ADAPTIVE AND POWER-AWARE FAULT TOLERANCE FOR FUTURE EXTREME-SCALE COMPUTING

by

Xiaolong Cui

Bachelor of Engineering

Xi'an Jiaotong University

2012

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of

Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

COMPUTER SCIENCE DEPARTMENT

This proposal was presented

by

Xiaolong Cui

Dr. Taieb Znati, Department of Computer Science, with joint appointment in Telecommunication Program, University of Pittsburgh

Dr. Rami Melhem, Department of Computer Science, University of Pittsburgh

Dr. John Lange, Department of Computer Science, University of Pittsburgh

Dr. Esteban Meneses, School of Computing, Costa Rica Institute of Technology

Dissertation Advisors: Dr. Taieb Znati, Department of Computer Science, with joint appointment in Telecommunication Program, University of Pittsburgh,

Dr. Rami Melhem, Department of Computer Science, University of Pittsburgh

ADAPTIVE AND POWER-AWARE FAULT TOLERANCE FOR FUTURE EXTREME-SCALE COMPUTING

Xiaolong Cui, PhD

University of Pittsburgh, 2016

As the demand for computing power continue to increase, both HPC community and Cloud service provides are building larger computing platforms to take advantage of the power and economies of scale. On the HPC side, a race is underway to build the world's first exascale supercomputer to accelerate scientific discoveries, big data analytics, etc. On the Cloud side, major IT companies are all expanding large-scale datacenters, for both private usage and public services. However, aside from the benefits, several daunting challenges will appear when it comes to extreme-scale.

This thesis aims at simultaneously solving two major challenges, i.e., power consumption and fault tolerance, for future extreme-scale computing systems. We come up with a novel power-aware computational model, referred to as Lazy Shadowing, to achieve high-levels of resilience in extreme-scale and failure-prone computing environments. Different approaches have been studied to apply this model. Accordingly, precise analytical models and optimization framework have been developed to quantify and optimize the performance, respectively.

In this work, I propose to continue the research in two aspects. Firstly, I propose to develop a MPI-based prototype to validate Lazy Shadowing in real environment. Using the prototype, I will run benchmarks and real applications to measure its performance and compare to state-of-the-art approaches. Then, I propose to further explore the adaptivity of Lazy Shadowing and improve its efficiency. Based on the specific system configuration, application characteristics, and QoS requirement, I will study the impact of process mapping the viability of partial shadowing.

TABLE OF CONTENTS

1.0 INTRODUCTION . . . 1

1.1 Problem Statement . . . 2

1.2 Research Overview . . . 4

1.2.1 Reward-based optimal Lazy Shadowing (completed) . . . 5

1.2.2 Applying Lazy Shadowing to extreme-scale systems (completed) 5

1.2.3 lsMPI: an implementation in MPI (partially completed) . . . 6

1.2.4 Smart shadowing (future) . . . 7

1.3 Contributions . . . 7

1.4 OUTLINE . . . 8

2.0 BACKGROUND . . . 9

3.0 LAZY SHADOWING: A NOVEL FAULT-TOLERANT COMPUTATIONAL MODEL . . . 11

4.0 REWARD-BASED OPTIMAL LAZY SHADOWING . . . 14

4.1 Cloud workload characteristics . . . 15

4.2 Optimization framework . . . 16

4.2.1 Reward Model . . . 16

4.2.2 Failure Model . . . 18

4.2.3 Power and Energy Models . . . 18

4.2.4 Income and Expense Models . . . 21

4.3 Profit-aware stretched replication . . . 22

4.4 Re-execution . . . 22

4.5 Performance evaluation . . . 23

4.6 Summary 27
5.0 APPLYING LAZY SHADOWING TO EXTREME-SCALE SYSTEMS 28
6.0 LSMPI: AN IMPLEMENTATION IN MPI 29
7.0 SMART SHADOWING 30
8.0 TIMELINE OF PROPOSED WORK 31
9.0 SUMMARY 32
BIBLIOGRAPHY 33

1.0 INTRODUCTION

As our reliance on IT continues to increase, the complexity and urgency of the problems our society will face in the future drives us to build more powerful and accessible computer systems. Among the different types of computer systems, High Performance Computing (HPC) and Cloud Computing systems are the two most powerful ones. For both of them, the computing power attributes to the massive amount of parallelism, which is supported by the massive amount of computing cores, memory modules, communication devices, storage components, etc.

Since CPU frequency flattens out in early 2000s, parallelism has become the "golden rule" to boost performance. In HPC, Terascale performance was achieved in the late 90s with fewer than 10,000 heavyweight single-core processors. A decade later, petascale performance required about ten times processors with multiple cores on each processor. Nowadays, a race is underway to build the world's first exascale machine to accelerate scientific discoveries and breakthroughs. It is projected that an exascale machine will achieve billion-way parallelism by using one million sockets each supporting 1,000 cores.

Similar trend is happening in Cloud Computing. As the demand for Cloud Computing accelerates, cloud service providers will be faced with the need to expand their underlying infrastructure to ensure the expected levels of performance, reliability and cost-effectiveness. As a result, lots of large-scale data centers have been and are being built by IT companies to exploit the power and economies of scale. For example, Microsoft, Google, Facebook, and Rackspace have hundreds of thousands of web servers in dedicated data centers to support their business.

Unfortunately, several challenging issues come with the increase in system scale. As today's HPC and Cloud Computing systems grow to meet tomorrow's compute power demand, the behavior of the 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. At the same time, the rapid growing power consumption, as a result of the increase in system components, is another major concern. At future extreme-scale, failure would become a norm rather than an exception, driving the system to significantly lower efficiency with unprecedented amount of power consumption.

1.1 PROBLEM STATEMENT

The system scale needed to address our future computing needs will come at the cost of increasing complexity, unpredictability, and operating expenses. As we approach future extreme-scale, two of the biggest challenges will be system resilience and power consumption, both being direct consequences of the increase in the number of components.

Regardless of the reliability of individual component, the system level reliability will continue to decrease as the number of components increases. It is projected that the Mean Time Between Failures (MTBF) of future extreme-scale systems will be at the order of hours or even minutes, meaning that many failures will occur every day. Without an efficient fault tolerance mechanism, faults will be so frequent that the applications running on the systems will be continuously interrupted, requiring the execution to be restarted every time there is a failure.

Also thanks to the continuous growth in system components, there has been a steady rise in power consumption in large-scale distributed systems. In 2005, the peak power consumption of a single supercomputer reached 3.2 Megawatts. This number was doubled only after 5 years, and reached 17.8 Megawatts with a machine of 3,120,000 cores in 2013. Recognizing this rapid upward trend, the U.S. Department of Energy has set 20 megawatts as the power limit for future exascale systems, challenging the research community to provide a 1000x improvement in performance with only a 10x increase in power. This huge imbalance makes system power a leading design constraint on the path to exascale.

