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

by

Xiaolong Cui

B.E. in Computer Science, Xi'an Jiaotong University, 2012

M.S. in Computer Science, University of Pittsburgh, 2017

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Xiaolong Cui

It was defended on

November 10, 2017

and approved by

Dr. Taieb Znati, Department of Computer Science, 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, University of Pittsburgh,

Co-advisor: 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, 2017

Two major trends in large-scale computing are the rapid growth in HPC with in particular an international exascale initiative, and the dramatic expansion of Cloud infrastructures accompanied by the Big Data passion. To satisfy the continuous demands for increasing computing capacity, future extreme-scale systems will embrace a multi-fold increase in the number of computing, storage, and communication components, in order to support an unprecedented level of parallelism. Despite the capacity and economies benefits, making the upward transformation to extreme-scale poses numerous scientific and technological challenges, two of which are the power consumption and fault tolerance. With the increase in system scale, failure would become a norm rather than an exception, driving the system to significantly lower efficiency with unforeseen power consumption.

This thesis aims at simultaneously addressing the above two challenges by introducing a novel fault-tolerant computational model, referred to as *Leaping Shadows*. Based on Shadow Replication, Leaping Shadows associates with each main process a suite of coordinated shadow processes, which execute in parallel but at differential rates, to deal with failures and meet the QoS requirements of the underlying application under strict power/energy constraints. In failure-prone extreme-scale computing environments, this new model addresses the limitations of the basic Shadow Replication model, and achieves adaptive and power-aware fault tolerance that is more time and energy efficient than existing techniques.

In this thesis, we first present an analytical model based optimization framework that demonstrates Shadow Replication's adaptivity and flexibility in achieving multi-dimensional

QoS requirements. Then, we introduce Leaping Shadows as a novel power-aware fault tolerance model, which tolerates multiple types of failures, guarantees forward progress, and maintains a consistent level of resilience. Lastly, the details of a Leaping Shadows implementation in MPI is discussed, along with extensive performance evaluation that includes comparison to checkpoint/restart. Collectively, these efforts advocate an adaptive and power-aware fault tolerance alternative for future extreme-scale computing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

For my beloved families.

1.0 INTRODUCTION

By applying computing to all kinds of areas, information technology (IT) has been transforming the way we understand and change the world. Over the past few decades, IT has realized the fast analysis of massive quantities of data and the rapid transmission of tremendous amount of information, facilitating countless advances in areas of science and technology. As our reliance on IT continues to increase, the complexity and urgency of the problems our society will face in the future necessitate the building of more powerful and ubiquitous computing systems.

Since CPU frequency flattened out in early 2000s, parallelism from all levels has become the "golden rule" to boost performance in the computer industry. Among the different types of modern computing systems, large-scale High Performance Computing (HPC) and Cloud Computing systems are the two most powerful ones. For both of them, the extraordinary computing capacity attributes to a massive amount of parallelism, which is achieved by equipping the system with hundreds or thousands of CPUs, accelerators, communication devices, storage components, etc.

With the increase in the number of components, it become more and more challenging for researchers to continuously offer more computing capacity with sustainable performance and reliability. As today's HPC and Cloud systems grow to meet tomorrow's demands, 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. At the same time, the rapid growing power consumption is another major concern. It is reported that the power required to run the machines as well as cool them has become the largest cost factor in a large-scale system's operating expenses [119]. In future-generation large-scale computing systems, failure would become a norm rather than

an exception, driving the system to significantly lower efficiency with unprecedented amount of power consumption.

1.1 EXTREME-SCALE COMPUTING

Nowadays, large-scale computing systems are embracing two transformative trends, i.e., the rapid growth in HPC with in particular an international exascale initiative, and the dramatic expansion of Cloud infrastructures accompanied by the Big Data explosion. With concerted efforts from researchers in various disciplines, a race is underway in the HPC community to build the world's first exascale machine, featuring a computing capability of exaFLOPS, to accelerate scientific discoveries and breakthroughs. It is projected that within the next decade an exascale machine will achieve billion-way parallelism by using one million sockets each supporting 1,000 cores [4, 90].

Similarly, remarkable expansion is happening in Cloud Computing. Due to its incomparable advantages, such as low entry cost and on demand resource provisioning, Cloud Computing has become the fastest growing segment in the software industry [8]. 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, numerous large-scale data centers have been and are being built by IT companies to exploit the power and economies of scale. For example, Google and Rackspace have hundreds of thousands of web servers in dedicated geo-distributed data centers to support their business.

The path to future extreme-scale computing involves several major roadblocks and numerous challenges inherent to the complexity and scale of these systems. The system scale needed to address our future computing needs will come at the cost of increasing unpredictability and operating expenses. As we approach extreme-scale, two of the biggest challenges will be power consumption and system resilience, both being direct consequences of the dramatic increase in the number of system components [9, 129].

As a result of the continuous growth in system scale, there has been a steady rise in

power consumption in large-scale systems. Google engineers, operating thousands of servers, warned that if power consumption continues to grow, power costs can easily overtake hardware costs by a large margin [12]. Figure 1 reveals the same concern in supercomputers. In 2005, the peak power consumption of a single supercomputer reached 3.2 Megawatts (MW). This number was doubled only after 5 years, and further climbed to 17.8 MW in 2013 with a machine of 3,120,000 cores [1]. Recognizing this upward trend, the U.S. Department of Energy has set 20 MW as the power limit for future exascale systems, so that a power budget of $20 million per year is preserved for any supercomputer [4]. Although largely pragmatic, this constraint challenges the HPC community to design future systems capable of sustaining 50 GFlops per Watt (GF/W). According to the Green500 list released in June 2017, the most energy-efficient supercomputer is the new TSUBAME 3.0 at the Tokyo Institute of Technology [1]. It achieved 14.110 GF/W during its 1.998-petaflop Linpack performance run. To enable future exascale system, combined efforts from hardware, OS, and software must improve energy efficiency by a factor of over 3.5X, making system power a leading design constraint on the path to exascale.

Due to the expected complexity and scale of future extreme-scale systems, another major roadblock is the increasing propensity of the system to diverse types of failures. Regardless of the reliability of individual component, the system level failure rate will continue to increase as the number of components increases, possibly by several orders of magnitude. 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, implying that many failures will occur every day [15]. Without an efficient fault tolerance mechanism, faults will be so frequent that the applications running on the systems will be constantly interrupted and restarted.

Today, fault tolerance in computing systems mainly relies on 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 [47, 74, 28]. Although checkpoint/restart is the most widely used technique in today's HPC systems, several studies predict that it is not likely to scale to future extreme-scale systems [54, 44, 103]. Given the anticipated increase in system level failure rates and the time to checkpoint large-

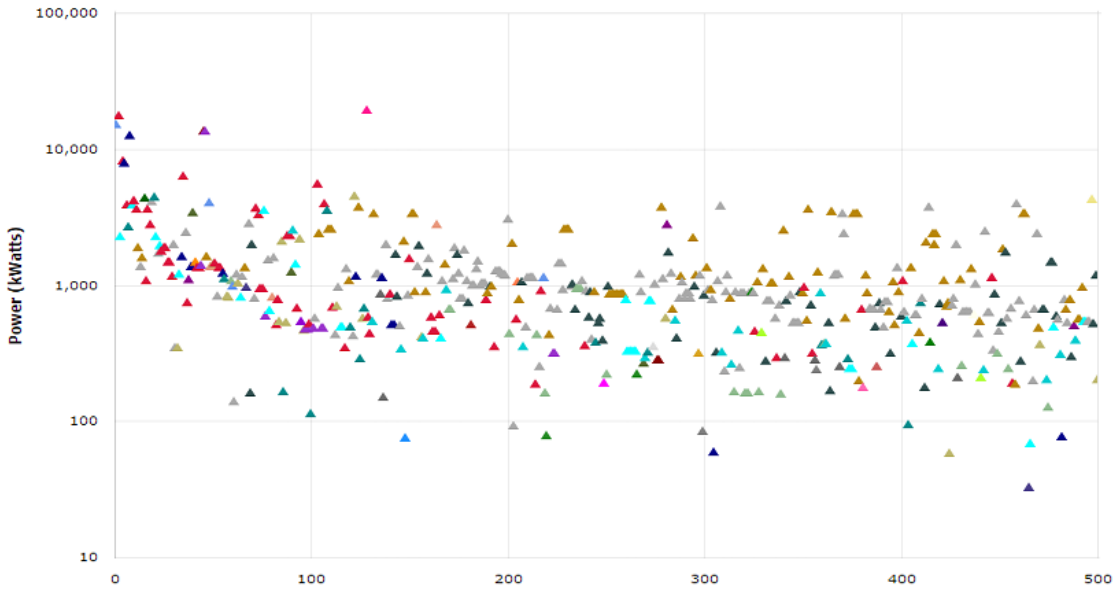


Figure 1: System power consumption of the top 500 supercomputers as of June 2017 [1].

scale compute-intensive and data-intensive applications, the time required to periodically checkpoint an application and restart its execution will approach the system's MTBF [23]. Consequently, applications will make little forward progress, thereby reducing considerably the overall system efficiency and wasting a lot of energy. Table 1 shows that at 100k nodes, only 35% of the time will be spent on useful work, while 55% of the time will be wasted on checkpointing and restarting. Furthermore, the nature and diversity of failures in extreme-scale systems are such that checkpoint/restart alone may not be an adequate approach. Based on this observation, recent work has proposed state machine replication as a scalable solution to handling diverse types of faults [55, 54].

State machine replication exploits hardware redundancy and executes multiple instances of a task across compute nodes in parallel to overcome failure [13, 137, 54]. Although this approach is extensively used to deal with failures in Cloud Computing and mission critical systems, it has never been used in any production HPC system due to its waste of resources. To replicate each process, state machine replication not only requires twice the amount of

Table 1: Projection of time spent on each part when using checkpoint/restart to guard a 168-hour job at different system scales [55]. 5 years MTBF per node, 15 minutes checkpoint time. Optimal checkpoint interval is derived using Daly's model [36].

# Nodes	work	checkpoint	recompute	restart
100	96%	1%	3%	0%
1,000	92%	7%	1%	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

compute nodes at least, but also increases the power consumption proportionally, which might exceed any imposed power budget.

Based on above analysis, neither of the two existing approaches would efficiently tolerate failure for future extreme-scale systems. And unfortunately, neither of them addresses the power cap issue. Therefore, achieving high resilience to failures under strict power constraints is a daunting and critical challenge that requires new fault tolerance models with scalability and power-awareness in mind.

1.2 RESEARCH OVERVIEW

There is a delicate interplay between fault tolerance and power consumption. Process replication and checkpoint/restart 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 [27, 146]. The trade-off between fault free operation and optimal power consumption has been explored in the literature [94, 97]. 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, this thesis aims at developing a novel fault-tolerant computational model that simultaneously addresses the power and resilience challenges for emerging extreme-scale computing systems. Our goal is to study the viability of and provide the justification for the following **thesis statement**:

"It is possible to persistently achieve Quality of Service with sustainable system efficiency, while operating under diverse types of failures and stringent power constraints, in future extreme-scale computing systems."

We seek to achieve this objective by developing an adaptive and power-aware fault tolerance model, referred to as *Leaping Shadows*. It builds on top of the recently proposed Shadow Replication model, and exploits novel techniques to address the limitations of Shadow Replication to meet the performance and resilience requirements of extreme-scale computing. With an extensive evaluation through the combination of analytical models and empirical experiments, this thesis demonstrates the viability of Leaping Shadows to achieve high tolerance to failure that is more time and energy efficient that state-of-the-art approaches.

The basic tenet of Shadow Replication is to associate with each original process a suite of coordinated "shadow processes", whose size depends on the criticality of the application and its QoS requirements. To tolerate failure while minimizing energy, the shadows are scheduled to execute in parallel with the original process, but on different nodes and at reduced rates. A shadow is an exact replica of its associated original process. If an original process fails, one of the associated shadows takes over and resumes the execution, with a potential increase in execution rate to mitigate delay.

Relying on Dynamic Voltage and Frequency Scaling (DVFS) to control execution rates, Mills studied the execution dynamics of Shadow Replication and its performance in HPC systems [96, 98, 95]. Through the use of modeling, simulation, and experimentation, Mills

demonstrated that Shadow Replication can achieve resilience more efficiently than both checkpoint/restart and traditional state machine replication when power is limited. However, Mills' study is limited to HPC systems and focuses exclusively on minimizing energy consumption with constraints on time to completion. In contrast, QoS requirements can be expressed in multiple dimensions that go beyond time and energy. Furthermore, the Shadow Replication model has several drawbacks that impact performance, expose vulnerability, and limit fault tolerance capability in extreme-scale, failure-prone computing environments.

To address the above limitations, this thesis builds on the computational model of Shadow Replication, and seeks to tolerate high rate of diverse types of failures in emerging power-constrained HPC and Cloud systems, while guaranteeing system efficiency and application QoS. Specifically, we have complemented Mills' work by developing analytical models and optimization frameworks for different objectives in the Cloud environments [34]. This work demonstrates Shadow Replication's adaptivity in balancing the trade-offs among performance, power, and resilience, and highlights its flexibility in achieving multi-dimensional QoS requirements. Additionally, we have proposed and studied the Leaping Shadows model which is depicted in Figure 2. Leaping Shadows exploits novel techniques of shadow collocation, leaping, and rejuvenation to address the *divergence* and *vulnerability* issues with Shadow Replication. Furthermore, we have applied Leaping Shadows to tolerate Silent Data Corruption (SDC), which is another class of failure that has become prevalent in current large-scale systems [55]. Last but not least, a prototype of Leaping Shadows in Message Passing Interface (MPI) has been implemented, to validate the computational model as well as measure its performance in real environments.

1.2.1 Contributions

This thesis consists of the following main contributions.

Reward-based optimal Shadow Replication. Shadow Replication is an adaptive and flexible computational model that can achieve multi-dimensional QoS requirements. The major challenge 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

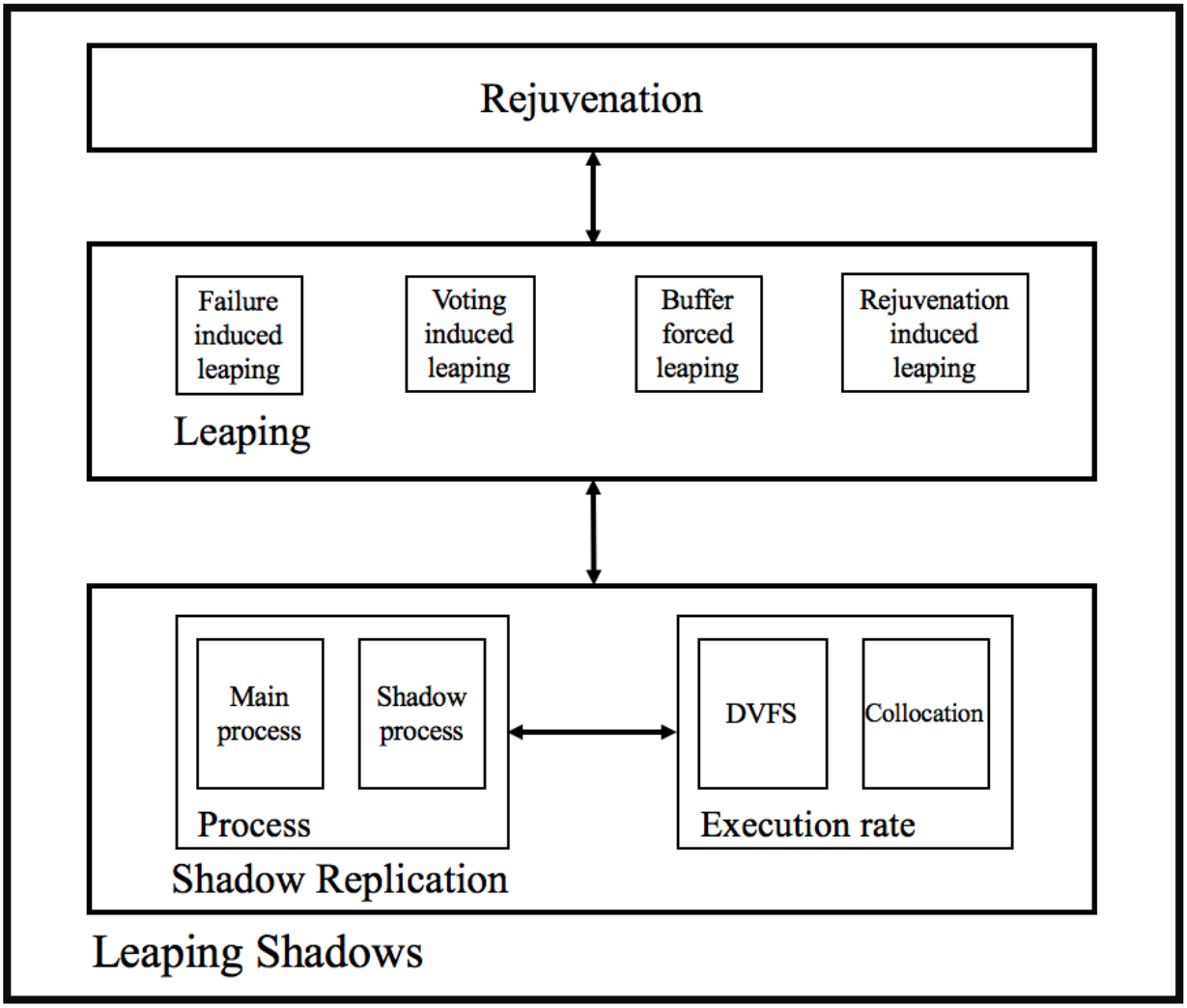


Figure 2: Overview of the Leaping Shadows computational model.

consumption, or their combinations. In this work we focus on the Service Level Agreement (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 [34, 33].

The Leaping Shadows model. Enabling Shadow Replication for resiliency in extreme-scale computing brings about a number of challenges and design decisions, including the applicability of this concept to a large number of tasks executing in parallel, the effective way to control shadows' execution rates, and maintenance of consistent resilience through a large number of failures. Taking into consideration the main characteristics of compute-intensive and highly-scalable applications, we devise novel ideas of shadow collocation, leaping, and rejuvenation, and integrate them with Shadow Replication to establish a more efficient and scalable model of Leaping Shadows [35].

Tolerance of Silent Data Corruption (SDC) with extended Leaping Shadows. Different from crash failures, which terminate the execution on a faulty processor, SDC allows a faulty processor to continue to completion but may silently generate incorrect results. In order to detect and correct a single SDC, Leaping Shadows associates 2 shadows with each main process. The primary shadow is scheduled to run at the same rate as its associated main, so that voting can be used to detect SDC in a timely manner, and correct results can be confirmed if no SDC occurs. In order to minimize energy, the secondary shadow initially executes at a reduced rate. As a result, it can also benefit from leaping to achieve forward progress with minimal overhead. Based on this idea, we have established precise analytical models and optimization frameworks to quantify and optimize the performance.

A proof-of-concept implementation of Leaping Shadows in MPI. Though Leaping Shadows has been evaluated analytically, a real implementation is needed for validation and performance measurement in real systems. We have developed *rsMPI* as a prototype of Leaping Shadows in Message Passing Interface (MPI), which is the de facto programming paradigm for HPC. Instead of a full implementation of MPI, the library is designed to be a separate layer between MPI runtime and user application, in order to take advantage of existing MPI performance optimizations that numerous researches have spent years on. This implementation transparently spawns a shadow for each main process during the initializa-

tion phase, manages the coordination between mains and shadows, and guarantees order and consistency for messages and non-deterministic events. In addition, we have implemented leaping and rejuvenation to efficiently tolerate a large number of crash failures.

Performance evaluation of Leaping Shadows. With the rsMPI implementation, extensive experiments have been conducted to measure its real system performance, using benchmark applications that represent a wide range of HPC workloads. As a first step, we measured the failure-free runtime overheads resulted from the enforced consistency protocol. This experiment revealed that the overheads depend on application and vary from 0.04% to 2.73%. Then we measured the scalability in order to predict the performance at extreme-scale, which suggested that the overheads would remain under 9.4% at 2^{20} processes. Lastly, we also implemented in-memory checkpoint/restart to compare with rsMPI in the presence of failures. The results demonstrated Leaping Shadows' ability to tolerate high failure rates, and to outperform in-memory checkpoint/restart in both execution time and resource utilization.

1.3 THESIS OUTLINE

The rest of this thesis is organized as follow: Chapter 2 reviews literature. It provides a background study of existing fault tolerance and power management techniques in large-scale computing systems. Chapter 3 introduces the Shadow Replication computational model, which forms the foundation of this work. In Chapter 4, we build a reward-based optimization framework for Shadow Replication in the Cloud environment. In Chapter 5, we introduce the Leaping Shadows model, which addresses the shortcomings of Shadow Replication in failure-prone, extreme-scale systems. Tolerance of silent data corruption is discussed in Chapter 6. Chapter 7 presents the details of a Leaping Shadows implementation as well as its performance evaluation. Finally, Chapter 8 concludes the thesis.

2.0 BACKGROUND

Research in fault tolerance and power management have been fruitful in the past few decades, with numerous mature techniques made available for various computing platforms, spanning from mobile devices to large-scale clusters. This chapter reviews the literature and provides a background study for the work presented in this thesis. Firstly, definitions of fault and failure are provided to set stage for how to achieve fault tolerance. The next two sections discuss two families of fault tolerance strategies that are widely adopted today, with a focus on large-scale computing systems. Finally, this chapter presents a survey of research work in the area of power and energy management.

