunity for the shadows to benefit from the faster execution of their mains. By copying the state of each main to its shadow, which is similar to the process of storing a checkpoint in a buddy in [14], forward progress is achieved for the shadows with minimized time and energy. This technique, referred to as *shadow leaping*, effectively limits the distance between main and shadow in progress. As a result, the recovery time after a failure, which depends on the distance between the failing main and its shadow, is also reduced. More importantly, we opportunistically overlap shadow leaping with failure recovery to avoid extra overhead.

Assuming a failure occurrence at time t_f, Figure 2 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 other tasks' progress, as illustrated in Figure 2(a). At the same time, as shown in Figure 2(b), the remaining main processes continue execution until the barrier at W_{syn}, and then become idle until t_r. Shadow leaping opportunistically takes advantage of this idle time to *leap forward* the shadows, so that all processes, including shadows, can resume execution from a consistent point afterwards. Shadow 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.

We summarize the integration of shadow collocation and shadow leaping with basic shadowing in Algorithm 1. User needs to specify M for the number of parallel tasks, and S for the number of shadow cores. These two parameters jointly determine the collocation ratio $\alpha = M/S$. The execution starts by simultaneously launching $2M$ processes, of which one main is associated with one shadow for each task (line 1). All processes are then grouped into S shadowed sets. Accordingly, the processes are mapped to cores so that each shadowed set has α cores for α mains and 1 core for all the associated shadows (line 2). During the execution, the system runs a failure monitor (e.g., by using heartbeat protocol [27]) that triggers corresponding actions when a failure is detected (line 4 to 15). A failure occurring in a vulnerable shadowed set results in an application fatal failure and forces a rollback (line 6 and 7). On the other hand, failure in a non-vulnerable shadowed set can be tolerated while making the target shadowed set vulnerable (line 9). In this case, failure of a main need to be treated differently from failure of a shadow. While a failure of shadow does not impact the normal execution and thus can be ignored,



(a) Faulty task behavior.

(b) Non-faulty task behavior.

Figure 2: The illustration of shadow leaping.

input : M, S
output: Application execution status

```
1  start M pairs of main and shadow;
2  map processes to cores;
3  while execution not done do
4      if failure detected in shadowed set SS_i then
5          if SS_i is vulnerable then
6              notify "Application fatal failure";
7              restart execution;
8          else
9              mark SS_i as vulnerable;
10             if failure happened to a main then
11                 promote its shadow to new main;
12                 trigger shadow leaping;
13             end
14         end
15     end
16 end
17 output "Application completes";
```

Algorithm 1: Lazy Shadowing

failure of a main triggers promotion of its shadow to a new main, which increases its rate to recover the failure and complete the task (line 11). Simultaneously, a shadow leaping is undertaken by all other shadows to align their states with those of their associated mains (line 12). This process continues until all tasks of the application are successfully completed.

D. Implementation issues

We are implementing a MPI based library for Lazy Shadowing, referred to as lsMPI. Inserted as a layer between application and MPI, lsMPI uses the MPI profiling hooks to intercept every MPI call. Currently, lsMPI delegates failure detection to ULFM [28], [29]. lsMPI should be portable across all MPI implementations once extensions by ULFM are added to MPI standard.

State consistency is required both during normal execution and following a failure. We design a consistency protocol to assure that the shadows see the same message order and MPI results as mains. For each message, the main sender sends a copy of the message to each of the main and shadow receivers. After getting the message, the main receiver sends an ACK to the shadow sender, so that the shadow sender can safely suppress sending the message and proceed. If a main fails, its associated shadow will become a new main and start sending out messages.

We assume that only MPI operations can introduce non-determinism. MPI_ANY_SOURCE receives may result in different message orders between the main and shadow. To deal with this, we always let the main receive a message ahead of the shadow and then forward the message source to its shadow. The shadow then issues a receive with the specific source. Other operations, such as MPI_Wtime() and MPI_Probe(), can be dealt with by always forwarding the result from the main to the shadow.

V. ANALYTICAL MODELS

In the following we develop analytical models to quantify the expected performance of Lazy Shadowing, as well as prove the bound on performance loss due to failures. 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 the collocated shadow processes. Note that process replication is a special case of Lazy Shadowing where $\alpha = 1$, so $M = S = \frac{N}{2}$ and $w = \frac{2W}{N}$.