Today, two approaches exist for fault tolerance. The first approach is rollback recovery, which rolls back and restarts the execution every time there is a failure. This approach is often equipped with checkpointing to periodically save the execution state to a stable storage so that execution can be restarted from a recent checkpoint in the case of a failure. Although checkpointing is the most widely used technique in today's HPC systems, it is strongly believed that it may not scale to future extreme-scale systems. 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 MTBF. Consequently, applications will make little forward progress, thereby reducing considerably the overall system efficiency.

The second approach, referred to as process replication, exploits hardware redundancy and executes multiple instances of the same task in parallel to overcome failure and guarantee that at least one task instance reaches completion. Although this approach is extensively used to deal with failures in Cloud Computing and mission critical systems, it has never been used in any HPC system due to its low system efficiency. To replicate each process, process replication requires at least double the amount of compute nodes, which also increases the power consumption proportionally.

Previous studies show that neither of the two approaches may be efficient for future extreme-scale systems. And unfortunately, neither of them addresses the power cap issue. Achieving high resilience to failures under strict power constraints is a daunting and critical challenge that requires new computational models with scalability, adaptability, and power-awareness in mind.

1.2 RESEARCH OVERVIEW

There is a delicate interplay between fault tolerance and power consumption. Checkpointing and process replication require additional power to achieve fault tolerance. Conversely, it has been shown that lowering supply voltages, a commonly used technique to conserve power, increases the probability of transient faults. The trade-off between fault free operation and optimal power 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 power 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 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, Mills, Znati, and Melhem have proposed a novel computational model, referred to as Lazy Shadowing, as a power-aware approach to achieve high-levels of resilience, through forward progress, in extreme-scale computing environments [Mills and et. al., 2014; Mills et al., 2014; Mills, 2014]. Based on Dynamic Voltage Frequency and Scaling (DVFS), Mills studied the computational model and its performance in terms of completion time and energy consumption in HPC systems. Through the use of analytical models, simulations, and experimentation, Mills demonstrated that Lazy Shadowing can achieve resilience more efficiently than both checkpointing and traditional replication when power is limited.

This thesis continues Mills' work on Lazy Shadowing, and seeks to simultaneously address the power and resilience challenges for future extreme-scale systems, so that both system efficiency and application QoS are guaranteed. Specifically, this thesis tries to answer 4 questions: 1) is Lazy Shadowing able to achieve objectives that go beyond time to completion, such as multiple simultaneous requirements defined in a SLA in the Cloud; 2) how to enable Lazy Shadowing when DVFS is not viable, and ensure forward progress while guaranteeing useful work in future extreme-scale systems; 3) is Lazy Shadowing a viable computational model in real environments; 4) how to make Lazy Shadowing reflective of the propensity of the processing elements to failures and adaptive to the QoS requirements. With these questions in mind, I have studied different techniques to enable the model, and developed analytical frameworks to optimize for different objectives in the Cloud and HPC environments. Next, I propose to continue the study in two aspects. Firstly, I propose to implement a prototype of Lazy Shadowing in the context of Message Passing Interface (MPI), to validate the computational model as well as measure its performance in real environment. Secondly, I propose to study the possibility of "smart shadowing" which further reduces the cost of shadowing by considering the specific system configuration, application characteristics, and QoS requirement.

1.2.1 Reward-based optimal Lazy Shadowing (completed)

Lazy Shadowing is a flexible computational model that goes beyond minimizing time to completion. The major challenge in Lazy Shadowing resides in determining jointly the execution rates of all task instances, both before and after a failure occurs, with the objective to optimize performance, resilience, power consumption, or their combinations. In this work we focus on the SLA requirements in the Cloud and develop a reward-based analytical framework, in order to derive the optimal execution rates for maximizing reward and minimizing energy costs under strict completion time constraints.

To define the reward-based analytical framework, we first define a failure model that considers the failure distribution of each process, and a power model that describes the power consumption characteristics under different states. Based on the failure model and power model, we then derive the expected income as a function of completion time, and the expected energy cost as a product of power and time. Lastly, the reward is defined as the optimization objective to balance between completion time and energy cost.

1.2.2 Applying Lazy Shadowing to extreme-scale systems (completed)

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 failures while minimizing delay and power consumption.

To control the processes' execution rate, DVFS can be applied while each process resides on one core exclusively. The effectiveness of DVFS, however, may be markedly limited by the granularity of voltage control, the number of frequencies available, and the negative effects on reliability. An alternative is to collocate multiple processes on each core while keeping all the cores executing at maximum frequency. Then time sharing can be used to achieve the desired execution rates for each collocated process. Since this approach collocates multiple processes on a core, it simultaneously reduces the number of compute nodes and the power consumption.

Furthermore, we identify a unique opportunity that ensures forward progress in failure-prone environments. Since each shadow process is associated with a main process, the lagging shadow can benefit from the faster execution of the main with minimal overhead. Specifically, when a failure occurs, Lazy Shadowing takes advantage of the recovery time and leaps forward each shadow by copying states from its associated main. This technique not only achieves forward progress for the shadow processes at minimized power and delay, but also reduces the recovery time after each failure.

1.2.3 lsMPI: an implementation in MPI (partially completed)

Though Lazy Shadowing has been shown to scale to future extreme-scale systems with our analytical models, a real implementation is still necessary for validation and performance measurement in real systems. I am implementing a prototype of Lazy Shadowing as a runtime for Message Passing Interface (MPI), which is the de facto programming paradigm for HPC. Instead of a full-feature MPI implementation, the runtime is designed to be a separate layer between MPI and user application, in order to take advantage of existing MPI performance optimizations that numerous researches have spent years on. The runtime will spawn the shadow processes at initialization phase, manage the coordination between main and shadow processes during execution, and guarantee consistency for messages and non-deterministic events. With this implementation, we will perform thorough experiments measuring its runtime overhead as well as performance under failures.

1.2.4 Smart shadowing (future)

Lazy Shadowing is a flexible and adaptive computational model that deserves further investigation. Previous studies have shown that different nodes tend to have different failure probabilities, e.g., 20% of the nodes account for 80% of the failures. The reason is complicated and may attribute to the manufacture process, heterogeneous architecture, environment factors (e.g. temperature), and/or workloads. I propose to apply machine learning techniques to learn the heterogeneity in failure distributions among a given system's nodes. Then I will study how the mapping from processes to physical cores can impact the performance and cost dichotomy. In addition, I will further consider allocating different number of shadow processes for different tasks to reduce cost while maintaining performance.

1.3 CONTRIBUTIONS

When completed, this thesis will make the following contributions:

- A reward-based framework for Lazy Shadowing to satisfy SLA requirements as well as maximize profit in Cloud Computing
- Study of optimization techniques that apply Lazy Shadowing to future extreme-scale systems
- A fully functional implementation of Lazy Shadowing for Message Passing Interface
- Exploration of Lazy Shadowing's adaptivity to different environments, workloads, and QoS requirements.