2.1 FAULT, FAILURE, AND FAULT TOLERANCE

There is a clear distinction between fault and failure. Generally, a *fault* refers to an exception or detect in the system at its lowest level [71, 59]. For example, a common fault in storage systems is hard disk drive malfunction. Such a fault is *reproducible* as it always re-occurs until the faulty disk is removed and replaced. On the contrary, a bit flip in the main memory caused by cosmic rays is usually transient, thus non-reproducible. Oftentimes fault is not visible to the application or end user. Instead, the externally visible manifestation of a fault is called a *failure* [59]. Using the disk fault example again, a failure when trying to read from the disk could be corrupted data, or inability to access, or even slowing down due to repeated read attempts. In cases where no distinction is made between a fault and the resulting failure, these terms are used interchangeably.

Failures in computing systems can be attributed to various factors, including software,

hardware, network, human, overloading, environment, etc [123]. The disk malfunction mentioned above is a hardware failure. Bugs, race conditions, and deadlock are examples of software failure. One example of failure induced by overloading is a low priority job terminated to make room for a high priority one. In the environment category, A/C failure leading to overheat is an example.

Due to the rich causes and varied effects, faults and their resulting behavior are typically abstracted by fault models, which are grouped in an hierarchical structure [59, 32, 121]. Below are three popular fault models.

- **Fail-stop** A faulty processor stops execution and this failure can easily be detected by other processors.
- **Silent data corruption** A faulty processor continues execution but may silently generate incorrect results.
- **Byzantine** A faulty processor continues execution but may behave in an arbitrary or even malicious way.

When studying fault tolerance, the correctness and efficiency of a particular approach is assessed with respect to a specific fault model. It is not hard to see that Byzantine fault model is the most generic in this list, as it covers the other two models, whereas fail-stop is the most restrictive, as it requires that a processor halt in response to a failure and that failure be detected, which may not be realistic in certain circumstances.

2.1.1 Fault Tolerance

Fault tolerance aims at guarding a system against faults, so that the system operates in accordance with established specifications [82]. A system is fault tolerant if it has build-in fault tolerance capability and never deviates from the expected behavior. To achieve fault tolerance, a system requires additional resources that exceed the minimum amount needed to satisfy the performance requirements of an application. This is referred to as redundancy and is the foundation of all fault tolerance techniques today.

Redundancy comes in four forms: time, information, hardware, and software [78]. To leverage time redundancy, a system re-executes partial or entire portion of lost work after a

failure, therefore coping with failure at the sacrifice of time resource. As the name suggests, information redundancy increases reliability through storing and processing additional information. A simple example is to replicate a file across multiple locations, such as Hadoop distributed file system [2]. Other examples involve more complicated encoding techniques, like Algorithm-based fault tolerance (ABFT) and RAID [68, 105]. Hardware redundancy refers to the use of extra hardware resources to execute redundant instances of a task. Software redundancy, on the other hand, consists of writing the same function using different methods and then comparing the output to detect and correct failures.

Research in fault tolerance has been a fertile ground, with significant progress on how we understand failures [122, 22, 21], detect failures [17, 55, 40], and mitigate the impact of failures [16, 66, 52]. Rollback recovery is the direct application of time redundancy. It is often optimized by checkpointing techniques to periodically save the execution state to a stable storage, with the anticipation that, in case of failure, computation can be restarted from a saved checkpoint [28]. Message logging protocols, which combine checkpointing with logging of non-deterministic events, allow a system to recover beyond the most recent consistent checkpoint [132]. Proactive fault tolerance relies on a prediction model to forecast faults, so that preventive measures, such as task migration or checkpointing, can be undertaken [58, 50, 26, 86]. Algorithm-based fault tolerance uses redundant information inherent to its algorithmic structure of the problem to achieve resilience [91, 18]. For the past 30 years, disk-based coordinated checkpoint/restart has been the primary fault tolerance technique in production HPC systems [54]. However, its foreseen low efficiency in emerging extreme-scale systems re-ignited the study of state machine replication as a promising alternative.

2.2 ROLLBACK RECOVERY

Rollback recovery is the dominant mechanism to ensure correctness in the presence of failures in current HPC environments, where failures are infrequent and considered as exception [38, 47, 42]. Based on rollback recovery, techniques available today can be classified into checkpoint-based approaches and log-based approaches [47].

2.2.1 Checkpoint-based Approach

To avoid rolling back to the very beginning every time there is a failure, checkpointing can be periodically invoked to save the intermediate execution state to a *stable storage*. Stable storage is an abstraction of a storage medium that persists through the anticipated failures and ensures that the stored information is available during recovery [47]. Typically, a stable storage is implemented on top of a magnetic disk based storage subsystem that is accessible through a network [125]. In the case of a failure, previously saved execution state can be retrieved to resume the computation.

In distributed systems, a global state consists of a collection of process state, each saved by its own process. In a message-passing system, however, rollback recovery is complicated by the issue of *rollback propagation* due to inter-process communication. When one process P_1 sends a message m to another process P_2, the event of sending at P_1 and the receipt at P_2 form a "happened before" relationship [80]. The rollback propagation issue is such that, if P_1 were to roll back to a state before sending m, P_2 needs to roll back to a state before receiving m. Otherwise, the states of the two processes would end up being *inconsistent* because they would show that m has not been sent but already received, which never exists in any correct execution. Therefore, checkpoint/restart in message passing systems needs to guarantee, either at checkpointing time or restart time, that a globally consistent state can be constructed from the saved checkpoints.

Coordinated checkpointing is a popular approach that coordinates the checkpoint writing process to record a globally consistent state. Specifically, all processes coordinate with each other to save individual states that collectively satisfy the "happened before" relationship [28]. As a result, only the last successfully saved checkpoint needs to be kept. One way to perform coordination is to quiesce the communication channels before writing a checkpoint [133, 64]. For example, barrier synchronization in the underlying application is a natural point to produce a checkpoint. At the same time, non-blocking protocols exist to save the communication state during the operation [31, 28, 79]. 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 [44, 113, 64].

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. Since each process is allowed to checkpoint its state independently, optimizations can be explored to checkpoint when the process state is small, or when there is abundant I/O bandwidth to access stable storage. However, the scheme requires that each process maintains multiple checkpoints, necessary to construct a consistent state during recovery. Furthermore, it can also suffer the well-known domino effect [111, 6, 65], which makes all saved checkpoints useless and forces the execution to roll back to its initial state. Although well-explored, uncoordinated checkpoint/restart has not been widely adopted in HPC environments, due to its complexities of handling recovery and its heavy dependency on applications [47, 62].

One hybrid approach of coordinated and uncoordinated checkpointing, known as communication induced checkpointing, aims at reducing coordination overhead by taking advantage of the communication patterns of an application[6]. The approach, however, may cause processes to store useless states. To address this shortcoming, "forced checkpoints" have been proposed [65] to avoid creating useless checkpoints. This approach, however, may lead to unpredictable checkpointing rates.

One of the largest overheads in any disk-based checkpointing technique is the time necessary to write the checkpoints to a stable storage. Given the amount of system memory and I/O bandwidth, it takes minutes to save a single global checkpoint to a centralized storage subsystem [95]. Incremental checkpointing attempts to reduce the state saved in a checkpoint by only writing the changes during the last checkpointing interval [3, 45, 85]. On a memory page granularity, this can be achieved using dirty-bit page flags [108, 45]. Hash based incremental checkpointing, on the other hand, makes use of hash algorithms to detect changes that are finer-grained than pages, at the cost of extra computation [29, 3]. However, whether the benefits in reducing checkpoint size offset the increased computation cost remains unclear [46, 101]. Copy-on-write checkpointing offloads the checkpointing process to a secondary task and only writes incremental checkpoints [85].

Another proposed scheme, known as in-memory checkpointing, minimizes the overhead of disk access by saving checkpoints in main memory [150, 149]. 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 [4]. Multi-level checkpointing, which consists of writing checkpoints to multiple storage media, can benefit from such an upgrade [100]. This, however, may lead to increased failure rates of individual nodes and complicate the checkpoint writing process.

2.2.2 Log-based Approach

Log-based rollback-recovery is often a natural choice for applications that frequently interact with the outside world. Log-based rollback-recovery combines checkpointing with logging of non-deterministic events, and allows the system to recover beyond the most recent consistent checkpoint [132]. To enforce determinism in the presence of non-deterministic events (e.g., message receipt), log-based rollback-recovery relies on the *piecewise deterministic (PWD)* assumption, which states that all non-deterministic events can be identified and *determinants* can be recorded to replay the events in their original order [7]. By replaying the non-deterministic events according to the logged determinants, a process can re-construct its state up to the first unlogged non-deterministic event, even if this state has not been checkpointed.

Depending on how the determinants are logged, log-based rollback-recovery has three flavors. Pessimistic logging takes a blocking approach, such that the application execution is suspended until the determinants have been safely stored in a stable storage [70, 72, 73]. Pessimistic logging simplifies recovery and garbage collection, at the cost of failure-free performance. Optimistic logging makes the optimistic assumption that logging will complete before a failure occurs, thus allowing determinants to be spooled to a stable storage asynchronously, avoiding blocking the application [132, 126, 128]. Optimistic logging reduces the failure-free overhead, but complicates recovery. Also, several checkpoints may need to be kept [47]. Finally, causal logging combines the benefits of optimistic and pessimistic logging to achieve low failure-free overhead while allowing simple recovery [92, 83, 7]. Different from optimistic logging, causal logging limits the rollback to the most recent checkpoint. This reduces the storage overhead and the amount of work at risk.

2.3 STATE MACHINE REPLICATION

In contrast to rollback recovery, forward recovery allows applications to proceed in the case of a failure instead of rolling back. The most well-know forward recovery technique is state machine replication, which has long been used for reliability and availability in mission critical systems and storage systems [120, 130, 151]. Potentially faulty entities are considered as black boxes that implement state machines, delivering identical outputs when presented with the same sequence of inputs. A similar idea to state machine replication is redundant multi-threading, which has been implemented in both hardware and software, and used to increase reliability in CPUs and GPUs [112, 141]. In the case where each black box is a process, this technique is also referred to as process replication, which instantiates multiple copies (replicas) of each process and let them execute the same code.

To ensure consistent state across replicas, all application messages must be delivered to all replicas of a given process, typically done by using a message ordering protocol [39, 11]. Additionally, if any non-deterministic event is involved in the computation, then extra consensus protocols must be enforced [81, 148]. By distributing replicas across the available compute nodes and minimizing the required state comparisons, low runtime overheads can be achieved with process replication, while masking a large number of failures from the underlying application. However, the undesired property of this technique is the increase in hardware resources, as well as the proportional increase in power consumption.

Although it was initially rejected in HPC communities, replication has recently been proposed to address the deficiencies of checkpoint/restart for upcoming extreme-scale systems [22, 49, 113, 54]. These studies predict that process replication achieves better scalability and system efficiency than checkpoint/restart in future failure-prone extreme-scale computing environments. Full and partial process replication have also been studied to augment existing checkpointing techniques, and to detect and correct silent data corruption [131, 43, 54, 55, 102, 84]. There are several different implementations of replication in the widely used MPI library and cloud environments, each with their different trade-offs and overheads [49, 54, 148]. The overhead can be negligible or up to 70% depending upon the communication patterns of the application [49].

2.4 POWER MANAGEMENT

Energy conservation is a major concern today. In computer systems, power and energy are critical in terms of both cost and availability. In mobile, battery-operated devices, power dissipation directly translates into a limitation on operation hours. In large-scale computing systems, energy costs account for a significant portion of the operating expenses [119]. Despite large or small systems, most of the energy consumed is converted into heat, resulting in wear and reduced reliability of hardware components [118, 5].

Energy saving can be targeted at all levels of a system, ranging from circuit level to architecture level and operation level [99, 139]. At the operation level, a large body of scheduling work has been explored, based on the observation that one can save energy by leveraging execution slack [60, 67, 69, 57, 87, 114, 136]. All of them resolve around two underlying mechanisms, i.e., power-down and dynamic speed scaling [5, 89].

Power-down, also known as dynamic resource sleeping, conserves energy by dynamically turning resources into low-power standby or sleeping modes, and then waking them up on demand. Each resource may be in an active running state, or in one of intermediate sleeping states, or in the completely power-off state. For instance, in the Intel Nehalem-EP processor, the clock and other components could be turned off to make a transition into a low-power mode [89]. The deeper the resource sleeps, the less power it consumes, but the more energy is needed to wake it up [5]. In addition to processor, some memory controllers support the dynamic switch of memory ranks between on and off states [104]. Similarly, disks may also exploit active, ready, and standby states to reduce power consumption [30, 107].

Dynamic speed scaling is another power management mechanism that allows dynamic tuning of the performance state of the target component to save power, i.e. it slows down when possible to reduce power consumption and speeds up when needed at the cost of greater power consumption. Dynamic Voltage Frequency Scaling (DVFS) is probably the best known example [51, 77, 109, 114, 57, 136, 87, 48, 106, 56]. This technique is widely available in today's processors, including Intel Xeon, AMD Atholon, and ATI co-processors. Generally, the reduction in power consumption is achieved through reducing the supply voltage , which in turn reduces the processor clock frequency [138]. Other examples of dynamic speed

scaling include multi-frequency memories and multi-speed disks, which dynamically scale the working frequency and thus the data rate [37, 63, 24]. In all of these methods, however, it is unavoidable that the transition between different performance states consumes additional energy and introduces latency [138].

2.5 SUMMARY

This chapter reviews related work in the fields of fault tolerance and power management. We begin with a clarification of what we mean by fault and failure, followed by how fault tolerance works in general. Then, state-of-the-art fault tolerance techniques are discussed, each with its advantages and limitations. We recognize the dominance of checkpoint/restart in current HPC systems, and point out the emergent process replication approach as it attracts growing attention. Lastly, we survey a number of power management techniques that build on top of power-down and dynamic speed scaling.

Motivated by the extreme-scale challenges, this thesis targets at work that lies at the intersection of fault tolerance and power management. Existing work in this area has mainly focused upon measuring power and energy consumption of fault tolerance techniques [93, 116, 97]. Instead, this thesis proposes a new fault tolerance model, *Leaping Shadows*, which explores the trade-offs between time and hardware redundancy to achieve fault tolerance with power awareness. The flexibility in balancing these trade-offs, together with the ability to rejuvenate after failures, allows Leaping Shadows to maximize system efficiency and maintain a persistent level of resilience to failure while minimizing energy consumption.

3.0 SHADOW REPLICATION

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.

Shadow Replication is a computational model that goes beyond adapting or optimizing well known and proven techniques, and explores radically different methodologies to fault tolerance [96, 98, 95]. The proposed solutions differ in the type of faults they manage, their design, and the fault tolerance protocols they use. It is not just a scale up of "point" solutions, but an exploration of innovative and scalable fault tolerance frameworks. When integrated, it will lead to efficient solutions for a "tunable" resiliency that takes into consideration the nature and requirements of the application.

The basic tenet of Shadow Replication 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, and a consistency protocol assures that each shadow process stays consistent with its associated main process. Shadow Replication achieves power awareness under QoS requirements by dynamically adjusting the execution rates in response to failures. To save power, the shadows initially execute at a lower rate than the main process. 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 a new main process, and possibly increases its execution rate to mitigate delay.

Since the individual failure rate is extremely low, in most instances the main process

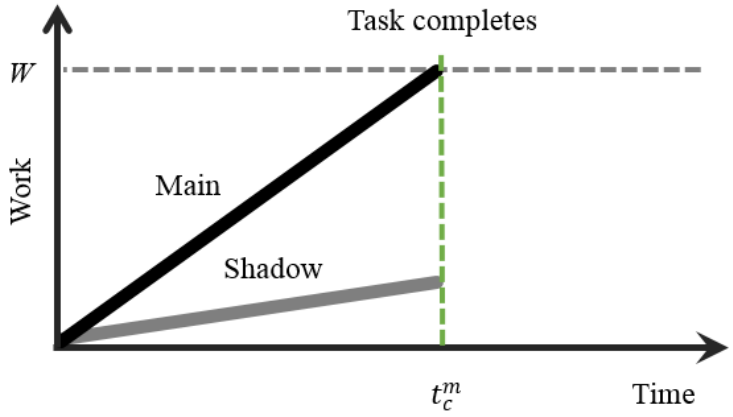
will not encounter any failure and complete the task. Consequently, the additional energy consumed by the slower shadow is significantly lower than that of a full-speed replica of the main, resulting in a lot of energy savings. Furthermore, a failure of a shadow process has no bearing on the behavior of its associated main process. These two properties make it possible for Shadow Replication to enable fault tolerance at a much lower cost.

3.1 EXECUTION MODEL

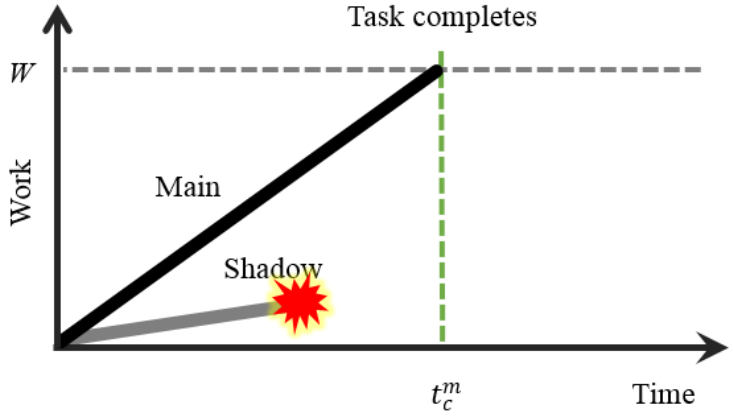
Assuming there is a Reliability Availability and Serviceability (RAS) system for fault detection [54], the Shadow Replication fault-tolerance model is formally defined as follows:

- A main process, $P_m(W,\ \sigma_m)$, whose responsibility is to executes a task of W workload at an execution rate 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 rates. All shadows start execution simultaneously with the main process at rate 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.

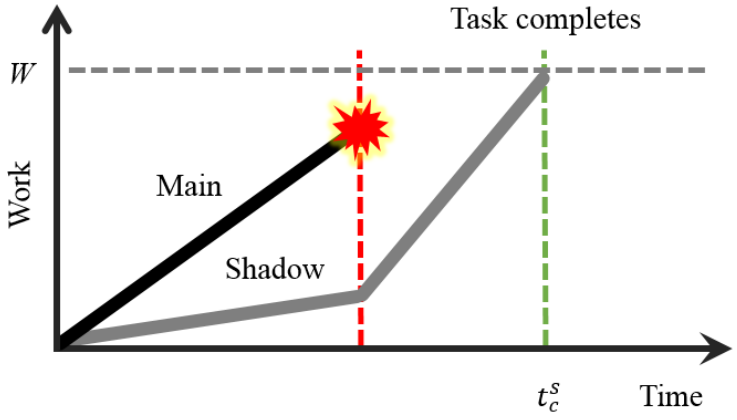
To illustrate the behavior of Shadow Replication, we limit the number of shadows to a single process and consider the scenarios depicted in Figure 3, assuming a single process failure. Figure 3(a) represents the case where neither the main nor the shadow fails. The main process, executing at a higher rate, completes the task at time t_c^m. At this time, the shadow process, progressing at a lower rate, stops execution immediately to save energy. Figure 3(b) represents the case where the shadow experiences a failure. This failure, however, has no impact on the progress of the main process, which still completes the task at t_c^m. Figure 3(c) depicts the case where 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 rate, completing the task at time t_c^s.



(a) No Failure



(b) Shadow Process Failure



(c) Main Process Failure

Figure 3: Shadow Replication for a single task using a pair of main and shadow.

3.2 ADAPTIVITY

In HPC and Cloud systems, performance, resilience, and power consumption are more often than not conflicting objectives. For example, achieving fault tolerance comes with a cost of redundant resources, which unavoidably lead to additional power and energy consumption. On the other hand, it has been shown that lowering supply voltages, a commonly used technique to conserve power, increases the probability of transient faults [27, 146], and introduces non-trivial performance degradation [143].

Shadow Replication is a pioneering work in exploring the trade-offs among failure-free operation, the imposed power constraints, and the time to solution of the supported application. Internally, Shadow Replication has a set of parameters, lying on two dimensions, that collectively determine its behavior along with costs. By configuring the shadow suite size based on fault tolerance needs, and dynamically adjusting the main and shadow processes' execution rates, Shadow Replication is able to guarantee a specific performance with certain fault tolerance capability and under a bounded power budget, thereby achieving adaptivity to the desired balance among the three conflicting objectives.

The size of shadow suite directly reflects the amount of redundancy needed. The more shadows in each suite, the more hardware resources are required to execute the replicas in parallel. Furthermore, the additional hardware resources place a higher demand on the power supply. Therefore, it is desirable to use as few shadows as possible, under the premise that the resilience requirements are met. It is well known that one can use $f+1$ replicas to tolerate f crash failures, and use $2f+1$ replicas to correct f silent failures. To deal with one crash failure, a single shadow that runs as a slower replica of its associated main would be sufficient to maintain acceptable response time. Like shown in Figure 3, two replicas guarantee that at least one can complete the task, if at most one crash failure could occur. Similarly, two shadows are sufficient to guard a main process against a silent failure. Specifically, by using a simple voting mechanism and comparing the outputs of three replicas, one can detect as well as correct a silent failure [55].