A. Application fatal failure probability

An application has to roll back when all replicas of a task have been lost. We call this an application fatal failure, which is inevitable even when every process is replicated. In order to take into account the overhead of rollback in the calculation of completion time and energy consumption, we first study the probability of application fatal failure. In this work we assume that once an application fatal failure occurs, execution always rolls back to the very beginning.

The impact of process replication on application fatal failure has been studied in [26] and results are presented in terms of Mean Number of Failures To Interrupt (MNFTI). Applying the same methodology, we derive the new MNFTI under Lazy Shadowing, as shown in Table I. As Lazy Shadowing can tolerate one failure in each shadowed set, the results are for different numbers of shadowed sets (S). Note that when processes are not shadowed, every failure would interrupt the application, i.e., MNFTI=1.

To further quantify the probability of application fatal failure, we use $f(t)$ to denote the failure probability density function of each core, and 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 an fatal failure occurs to an application using N cores 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 fatal 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

There are two types of delay due to failures. If a failure does not lead to an application fatal failure, the delay corresponds to the catching up of the shadow of the failing main (see Figure 2(a)). Otherwise, a possible larger (rollback) delay will be introduced by an application fatal failure. In the following we consider both delays step by step. First we discuss the case of k failures without application fatal failure. Should a failure occur during the recovery of a previous failure, its recovery would overlap with the ongoing recovery. To study the worst case behavior, we assume failures do not overlap, so that the execution is split into $k+1$ intervals, as illustrated in Figure 3. Δ_i $(1 \le i \le k+1)$ represents the i^{th} execution interval, and τ_i $(1 \le i \le k)$ is the recovery time after Δ_i.

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

Theorem 1: Assuming that failures do not overlap and no application fatal failure occurs, then using Lazy Shadowing,

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

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

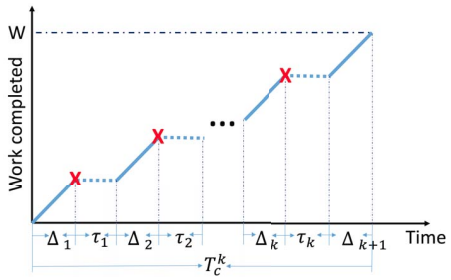


Figure 3: Application's progress with shadow catching up delays.

Proof: Lazy Shadowing guarantees that all the shadows reach the same execution point as the mains (See Figure 2) after a previous recovery, so every recovery time is proportional to its previous execution interval. That is, $\tau_i = \Delta_i \times (1-\sigma_s^b)$. According to Figure 3, 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 with the number of failures, it turns out to be well bounded, as a benefit of shadow 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, $\sum_{i=1}^{k} \Delta_i = w - \Delta_{k+1} \leq w$. Therefore, $(1-\sigma_s^b)\sum_{i=1}^{k} \Delta_i \leq (1-\sigma_s^b)w$. ■

Typically, the number of failures to be encountered is stochastic. 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 calculated as $P_c = F(w)$. Then the probability that there are k failures among the N cores is

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

The following theorem expresses the expected completion time, T_{total}, considering all possible number of failures.

Theorem 2: Assuming that failures do not overlap, then using Lazy Shadowing, $T_{total} = T_c/(1-P_a)$, where $T_c = \sum_i T_c^i \cdot P_s^i$.

Proof: Without application fatal failure, the completion time considering all possible values of k can be averaged as $T_c = \sum_i T_c^i \cdot P_s^i$. If an application fatal failure occurs, however, the application needs to roll back to the beginning. With the probability of rollback calculated as P_a in Section V-A, 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 shadows so that the workload assigned to each task is significantly increased, i.e., $w = 2W/N$. Different from cases where $\alpha \geq 2$, failures do not incur any delay except for application fatal failures. As a result, without application fatal failure the completion time under process replication 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 rollback is $T_{total} = T_c/(1-P_a) = 2W/N/(1-P_a)$. ■

C. Expected energy consumption

Power consumption consists of two parts, dynamic power, p_d, which exists only when a 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}:

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

Even using the same amount of cores, Lazy Shadowing can save power and energy, since main cores are idle during the recovery time after each failure, and the shadows can achieve forward progress through shadow leaping. During normal execution, 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

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

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 models above leads us to identify several important factors that determine the performance. These factors can be classified into three categories, i.e., system, application, and algorithm. The system category includes static power ratio ρ ($\rho = p_s/p$), total number of cores N, and MTBF of each core; the application category is mainly the total workload, W; and shadowing ratio α in the algorithm category determines 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