1.4 OUTLINE

The rest of this proposal is organized as follow: Chapter 2 introduces fault tolerance in large-scale distributed systems, and Chapter 3 introduces Lazy Shadowing computational model. In Chapter 4 we build a reward-based optimization framework for Lazy Shadowing in the Cloud environment. In Chapter 5, we explore techniques to efficiently apply Lazy Shadowing in extreme-scale systems. Implementation issues are discussed in Chapter 6. Adaptivity and smart shadowing are discussed in Chapter 7. Chapter 8 and 9 lists the timeline and concludes the proposal, respectively.

2.0 BACKGROUND

Rollback recovery is the dominant mechanism to achieve fault tolerance in current HPC environments [Elnozahy and et. al., 2002]. 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. Specifically, all processes coordinate with one another to produce individual states that satisfy the "happens before" communication relationship [Chandy and Ramamoorthy, 1972], which is proved to provide a consistent global state. The major benefit of coordinated checkpointing stems from its simplicity and ease of implementation. Its major drawback, however, is the lack of scalability, as it requires global coordination [Elnozahy and Plank, 2004; Riesen et al., 2010].

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 [Randell, 1975; Alvisi et al., 1999; Helary and et. al., 1997]. One hybrid approach, known as communication induced checkpointing, aims at reducing coordination overhead [Alvisi et al., 1999]. The approach, however, may cause processes to store useless states. To address this shortcoming, "forced checkpoints" have been proposed [Helary and et. al., 1997]. This approach, however, may lead to unpredictable checkpointing rates. 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 [Agarwal and et. al.; Elnozahy and Zwaenepoel, 1992; Li et al., 1994]. This can be achieved using dirty-bit page flags [Plank and Li, 1994; Elnozahy and Zwaenepoel, 1992]. Hash based incremental checkpointing, on the other hand, makes use of hashes to detect changes [chang Nam et al., 1997; Agarwal and et. al.]. Another proposed scheme, known as in-memory checkpointing, minimizes the overhead of disk access [Zheng and et. al., 2004; Zheng et al., 2012]. 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 [Ahern and et. al., 2011]. Multi-level checkpointing , which consists of writing checkpoints to multiple storage targets, can benefit from such a strategy [Moody et al., 2010]. 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 [Schneider, 1990]. Although it is initially rejected in HPC communities, replication has recently been proposed to address the deficiencies of checkpointing for upcoming extreme-scale systems [Cappello, 2009; Engelmann and Böhm, 2011]. Full and partial process replication have also been studied to augment existing checkpointing techniques, and to detect and correct silent data corruption [Stearley and et. al., 2012; Elliott and et. al.; Ferreira et al.; Fiala and et. al., 2012]. 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 [Reinhardt and et. al., 2000]. However, the focus of [Reinhardt and et. al., 2000] is on transient faults within CPU while we aim at tolerating both permanent and transient faults across all system components.

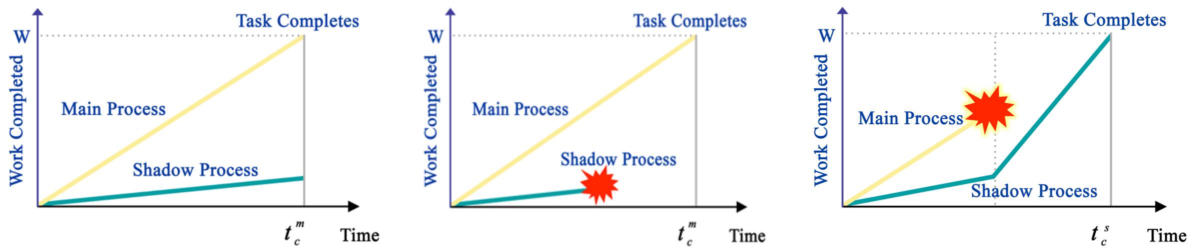
3.0 LAZY SHADOWING: A NOVEL FAULT-TOLERANT COMPUTATIONAL MODEL

It is without doubt that our understanding of how to build reliable systems out of unreliable components has led the development of robust and fairly reliable large-scale software and networking systems. The inherent instability of extreme-scale distributed systems of the future in terms of the envisioned high-rate and diversity of faults, however, calls for a reconsideration of the fault tolerance problem as a whole.

Lazy Shadowing is a novel computational model that goes beyond adapting or optimizing well known and proven techniques, and explores radically different methodologies to fault tolerance [Mills and et. al., 2014; Mills et al., 2014; Mills, 2014]. The proposed solutions differ in the type of faults they manage, their design, and the fault tolerance protocols they use. When integrated, it will lead to efficient solutions for a "tunable" resiliency that takes into consideration the nature of the data and the requirements of the application.

The basic tenet of Lazy Shadowing is to associate with each main process a suite of shadows whose size depends on the "criticality" of the application and its performance requirements. Each shadow process is an exact replica of the original main process, but it executes at a reduced rate to save power when possible. If the main process completes the task successfully, the associated shadows will be terminated immediately. If the main process fails, however, one of the shadow processes will be promoted to be a new main process, and possibly increase its execution rate to mitigate delay.

Assuming the fail-stop fault model, where a processor stops execution once a fault occurs and failure can be detected by other processors [Gärtner, 1999; Cristian, 1991], we define the Lazy Shadowing fault-tolerance model as follows:



(a) No Failure (b) Shadow Process Failure (c) Main Process Failure

Figure 3.1: Lazy Shadowing for a single task and single replica

- A main process, $P_m(W,\ \sigma_m)$, whose responsibility is to executes a task of size W at a speed of σ_m;
- A suite of shadow processes, $P_s(W,\ \sigma_b^s,\ \sigma_a^s)$ $(1 \leq s \leq \mathcal{S})$, where $\mathcal{S}$ is the size of the suite. The shadows execute on separate computing nodes. Each shadow process is associated with two execution speeds. All shadows start execution simultaneously with the main process at speed σ_b^s $(1 \leq s \leq \mathcal{S})$. Upon failure of the main process, all shadows switch their executions to σ_a^s, with one shadow being designated as the new main process. This process continues until completion of the task.

To illustrate the behavior of Lazy Shadowing, we limit the number of shadows to a single process and consider the scenarios depicted in Figure 3.1, assuming a single process failure. Figure 1(a) represents the case when neither the main nor the shadow fails. The main process, executing at a higher speed, completes the task at time t_c^m. At this time, the shadow process, progressing at a lower speed, stops execution immediately. Figure 1(b) represents the case when the shadow fails. This failure, however, has no impact on the progress of the main process, which still completes the task at t_c^m. Figure 1(c) depicts the case when the main process fails while the shadow is in progress. After detecting the failure of the main process, the shadow begins execution at a higher speed, completing the task at time t_c^s. When possible, the shadow execution speed upon failure must be set so that t_c^s does not exceed t_c^m. Given that the failure rate of an individual node is much lower than the aggregate system failure, it is very likely that the main process will always complete its execution successfully, thereby achieving fault tolerance at a significantly reduced cost of energy consumed by the shadow.