In addition to shadow suite size, another dimension of control consists of the execution rates of both the main and shadow processes, before and after failure. A closer look at the

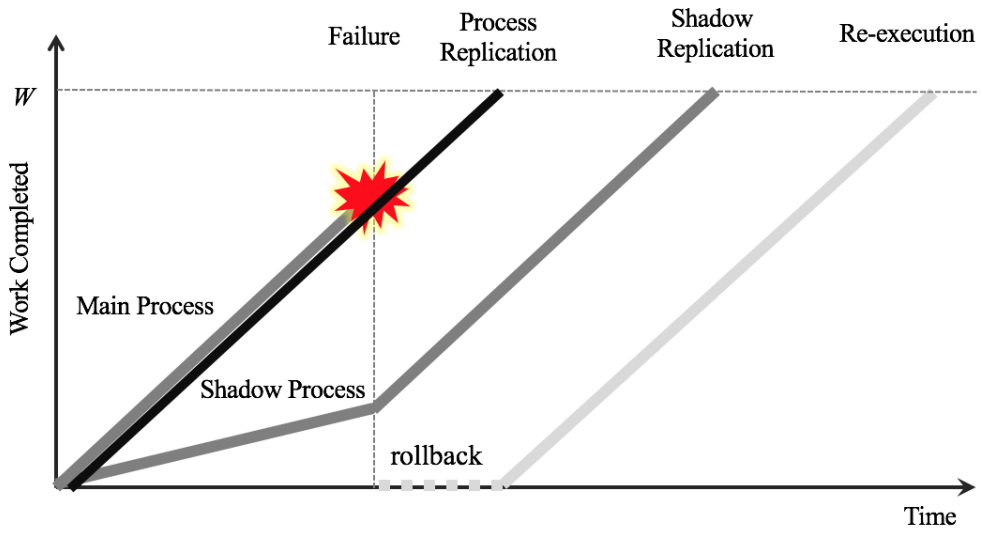


Figure 4: Illustration of Shadow Replication's ability to converge to either re-execution or traditional process replication.

model reveals that Shadow Replication is a generalization of traditional fault tolerance techniques, namely re-execution and process replication. If it allows for flexible completion time, Shadow Replication 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, Shadow Replication converges to process replication, as the shadow must execute simultaneously with the main at the maximum rate. Assuming dual modular redundancy to deal with a crash failure, this adaptability of Shadow Replication is further illustrated in Figure 4. It is not difficult to imagine that by exploring the combination of execution rates, Shadow Replication covers a "spectrum" of fault tolerance strategies, including both re-execution and process replication. The flexibility of the Shadow Replication model provides the basis for the design of a fault tolerance strategy that strikes a balance between task completion time and energy saving.

3.3 EXECUTION RATE CONTROL

The Shadow Replication model relies on the fact that power savings can be achieved by reducing process execution rate. Furthermore, Shadow Replication needs to dynamically adjust the process execution rates to maintain satisfactory response time while saving power and energy. So far, two techniques have been explored to achieve the desired process execution rate. The first technique directly relates to a hardware feature, while the second one can be easily done via process mapping on any hardware platform.

DVFS is the first technique that our lab studied to reduce the execution rate of a process. While running each main and shadow process on a separate CPU, DVFS allows to reduce the CPU frequency by lowering the supply voltage. In the case of a failure, DVFS also allows to dynamically increase the CPU frequency to speed up a process. It is well known that one can reduce the dynamic CPU power consumption at least quadratically by reducing the frequency linearly, thereby saving power and energy simultaneously.

An alternative approach to DVFS is to collocate multiple processes on a single CPU, while keeping each CPU running at the maximum frequency [35]. The desired process execution rate can be achieved by controlling the *collocation ratio*, which is defined as the number of processes that time-share a CPU. An example of collocation is depicted in Figure 5, with a collocation ratio of 3 for shadow processes. A *shadowed set* refers to a set of mains and their collocated shadows. The advantages and disadvantages of collocation will be further discussed in later chapters.

In terms of power and energy saving, DVFS has a different effect from collocation. When applying collocation, it is straightforward that fewer hardware resources are required to support the same number of processes than using DVFS. For example, only 12 cores are required to simultaneously execute 18 processes, as shown in Figure 5. This is a 33.3% saving in hardware resources compared to DVFS. As a result, collocation brings in reduction in power and energy consumption proportionally to the reduction in hardware resources. On the other hand, although DVFS requires the same amount of hardware as traditional process replication, it saves CPU dynamic power on each process that executes at a reduced rate, thus achieving energy savings. This thesis does not carry out a quantitative comparison between

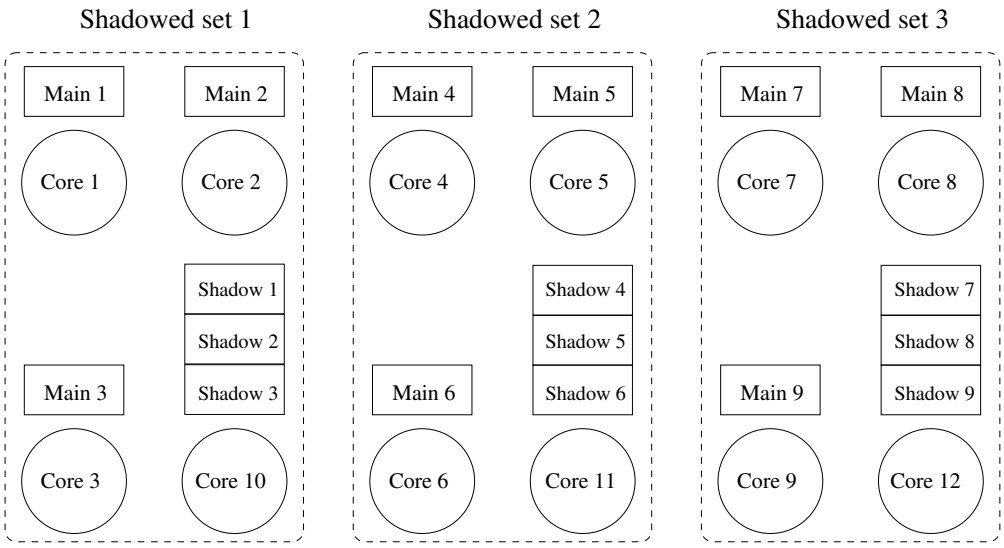


Figure 5: An example of collocation. Nine mains and their associated shadows are grouped into three logical shadowed sets. By collocating every three shadows on a core, twelve cores are required.

these two techniques, because the exact power and energy consumption largely depend on the specific architecture, such as the ratio between CPU dynamic power and static power, and the relationship between power reduction and frequency reduction due to DVFS.

3.4 SUMMARY

Based on DVFS, Mills studied the Shadow Replication computational model and associated one shadow process with each main process to tolerate fail-stop failures in HPC systems. For HPC throughput consideration, his work assumes that the main process always executes at the maximum rate. Even with this restriction, Mills successfully demonstrates that Shadow Replication can achieve resilience more efficiently than both checkpoint/restart and process replication when power is limited [96, 98, 95].

Although DVFS is widely available in today's processors, its effectiveness, however, may

be markedly limited by the granularity of voltage control, the range of frequencies available, and the negative effects on reliability [51, 146, 147]. At the same time, there are several issues with the Shadow Replication model that have been later identified and that question the applicability of Shadow Replication to tolerating high rate and diverse types of failures in extreme-scale computing environments. In the following chapters, this thesis presents novel work that not only addresses the limitations of the basic Shadow Replication model to achieve better adaptivity, sustainability, and scalability, but also verifies the model with prototype implementation and performance evaluation.

4.0 REWARD-BASED OPTIMAL SHADOW REPLICATION

In the most straightforward form of Shadow Replication, each main is associated with one shadow to tolerate a crash failure, and with two shadows to tolerate a silent failure. Similar to traditional process replication, Shadow Replication ensures successful task completion by running the mains and shadows in parallel. Contrary to process replication, however, Shadow Replication exploits differential and dynamic execution rates, thereby enabling a parameterized trade-off between response time, energy consumption and resilience.

After figuring out the shadow suite size according to the fault tolerance requirements, a major challenge resides in determining jointly the execution rates of all task instances, both before and after a failure occurs, with the consideration of any objective and any constraint. This chapter addresses the above challenge by formulating it into a generic optimization problem, so that standard and well-known math methods (e.g., quasi-newton) can be naturally applied to solve it and derive the optimal execution rates. With this optimization framework, this chapter also provides a case study in the Cloud with a series of analytical models for failure, power, energy, etc.

4.1 GENERIC OPTIMIZATION FRAMEWORK

As mentioned in Section 3.2, large-scale computing systems typically aim at maximizing performance, resilience, power/energy consumption, or any of their combination. Given a specific objective, by tuning its execution rates Shadow Replication can be optimized with respect to that objective, while satisfying the provided constraints, if any. A generic optimization framework for this purpose is depicted in Figure 6.

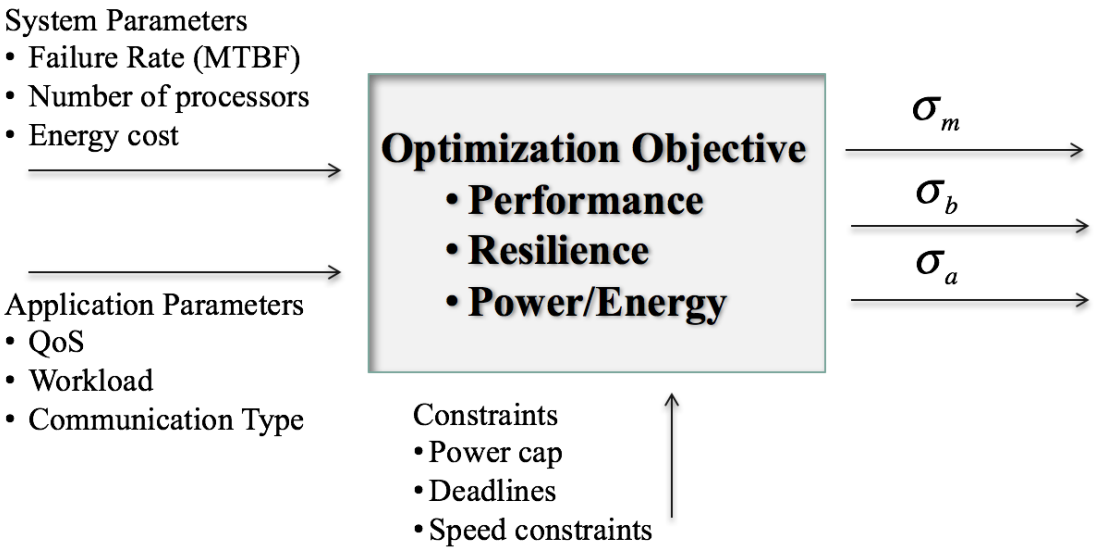


Figure 6: A generic optimization framework for Shadow Replication.

As shown in Figure 6, the inputs to the problem consist of system parameters and application parameters. System parameters include the failure distribution, system scale, and power consumption characteristics. Application parameters could be QoS requirements, total workload, and/or the communication and synchronization patterns. In addition, constraints, such as power cap and deadline, can be specified in observation of resource limitations. Finally, the outputs are the derived execution rates for all processes, which optimize the given objective.

The above optimization framework can be tailored to various computing environments for different needs. For example, in HPC systems, the problem can be simplified by fixing σ_m and σ_a at the maximum CPU rate and only optimize σ_b to minimize energy consumption under deadline constraint [96]. The following section discusses another case study in the Cloud environment that relaxes the constraints on σ_m and σ_a [34]. The resulting work, referred to as reward-based optimal Shadow Replication, considers Service Level Agreements (SLA) in the Cloud and optimizes a "reward" for Cloud service providers. The flexibility in the definition of reward demonstrates the ability of Shadow Replication to achieve objectives

that go beyond time and energy.

4.2 REWARD-BASED OPTIMAL SHADOW REPLICATION

Cloud Computing has emerged as an attractive platform for diverse compute- and data-intensive applications, as it allows for low-entry costs, on demand resource provisioning, and reduced complexity of maintenance [134]. 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 and cost-effectiveness, resulting in a multi-fold increase in the number of computing, storage and communication components in their data centers.

Two direct implications of the ever-growing large-scale data centers are the increasing energy costs, which build up the operating expenditure, and service failures, which subject the CSP to a loss of revenue. Therefore, Service Level Agreement (SLA) becomes a critical aspect for a sustainable cloud computing business. In its basic form, an SLA is a contract between the CSPs and consumers that specifies the terms and conditions under which the service is to be provided, including expected response time and reliability.

To understand the question of how fault tolerance might impact power consumption and ultimately the expected profit of CSPs, this section studies the application of Shadow Replication to satisfying SLA requirements in the presence of crash failures in Cloud computing. The rest of the section is organized as follows. We begin by describing a parallel computing model typically used in cloud computing for compute- and data-intensive applications. We then present our analytical models and optimization problem formulation, followed by experiments and evaluation.

4.2.1 Cloud Workloads

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 sci-

ences and discovery. These applications exhibit different behaviors, in terms 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 [88, 53]. This model is directly reflective of the *MapReduce* computational model, which is predominately used in Cloud Computing [117]. An instance of this model, is depicted in Figure 7.

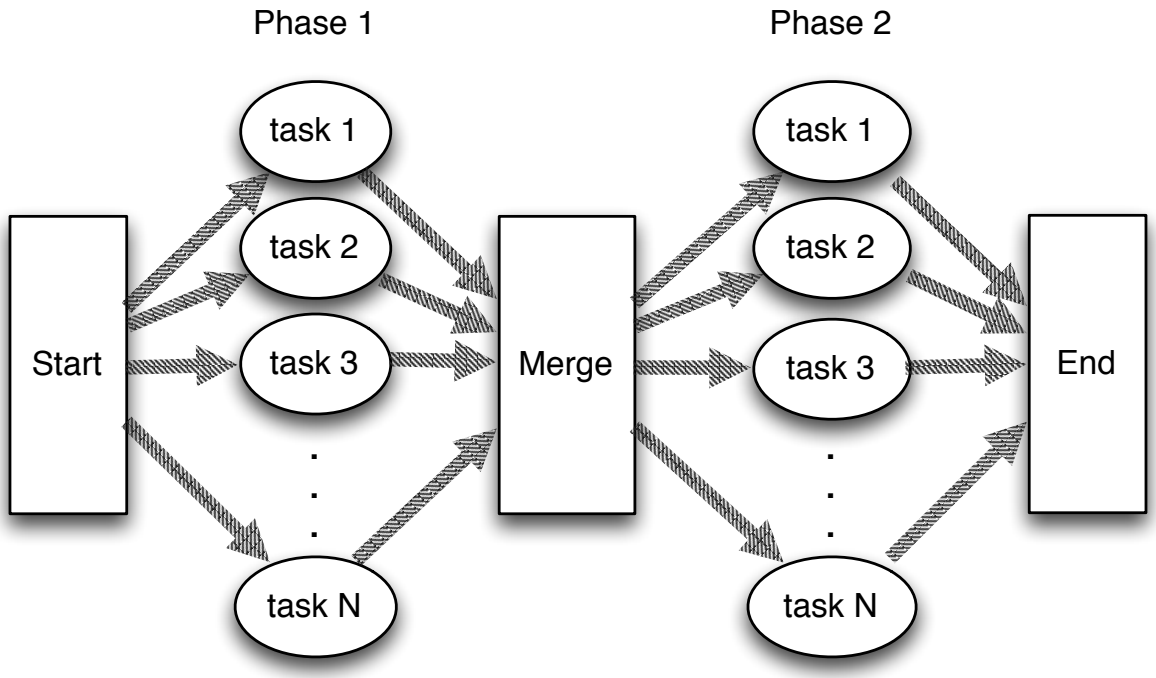


Figure 7: Cloud computing execution model with 2 phases.

Each job has a targeted response time defined by the terms of the SLA. Further, the SLA defines the amount to be paid for completing the job by the targeted response time as well as the penalty to be incurred if the targeted response time is not met.

Each task is mapped to one compute core and executes at a rate, 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 execution rate. This is also the minimal response time of the entire phase.

4.2.2 Optimization

This part describes an optimization problem for a single job on top of the Cloud computing execution model described above. Using this framework we compute profit-optimized execution rates for Shadow Replication with dual modular redundancy (i.e., one shadow process per main process):

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

We assume that processor execution rates 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 expectation. In our model, the expected profit is defined as the expected reward minus the expected expense.

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

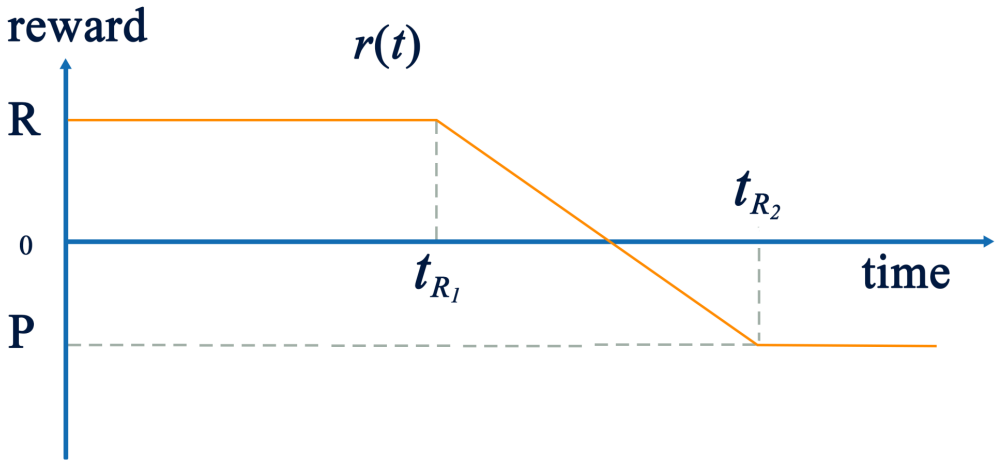


Figure 8: A reward model for Cloud computing.

4.2.2.1 Reward Model The Cloud SLA terms and conditions can be diverse and complex. To focus on the performance and reliability aspects of the SLA, we define the reward model based on job completion time. Platform as a Service (PaaS) companies will continue to become more popular, causing an increase in SLAs using job completion time as their performance metric. We are already seeing this appear in web-based remote procedure calls and data analytic requests.

As depicted in Figure 8, customers expect that their job deployed on Cloud finishes by a mean response time t_{R_1}. As a return, the service provider earns a certain amount of reward, denoted 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. It would be trivial to extend this reward model to any convex function, but we will not bother to do so.

When applying Shadow Replication with dual modular redundancy to deal with crash

failures, 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 3(a) and Figure 3(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 3(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.

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

4.2.2.2 Failure Model Failure can occur at any point during the execution of the main or shadow process. Our assumption is that at most one failure occurs, therefore if the main process fails it is implied that the shadow will complete the task without failure. We can make this assumption because we know the failure of any one node is rare, thus the failure of any two specific nodes is very unlikely.

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 section 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 failures (MTBF) is $\frac{1}{\lambda}$.

4.2.2.3 Power and Energy Models This work assumes DVFS as the underlying execution rate control mechanism. DVFS has been widely exploited as a technique to reduce CPU dynamic power [106, 56]. It is well known that one can reduce the dynamic CPU power consumption at least quadratically by reducing the execution rate linearly. The dynamic CPU power consumption of a computing node executing at rate 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

rate. In this thesis we define static power as a fixed fraction of the node power consumed when executing at maximum rate, 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 rate 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 rate [115, 145], therefore the overall system power when executing at rate 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 Equation 4.5, the energy consumed by a process executing at rate σ during an interval T is given by

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

Corresponding to Figure 3, there are three 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 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 Figure 3(a).

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

The product of the first two factors 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 Figure 3(b).

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

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 3(c).

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

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 by some processes while waiting in idle.

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

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 second 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.

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

4.2.2.4 Reward and Expense Reward is the amount paid by customer for the cloud computing services that they utilize. It depends on the reward function $r(t)$, depicted in Figure 8, 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 expected reward is approximated by the following equation.

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

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:

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

However, the expenditure of running the cloud computing service is more than just energy, and must include hardware, maintenance, and human labor. These costs can be

accounted for by amortizing these costs into the static power factor, ρ. Because previous studies have suggested that energy will become a dominant factor [48, 110], we decided to focus on this challenge and leave other aspects to future work.

Table 2: Symbols used in reward-based optimal Shadow Replication.

Symbols	Definition
W	Task workload
N	Number of tasks
$r(t)$	Reward function
R, P	Maximum reward and penalty
t_{R_1}, t_{R_2}	Response time thresholds
C	Unit price of energy
ρ	Static power ratio
t_c^m, t_c^s, t_c^j	Completion time of main, shadow, and the whole job
$f_m()$, $f_s()$	Failure density function of main and shadow
λ	Failure rate
P_f	System level failure probability
σ_m, σ_b, σ_a	rates of main, shadow before and after failure

Based on the above formulation of the optimization problem, the MATLAB Optimization Toolbox was used to solve the resulting nonlinear optimization problem. The parameters of this problem are listed in Table 2.

4.3 PROFIT-AWARE STRETCHED REPLICATION

We compare Shadow Replication to two other replication techniques, traditional replication and profit-aware stretched replication. Traditional replication requires that the two processes always execute at the same rate σ_{max}. Unlike traditional replication, Shadow Replication is dependent upon failure detection, enabling the replica to increase its execution rate 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 rates to meet the expected response time, without the need for failure detection. In profit-aware stretched replication, both the main and shadow execute at rate σ_r, derived 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:

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

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 Equation 4.12 with E_4 fixed at 0 and then compute the expected expense using Equation 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, in which 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 reward for re-execution is the weighted average of the two cases:

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

For one task, if no failure occurs then the expected energy 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 by re-execution for completing a job of N tasks is

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

4.5 EVALUATION

This section evaluates the expected profit of each of the fault tolerance methods discussed above under different system environments. 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 rate.
- 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 rates can be expressed as a fraction of maximum rate. 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 Figure 4.3 and 4.4. The price of energy C 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 maximum 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 maximum and minimum rewards are 1.3 hours and 2.6 hours respectively. Since the maximum power consumption is 1 unit, the energy needed for the task with one process at maximum rate is also 1 unit.

4.5.1 Sensitivity to Static Power

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.

Table 3: Optimal execution rates 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.