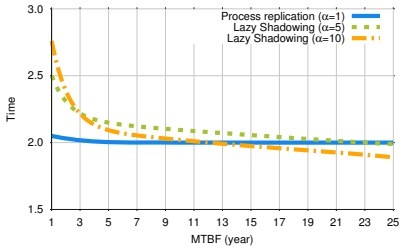
We compare with both process replication and checkpointing. The completion time with checkpointing is calculated with Daly's model [30] assuming 10 minutes for both checkpointing and restart. The energy consumption is then derived with Equation 4. It is important to point out that we always assume the same number of cores, so that process replication and Lazy Shadowing do not use extra cores for the replicas.

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. 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 if failures are uniformly distributed [30]. Specifically, $\Delta_i = w/(k+1)$, and $T_c^k = w + w*(1-\sigma_s^b)*\frac{k}{k+1}$. Further, we assume that each shadow gets a fair share of its core's execution rate so that $\sigma_s^b = \frac{1}{\alpha}$. To calculate 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 is half of the recovery time, i.e., $T_l = 0.5*(T_{total} - w)$.

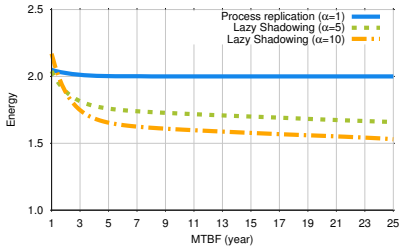
The first study uses $N = 1$ million cores, $W = 1$ million hours, and static power ratio $\rho = 0.5$. Our results show that at extreme-scale, the completion time and energy consumption of checkpointing are orders of magnitude larger than those of Lazy Shadowing and process replication. Thus, we choose not to plot a separate graph for checkpointing in the interest of space. Figure 4(a) reveals that the most time efficient choice largely depends on MTBF. When MTBF is high, Lazy Shadowing requires less time as more cores are used for main processes and less workload is assigned to each process. As MTBF decreases, process replication outperforms Lazy Shadowing as a result of the increased likelihood of rollback for Lazy Shadowing. 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 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 5. 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 5(a) clearly shows that each of the three fault tolerance alternatives has its own advantage. Specifically, checkpointing is the best choice for small systems at the scale of 10,000 cores, Lazy Shadowing outperforms others for systems with 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.



(a) Expected completion time



(b) Expected energy consumption

Figure 4: Comparison of time and energy for different core level MTBF. $W = 10^6$ hours, $N = 10^6$, $\rho = 0.5$.

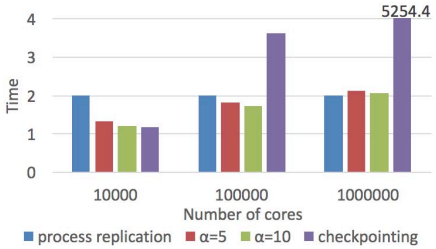
When MTBF is changed to 25 years, the performance of checkpointing improves a lot, but is still much worse than that of the other two approaches. Lazy Shadowing benefits much more than process replication from the increased MTBF. As a result, Lazy Shadowing is able to achieve shorter completion time than process replication when N reaches 1,000,000.

C. Impact of workload

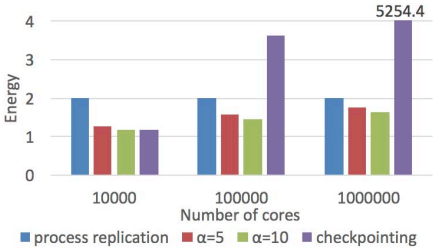
To a large extent, workload determines the time exposed to failures. With other factors being the same, an application with a larger workload is likely to encounter more failures during its execution. Fixing N at 1,000,000, we increase W from 1,000,000 hours to 12,000,000 hours. Figure 6 assumes a MTBF 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 when workload reaches 6,000,000 hours. Considering energy consumption, however, Lazy Shadowing is able to achieve the most saving in all cases.