A closer look at the model reveals that shadow replication is a generalization of traditional fault tolerance techniques, namely re-execution and traditional replication. If it allows for flexible completion time, shadow replication would take advantage of the delay laxity to trade time redundancy for energy savings. It is clear, therefore, that for a large response time, Lazy Shadowing converges to re-execution, as the shadow remains idle during the execution of the main process and only starts execution upon failure. If the target response time is stringent, however, Lazy Shadowing converges to pure replication, as the shadow must execute simultaneously with the main at the same speed. The flexibility of the Lazy Shadowing model provides the basis for the design of a fault tolerance strategy that strikes a balance between task completion time and energy saving, thereby maximizing profit.

4.0 REWARD-BASED OPTIMAL LAZY SHADOWING

Cloud Computing has emerged as an attractive platform for increasingly diverse compute- and data-intensive applications, as it allows for low-entry costs, on demand resource provisioning and allocation and reduced cost of maintaining internal IT infrastructure [?]. Cloud computing will continue to grow and attract attention from commercial and public market segments. Recent studies predict annual growth rate of 17.7 percent by 2016, making cloud computing the fastest growing segment in the software industry.

As the demand for cloud computing accelerates, cloud service providers (CSPs) will be faced with the need to expand their underlying infrastructure to ensure the expected levels of performance, reliability and cost-effectiveness, resulting in a multi-fold increase in the number of computing, storage and communication components in their datacenters. The direct implication of large datacenters is increased management complexity and propensity to failure.

Failure to deliver the service as specified subjects the CSP to pay a penalty, resulting in a loss of revenue. In addition, CSPs face rising energy costs of their large-scale datacenters. This raises the question of how fault tolerance might impact power consumption and ultimately the expected profit of the service providers. In this work, we address the above trade-off challenge by studying the application of Lazy Shadowing for resilience in cloud computing.

4.1 CLOUD WORKLOAD CHARACTERISTICS

Cloud computing workload ranges from business applications and intelligence, to analytics and social networks mining and log analysis, to scientific applications in various fields of sciences and discovery. These applications exhibit different behaviors, in term of computation requirements and data access patterns. While some applications are compute-intensive, others involve the processing of increasingly large amounts of data. The scope and scale of these applications are such that an instance of a job running one of these applications requires the sequential execution of multiple computing phases; each phase consists of thousands, if not millions, of tasks scheduled to execute in parallel and involves the processing of a very large amount of data [**??**]. This model is directly reflective of the *MapReduce* computational model, which is predominately used in Cloud Computing [**?**]. An instance of this model, is depicted in Figure 4.1.

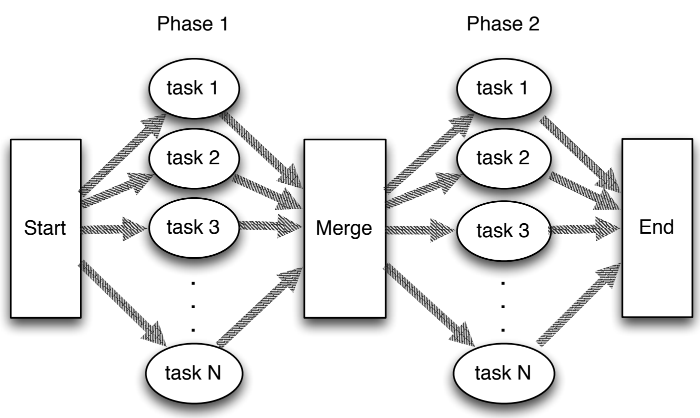


Figure 4.1: Cloud computing execution model with 2 phases.

Each task is mapped to one compute core and executes at a speed, σ. The partition of the job among tasks is such that each task processes a similar workload, W. Consequently, baring failures, tasks are expected to complete at about the same time. Therefore, the minimal response time of each task, when no failure occurs, is $t_{min} = \frac{W}{\sigma_{max}}$, where σ_{max} is the maximum speed. This is also the minimal response time of the entire phase.

As the number of tasks increases, however, the likelihood of a task failure during an execution of a given phase increases accordingly. This underscores the importance of an energy-efficient fault-tolerance model to mitigate the impact of a failing task on the overall delay of the execution phase. Lazy Shadowing is a perfect match for the needs. We can easily apply Lazy Shadowing by issuing one main and shadow pair for each task, and the execution can be performed phase by phase, just as previously described.

4.2 OPTIMIZATION FRAMEWORK

In this section, we describe a profit-based optimization framework for the cloud-computing execution model previous described. Using this framework we compute profit-optimized execution speeds by optimizing the following objective function:

$$
\begin{aligned}
&\max_{\sigma_m,\sigma_b,\sigma_a} E[profit] \\
&s.t. 0 \leq \sigma_m \leq \sigma_{max} \\
&0 \leq \sigma_b \leq \sigma_m \\
&0 \leq \sigma_a \leq \sigma_{max}
\end{aligned}
\tag{4.1}
$$

We assume that processor speeds are continuous and use nonlinear optimization techniques to solve the above optimization problem.

In order to earn profit, service providers must either increase income or decrease expenditure. We take both factors into consideration for the purpose of maximizing profit while meeting customer's requirements. In our model, we set the expected profit to be expected income minus expected expense.

$$E[\text{profit}] = E[\text{income}] - E[\text{expense}] \tag{4.2}$$

4.2.1 Reward Model

As depicted in Figure 4.2, customers expect that their job deployed on cloud finishes by a mean response time t_{R_1}. As a return, the provider earns a certain amount of reward, denoted

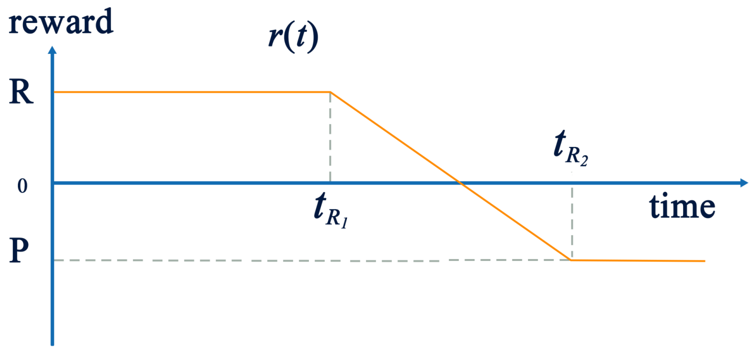


Figure 4.2: A reward function

by R, for satisfying customer's requirements. However, if the job cannot be completed by the expected response time, the provider loses a fraction of R proportional to the delay incurred. For large delay, the profit loss may translate into a penalty that the CSP must pay to the customer. In this model, the maximum penalty P is paid if the delay reaches or exceeds t_{R_2}. The four parameters, R, P, t_{R_1} and t_{R_2}, completely define the reward model.