ρ	σ_m	σ_b	σ_a
0.0	0.77	0.65	1.00
0.1	0.78	0.66	1.00
0.2	0.83	0.66	1.00
0.3	0.84	0.68	1.00
0.4	0.85	0.70	1.00
0.5	0.86	0.72	1.00
0.6	0.87	0.73	1.00
0.7	0.91	0.81	1.00
0.8	1.00	1.00	1.00
0.9	1.00	1.00	1.00
1.0	1.00	1.00	1.00

Table 3 shows how the profit-optimized execution rates of Shadow Replication will change as static power increases. The execution rates increase to reduce the execution time as static power ratio increases. Observe that σ_a is always equal to σ_{max}, which means that after sensing the failure of the main process, the shadow process should always shift to maximum rate. This is expected because the optimization will reduce the amount of work done by the shadow process before failure, resulting in the maximum execution rate after failure, thus minimizing the amount of repeated work.

The potential profit gains achievable by using profit-aware replication techniques decreases as static power increases, as is shown in Figure 9. The reason is that our profit-aware techniques rely upon the fact that one can reduce energy costs by adjusting the execution rates. Modern systems have a static power between 40%-70% [20], and it is reasonable to suspect that this will continue to be the case in the near future. Within this target range, 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.

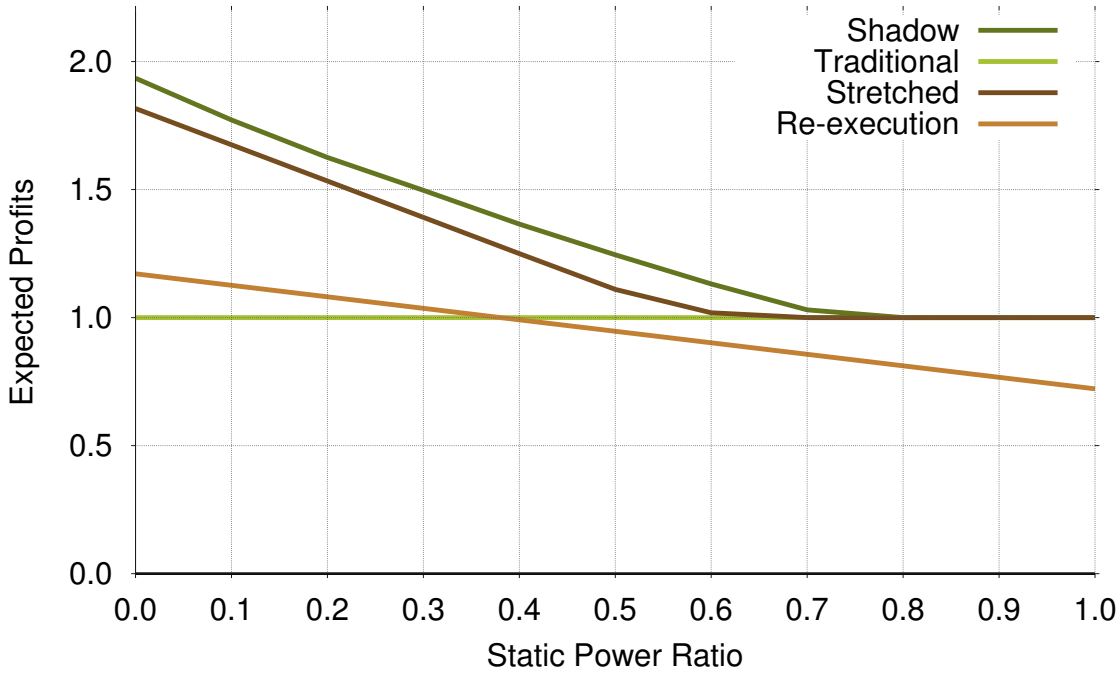


Figure 9: Profit for different static power ratio. MTBF=5 years, N=100000, W=1 hour, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

4.5.2 Sensitivity to Response Time

Response time is critical in the negotiation of SLA as customers always expect their tasks to complete as soon as possible. In this section we show a sensitivity study with respect to task response time. We vary the first threshold t_{R_1} from the minimum response time t_{min} to $1.9t_{min}$, and set the second threshold t_{R_2} to be always $2t_{R_1}$. We do not show results for varying the reward and penalty values of the SLA. The reason is that changing these values have no effect on the choice of fault tolerance methods because they are all affected in a similar way.

In Table 4 we see that Shadow Replication adapts the execution rates to take advantage of the available laxity, reducing its rates as laxity increases. It is clear that Shadow Replication has two different execution strategies separated by $t_{R_1} = 1.4$: when time is critical, it uses

Table 4: Optimal execution rates for different response time threshold. ρ=0.5, MTBF=5 years, N=100000, W=1 hour.

t_{R_1}	σ_m	σ_b	σ_a
1.0	1.00	1.00	1.00
1.1	0.94	0.88	1.00
1.2	0.89	0.79	1.00
1.3	0.86	0.72	1.00
1.4	1.00	0.00	1.00
1.5	1.00	0.00	1.00
1.6	0.84	0.00	1.00
1.7	0.74	0.00	1.00
1.8	0.64	0.00	1.00
1.9	0.64	0.00	1.00

both a main and a shadow from the very beginning to guarantee that task can be completed on time; when time is not critical, it mimics re-execution and starts its shadow only after a failure. Also note that as t_{R_1} approaches t_{min}, the rates of the main process and the shadow process converge, effectively causing Shadow Replication to mimic traditional replication when faced with time-critical jobs.

Figure 10 shows the effect that targeted response time has upon the profitability of each fault tolerance method. Compared to traditional replication, all the other methods increase their profit as the targeted response time increases, 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.

4.5.3 Sensitivity to Number of Tasks

The number of tasks has a direct influence upon the system level failure probability because as the number of tasks increase the probability that failure will occur to at least one task

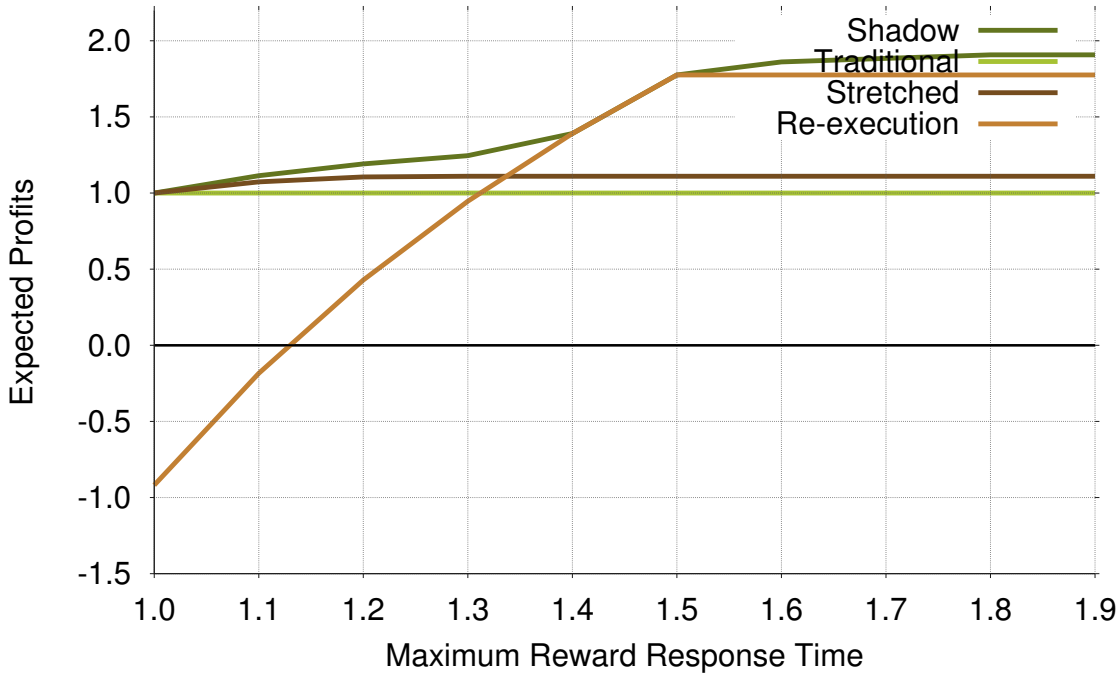


Figure 10: Profit for different response time threshold. ρ=0.5, MTBF=5 years, N=100000, W=1 hour.

increases. Recall that even one failure can hurt the total reward significantly, and keep the other processes waiting. Thus, Shadow Replication will adjust its execution rates to reduce the waiting time.

Table 5 is similar to Table 4 in that there are also two execution strategies. When there are few parallel tasks, shadow replication chooses to execute the main processes at nearly full rate and keeps the shadow processes dormant. The reason is that it is very likely that all main processes can finish their tasks successfully, and the need for redundancy is thus less significant. The other case is when there is a huge number of tasks to execute, the shadow process would keep running at a slower rate than the main to protect the main as well as save energy. Since the system level failure probability is already 0.9 when N is 100000, the rates stay the same when $N \geq 100000$.

Figure 11 confirms that for small number of tasks re-execution is more profitable than

Table 5: Optimal execution rates for different number of tasks. ρ=0.5, MTBF=5 years, W=1 hour, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

N	σ_m	σ_b	σ_a
100	0.80	0.00	1.00
1,000	0.84	0.00	1.00
10,000	1.00	0.00	1.00
100,000	0.86	0.72	1.00
1,000,000	0.86	0.72	1.00
10,000,000	0.86	0.72	1.00

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.

4.5.4 Sensitivity to Failure Vulnerability

The ratio between task size and node MTBF represents the task's 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 seen in Table 6, when the vulnerability to failure is low the execution rates for the shadow process is such that no work is done before failure. However, as the vulnerability increases, the shadow process performs more work before failure. This is analogous to what we observed as we increased the number of tasks (Table 5). 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 12.

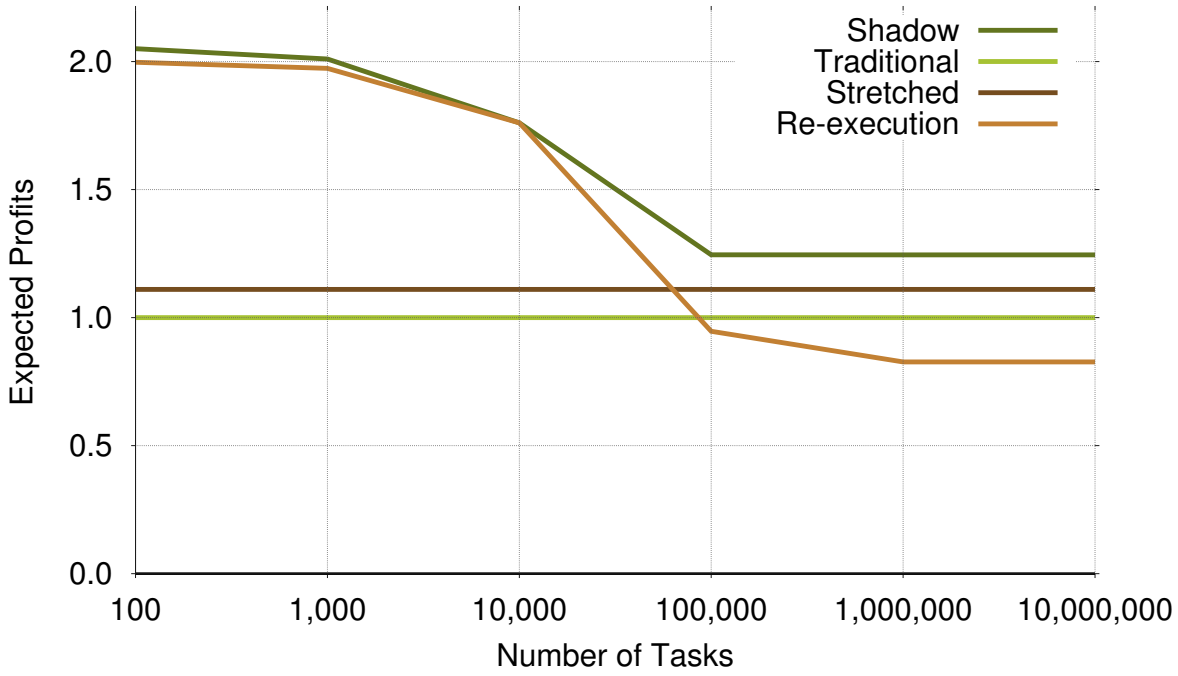


Figure 11: Profit for different number of tasks. ρ=0.5, MTBF=5 years, W=1 hour, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

Table 6: Optimal execution rates for different task size over MTBF. ρ=0.5, N=100000, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

$W/MTBF$	σ_m	σ_b	σ_a
2E-10	0.79	0.00	1.00
2E-09	0.79	0.00	1.00
2E-08	0.80	0.00	1.00
2E-07	0.84	0.00	1.00
2E-06	1.00	0.00	1.00
2E-05	0.86	0.72	1.00
2E-04	0.86	0.72	1.00
2E-03	0.86	0.72	1.00

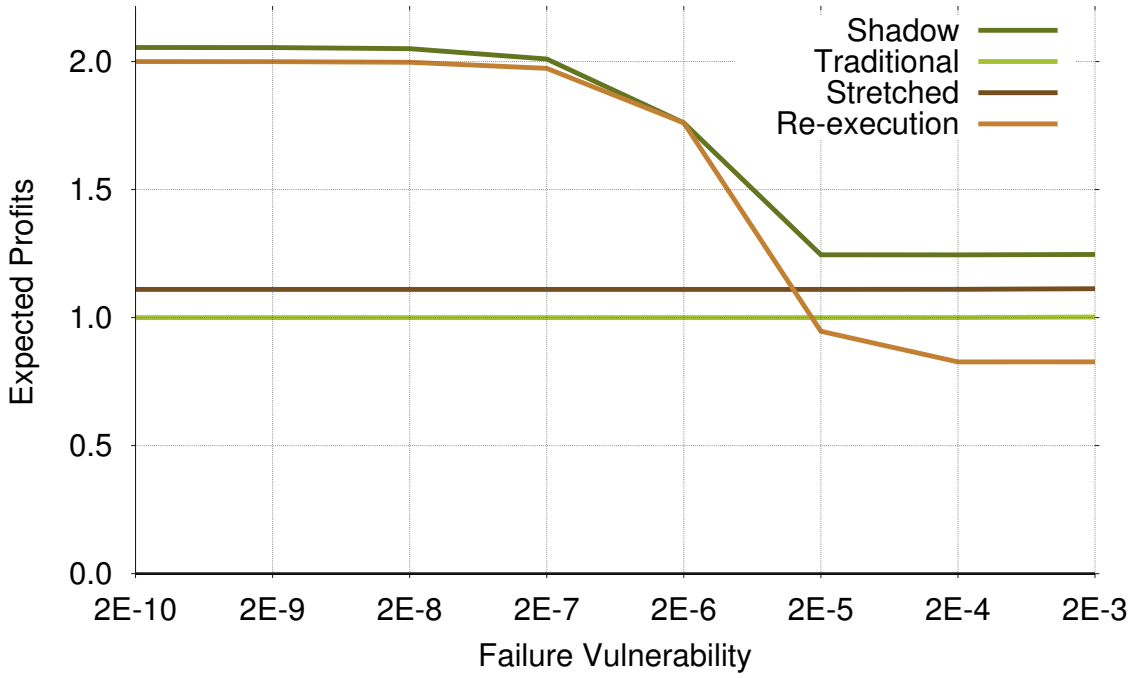


Figure 12: Profit for different task size over MTBF. ρ=0.5, N=100000, t_{R_1}=1.3 hours, t_{R_2}=2.6 hours.

4.5.5 Application Comparison

To compare the potential benefit of Shadow Replication, we evaluate the expected profit of each resilience technique using three different benchmark applications representing a wide range of Cloud workloads [117]: Business Intelligence, Bioinformatics and Recommendation System. The business intelligence benchmark application is a decision support system for a wholesale supplier. It emphasizes executing business-oriented ad-hoc queries using Apache Hive. The bioinformatics application performs DNA sequencing, allowing genome analysis on a wide range of organisms. The recommendation system is similar to those typically found in e-commerce sites which, based upon browsing habits and history, recommends similar products.

Using the results of the experiments reported in [117], we derived the time required to

Table 7: Cloud benchmark applications [117].

Application	Processing Rate
Business Intelligence	3.3 (MB/s)
Bioinformatics	6.6 (MB/s)
Recommendation System	13.2 (MB/s)

process data for each application type (Table 7). We assume that these processing rates per task will not change when scaling the applications to future cloud environments. This is a reasonable assumption given that map-reduce tasks are loosely coupled and data are widely distributed, therefore data and task workload will scale linearly.

In Figure 13 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 I/O intensive, resulting in longer running tasks, whereas recommendation systems tend to require little I/O, 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 Replication 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.

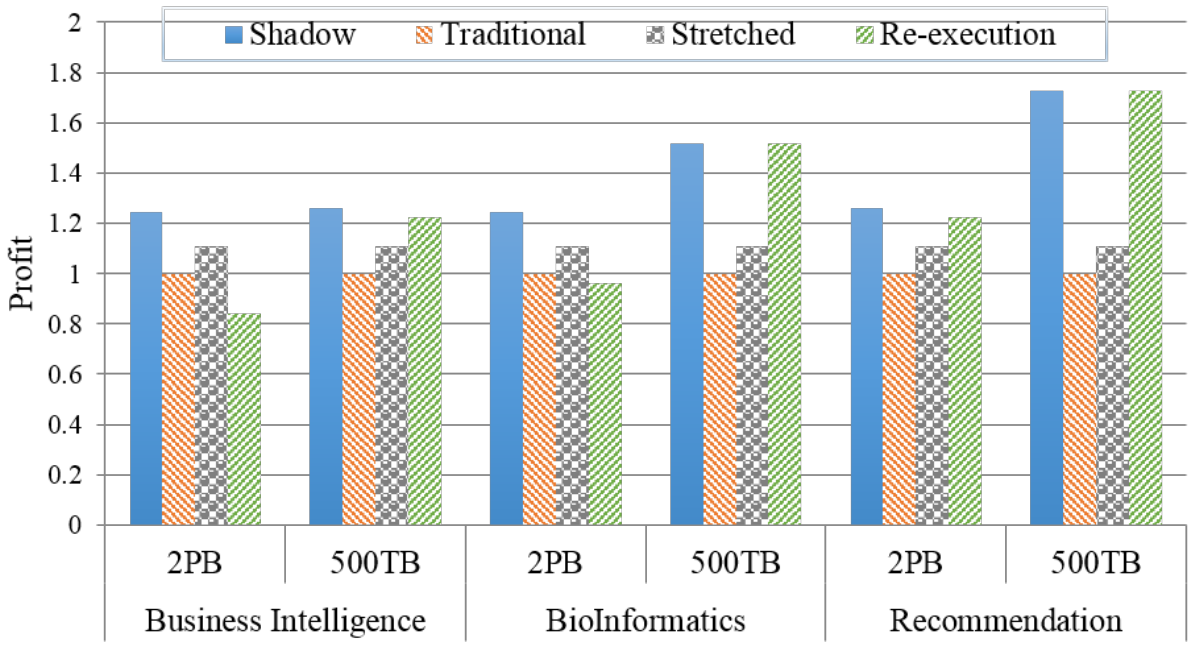


Figure 13: Application comparison. ρ=0.5, N=500000, t_{R_1}=1.3t_{min}, t_{R_2}=2.6t_{min}.

4.6 SUMMARY

In this work we focus on the objective of satisfying SLA in Cloud Computing and demonstrate that Shadow Replication is capable of achieving multi-dimensional QoS goals. To assess the performance of the Shadow Replication, 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 LEAPING SHADOWS

In today's large-scale HPC systems, an increasing portion of the computing capacity is wasted due to failures and recoveries. It is expected that future exascale machines, featuring a computing capability of exaFLOPS, will decrease the mean time between failures to a few hours, making fault tolerance extremely challenging. Enabling Shadow Replication for resiliency in extreme-scale computing brings about a number of challenges and design decisions, 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 a novel fault tolerance model of *Leaping Shadows*. Leaping Shadows is a Shadow Replication based model that associates a suite of shadows to each main process. To achieve fault tolerance, shadows execute simultaneously with the mains, but on different nodes. Furthermore, to save power, shadows execute at a lower rate than their associated mains. When a main fails, the corresponding shadow increases its execution rate to speed up recovery. This chapter focuses on tolerating crash failures under the fail-stop fault model, whereby a failed process halts and its internal state and memory content are irretrievably lost. As a consequence, each main process is replicated with one shadow. A study of tolerating silent failures will be discussed in Chapter 6.

To achieve higher efficiency and better scalability, however, we adopt radically different methodologies from the Shadow Replication model in the design of Leaping Shadows. In the original Shadow Replication model, shadows are designed to substitute for their associated mains when failure occurs. The tight coupling and ensuing fate sharing between a main and its associated shadow increase the implementation complexity and reduce the efficiency of

the system to deal with failures. In this new model, we deviate from the original design, and use each shadow as a "rescuer", whose role is to restore the associated main to its exact state before failure.

Instead of using DVFS to achieve the desired execution rates, this work applies collocation to shadow processes for the first time, in order to simultaneously save power and computing resources. In addition, two issues, *divergence* and *vulnerability*, have been identified that limit Shadow Replication's effectiveness in large-scale, failure-prone computing environments. This chapter discusses innovative techniques of *leaping* and *rejuvenation* as the provided solutions, respectively.

5.1 SHADOW COLLOCATION

In HPC, throughput consideration requires that the rate of the main, σ_m, and the rate of the shadow after failure, σ_a, be set to the maximum. The initial execution rate of the shadow, σ_b, however, can be derived by balancing the trade-offs between delay and energy. For a delay-tolerant, energy-stringent application, σ_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 $\sigma_b = \sigma_m$ 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.