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

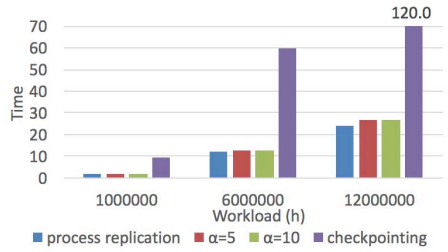


(a) Expected completion time

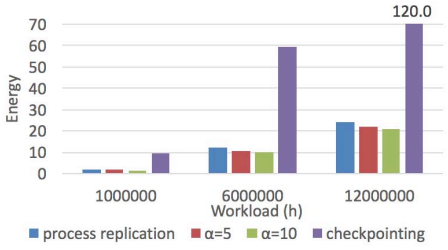


(b) Expected energy consumption

Figure 5: Comparison of time and energy for different number of cores. $W = N$, MTBF=5 years, $\rho = 0.5$.



(a) Expected completion time



(b) Expected energy consumption

Figure 6: Comparison of time and energy for different workloads. $N = 10^6$, MTBF=25 years, $\rho = 0.5$.

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 7. 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

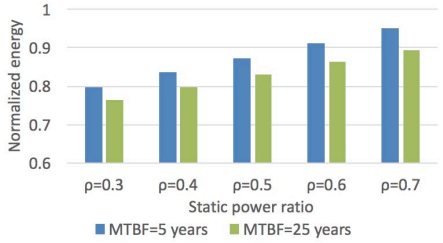


Figure 7: Impact of static power ratio on energy consumption. $W = 10^6$ hours, $N = 10^6$, α=5.

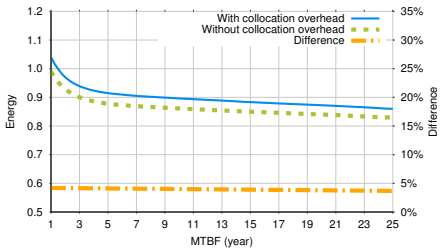


Figure 8: Impact of collocation overhead on energy consumption. $W = 10^6$ hours, $N = 10^6$, ρ=0.5, α=5.

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.

E. Adding collocation overhead

Lazy Shadowing increases memory requirement[1] 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 8 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. 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 shadow leaping can take advantage of the recovery time after each failure and achieve forward progress for shadow processes that fall behind. The results for other values of α have similar behavior. 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

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

efficiently. Existing fault tolerance methods rely on either time or hardware redundancy.

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 analytical 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.

In the future, we plan to explore the combination of Lazy Shadowing with Checkpointing, so that if an application fatal failure occurs computation can restart from an intermediate state. Another future direction is to use dynamic and partial shadowing for platforms where nodes exhibit different "health" status, e.g., some nodes may be more reliable while others are more likely to fail [31]. With this taken into account, we can apply dynamic scheduling of shadows only for mains that are likely to fail, to further reduce the resource requirement for shadowing.

ACKNOWLEDGMENT

This research is based in part upon work supported by the Department of Energy under contract DE-SC0014376.

REFERENCES

[1] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, pp. 44:1–44:12.

[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] 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.

[5] B. Mills and et. al., "Shadow computing: An energy-aware fault tolerant computing model," in *ICNC*, 2014.

[6] X. Cui and et. al., "Shadow replication: An energy-aware, fault-tolerant computational model for green cloud computing," *Energies*, vol. 7, no. 8, pp. 5151–5176, 2014.

[7] X. Cui, B. Mills, T. Znati, and R. Melhem, "Shadows on the cloud: An energy-aware, profit maximizing resilience framework for cloud computing," in *CLOSER*, April 2014.

[8] E. Elnozahy and et. al., "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[9] 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.

[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, E. Elnozahy, S. Rao, S. A. Husain, and A. de Mel, "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] S. Agarwal and et. al., "Adaptive incremental checkpointing for massively parallel systems," in *ICS 04*, St. Malo, France.

[14] 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.

[15] G. Zheng, X. Ni, and L. V. Kal, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *DSN-W*, June 2012, pp. 1–6.

[16] 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.

[17] 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.

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

[19] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *PDCN*, 2011, pp. 15–17.

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

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

[22] 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.

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

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

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

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

[27] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 561–580, May 2002.

[28] W. Bland, A. Bouteiller, and et. al., "An evaluation of user-level failure mitigation support in mpi," ser. EuroMPI. Springer-Verlag, 2012, pp. 193–203.

[29] W. Bland and et. al., "Post-failure recovery of mpi communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, 2013.

[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.

[31] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *SC*, Nov 2012, pp. 1–11.