There are two facts that the service provider must take into account when negotiating the terms of the SLA. The first is the response time of the main process assuming no failure (Figure 1(a) and Figure 1(b)). This results in the following completion time:

$$t_c^m = W/\sigma_m \tag{4.3}$$

If the main process fails (shown in Figure 1(c)), the task completion time by shadow process is the time of the failure, t_f, plus the time necessary to complete the remaining work.

$$t_c^s = t_f + \frac{W - t_f \times \sigma_b}{\sigma_a} \tag{4.4}$$

4.2.2 Failure Model

We assume that two probability density functions, $f_m(t_f)$ and $f_s(t_f)$, exist which express the probabilities of the main and shadow process failing at time t_f separately. The model does not assume a specific distribution. However, in the remainder of this paper we use an exponential probability density function, $f_m(t_f) = f_s(t_f) = \lambda e^{-\lambda t_f}$, of which the mean time between failure (MTBF) is $\frac{1}{\lambda}$.

4.2.3 Power and Energy Models

Dynamic voltage and frequency scaling (DVFS) has been widely exploited as a technique to reduce CPU dynamic power [**??**]. It is well known that one can reduce the dynamic CPU power consumption at least quadratically by reducing the execution speed linearly. The dynamic CPU power consumption of a computing node executing at speed σ is given by the function $p_d(\sigma) = \sigma^n$ where $n \geq 2$.

In addition to the dynamic power, CPU leakage and other components (memory, disk, network etc.) all contribute to static power consumption, which is independent of the CPU speed. We define static power as a fixed fraction of the node power consumed when executing at maximum speed, referred to as ρ. Hence node power consumption is expressed as $p(\sigma) = \rho \times \sigma_{max}^n + (1-\rho) \times \sigma^n$. When the execution speed is zero the machine is in a sleep state, powered off or not assigned as a resource; therefore it will not be consuming any power, static or dynamic. Throughout this thesis we assume that dynamic power is cubic in relation to speed [**??**], therefore the overall system power when executing at speed σ is defined as:

$$p(\sigma) = \begin{cases} \rho\sigma_{max}^3 + (1-\rho)\sigma^3 & \text{if } \sigma > 0 \\ 0 & \text{if } \sigma = 0 \end{cases} \tag{4.5}$$

Using the power model given by 4.5, the energy consumed by a process executing at speed σ during an interval T is given by

$$E(\sigma, T) = p(\sigma) \times T \tag{4.6}$$

Corresponding to 3.1, there are three failure cases to consider: main and shadow both succeed, shadow fails and main fails. As described earlier, the case of both the main and shadow failing is very rare and will be ignored. The expected energy consumption for a single task is then the weighted average of the expected energy consumption in the three cases.

First consider the case where no failure occurs and the main process successfully completes the task at time t_c^m, corresponding to 1(a).

$$\begin{aligned} E_1 =&(1 - \int_0^{t_c^m} f_m(t)dt) \times (1 - \int_0^{t_c^m} f_s(t)dt)\times \\ &(E(\sigma_m, t_c^m) + E(\sigma_b, t_c^m)) \end{aligned} \tag{4.7}$$

The first line is the probability of fault-free execution of the main process and shadow process. Then we multiple this probability by the energy consumed by the main and the shadow process during this fault free execution, ending at t_c^m.

Next, consider the case where the shadow process fails at some point before the main process successfully completes the task, corresponding to 1(b).

$$\begin{aligned} E_2 =&(1 - \int_0^{t_c^m} f_m(t)dt)\times \\ &\int_0^{t_c^m} (E(\sigma_m, t_c^m) + E(\sigma_b, t)) \times f_s(t)dt \end{aligned} \tag{4.8}$$

The first factor is the probability that the main process does not fail, and the probability of shadow fails is included in the second factor which also contains the energy consumption since it depends on the shadow failure time. Energy consumption comes from the main process until the completion of the task, and the shadow process before its failure.

The one remaining case to consider is when the main process fails and the shadow process must continue to process until the task completes, corresponding to Figure 1(c).

$$\begin{aligned} E_3 =&(1 - \int_0^{t_c^m} f_s(t)dt) \times \int_0^{t_c^m} (E(\sigma_m, t)+ \\ &E(\sigma_b, t) + E(\sigma_a, t_c^s - t))f_m(t)dt \end{aligned} \tag{4.9}$$

Similarly, the first factor expresses the probability that the shadow process does not fail. In this case, the shadow process executes from the beginning to t_c^s when it completes the task. However, under our "at most one failure" assumption, the period during which shadow process may fail ends at t_c^m, since the only reason why shadow process is still in execution after t_c^m is that main process has already failed. There are three parts of energy consumption, including that of main process before main's failure, that of shadow process before main's failure, and that of shadow process after main's failure, all of which depend on the failure occurrence time.

The three equations above describe the expected energy consumption by a pair of main and shadow processes for completing a task under different situations. However, under our system model it might be the case that those processes that finish early will wait idly and consume static power if failure delays one task. If it is the case that processes must wait for all tasks to complete, then this energy needs to be accounted for in our model. The probability of this is the probability that at least one main process fails, referred to as the system level failure probability.

$$P_f = 1 - (1 - \int_0^{t_c^m} f_m(t)dt)^N \tag{4.10}$$

Hence, we have the fourth equation corresponding to the energy consumed while waiting in idle.

$$\begin{aligned} E_4 = &(1 - \int_0^{t_c^m} f_m(t)dt) \times (1 - \int_0^{t_c^m} f_s(t)dt) \times \\ &2P_f \times E(0, t_c^j - t_c^m) + \int_0^{t_c^m} f_s(t)dt \times \\ &(1 - \int_0^{t_c^m} f_m(t)dt) \times P_f \times E(0, t_c^j - t_c^m) \end{aligned} \tag{4.11}$$

Corresponding to the first case, neither main process nor shadow process fails, but both of them have to wait in idle from task completion time t_c^m to the last task's completion (by a shadow process) with probability P_f. Under the second case, only the main process has to wait if some other task is delayed since its shadow process has already failed. These two aspects are accounted in the first and last two lines in E_4 separately. We use the expected shadow completion time t_c^j as an approximation of the latest task completion time which is also the job completion time.

By summing these four parts and then multiplying it by N we will have the expected energy consumed by Shadow Replication for completing a job of N tasks.

$$E[\text{energy}] = N \times (E_1 + E_2 + E_3 + E_4) \tag{4.12}$$

4.2.4 Income and Expense Models

The income is the reward paid by customer for the cloud computing services that they utilize. It depends on the reward function $r(t)$, depicted in 4.2, and the actual job completion time. Therefore, the income should be either $r(t_c^m)$, if all main processes can complete without failure, or $r^*(t_c^s)$ otherwise. It is worth noting that the reward in case of failure should be calculated based on the last completed task, which we approximate by calculating the expected time of completion allowing us to derive the expected reward, i.e. $r^*(t_c^s) = \frac{\int_0^{t_c^m} r(t_c^s) \times f_m(t)dt}{\int_0^{t_c^m} f_m(t)dt}$. Therefore the income is estimated by the following equation.

$$E[\text{income}] = (1 - P_f) \times r(t_c^m) + P_f \times r^*(t_c^s) \tag{4.13}$$

The first part is the reward earned by the main process times the probability that all main processes would complete tasks without failure. If at least one main process fails, that task would have to be completed by a shadow process. As a result, the second part is the reward earned by shadow process times the system level failure probability.