To control the shadows' execution rates, DVFS can be applied while each shadow resides on one processor 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 [51, 76, 27, 146]. An alternative is to collocate shadows. We use the term processor to represent the resource allocation unit (e.g., a CPU core, a multi-core CPU, or a cluster node), so that our discussion is agnostic to the granularity of the hardware platform. While each main process occupies a processor, we collocate multiple shadows on each processor and use time sharing to achieve the desired execution rates.

To execute an application of M parallel tasks, $N = M + S$ processors are required, where M is a multiple of S. Each main is allocated one processor (referred to as *main processor*), while $\alpha = M/S$ (referred to as *collocation ratio*) shadows are collocated on a processor (referred to as *shadow processor*). The N processors are grouped into S sets, each of which we call a *shadowed set*. Each shadowed set contains α main processors and 1 shadow processor. This has been illustrated in Figure 5.

Contrary to traditional process replication, shadow collocation reduces the number of processors required to achieve fault-tolerance, thereby reducing power and energy consumption. Furthermore, the collocation ratio can be adapted to reflect the propensity of the system to failure. This flexibility, however, comes at the increased cost of memory requirement at the shared processor. It is to be noted that this limitation is not intrinsic to Leaping Shadows, as in-memory checkpoint/restart and multi-level checkpoint/restart also require additional memory to store checkpoints.

Under Shadow Replication, collocation has an important ramification with respect to the resilience of the system. Specifically, only one failure can be tolerated in each shadowed set. If a shadow processor fails, all the shadows in the shadowed set will be lost, although this does not interrupt the execution of the mains. On the other hand, if a main processor 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 processor, will result in losing all the shadows in the shadowed set, thereby losing the tolerance to any other failures. After the first failure, a shadowed set becomes *vulnerable*. To address this issue, Leaping Shadows applies rejuvenation, to be discussed in Section 5.3, to maintain a persistent level of system resilience.

5.2 LEAPING

In the basic form of Shadow Replication, failures can have a significant impact on the performance. Since a shadow executes at a lower rate than its associated main, a computational *divergence* will occur between the pair of processes. As shown in Figure 14, the larger this

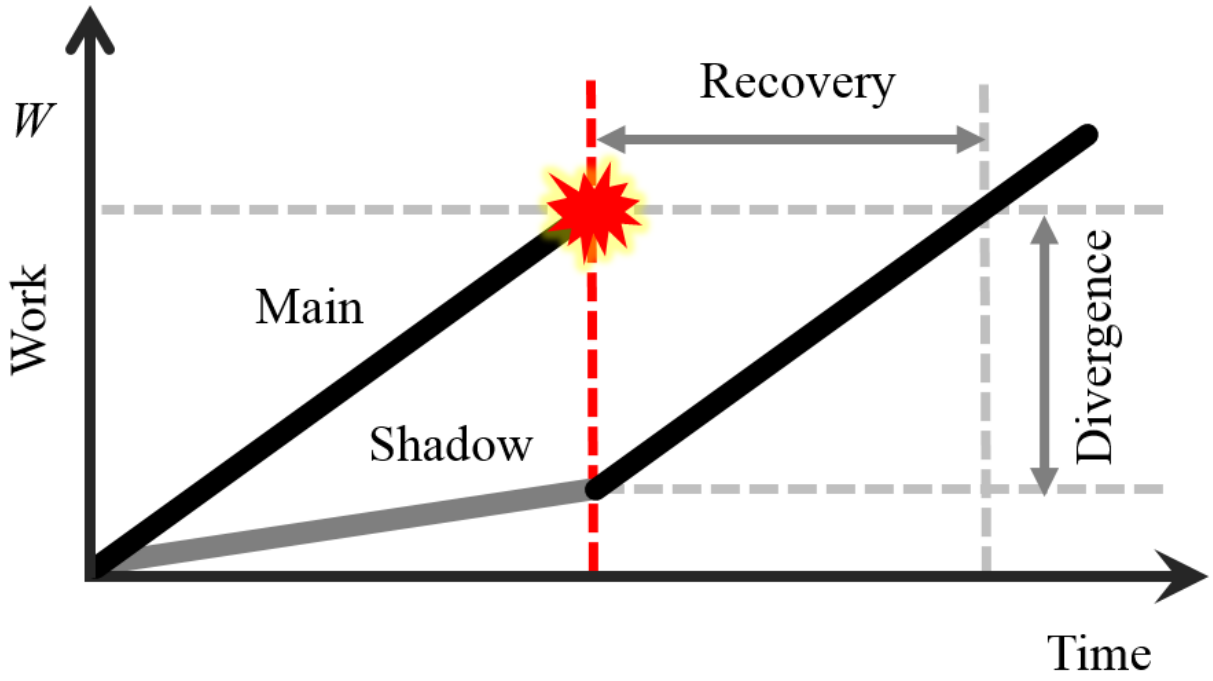


Figure 14: Illustration of divergence between a main and its shadow.

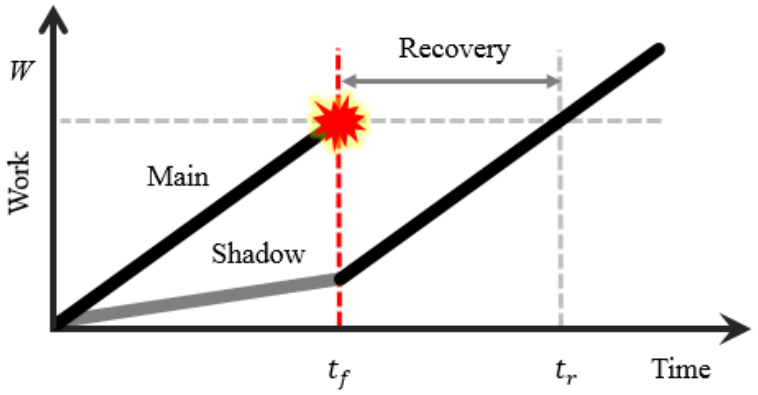
divergence, the more time the lagging shadow will need to "catch up" in the case of a failure, introducing delay to the recovery process. This problem deteriorates as dependencies incurred by messages and synchronization barriers would propagate the delay of one task to others. At the same time, divergence has another side-effect in message passing systems. Specifically, forwarding messages from mains to shadows in a message passing system will cause undesired message accumulation. Similarly, the message buffer will bear higher pressure as divergence increases.

Fortunately, the association with the mains provides an unique opportunity for the shadows to benefit from the faster execution of their mains. By copying the state of a main to its shadow, which is similar to the process of storing a checkpoint in a buddy in [150], forward progress is achieved for the shadow with minimized time and energy. This technique, referred to as *leaping*, effectively limits the divergence between main and shadow. As a result, there is no need of concern for buffer overflow, and the recovery time after a failure, which

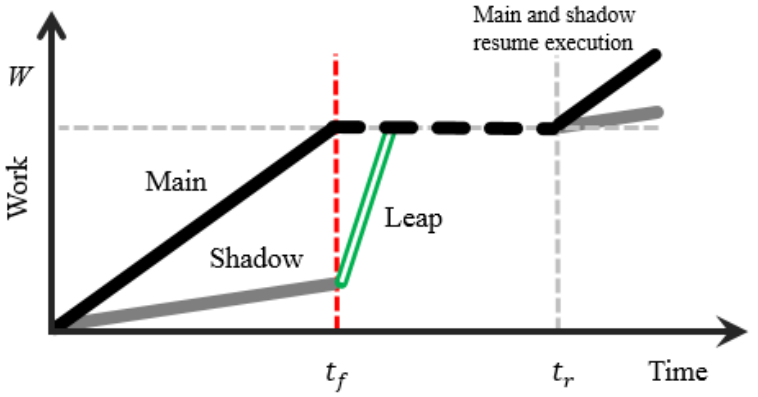
depends on the divergence between the failing main and its shadow, is also reduced.

While recovery may lead to delay due to divergence, we opportunistically overlap shadow leaping with failure recovery to avoid extra overhead. Assuming a failure occurrence at time t_f, Figure 15 shows the concept of leaping overlapped with failure recovery. 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 15(a). At the same time, as shown in Figure 15(b), the remaining main processes are blocked at the next synchronization point, which is assumed to take place shortly after t_f. 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. 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 the total energy consumption. Note that leaping is applicable, whether shadows are collocated or use DVFS. However, in the case of collocation, the leaping for some shadows could not overlap with the recovery. This will be further discussed in the next section when we integrate leaping with rejuvenation.

The main objective of leaping is to ensure forward progress in the presence of failure. However, later on we have identified that leaping is useful in a variety of scenarios. Correspondingly, we define multiple types of leaping, each of which applies to a particular scenario. Above mentioned leaping is referred to as *failure-induced leaping*, as it is triggered by a failure. As also mentioned above, message buffer pressure increases with divergence in message passing systems. If failure-induced leaping is not frequent enough, there may be a need to force a leaping to avoid buffer overflow, thus this type of leaping is referred to as *buffer-forced leaping*. Furthermore, in following chapters we will demonstrate that leaping can be used to achieve forward progress, for both shadow and main processes, in another two scenarios, resulting in *rejuvenation-induced leaping* and *voting-induced leaping*. In all scenarios, leaping always takes place between a main and its associated shadow, and thus does not require global coordination. The process which provides the leaping state is referred to as the *leap-provider*, while the process which receives the leaping state and rolls forward is referred to as the *leap-recipient*.



(a) Faulty task behavior.



(b) Non-faulty task behavior.

Figure 15: Illustration of shadow leaping after a failure.

5.3 REJUVENATION

Another shortcoming of Shadow Replication is that failures can impact the resilience of the system. Upon failure of a main, the system can only rely on an "orphan" shadow to complete the task. This is even worse when shadows are collocated. As discussed in Section 5.1, each shadowed set can only tolerate a single failure. A trivial approach to address this shortcoming is to associate a "suite" of shadows with each main. Such an approach, however, is resource wasteful and costly in terms of energy consumption. Instead, Leaping Shadows embraces

rejuvenation techniques to improve resource efficiency, when a processor on which a failure occurred can be rebooted to start new processes[1].

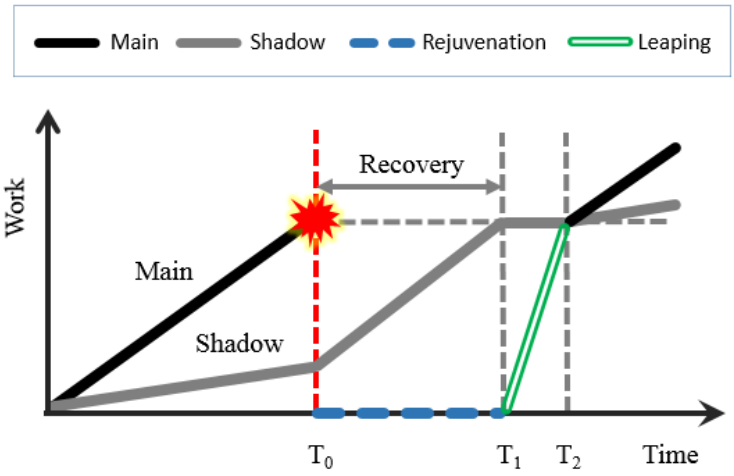
The main objective of rejuvenation is to enable the system to maintain its intended level of resilience, in the event of multiple failures. The proposed approach is to use the rescuer shadow to *rejuvenate* the failed main. Specifically, while the shadow is executing at a high rate to reach the state at which the main failed, a new process is launched to replace the failed main. Furthermore, rather than starting the new process from its initial state, *rejuvenation-induced leaping* is invoked to synchronize the new process' state with that of the recovering shadow. Similar to leaping, rejuvenation is applicable, in spite of the underlying execution rate control mechanism. The following discussion focuses on collocation as it represents the most comprehensive scenario.

Figure 16 illustrates the failure recovery process with rejuvenation, assuming that a main M_i fails at time T_0. In order for its shadow S_i to speed up, the shadows collocated with S_i are temporarily suspended. Meanwhile, the failed processor is rebooted and then a new process is launched for M_i. When, at T_1, S_i catches up with the state of M_i before its failure, leaping is performed to advance the new process to the current state of S_i.

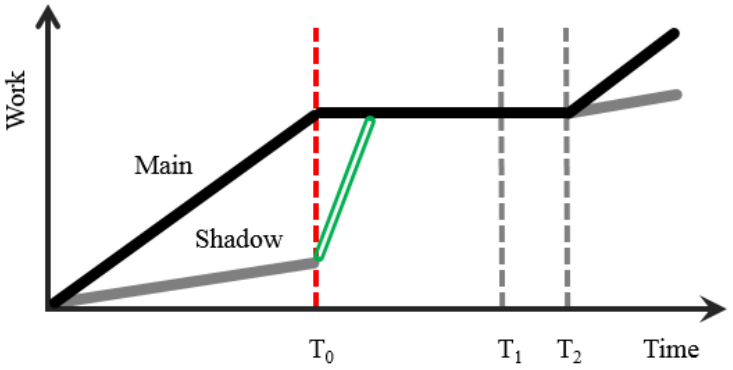
Because of the failure of M_i, the other mains are blocked at the next synchronization point, which is assumed to take place shortly after T_0. During the idle time, a leaping is opportunistically performed to transfer state from each living main to its shadow. Therefore, this leaping has minimal overhead as it overlaps with the recovery, as shown in Figure 16(b). Figure 16(c) shows that leaping for the shadows collocated with S_i are delayed until they resume execution when the recovery completes at T_1. After the leaping finishes at T_2, all mains and shadow resume normal execution, thereby bringing the system back to its original level of fault tolerance.

Figure 16 and the above description assume that the time for rebooting is no longer than the recovery time. If the new M_i is not yet ready when S_i catches up at T_1, however, two design choices are possible. In the first, S_i can assume the role of a main and continue execution. In the second, S_i can wait until the launching of the new M_i is complete. The first option requires that all non-failed processes update their internal mapping to identify the

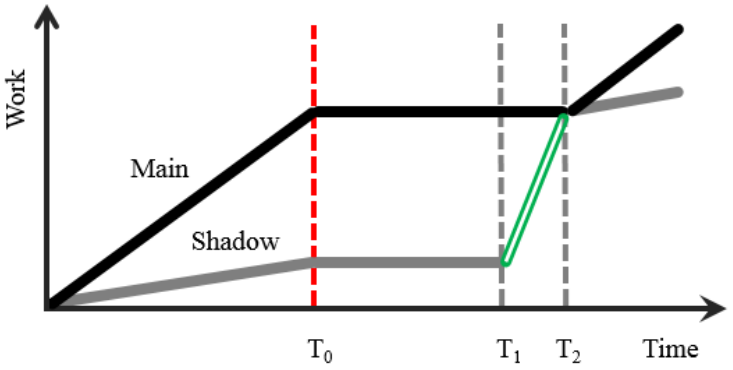
[1]Equivalently, a spare processor can be used for this purpose.



(a) Faulty task



(b) Non-faulty tasks in different shadowed sets



(c) Non-faulty tasks in the same shadowed set

Figure 16: Recovery and rejuvenation after a main process fails.

shadow as a new main and continue to correctly receive messages. This not only complicates the implementation, but also requires expensive global coordination that is detrimental to scalability. We, therefore, chose the second design option.

The above analysis focuses on rejuvenating a failed main process. Failure of a shadow can be addressed in a similar manner. Each failure requires a rebooting of the target processor to launch replacing process(es), but the only difference is that the leap-provider and leap-recipient are reversed, i.e., the main process becomes the leap-provider and shadow becomes the leap-recipient. Collocation makes the rejuvenation of shadow process slightly more complicated, since a shadow processor failure will impact all the collocated shadows on that processor. Specifically, if a shadow processor fails, all the shadows in a shadowed set are lost. To rejuvenate, the failed processor is rebooted and then a new process is launched to replace each of the failed shadow processes. It is to be noted that all the mains can continue execution while rebooting the processor. When the newly launched shadows become ready, rejuvenation-induced leaping is invoked to synchronize every shadow with its main.

5.4 ANALYTICAL MODELS

In the following we develop analytical models to quantify the expected performance of Leaping Shadows, 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 processors, and W is the application workload. M of the N processors are allocated for main processes, each having a workload of $w = \frac{W}{M}$, and the rest S processors are for the collocated shadow processes. Note that process replication is a special case of Leaping Shadows where $\alpha = 1$, so $M = S = \frac{N}{2}$ and $w = \frac{2W}{N}$.

Section 5.3 discusses rejuvenation to maintain a persistent level of system resilience. In certain situations, the underlying assumption, that failed processor can be rebooted or standby processors are available, may not hold. If this is the case, the scheme discussed in Section 5.3 becomes invalid, and one has to take the risk that a task may lose both the main and shadow processes, resulting in the entire application being re-executed. To be conservative and emphasize on the benefits of leaping in our assessment of Leaping Shadows,

we consider the general case where rejuvenation is not applicable.

5.4.1 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 [25] and results are presented in terms of Mean Number of Failures To Interrupt (MNFTI), i.e., the mean number of failures to cause an application fatal failure. Applying the same methodology, we derive the new MNFTI under collocated Leaping Shadows, as shown in Table 8. As each shadowed set can tolerate one failure, the results are for different numbers of shadowed sets (S). Table 8 reveals that the MNFTI almost doubles when the number of shadowed set increases by a factor of 4. At 2^{20} shadowed sets, the application is expected to go through 1816 processor failures before observing an interrupt. Note that when processes are not shadowed, every failure would interrupt the application, i.e., MNFTI=1.

Table 8: Application Mean Number of Failures To Interrupt (MNFTI) when Leaping Shadows 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

To further quantify the probability of application fatal failure, we use $f(t)$ to denote

the failure probability density function of each processor, and then $F(t) = \int_0^t f(\tau)d\tau$ is the probability that a processor fails in the next t time. Since each shadowed set can tolerate one failure, then the probability that a shadowed set with α main processors and 1 shadow processor does not fail by time t is the probability of no failure plus the probability of one failure, i.e.,

$$P_g = \Big(1 - F(t)\Big)^{\alpha+1} + \binom{\alpha+1}{1} F(t) \times \Big(1 - F(t)\Big)^{\alpha} \tag{5.1}$$

and the probability that an fatal failure occurs to an application using N processors within t time is the complement of the probability that none of the shadowed sets fails, i.e.,

$$P_a = 1 - (P_g)^S \tag{5.2}$$

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.

5.4.2 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 15(a)). Otherwise, a possibly 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 17. Δ_i $(1 \leq i \leq k+1)$ represents the i^{th} execution interval, and τ_i $(1 \leq i \leq k)$ 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,*

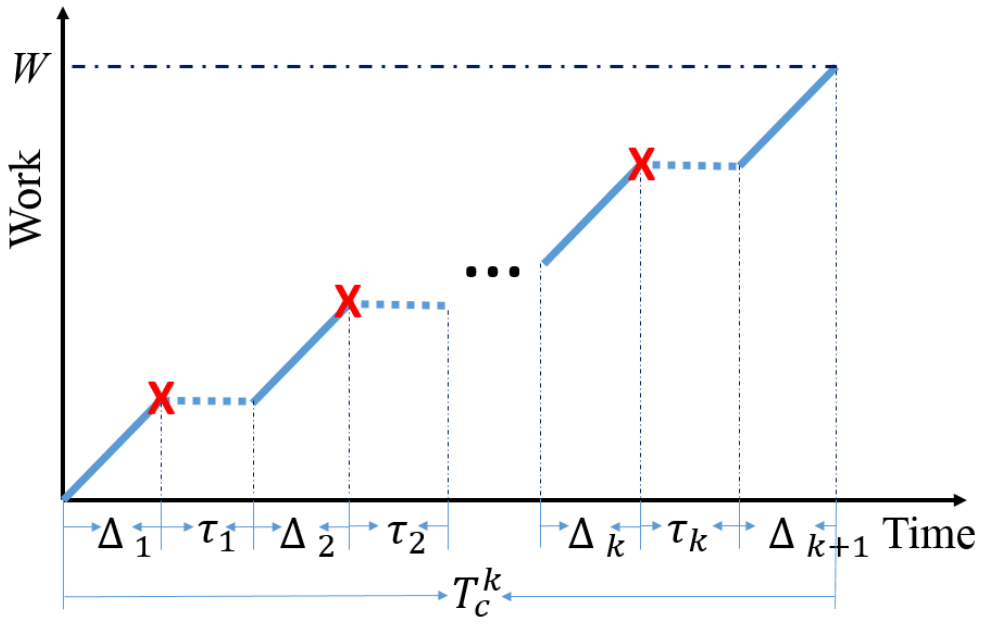


Figure 17: Application progress with shadow catching up delays.

then using Leaping Shadows,

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

Proof. Leaping Shadows guarantees that all the shadows reach the same execution point as the mains (See Figure 15) after a previous recovery, so every recovery time is proportional to its previous execution interval. That is, $\tau_i = \Delta_i \times (1 - \sigma_b)$. According to Figure 17, the completion time with k failures is $T_c^k = \sum_{i=1}^{k+1} \Delta_i + \sum_{i=1}^{k} \tau_i = w + (1 - \sigma_b) \sum_{i=1}^{k} \Delta_i$ □

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* $(1 - \sigma_b)w$.