If C is the charge expressed as dollars per unit of energy consumption (e.g. kilowatt hour), then the expected expenditure would be C times the expected energy consumption for all N tasks:

$$E[\text{expense}] = C \times E[\text{energy}] \tag{4.14}$$

4.3 PROFIT-AWARE STRETCHED REPLICATION

Unlike traditional replication Shadow Replication is dependent upon failure detection, enabling the replica to increase its execution speed upon failure and maintain the targeted response time thus maximizing profit. While this is the case in many computing environments, there are cases where failure detection may not be possible. To address this limitation, we propose profit-aware stretched replication, whereby both the main process and the shadow execute independently at stretched speeds to meet the expected response time, without the need for failure detection. In profit-aware stretched replication both the main and shadow execute at speed σ_r, found by optimizing the profit model. For both traditional replication and stretched replication, the task completion time is independent of failure and can be directly calculated as:

$$t_c = \frac{W}{\sigma_{max}} \text{ or } t_c = \frac{W}{\sigma_r} \tag{4.15}$$

Since all tasks will have the same completion time, the job completion time would also be t_c. Further, the expected income, which depends on negotiated reward function and job completion time, is independent of failure:

$$E[income] = r(t_c) \tag{4.16}$$

Since both traditional replication and profit-aware stretched replication are special cases of our Shadow Replication paradigm where $\sigma_m = \sigma_b = \sigma_a = \sigma_{max}$ or $\sigma_m = \sigma_b = \sigma_a = \sigma_r$ respectively, we can easily derive the expected energy consumption using 4.12 with E_4 fixed at 0 and then compute the expected expense using 4.14.

4.4 RE-EXECUTION

Contrary to replication, re-execution initially assigns a single process for the execution of a task. If the original task fails, the process is re-executed. In the cloud computing execution framework this is equivalent to a checkpoint/restart, the checkpoint is implicitly taken at the end of each phase and because the tasks are loosely coupled they can restart independently.

Based on the one failure assumption, two cases must be considered to calculate the task completion time. If no failure occurs, the task completion time is:

$$t_c = \frac{W}{\sigma_{max}} \tag{4.17}$$

In case of failure, however, the completion time is equal to the sum of the time elapsed until failure and the time needed for re-execution. Again, we use the expected value $t_f^* = \frac{\int_0^{t_c} t \times f_m(t)dt}{\int_0^{t_c} f_m(t)dt}$ to approximate the time that successfully completed processes have to spend waiting for the last one.

Similar to Shadow Replication, the income for re-execution is the weighted average of the two cases:

$$E[\text{income}] = (1 - P_f) \times r(t_c) + P_f \times r(t_c + t_f^*) \tag{4.18}$$

For one task, if no failure occurs then the expected energy consumption can be calculated as

$$E_5 = (1 - \int_0^{t_c} f_m(t)dt) \times (E(\sigma_{max}, t_c) + P_f \times E(0, t_f^*)) \tag{4.19}$$

If failure occurs, however, the expected energy consumption can be calculated as

$$E_6 = \int_0^{t_c} (E(\sigma_{max}, t) + E(\sigma_{max}, t_c)) \times f_m(t)dt \tag{4.20}$$

Therefore, the expected energy consumption by re-execution for completing a job of N tasks is

$$E[energy] = N \times (E_5 + E_6) \tag{4.21}$$

4.5 PERFORMANCE EVALUATION

This section evaluates the expected profit of each of the fault tolerance methods discussed above under different system environment. We have identified 5 important parameters which affect the expected profit:

- Static power ratio ρ, which determines the portion of power that is unaffected by the execution speed.

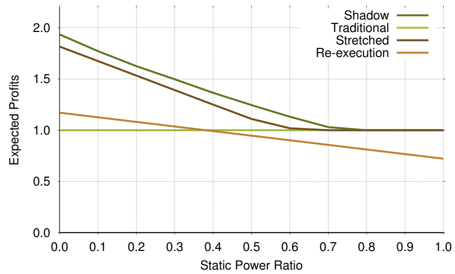
- SLA - The amount of reward, penalty and the required response times.
- N - The total number of tasks.
- MTBF - The reliability of an individual node.
- Workload - The size, W, of each individual task.

Without loss of generality, we normalize σ_{max} to be 1, so that all the speeds can be expressed as a fraction of maximum speed. Accordingly, the task workload W is also adjusted such that it is equal to the amount of time (in hours) required for a single task, preserving the ratios expressed in 4.3 and 4.4. The price of energy is assumed to be 1 unit. We assume that R in our reward model is linearly proportional to the number of tasks N and the maximal reward for one task is 3 units, so the total reward for a job is $3 \times N$ units. However, for the analysis we look at the average of expenditure and income on each task by dividing the total expenditure and income by N. In our basic configuration we assume that the static power ratio is 0.5, the task size is 1 hour, the node MTBF 5 is years, the number of tasks is 100000, and the response time thresholds for maximal and minimal rewards are 1.3 hours and 2.6 hours respectively. Since the maximal power consumption is 1 unit, the energy needed for the task with one process at maximal speed is also 1 unit.

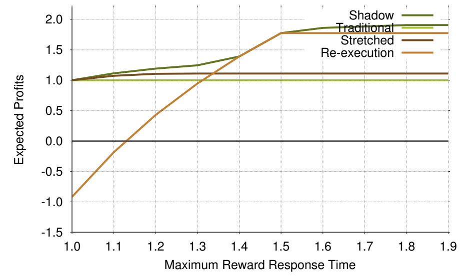
With various architectures and organizations, servers deployed at different data centers will have different characteristics in terms of power consumption. The static power ratio is used to abstract the amount of static power consumed versus dynamic power.

The potential profit gains achievable by using profit-aware replication techniques decreases as static power increases, as is shown in Figure 3(a). The reason is that our profit-aware techniques rely upon the fact that one can reduce energy costs by adjusting the execution speeds. Modern systems have a static power between 40%-70% and it is reasonable to suspect that this will continue to be the case. Within this target range of static power, Shadow Replication can achieve, on average, 19.3% more profit than traditional replication, 8.9% more than profit-aware stretched replication, and 28.8% more than re-execution.

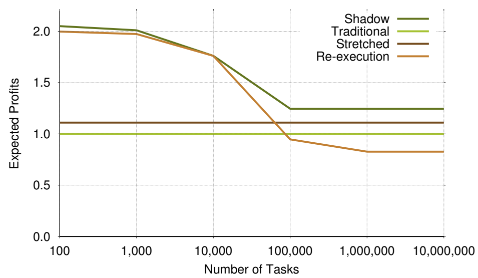
Figure 3(b) shows the effect that targeted response time has upon the profitability of each fault tolerance method. We vary the first threshold t_{R_1} from the minimal response time t_{min} to $1.9t_{min}$, and set the second threshold t_{R_2} to be always $2t_{R_1}$. Compared to traditional replication, all the other methods increase their profit as the targeted response time increases,



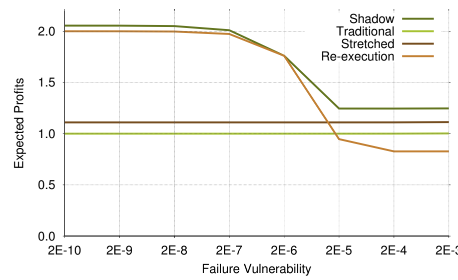
(a) Profit for different static power ratio. MTBF=5 years, N=100000, W=1 hour, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.



(b) Profit for different response time threshold. ρ=0.5, MTBF=5 years, N=100000, W=1 hour.



(c) Profit for different task size over MTBF. ρ=0.5, N=100000, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.



(d) Profit for different task size over MTBF. ρ=0.5, N=100000, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

this is expected because each of the other techniques can make use of increased laxity in time to increase profit. Re-execution is the most sensitive to the target response time since it fully relies upon time redundancy, showing that it should only be used when the targeted response time is *not* stringent. Again, Shadow Replication always achieves more profit than traditional replication and profit-aware stretched replication, and the profit gains are 52.8% and 39.0% on average.

Figure 3(c) confirms that for small number of tasks re-execution is more profitable than replication. However, re-execution is not scalable as its profit decreases rapidly after N reaches 10000. At the same time, traditional replication and profit-aware stretched replication are not affected by the number of tasks because neither are affected by the system level failure rate. On average, Shadow Replication achieves 43.5%, 59.3%, and 18.4% more profits than profit-aware stretched replication, traditional replication and re-execution, respectively.

The ratio between task size and node MTBF represents the tasks vulnerability to failure, specifically it is an approximation of the probability that failure occurs during the execution of the task. In our analysis we found that increasing task size will have the same effect as reducing node MTBF. Therefore, we analyze these together using the vulnerability to failure, allowing us to analyze a wider range of system parameters. As expected re-execution is desired when the vulnerability to failure is low. As always, Shadow Replication can adjust its execution strategy to maximize the profits, as shown in Figure 3(d).

Lastly, we evaluate the expected profit of each resilience technique using three different benchmark applications representing a wide range of application [?]: Business Intelligence, Bioinformatics and Recommendation System. Using the results of the experiments reported in [?], we derived that the time required to process data for above application types are 3.3 MB/s, 6.6 MB/s, and 13.2 MB/s, respectively.

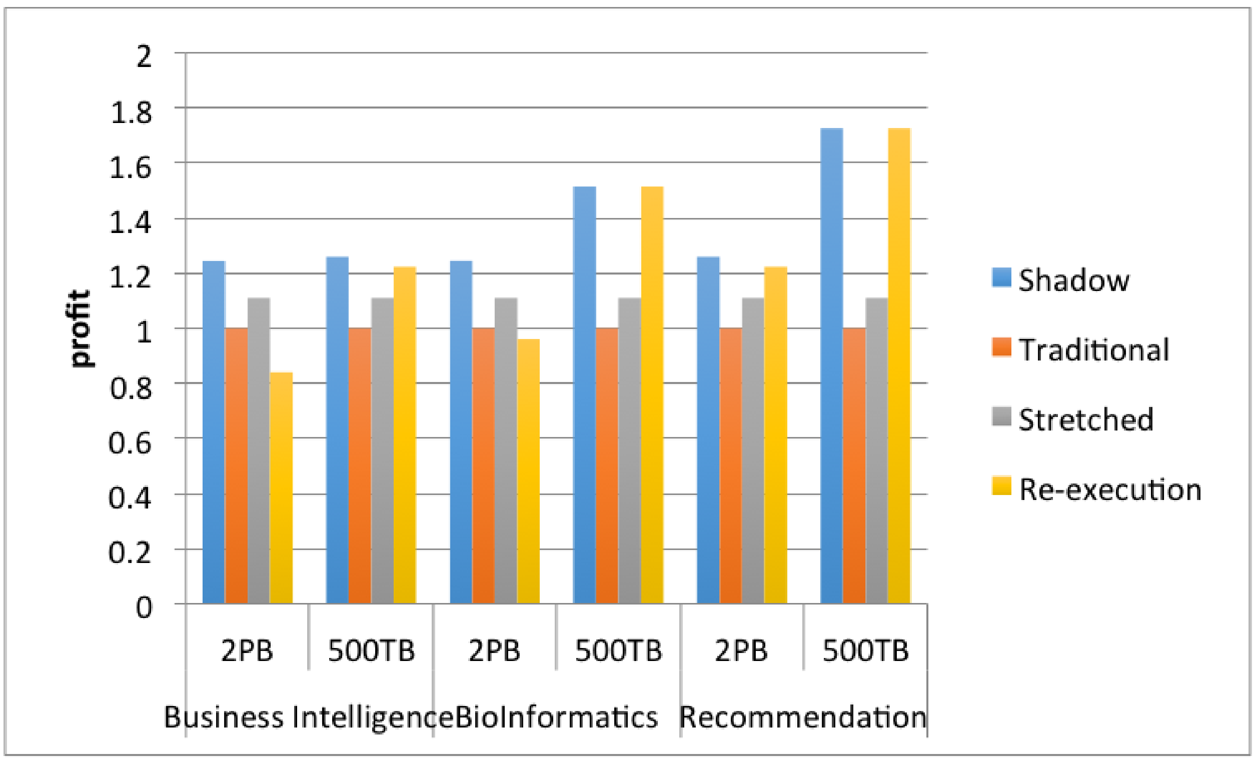


Figure 4.3: Application comparison. ρ=0.5, N=500000, t_{R_1}=1.3t_{min}, t_{R_2}=2.6t_{min}.

In Figure 4.3 we compare the expected profit for each application using each of the 4 resilience techniques. We consider two data sizes expected in future cloud computing environments, 500TB and 2PB. The figure shows that for business intelligence applications, Shadow Replication achieves significantly larger profits for both data sizes. This is because business intelligence applications tend to be IO intensive resulting in longer running tasks. Whereas recommendation systems tend to require little data IO resulting in shorter running tasks making re-execution as good as Shadow Replication. Bioinformatics tends to be in between these two applications resulting in shadow computing performing better when processing large datasets (2 PB) but not outstanding on smaller datasets (500 TB). The take away from this evaluation is that for the shown system parameters if phase execution is short, then re-execution performs as well as Shadow Replication. Alternatively, if a phase is long (20 minutes or greater), then Shadow Replication can be as much as 47.9% more profitable than re-execution. The previous sensitivity analysis can be used to extrapolate expected profit for different system parameters.