Proof. From above theorem we can see the delay from k failures is $(1 - \sigma_b) \sum_{i=1}^{k} \Delta_i$. 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, $(1 - \sigma_b) \sum_{i=1}^{k} \Delta_i \leq (1 - \sigma_b)w$. □

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 processor during the execution can be calculated as $P_c = F(w)$. Then the probability that there are k failures among the N processors is

$$P_s^k = \binom{N}{k} {P_c}^k (1 - P_c)^{N-k} \tag{5.3}$$

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 Leaping Shadows,* $T_{total} = T_c/(1 - P_a)$*, 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 5.4.1, the total expected completion time is $T_{total} = T_c/(1 - P_a)$. □

Process replication is a special case of Leaping Shadows where the collocation ratio for shadows is 1, so we can apply the above theorem to derive the expected completion time for process replication, when it uses the same amount of processors:

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 processors 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 $T_{total} = T_c/(1 - P_a) = 2W/N/(1 - P_a)$. □

5.4.3 Expected Energy Consumption

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

$$En = N \times p \times T_{total} \tag{5.4}$$

Even using the same amount of processors, Leaping Shadows can save power and energy, since main processors are idle during the recovery time after each failure, and the shadows can achieve forward progress through failure-induced leaping. During normal execution, all the processors consume static power as well as dynamic power. During recovery time, however, the main processors are idle and consume only static power, while the shadow processors first perform leaping and then become idle. Altogether, the expected energy consumption for Leaping Shadows can be modeled as

$$En = N \times p_s \times T_{total} + N \times p_d \times w + S \times p_l \times T_l. \tag{5.5}$$

with p_l denoting the dynamic power consumption of each processor during leaping and T_l the expected total time spent on leaping.

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

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

5.5 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 processors N, and MTBF of each processor; the application category is mainly the total workload, W; and collocation ratio α in the algorithm category determines the number of main processors and shadow processors ($N = M + S$ and $\alpha = M/S$). In this section, we evaluate each performance metric of Leaping Shadows, with the influence of each of the factors considered.

5.5.1 Comparison to Checkpoint/restart and Process Replication

We compare with both process replication and checkpoint/restart, assuming the same number of processors to use. The completion time with checkpoint/restart is calculated with Daly's model [36] assuming 10 minutes for both checkpointing and restart. The energy consumption is then derived with Equation 5.4. It is important to point out that we always assume the same number of processors, so that process replication and Leaping Shadows do not use extra processors 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 [36]. Specifically, $\Delta_i = w/(k+1)$, and $T_c^k = w + w * (1 - \sigma_b) * \frac{k}{k+1}$. Further, we assume that each shadow gets a fair share of its processor's execution rate so that $\sigma_b = \frac{1}{\alpha}$. To calculate Equation 5.5, we assume that the dynamic power during leaping is twice of that during normal execution, i.e., $p_l = 2 * p_d$, and the time for leaping is half of the recovery time, i.e., $T_l = 0.5 * (T_{total} - w)$.

The first study uses $N = 1$ million processors, effectively simulating future extreme-scale computing environments, and assumes that $W = 1$ million hours, and static power ratio

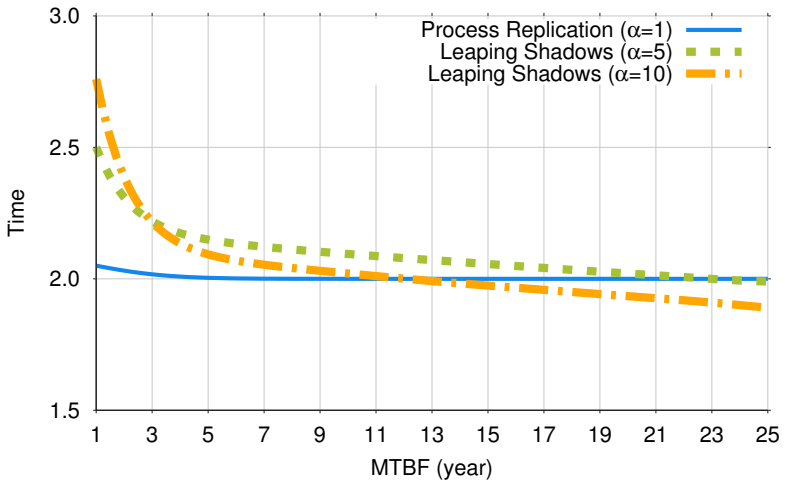
$\rho = 0.5$. Our results show that at extreme-scale, the expected completion time and energy consumption of checkpoint/restart are orders of magnitude larger than those of Leaping Shadows and process replication. Therefore, we choose not to plot a separate graph for checkpoint/restart.

Figure 18(a) reveals that the most time efficient choice largely depends on MTBF. When MTBF is high, Leaping Shadows requires less time as more processors are used for main processes and less workload is assigned to each process. As MTBF decreases, process replication outperforms Leaping Shadows as a result of the increased likelihood of rollback for Leaping Shadows. In terms of energy consumption, Leaping Shadows has much more advantage over process replication. For MTBF from 2 to 25 years, Leaping Shadows 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), Leaping Shadows with $\alpha = 10$ consumes more energy because of extended execution time.

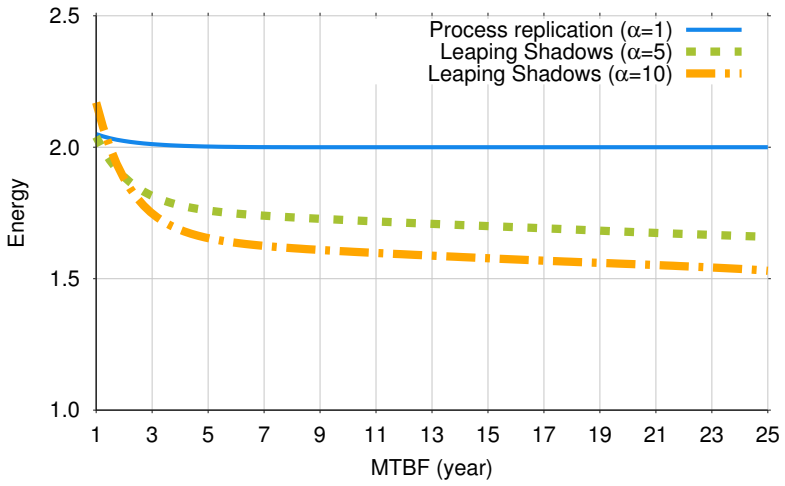
5.5.2 Impact of Processor Count

The system scale, measured in number of processors, 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 19. Please note that the time and energy for checkpoint/restart 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 19(a) clearly shows that each of the three fault tolerance alternatives has its own advantage. Specifically, checkpoint/restart is the best choice for small systems at the scale of 10,000 processors, Leaping Shadows outperforms others for systems with 100,000 processors, while process replication has slight advantage over Leaping Shadows for larger systems. On the other hand, Leaping Shadows wins for all system sizes when energy consumption is the objective.

When MTBF is changed to 25 years, the performance of checkpoint/restart improves a lot, due to the reduced frequency of checkpointing and decreased need of restarting. However, compelled to rollback every time there is a failure, checkpoint/restart is still orders of



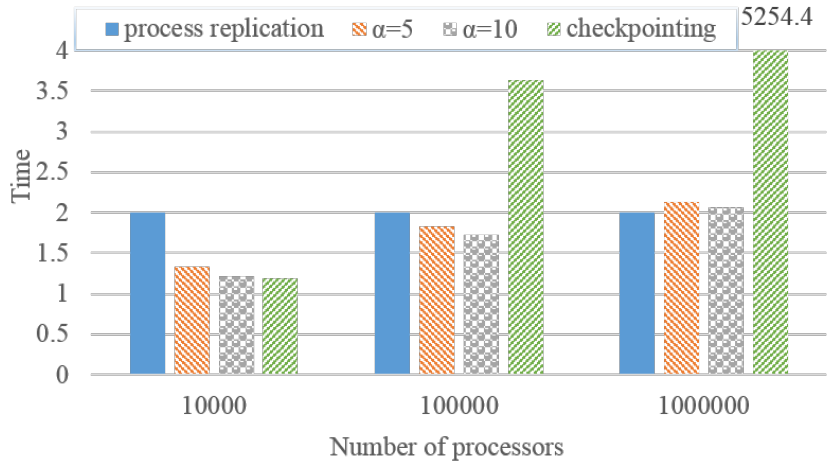
(a) Expected completion time



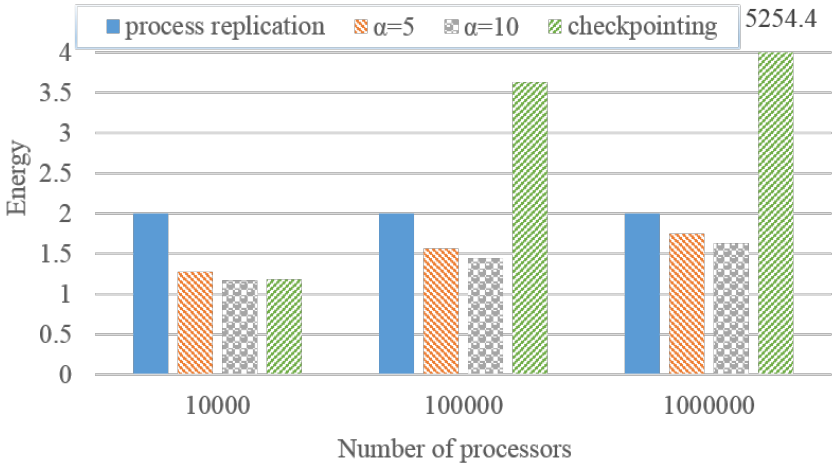
(b) Expected energy consumption

Figure 18: Comparison of time and energy for different processor level MTBF. $W = 10^6$ hours, $N = 10^6$, $\rho = 0.5$.

magnitude worse in time and energy than that of the other two approaches. Leaping Shadows benefits much more than process replication from the increased MTBF. As a result, Leaping Shadows is able to achieve shorter completion time than process replication when N reaches 1,000,000.



(a) Expected completion time



(b) Expected energy consumption

Figure 19: Comparison of time and energy for different number of processors. $W = N$, MTBF=5 years, $\rho = 0.5$.

5.5.3 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. Hence, it is intuitive that workload would impact the performance comparison.

Fixing N at 1,000,000, we increase W from 1,000,000 hours to 12,000,000 hours. Figure 20 assumes a MTBF of 25 years and shows both the time and energy. Checkpoint/restart 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, Leaping Shadows is able to achieve the most savings in all cases. When MTBF of 5 years is used, the difference is that process replication consumes less energy than Leaping Shadows when W reaches 6,000,000.

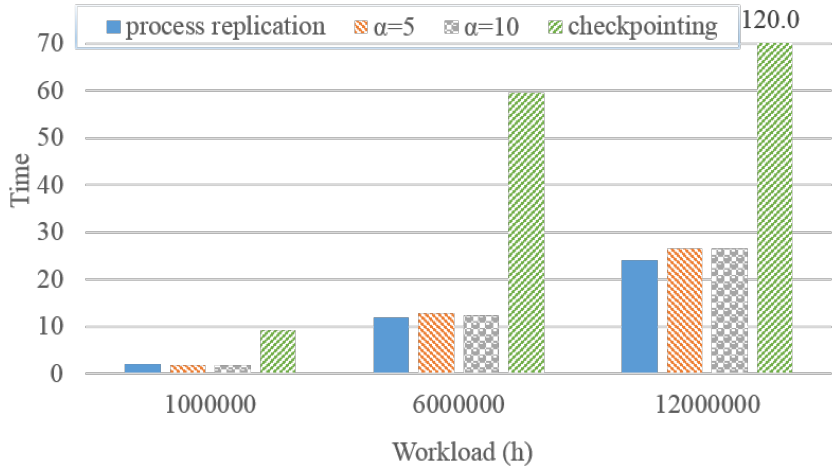
5.5.4 Impact of Static Power Ratio

With various architectures and organizations, servers vary in terms of power consumption. The static power ratio ρ is used to abstract the amount of static power consumed versus dynamic power. Considering modern systems, we vary ρ from 0.3 to 0.7 and study its effect on the expected energy consumption. The results for Leaping Shadows with $\alpha = 5$ are normalized to that of process replication and shown in Figure 21. The results for other values of α have similar behavior and thus are not shown. Leaping Shadows achieves more energy saving when static power ratio is low, since it saves dynamic power but not static power. When static power ratio is low ($\rho = 0.3$), Leaping Shadows is able to save 20%-24% energy for the MTBF of 5 to 25 years. The saving decreases to 5%-11% when ρ reaches 0.7.

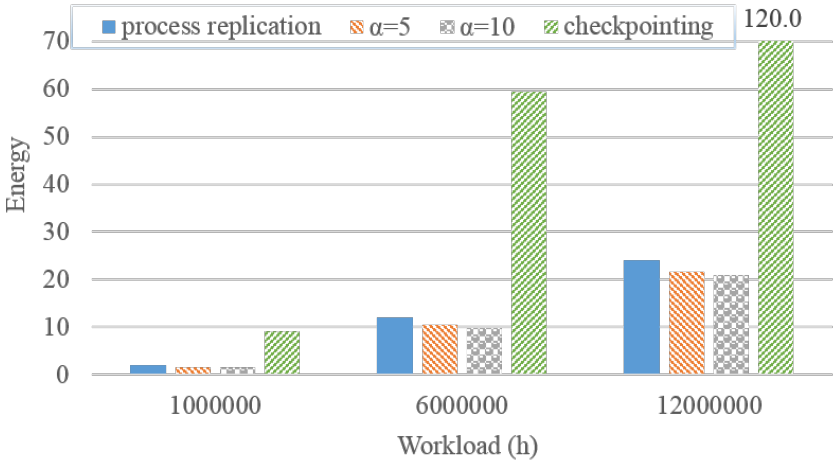
5.5.5 Adding Collocation Overhead

Leaping Shadows increases memory requirement[2] when multiple shadows are collocated. Moreover, this may have an impact on the execution rate of the shadows due to cache contention and context switch. To capture this effect, we re-model the rate of shadows as $\sigma_b = \frac{1}{\alpha^{1.5}}$. Figure 22 shows the impact of collocation overhead on expected energy consumption for Leaping Shadows 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%.

[2]Note that this problem is not intrinsic to Leaping Shadows, as in-memory checkpoint/restart also requires extra memory.



(a) Expected completion time



(b) Expected energy consumption

Figure 20: Comparison of time and energy for different workloads. $N = 10^6$, MTBF=25 years, $\rho = 0.5$.

The reason is that failure-induced 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%.

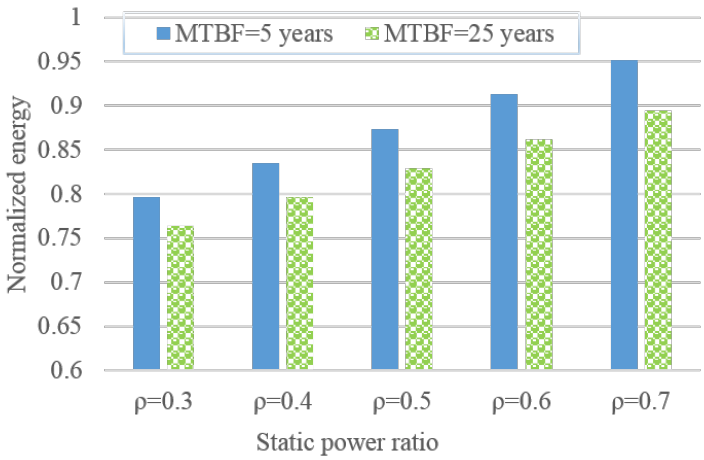


Figure 21: Impact of static power ratio on energy consumption. $W = 10^6$ hours, $N = 10^6$, α=5. All energies are normalized to that of process replication.

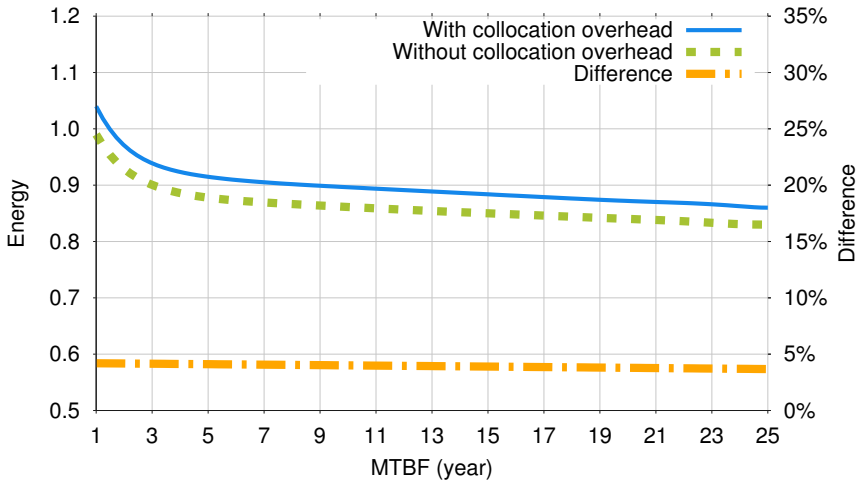


Figure 22: Impact of collocation overhead on energy consumption. $W = 10^6$ hours, $N = 10^6$, ρ=0.5, α=5.

5.6 SUMMARY

As the scale and complexity of HPC systems continue to increase, both the failure rate and power consumption are expected to increase dramatically, making it extremely challenging to deliver the designed performance. Existing fault tolerance methods rely on either time or hardware redundancy. Neither of them appeals to the next generation of supercomputing, as the first approach may incur significant delay while the second one constantly wastes over 50% of the system resources.

In this work, we present a comprehensive discussion of the techniques that enable Leaping Shadows to achieve scalable resilience in future extreme-scale computing systems. 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 traditional process replication and checkpoint/restart, we identify the scenarios where each of the alternatives should be chosen for best performance.

6.0 TOLERANCE OF SILENT DATA CORRUPTION

Modern scientific discoveries and business intelligence rely heavily on large-scale simulation and data analytics. The next generation of parallel applications will require massive computing capacity to support the execution of predictive models and analysis of massive quantities of data, with significantly higher resolution and fidelity than what is possible within existing computing infrastructures. In order to deliver the desired performance of emerging applications, future HPC and Cloud computing infrastructure is expected to continue to grow in both scale and complexity, resulting in an urgent need for efficient and scalable fault tolerance solutions to all kind of causes and symptoms.

A significant body of work targets at improving the scalability of checkpoint/restart, while also lots of efforts are devoted to making process replication based approaches a viable alternative. Most of the existing work assumes a fail-stop fault model, such that failures are detectable by monitoring hardware or network. Silent data corruption (SDC) is yet a another class of failures. Different from crash failures, SDC may remain undetected and corrupt the intermediate or final results. It materializes as bit flips in storage (both volatile and non-volatile memory) or even within processing cores. In modern computers, a single bit flip in memory can be detected with cyclic redundancy check (CRC) and corrected with error correction code (ECC). Double bit flips, however, is beyond the hardware fault tolerance capability in most systems. Meanwhile, even single bit flips in the processor core remain undetected as only caches feature ECC while register files or even ALUs typically do not [55].

In large-scale production systems, SDC has become a major concern for the user, which can not only cause data loss, but also has serious impact on the integrity of job outputs, jeopardizing scientific research or business decisions. SDC is one of the most critical problems in cloud data processing [142]. As the capacity of and memory and disks grows, the likelihood

of SDC caused by hardware failures also increases. For example, a hard drive failure resulted in Facebook temporarily losing over 10% of photos published on their social network [142]. Similarly, SDC due to software bugs also becomes prevalent as the scale of cloud systems keeps expanding. For example, Amazon Simple Storage Service (S3) once experienced a data corruption problem caused by a load balancer bug [10]. In HPC systems, SDC is also attracting more attention. It is reported that, due to the high density of DIMMs, Cray XT5 at Oak Ridge National Labs encounters double bit flips on a daily basis [61]. Although previous studies have shown that disk errors and DRAM errors in large-scale production systems are happening often enough to require attention, little research has been done to answer to this challenge [55].

This chapter applies the Leaping Shadows model, discussed in Chapter 5, to deal with SDC in large-scale computing systems. In order to detect and correct SDC, we come up with a new scheme that equips Leaping Shadows with triple modular redundancy, similar to [55]. This new scheme inherits the adaptivity and power-awareness from Leaping Shadows, and optimizes a combination of process execution rates to balance the trade-offs among time, hardware, and power. In the following sections, we first describe a parallel programming paradigm typically used in compute- and data-intensive applications. Next, we present the new Leaping Shadows scheme with its execution model. Then we build analytical models to study its performance and power requirements, and form an optimization framework, which derive the optimal execution parameters. Lastly, comparative analysis and evaluation results are given.

6.1 PARALLEL PROGRAMMING PARADIGM

Parallel computing frameworks, such as MPI, MapReduce, and Pregel, have been widely adopted for large-scale data analytics and simulation. To efficiently handle the sheer amount of data, and to utilize a cluster of nodes and/or multiple processors within a node, these frameworks typically arrange a job into hundreds or thousands of tasks scheduled to execute in parallel. There are two widely adopted approaches to parallelism. In task-parallelism, we

partition the various tasks carried out in solving the problem among the processors. In data-parallelism, we partition the data used in solving the problem among the processors, and each processor carries out more or less similar operations on its part of the data. Despite the parallelism approach, the parallel processing paradigm is abstracted by the Bulk Synchronous Parallel (BSP) model [127].

According to the BSP model, there is a set of processors which may follow different threads of computation, with each processor equipped with fast local memory and interconnected by a communication network. A BSP computation proceeds in a series of global supersteps, which consists of three components:

- Computation: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor.

- Communication: The processes exchange data between themselves to facilitate remote data storage capabilities.

- Barrier synchronization: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier.

Pregel is directly inspired by the BSP model. In MPI programs, whose main body usually consists of a loop, each iteration could be a BSP superstep. This is also true for iterative MapReduce applications, in which each superstep includes a map or a reduce phase.