4.6 SUMMARY

To assess the performance of the Lazy Shadowing, an analytical framework is developed and an extensive performance evaluation study is carried out. In this study, system properties that affect the profitability of fault tolerance methods, namely failure rate, targeted response time and static power, are identified. The failure rate is affected by the number of tasks and vulnerability of the task to failure. The targeted response time represents the clients' desired job completion time. Our performance evaluation shows that in all cases, Shadow Replication outperforms existing fault tolerance methods. Furthermore, shadow replication will converge to traditional replication when target response time is stringent, and to re-execution when target response time is relaxed or failure is unlikely.

5.0 APPLYING LAZY SHADOWING TO EXTREME-SCALE SYSTEMS

6.0 LSMPI: AN IMPLEMENTATION IN MPI

7.0 SMART SHADOWING

8.0 TIMELINE OF PROPOSED WORK

Table 8.1: Timeline of Proposed Work.

Date	**Content**	**Deliverable results**
Jan - Feb	Explore restoring in approximate computing in Section **??** of Chapter **??**	Pin-based framework for restoring approximation
Mar - May	Integrate restoring with information leakage in Section **??** of Chapter **??**	Experimental data of memory performance and security
Jun - Sep	Study restoring in Hybrid Memory Cube (HMC) in Section **??** of Chapter **??**	Modified simulator of temperature effect of restoring in HMC
Jul - Oct	Thesis writing	Thesis ready for defense
Oct - Dec	Thesis revising	Completed thesis

The proposed works will be undertaken as shown in the Table 8.1. I will start the effort with task (1) to develop Pin-based framework for approximate computing of restoring. This task involves program annotation, chip generation, QoS evaluation, and conventional performance simulation, etc. While multiple complicated subtasks are there, this task has been partially finished, and will not take much time to complete. Afterwards, I'll move to task (2) to study information leakage in restoring scenario, and this task is partially on basis of the previous approximation work. With the completion of task (2), the overall goal of exploring DRAM restoring in application level have been reached, and then I'll start the study restoring's temperature effect in HMC, i.e., task (3). The general infrastructure can be borrowed from my previous HMC work [**?**]. This task might be performed concurrently with other jobs, and thus might take more time to finish. At the end of task (3), the holistic exploration of DRAM restoring is considered finished, and thus I'll summarize all the tasks into my final thesis.

9.0 SUMMARY

Current fault tolerance approaches rely exclusively on either time or hardware redundancy to hide failures from being seen by users. Rollback recovery, which exploits time redundancy, requires full or partial re-execution when failure occurs. Such an approach can incur a significant delay, and high power costs due to extended execution time. On the other hand, Process Replication relies on hardware redundancy and executes multiple instances of the same task in parallel to guarantee completion with minimal delay. This solution, however, requires a significant increase in hardware resources and increases the power consumption proportionally.

BIBLIOGRAPHY

Agarwal, Saurabh and al. et. . Adaptive incremental checkpointing for massively parallel systems. In *ICS 04*, St. Malo, France.

Ahern, Sean and al. et. . Scientific discovery at the exascale, a report from the doe ascr 2011 workshop on exascale data management, analysis, and visualization, 2011.

Alvisi, L., Elnozahy E., Rao S., Husain S. A., and Mel A.de . An analysis of communication induced checkpointing. In *Fault-Tolerant Computing*, 1999. doi: 10.1109/FTCS.1999.781058.

Cappello, Franck. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009.

Chandy, K.M. and Ramamoorthy C.V. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on*, C-21(6):546–556, June 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009007.

Nam, Hyochang , Kim Jong, Lee Sunggu, and Lee Sunggu. Probabilistic checkpointing. In *In Proceedings of Intl. Symposium on Fault-Tolerant Computing*, pages 153–160, 1997.

Cristian, Flavin. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, February 1991. ISSN 0001-0782. doi: 10.1145/102792.102801.

Elliott, James and al. et. . Combining partial redundancy and checkpointing for HPC. In *ICDCS '12*, Washington, DC, US. ISBN 978-0-7695-4685-8. doi: 10.1109/ICDCS.2012.56.

Elnozahy, E. and al. et. . A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/568522.568525.

Elnozahy, Elmootazbellah and Zwaenepoel Willy. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *TC*, 41:526–531, 1992.

Elnozahy, E.N. and Plank J.S. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *DSC*, 1(2):97 – 108, april-june 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.15.

Engelmann, Christian and Böhm Swen. Redundant execution of hpc applications with mr-mpi. In *PDCN*, pages 15–17, 2011.

Ferreira, Kurt, Stearley Jon, Laros James H., III, Oldfield Ron, Pedretti Kevin, Brightwell Ron, Riesen Rolf, Bridges Patrick G., and Arnold Dorian. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063443.

Fiala, David and al. et. . Detection and correction of silent data corruption for large-scale high-performance computing. SC, Los Alamitos, CA, USA, 2012. ISBN 978-1-4673-0804-5.

Gärtner, Felix C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, March 1999. ISSN 0360-0300. doi: 10.1145/311531.311532.

Helary, J and al. et. . Preventing useless checkpoints in distributed computations. In *RDS*, 1997. doi: 10.1109/RELDIS.1997.632814.

Li, K., Naughton J. F., and Plank J. S. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, August 1994. ISSN 1045-9219. doi: 10.1109/71.298215.

Mills, Bryan. *Power-Aware Resilience for Exascale Computing.* PhD thesis, University of Pittsburgh, 2014.

Mills, Bryan and al. et. . Shadow computing: An energy-aware fault tolerant computing model. In *ICNC*, 2014.

Mills, Bryan, Znati Taieb, Melhem Rami, Grant Ryan E., and Ferreira Kurt B. Energy consumption of resilience mechanisms in large scale systems. In *Parallel, Distributed and Network-Based Processing (PDP), 22st Euromicro International Conference*, Feb 2014.

Moody, Adam, Bronevetsky Greg, Mohror Kathryn, and Supinski Bronis. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, pages 1–11, 2010. ISBN 978-1-4244-7559-9. doi: http://dx.doi.org/10.1109/SC.2010.18.

Plank, J.S. and Li Kai. Faster checkpointing with n+1 parity. In *Fault-Tolerant Computing*, pages 288–297, June 1994. doi: 10.1109/FTCS.1994.315631.

Randell, B. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808467.

Reinhardt, Steven K and al. et. . *Transient fault detection via simultaneous multithreading*, volume 28. ACM, 2000.

Riesen, Rolf, Ferreira Kurt, Stearley Jon R., Oldfield Ron, III James H. Laros, Pedretti Kevin T., and Brightwell Ron. Redundant computing for exascale systems, December 2010.

Schneider, Fred B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. doi: 10.1145/98163.98167.

Stearley, J. and al. et. . Does partial replication pay off? In *DSN-W*, pages 1–6, June 2012. doi: 10.1109/DSNW.2012.6264669.

Zheng, G., Ni Xiang, and Kal L. V. A scalable double in-memory checkpoint and restart scheme towards exascale. In *DSN-W*, pages 1–6, June 2012. doi: 10.1109/DSNW.2012.6264677.

Zheng, Gengbin and al. et. . FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing*, pages 93–103, 2004. doi: 10.1109/CLUSTR.2004.1392606.