In today's parallel computing frameworks, re-execution on top of a heartbeat protocol is mainly used to provide fault tolerance and to deal with staggers. This approach works fine for system scales such that failures are rare and applications that are delay-tolerant. If failures are frequent, however, large delay can be incurred since one faulty task or stagger may delay the whole job execution. In addition, re-execution only handles crash failures. This is not acceptable for applications with strict response time requirements or applications that are vulnerable to SDC. In contrast, Leaping Shadows, if applied, will enable these frameworks to trade-off among multiple objectives, while respecting any hard or soft deadline and handling both crash failures and SDC.

6.2 LEAPING SHADOWS EXECUTION MODEL

Different from crash failures which crash a processor, silent data corruption allows a faulty processor to continue to completion but may silently generate incorrect results. To deal with f failures, $(2f + 1)$ replicas are needed, and voting is required periodically to detect failure. For example, when at most 1 silent data corruption could occur, 3 replicas are sufficient to detect and tolerate the failure. In the following, we will focus on tolerating one silent data corruption per task.

Compared to traditional replication techniques, Leaping Shadows can deal with silent data corruption with higher efficiency and less resource requirement. For each task, Leaping Shadows associates two shadows with each main process. One shadow is designated as the primary shadow, and its duty is to compare with its associated main at a voting point to detect SDC. Based on the BSP model, each barrier synchronization is naturally a voting point. In order to detect failure as soon as possible, the primary shadow executes at the same rate as its associated main. In addition to a primary shadow, a secondary shadow is needed to correct a SDC, once occurred, through voting. To save energy, the secondary shadow executes at a potentially lower rate than the other two replicas, and dynamically speeds up if failure is detected.

To take advantage of the forward progress of the fast replicas, Leaping Shadows performs a leaping at every voting point to leap forward the secondary shadow when possible. Specifically, when the main and the primary shadow both arrive at a voting point and they reach agreement, the results and execution state are copied to the secondary shadow to achieve forward progress. This is illustrated in Figure 23. Since this leaping is triggered by a voting, it is referred to as *voting-induced leaping*. If the main or the primary shadow experiences a SDC, the failure will be detected at the next voting point. At this time, the secondary shadow speeds up to reach the specific voting point, and participates in the voting to detect which replica fails. Then a leaping from one correct replica to the failed replica can rejuvenate the failed one, after which all replicas resume normal execution. Note that the secondary shadow is only useful when one of the other two replicas fails. If the secondary shadow fails, it will automatically get rejuvenated by leaping.

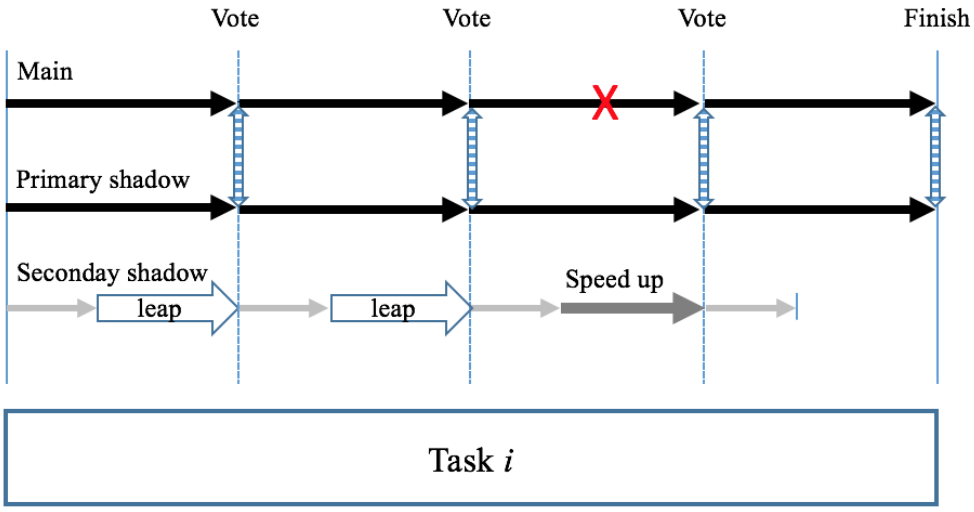


Figure 23: Tolerating SDC by applying Leaping Shadows with triple modular redundancy.

6.3 ANALYTICAL MODELS AND OPTIMIZATION FRAMEWORK

To assess the efficiency of Leaping Shadows to deal with SDC, following subsections develop analytical models for the expected response time and energy consumption under a failure distribution. Then using these analytical models, an optimization problem is formulated to minimize the expected energy consumption of Leaping Shadows under strict deadline constraint. This not only demonstrates Leaping Shadows' adaptivity to the desired trade-off, but also provides a framework with which we can perform comparative performance analysis to state-of-the-art approaches.

6.3.1 Notations

Let W denote the total workload to process a task. Let N denote the number of BSP supersteps, which is also the number of voting points. Then each voting interval has a workload of $w = \frac{W}{N}$. Let σ_{max} denote the maximum execution rate. $R_{min} = \frac{W}{\sigma_{max}}$ is the minimal response time. Let $\overline{R} = (1+\alpha)R_{min}$ $(0 \leq \alpha \leq 1)$, where α is called laxity factor,

denote the target response time considering fault tolerance overhead. Let λ denote the failure rate, and $f(t)$ denote the failure density function. Let $E(\sigma, [t_1, t_2])$ denote the energy consumption of a replica when executing at rate σ for an interval from t_1 to t_2. For each leaping, let T_l denote the time overhead, and E_l denote the energy cost.

To deal with silent data corruption, the Leaping Shadows model associates two shadows with the main process for each task. According to the execution model described in above section, there are three execution rates:

- σ_m, the execution rate of the main and the primary shadow
- σ_b, the execution rate of the secondary shadow before a SDC is detected
- σ_a, the execution rate of the secondary shadow after a SDC is detected

6.3.2 Response Time

Assuming there is at most one silent data corruption per task, two scenarios need to be considered, i.e., a SDC occurs to the main process or the primary shadow, or neither of them fails. The failure of the secondary shadow has no impact on the response time, thus is ignored in the following analysis.

In the first scenario, where no failure occurs, the execution time is determined by the main process, as $t_r^m = \frac{W}{\sigma_m}$. Considering the time for leaping, the total response time is $t_{rl}^m = \frac{W}{\sigma_m} + N \times T_l$.

In the second scenario, where a fast replica fails, the delay is the time for the secondary shadow to catch up with respect to a voting interval. The time for the main process to complete a voting interval is $t_v = \frac{w}{\sigma_m}$. The time required by the secondary shadow to complete the remaining work in the current interval is $t_d = \frac{w - t_v \times \sigma_b}{\sigma_a}$. The execution time is $t_r^s = t_r^m + t_d$. Considering the time for leaping, the total response time is $t_{rl}^s = t_r^s + N \times T_l$.

6.3.3 Power and Energy Consumption

Dynamic voltage and frequency scaling (DVFS) is assumed in this work to achieve the desired execution rate of a process. It is well known that one can reduce the dynamic processor power consumption at least quadratically by reducing the frequency linearly. The dynamic

processor power consumption executing at rate σ is given by the function $p_d(\sigma) = \sigma^n$ where $n \geq 2$. Throughout this section, we assume that dynamic power is cubic in relation to the processor frequency.

In addition to the dynamic power, processor leakage and other components (memory, disk, network etc.) all contribute to static power consumption, which is independent of the processor frequency. In this section, we define static power as a fixed fraction of the total power consumed when executing at maximum rate, referred to as ρ. Hence, a processor's power consumption is expressed as $p(\sigma) = \rho \times \sigma_{max}^3 + (1 - \rho) \times \sigma^3$.

Next we calculate the expected energy consumption for a single task under Leaping Shadows. Corresponding to the two scenarios in the above response time analysis, the energy consumption also falls into two cases. If neither of the main process and primary shadow fails, the energy consumption of the three replicas weighted by its probability is

$$E_1 = (1 - \int_0^{t_r^m} f(t)dt)^2 \times \{2E(\sigma_m, [0, t_r^m]) + E(\sigma_b, [0, t_r^m])\} \tag{6.1}$$

The first factor is the probability that neither of them fails, as $\int_0^{t_r^m} f(t)dt$ is the probability that a replica encounters a SDC during task execution. The second factor models the energy of the three replicas from task start to end, with two replicas executing at rate σ_m and one replica executing at σ_b.

If the main or the primary shadow fails, the energy consumption weighted by its probability is

$$\begin{aligned} E_2 =& 2 \times (1 - \int_0^{t_r^m} f(t)dt) \times \int_0^{t_r^m} f(t)dt \times \\ & \{2E(\sigma_m, [0, t_r^m]) + E(\sigma_b, [0, t_r^m]) + 2E(0, [t_r^m, t_r^s]) + E(\sigma_a, [t_r^m, t_r^s]\} \end{aligned} \tag{6.2}$$

The first line calculates the probability that one fast replica fails while the other successfully completes. In addition to the energy in the first scenario, also accounted is the energy consumed during the secondary shadow catches up. This energy corresponds to the main process and the primary shadow idly waiting and the secondary shadow speeding up to reach the next voting point.

All in all, the total energy consumption is the sum of the above two, plus the energy cost of leaping, i.e., $E_{total} = E_1 + E_2 + N \times E_l$.

6.3.4 Optimization

When applying Leaping Shadows to deal with SDC, an optimization problem formulation is also needed to derive the optimal execution rates for both the main and shadow processes. With the generic optimization framework introduced in Chapter 3, we define the optimization objective as minimizing energy under response time constraint:

$$\begin{aligned} \min_{\sigma_m,\sigma_b,\sigma_a} \quad & E_{total}(W, N, \overline{R}, \rho, \lambda, \sigma_{max}, T_l, E_l) \\ s.t. \quad & 0 \leq \sigma_m \leq \sigma_{max} \\ & 0 \leq \sigma_b \leq \sigma_m \\ & \sigma_b \leq \sigma_a \leq \sigma_{max} \\ & t^s_{rl} \leq \overline{R} \end{aligned} \tag{6.3}$$

The first constraint says the execution rate of the main process and primary shadow should observe the physical processor limit. The second constraint indicates that the initial rate of the secondary shadow should not exceed that of the main process and primary shadow. The third constraint ensures that the secondary shadow could speed up after detecting a failure. The last constraint guarantees that the deadline is met even in the case of failure. Same as before, non-linear optimization techniques can be used to solve the above problem, and the output will be the three optimal execution rates.

6.4 EVALUATION

Using the optimization framework developed above, this section evaluates the performance of Leaping Shadows by comparing with traditional process replication using triple modular redundancy [55], under various environments and different application requirements. With an uniform treatment of the replicas, process replication requires all three replicas execute at the same rate before and after a failure. In the comparison, however, we also optimize this single rate for process replication in a way similar to Section 6.3.4[1], in order to demonstrate

[1]Process replication is a special case of Leaping Shadows where $\sigma_m = \sigma_b = \sigma_a$.

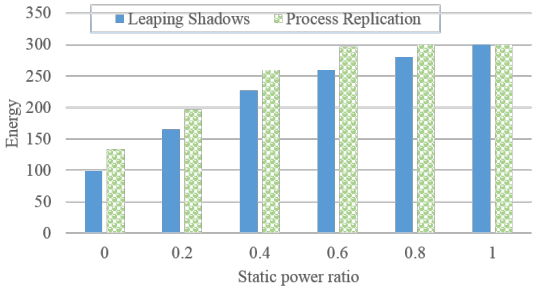
the benefits of the unique design in Leaping Shadows.

After careful analysis of the analytical models, we identify the important parameters of static power ratio, laxity in deadline, workload, number of voting interval, and leaping cost, and study the impact of each parameter. With the understanding that failure rate of each processor will remain more or less the same in the near future, We fix MTBF to be 5 years for realistic consideration. The comparison results are shown in Figure 24.

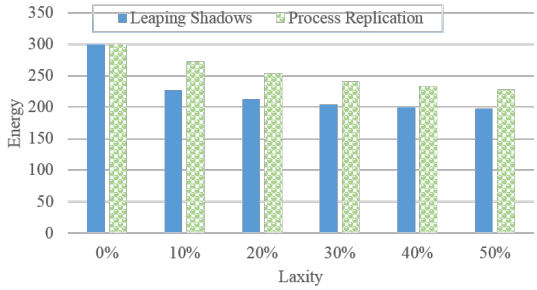
Figure 24(a) reveals the energy consumption of the two compared approaches at different static power ratios, when laxity is 50% and number of voting intervals is 10. When static power ratio is less than 0.8, both approaches reduces the execution rates to minimize energy. At 0 static power ratio, Leaping Shadows saves 25.6% energy compared to process replication. The saving decreases as static power ratio increases, and finally Leaping Shadows converges to process replication when static power ratio reaches 1. Modern computers has a static power ratio between 40% and 70% [20]. Within this target range of static power ratio, Leaping Shadows achieves 12.5% energy savings with respect to process replication.

With 10 voting intervals and 0.3 as the static power ratio, the impact of laxity is illustrated in Figure 24(b). When there is no laxity, Leaping Shadows is forced to execute all three replicas at the maximum rate, which is essentially process replication. As laxity increases, both compared approaches have more room to slow down, thereby reducing energy. By coupling two fast replicas with a slow one and using leaping to achieve forward progress for the slow replica, Leaping Shadows always saves 13.5%-16.9% energy compared to process replication when there is laxity.

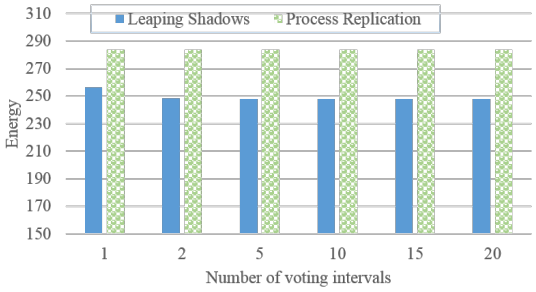
One interesting questions is how the number of voting intervals changes the picture. Without considering the overhead of leaping, intuition tells us that the more voting intervals, the less effect a failure can have on the total execution time of Leaping Shadows, and thus the better performance. This is mostly true, according to Figure 24(c). However, the figure also shows that after the number of voting intervals reaches 5, its impact becomes negligible. We also study this behavior with static power ratio changed to 0.3, as shown in Figure 24(d). As a result, both of the two compared approaches are able to reduce energy consumption by further slowing down. Although the difference between them decreases at a small number of voting intervals, Leaping Shadows keeps increasing its energy savings with the number of



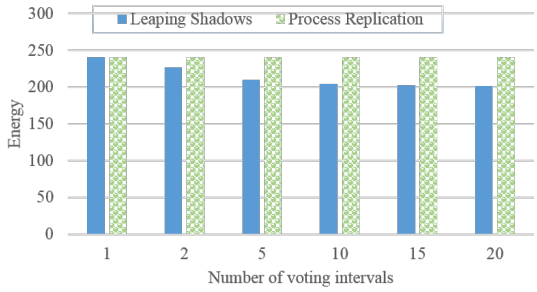
(a) Impact of static power ratio. $\alpha = 50\%$, $N = 10$, $W = 100$ hours.



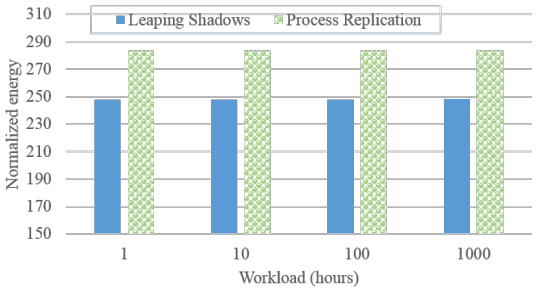
(b) Impact of laxity. $\rho = 0.3$, $N = 10$, $W = 100$ hours.



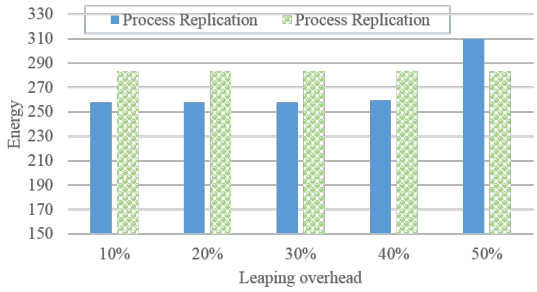
(c) Impact of voting interval. $\alpha = 50\%$, $\rho = 0.5$, $W = 100$ hours.



(d) Impact of voting interval. $\alpha = 50\%$, $\rho = 0.3$, $W = 100$ hours.



(e) Impact of task workload. $\alpha = 25\%$, $\rho = 0.5$, $N = 10$.



(f) Impact of leaping cost. $\alpha = 50\%$, $\rho = 0.5$, $N = 10$, $W = 100$ hours.

Figure 24: Comparison between Leaping Shadows and process replication for energy consumption under silent data corruption. MTBF=5 years.

voting intervals, up to 20.

The next parameter studied is the total workload, which determines the failure-free execution time, and thus the propensity of the task to failures. For Figure 24(e), we vary the task workload from 1 hour to 1000 hours. Compared to the 5 years' MTBF, all the workloads considered are relatively small, thus the failure of a given task is very unlikely. Therefore, there is only slight change in the energy consumption.

All the above experiments ignore the overhead of leaping. Assuming each time leaping consumes 1 unit of energy, the last experiment studies the impact of leaping overhead, which is defined as a fraction of the minimum response time. Figure 24(f) reveals that when the overhead is below 30%, it has negligible impact. At 40% overhead, Leaping Shadows is forced to slightly increase its rates, and thus incurs a slightly higher energy consumption. When overhead is 50%, it essentially offsets the laxity, which is 50%, and leaves no room for Leaping Shadows to slow down. As a result, all replicas need to execute at the maximum rate and end up with consuming more energy than process replication, which does not perform leaping at all.

6.5 SUMMARY

Scientific research and engineering development are increasingly relying on computational modeling, simulation, and data analytics to augment theoretical analysis. Behind the wheel, data has become the ultimate driving force that yields insights and propels innovation. With a torrent of data generated every second from distributed sensors, social media, software logs and so on, it is critical to analyze and visualize the data in a timely manner, at massively parallel scale, and with fault tolerance capabilities.

Leaping Shadows is a novel fault-tolerant computational model that unifies HPC and Big Data analytics. The flexibility within the model allows it to embrace different optimization techniques in accordance with the underlying workloads, whether compute-intensive or data-intensive. By designing the execution model of Leaping Shadows in accordance with the generic Bulk Synchronous Parallel model, Leaping Shadows can be applied to both HPC

and Cloud environments, to deal with different types of failures, or multiple types of failures at the same time.

While previous chapters discuss on tolerance of crash failures, this chapter extends the Leaping Shadows model and studies the tolerance of silent data corruption. By exploring the interplay between performance, fault-tolerance, and energy consumption, Leaping Shadows is predicted to save a significant amount of energy (up to 64.9%) compared to existing fault tolerance approaches, while respecting strict response time requirements. In the future, we plan to implement this model and perform intensive empirical evaluation to verify the accuracy of the analytical models.

7.0 rsMPI: AN IMPLEMENTATION IN MPI

In a complex system like the supercomputers we have today, the performance of Leaping Shadows is subject to both the hardware configuration, such as failure detection ability and the execution rate control mechanism, and software behavior, such as communication patterns and the amount of synchronization. It's extremely difficult for an analytical framework to precisely capture every detail of Leaping Shadows when it runs in a real environment. Therefore, an executable prototype is necessary to prove its validity as well as measure its actual performance.

As a proof of concept, we implement Leaping Shadows in a runtime library, referred to as rsMPI[1], for Message Passing Interface (MPI), which is the de facto programming paradigm for HPC. With the understanding that MPI standard keeps evolving to support new features and semantics, we focus on the most essential aspects and make rsMPI MPI-1 compliant. Currently, rsMPI focuses on tolerating crash failures and associates one shadow process with each main process. With the rejuvenation technique introduced in Chapter 5, applying Leaping Shadows with dual modular redundancy minimizes the hardware and power costs, while being able to tolerate multiple failures. It is to be noted that, it is straightforward to extend this implementation to associate two shadows with each main, for the purpose of dealing with silent failures, as discussed in Chapter 6.

Instead of a full-feature MPI implementation, rsMPI library is designed to be a separate layer between MPI and user application, and it uses function wrappers on top of the MPI profiling hooks to intercept every MPI call. This is similar to the rMPI and redMPI implementations [54, 55]. There are three benefits for this choice: 1) we can save tremendous time and efforts of rebuilding MPI from scratch; 2) we can take advantage of existing MPI

[1]The source code is available on BitBucket at git@bitbucket.org:Michael870/lsmpi.git

performance optimization that numerous researches have spent years on, such as leveraging Remote Directory Memory Access (RDMA) and OS bypass [144]; and 3) the library is portable across all MPI implementations. When linked to an MPI application, rsMPI transparently spawns the shadow processes at the initialization phase, manages the coordination between main and shadow processes during execution, and guarantees order and consistency for messages and non-deterministic events.

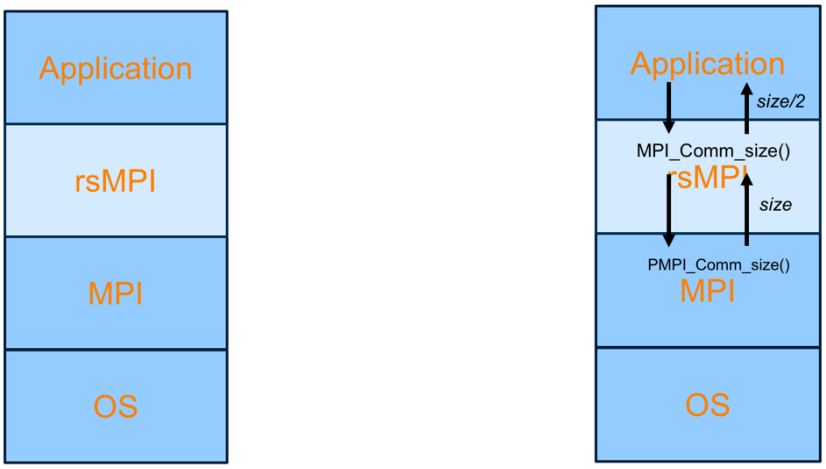
To avoid the drawbacks of DVFS, current rsMPI applies collocation to achieve the desired execution rates for the shadows. While each main executes on a separate processor at the maximum rate for HPC's throughput consideration, shadows are configured to collocate and execute at a slower rate based on a user configuration file. According to the user-provided number of processes and collocation ratio, rsMPI generates an MPI rankfile and provides it to the MPI runtime to control the process to processor mapping. Note that rsMPI always maps the main and shadow of a same task onto different nodes. This is required to prevent a fault on one node from affecting both a main and its associated shadow.

Despite the overview of the rsMPI library above, many challenges lie in efficiently integrating MPI with Leaping Shadows to support fault tolerance with high performance. One challenge is the preservation of the rich communication semantics in MPI standard. To be MPI-1 compliant, rsMPI needs to support both blocking and non-blocking point-to-point communications and blocking collective communications. A second challenge lies in coordination between the main and shadow process, in order to correctly terminate, carry out leaping, and perform failure recovery. Another challenge is that state consistency between mains and shadows needs to be maintained at all times. Last but not least, the side effects of divergence need to be addressed. The following sections discuss solutions to the above challenges by presenting the rsMPI implementation details.

7.1 FUNCTION WRAPPERS

As illustrated in Figure 25, rsMPI is positioned between the MPI runtime layer and the user application layer. Externally, rsMPI a library of function wrappers, one for each MPI prim-

itive. As a result, user applications can be linked to rsMPI without modification to use MPI functions. When the user application makes an MPI function call, rsMPI library intercepts this call to enforce the Leaping Shadows logic. Internally, rsMPI uses the original MPI procedures by interacting with the MPI profiling interface. For example, to hide the existence of shadows from the user application, rsMPI adds a wrapper for the MPI_Comm_size() function, which is to retrieve the number of processes (i.e., MPI ranks) in an MPI communicator. When the user application calls MPI_Comm_size(), rsMPI internally calls PMPI_Comm_size() using the MPI profiling interface to retrieve a *size*, counting both mains and shadows. Then rsMPI returns *size/2* to the above application, giving an illusion that the same number of processes are launched. This is illustrate in Figure 25(b).



(a) Position of the rsMPI library in software stack. (b) Example of MPI_Comm_size() function call.

Figure 25: rsMPI is inserted between the MPI and application layers to intercept MPI calls from the above application.

A static mapping between rsMPI ranks and application-visible MPI ranks is maintained so that each process can retrieve its identity. For example, if the user specifies N ranks

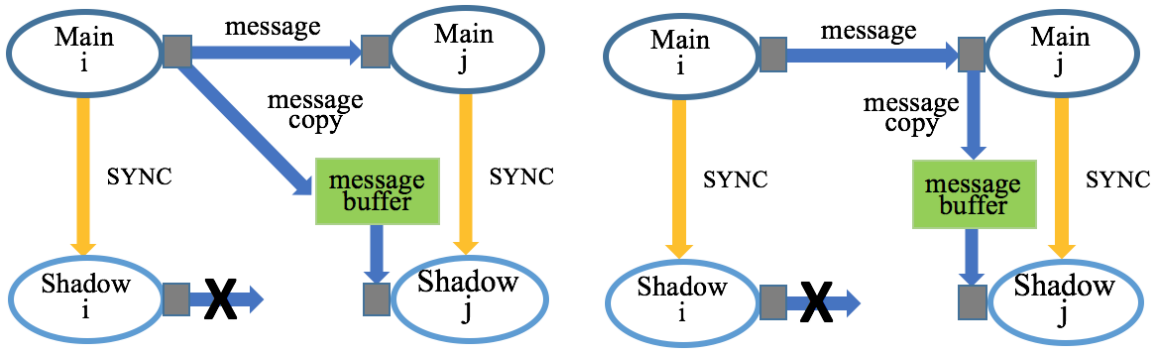
to launch, rsMPI will automatically translate it into $N + N$ ranks[2], with the first N ranks being the mains, and the next N ranks being their associated shadows in the corresponding order. By maintaining this main to shadow mapping, rsMPI guarantees that each process, whether main or shadow, gets its correct execution path. This logic is enforced by wrapping the MPI_Comm_rank() function. The wrappers for the MPI communication primitives will be discussed in the next section, where we introduce the consistency protocols.

7.2 MESSAGE PASSING AND CONSISTENCY

State consistency between mains and shadows is required both during normal execution and following a failure to leap-forward the shadows. Specifically, rsMPI needs to maintain sequential consistency so that each shadow sees the same message order and operation results as its main. Instead of having two parallel replica groups in which main communicates with main and shadow communicates with shadow [54], we choose to let the mains forward each message to the shadows. This allows us to speed up a single lagging shadow when a main fails. At the same time, the shadows are suppressed from sending out messages. As a consequence, two consistency protocols are explored, as depicted in Figure 26. In the sender-forwarding protocol, as shown in Figure 26(a), each main sender is responsible for forwarding each application message to the shadow of the receiver. In the receiver-forwarding protocol, as shown in Figure 26(b), the main receiver is responsible for forwarding each received message to its associated shadow. We assume that two copies of the same message are sent in an atomic manner.

For MPI point-to-point communication routines, we add wrappers to implement the above consistency protocols. For sending functions, such as MPI_Send() and MPI_Isend(), the sender-forwarding protocol requires the main to duplicate the message, while in the receiver-forwarding protocol the main just does the normal sending. Again, the shadow is suppressed from sending, in both protocols. For receiving functions, such as MPI_Recv() and MPI_Irecv(), the sender-forwarding protocol requires that both the main and the shadow do

[2]This ignores the ranks for coordinator processes to be discussed later.



(a) Sender-forwarding protocol.

(b) Receiver-forwarding protocol.

Figure 26: Consistency protocols for rsMPI.

one receiving from the main process at the sending side. In the receiver-forwarding protocol, MPI_Recv() is modified such that the main performs a receiving followed by a sending to forward the message, and the shadow receives from its own main. For MPI_Irecv(), it is slightly different that message forwarding is moved to MPI_Test() and MPI_Wait(), so that the message is assured to have arrived.

Collective communications, such as MPI_Bcast() and MPI_Allreduce(), are also supported in rsMPI. To prevent the mains from slowing down by the shadows due to the synchronous semantic of collective communication, rsMPI maintains communicators that only contain the mains, and calls the original MPI function among the mains to perform a collective communication. Since the original MPI collective function is used, rsMPI automatically benefits from the optimization that has been put into the collective communications, such as [135]. To keep the shadows consistent, results, if any, are forwarded from mains to shadows, so that shadows do not need to perform collective operation.

Assuming that only MPI operations can introduce non-determinism, the SYNC message shown in Figure 26 is used to enforce consistency when necessary. For example, under the sender-forwarding protocol MPI_ANY_SOURCE receiving may result in different message orders between a main and its shadow. To address this issue, we serialize the receiving of

MPI_ANY_SOURCE messages by having the main finish the receiving and then use a SYNC message to forward the message source to its shadow, which then performs a normal receiving from the specific source. Other operations, such as MPI_Wtime() and MPI_Probe(), require both protocols to use the SYNC messages. Similar to MPI_ANY_SOURCE receiving, they are dealt with by forwarding each result from a main to its associated shadow.

Table 9: Comparison between sender-forwarding protocol and receiver-forwarding protocol. N is the number of application message. D is the number of non-deterministic event. P is the number of process.

	Sender-forwarding	Receiver-forwarding
Number of application message	$2N$	$2N$
Number of SYNC message	D	$\leq D$
Shadow blocks main	Possible	Possible
Number of socket connection	P^2	P
Communication optimization	Hard	Easy

From the above analysis, we already see differences between the sender-forwarding protocol and the receiver-forwarding protocol. When deployed, they are expected to have further disparity on the cost and performance. A more detailed comparison is given in Table 9. If an MPI application sends N messages, both protocols double the number of application messages to be sent. Also, in both protocols, a slower shadow may block a faster main when message is forwarded from the main to the shadow. Therefore, flow control and buffer management need to be considered (to be discussed in following section). As mentioned above, the receiver-forwarding protocol does not need SYNC message in the case of MPI_ANY_SOURCE receiving, thus requiring fewer SYNC messages than the number of non-deterministic event, D. Furthermore, assuming an all-to-all connection of P processes, the sender-forwarding protocol implies P^2 socket connections between mains and shadows,

while the receiver-forwarding protocol only requires P connections. Lastly, since each shadow only communicates with its associated main under the receiver-forwarding protocol, it is easier to optimize communication via process to processor mapping. For example, a heuristic may be that each shadow is placed in the same rack as its main, but not on the same node. Given the advantages of the receiver-forwarding protocol, we focus on this protocol in the following discussion and performance evaluation.

7.3 COORDINATION BETWEEN MAIN AND SHADOW

To facilitate the correct execution of all processes, rsMPI adds a coordinator process to each shadowed set. Coordinators do not execute user application code, but just wait for rsMPI defined control messages. When a control message arrives, the coordinator carries out a minimal amount of coordination work accordingly. There are three types of control messages: termination, failure, and leaping. They corresponds to three actions:

- When a main process finishes, the coordinator in the shadowed set forces the associated shadow process to terminate immediately.
- When a main process fails, the coordinator speeds up the associated shadow by temporarily suspending the other collocated shadows, until the recovery is complete.
- When a main process initiates a failure-induced leaping, the coordinator triggers leaping at the associated shadow process.

To minimize resource usage, each coordinator is collocated with the shadows in the shadowed set. A coordinator performs minimal work, as its main task is to simply handle incoming control messages. As such, the impact of the coordinator on the execution rate of the collocated shadows is negligible. To separate control messages from data messages, rsMPI uses a dedicated MPI communicator for the control messages. This Control Communicator is created by the wrapper of the MPI_Init() function. In addition, to ensure fast response and minimize the number of messages, coordinators also use OS signals to communicate with their collocated shadows. This is illustrated in Figure 27, assuming a collocation ratio of 2.

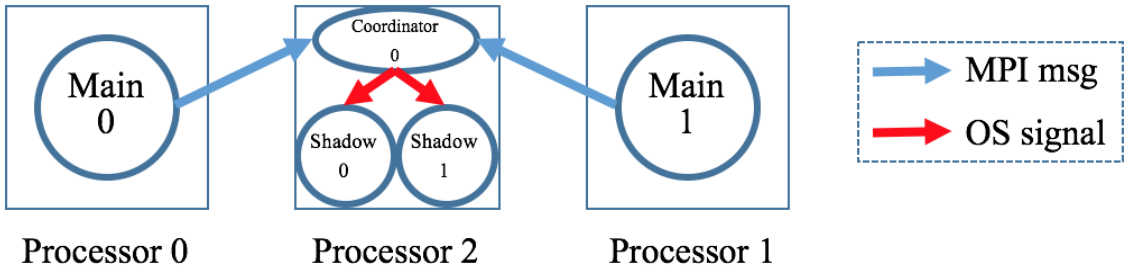


Figure 27: A coordinator is added to each shadowed set. In this example, collocation ratio is 2 and each shadowed set contains 2 mains and 2 shadows.

7.4 FLOW CONTROL

As discussed in Section 7.2, rsMPI requires that mains forward application messages to shadows. Since the shadows are scheduled to execute at a slower rate than the mains, an implication is that a shadow may block a main when a message is forwarded from the main to the shadow. As a consequence, this may slow down the mains to proceed at the same rate as the shadows, significantly hurting the performance. This has been confirmed with our previous implementation where MPI communication is used for the forwarded messages. In many MPI implementations, two protocols are used for the communication. The eager protocol, which is used to transfer small messages, pushes the entire message to the receiver side regardless of the receiver being ready or not. In the rendezvous protocol, which is used for large messages, the sender is blocked until the receiver posts a matching receive. In order to prevent the shadows from blocking mains, lots of efforts need to be spent on tuning the eager threshold and buffer sizes for each application, not to mention that the eager threshold cannot be increased infinitely[3]. Due to message forwarding, another issue is that the divergence mentioned in Chapter 5 will cause messages to accumulate at the shadow side, eventually leading to buffer overflow if no action is taken. With the previous implementation, we are not able to monitor or manipulate the buffers, which are managed

[3]In OpenMPI 1.10, the limit is 160 KB

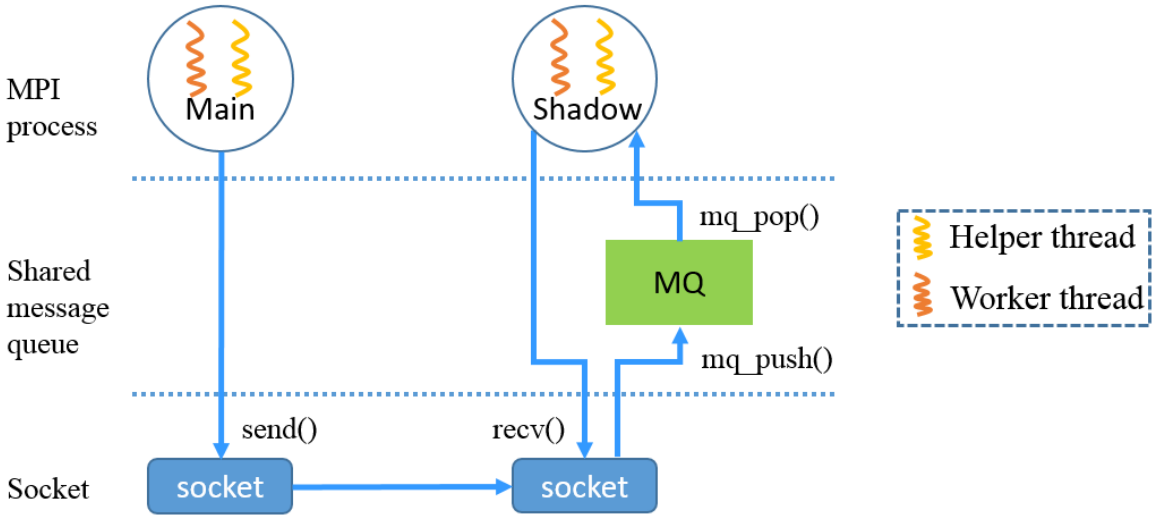


Figure 28: A layered architecture for flow control in rsMPI.

by MPI runtime.

To simultaneously addressing these two issues, rsMPI adopts a layered architecture, as shown in Figure 28, to establish communication channels between mains and shadows with flow control capability. Under the receiver-forwarding protocol, each pair of main and shadow only requires one such channel.

At the lowest level, a socket connection to transfer application messages is established between each main and its associated shadow. When Remote Direct Memory Access (RDMA) is available, such as on the InfiniBand interconnects, rsMPI uses the rsocket library, which provides a socket programming interface but internally uses RDMA for communication. Rather than copying data to the buffers of the operating system, RDMA enables the network adapter to transfer data directly from the main process to its shadow. The zero-copy networking feature of RDMA considerably reduces latency, thereby enabling fast transfer of data. When RDMA is not available, rsMPI reverts back to TCP connections.

At each shadow, a circular message queue, referred to as MQ, is implemented to store the messages that have been sent by main but not yet consumed by shadow. Similar to

Linux socket buffer or Unix mbuf, MQ consists of two layers. The control layer contains fixed-length elements, each of which stores the meta-data of a message. The data layer is the actual buffer where variable-length message contents are stored. The control layer and data layer are connected via pointers.

At the process level, a helper thread is added to each main and shadow process. The shadow helper thread is responsible for quickly responding to messages sent by the main on the socket connection, thus avoiding blocking the main. Every time there is a message, the shadow helper thread calls socket recv() to receive the message, and then stores it in its MQ. The main helper thread is mainly used to assist buffer-forced leaping. To avoid buffer overflow, a threshold is specified for MQ. If the threshold is reached, the shadow helper thread initiates a buffer-forced leaping, and notifies the corresponding main helper thread to participate in the leaping at the same time. After the buffer-forced leaping, obsolete messages in the MQ can be safely clearly. To minimize the impact of the helper threads on the performance of the compute threads, each helper thread is forced to relinquish CPU until a message arrives.

7.5 LEAPING

With the focus on dealing with crash failures, three types of leaping are implemented in rsMPI. To facilitate the following discussion, we summarize each leaping type along with the leap-provider and leap-recipient information in Table 10.

Checkpoint/restart requires each process to save its execution state, which can be used later to retrieve the state of the computation. Leaping, in all types, is similar to the checkpointing process, except that the state is directly transferred between a pair of main and shadow, thus requiring no additional storage space. To reduce the size of data involved in saving state, rsMPI uses a similar approach as application-level checkpointing [14, 102], and requires users to identify necessary data as process state using the following API:

```
void  leap_register_state(void *addr, int count, MPI_Datatype dt);
```

Table 10: Leaping types used in rsMPI.

Leaping type	Leap-provider	Leap-recipient
Failure-induced leaping	Main process	Shadow process
Buffer-forced leaping	Main process	Shadow process
Rejuvenation-induced leaping	Shadow process	Main process

For each data set to be registered, three arguments are needed: a pointer to the memory address of the data set, the number of data items in the data set, and the datatype. Application developer could use domain knowledge to identify only necessary state data, or use compiler techniques to automate this process [19]. Internally, rsMPI uses a linked list to keep track of all registered data. After each call of leap_register_state(), rsMPI adds a node to its internal linked list to record the three parameters. During leaping, the linked list is traversed to retrieve all registered data as the process state.

MPI communication channels are used in rsMPI to transfer process state. Although multiple data sets can be registered as a process' state, only a single message needs to be transferred, as MPI supports derived datatypes. To isolate state messages from application messages, rsMPI uses the Control Communicator to transfer process state. By using a coordinator to synchronize the leaping and relying on MPI messages to rapidly transfer state, the overhead of leaping is minimized.

To make sure a pair of main and shadow stay consistent after a leaping, not only user-defined states should be transferred, but also lower level states, such as program counter and message buffers, need to be correctly updated. Specifically, the leap-recipient needs to satisfy two requirements: 1) Discard all obsolete messages, if any, after the leaping; 2) Resume execution at the same point as the leap-provider. We discuss our solutions below, under the assumption that the application's main body consists of a loop of iterations, which is true in most HPC applications.

In failure-induced and buffer-forced leaping, shadow is the leap-recipient. To correctly

discard all obsolete messages at a shadow without throwing away useful ones, rsMPI requires every main and shadow maintain a counter for messages consumed. During leaping, the counter value at the main is transferred to the shadow, so that the latter knows how many messages to discard. Then the shadow can easily remove the obsolete messages because messages stored in MQ at the shadow are in the same order as those consumed by the main. In rejuvenation-induced leaping, the main is the leap-recipient, so there is no obsolete message to discard.

To resume execution from the same point, we restrict leaping to always occur at specific points, and use an internal counter to make sure that both the leap-recipient and leap-provider start leaping from the same point. For example, when a main initiates a failure-induced leaping, the coordinator will trigger a rsMPI-defined signal handler at the associated shadow. The signal handler does not carry out leaping, but sets a flag for leaping and receives from its main a counter value that indicates the leaping point. Only when both the flag is set and counter value matches will the shadow start leaping. In this way, it is guaranteed that after leaping the leap-recipient and leap-provider will resume execution from the same point. To balance the trade-off between implementation overhead and flexibility, we choose MPI receive operations as the only possible leaping points.

7.6 EVALUATION

We deployed rsMPI on a cluster of 30 nodes (600 cores) for testing and benchmarking. Each node consists of a 2-way SMPs with Intel Haswell E5-2660 v3 processors of 10 cores per socket (20 cores per node). Each node is configured with 128 GB of local memory. Nodes are connected via 56 GB/s FDR InfiniBand. To maximize the computing capacity, we used up to 20 cores per node.

Benchmarks from the Sandia National Lab Mantevo Project and NAS Parallel Benchmarks (NPB) are used, and we evaluated rsMPI with various problem sizes and number of processes. CoMD is a proxy for molecular dynamics application. MiniAero is an explicit unstructured finite volume code that solves the Navier-Stokes equations. Both

MiniFE and HPCCG are unstructured implicit finite element codes, but HPCCG uses MPI_ANY_SOURCE receive operations and can demonstrate rsMPI's capability of handling non-deterministic events. LU, EP, and FT from NPB represent lower-upper Gauss-Seidel solver, embarrassingly parallel, and fast Fourier Transform, respectively. In addition, we choose the LULESH benchmark developed at the Lawrence Livermore National Lab, since it is a widely studied proxy application in DOE co-design efforts for exascale [75]. These applications cover key simulation workloads and represent both different communication patterns and computation-to-communication ratios.

We also implemented checkpoint/restart to compare with rsMPI in the presence of failures. To be optimistic, we chose double in-memory checkpointing that is much more scalable than disk-based checkpointing [150]. Same as leaping in rsMPI, our application-level checkpointing provides an API for process state registration. This API requires the same parameters, but internally, it allocates extra memory in order to store 2 checkpoints, one for the local process and one for a remote "buddy" process. Another provided API is checkpoint(), which inserts a checkpoint in the application code. For fairness, MPI messages are used to transfer state between buddies. For both rsMPI and checkpoint/restart, we assume a 60 seconds rebooting time after a failure. All figures in this section show the average of 5 runs with the standard deviation.

7.6.1 Measurement of Runtime Overhead

While the hardware overhead for rsMPI is straightforward (e.g., collocation ratio of 2 results in the need for 50% more hardware), the runtime overhead due to the enforced consistency and coordination depends on applications. To measure this overhead we ran each benchmark application linked to rsMPI and compared the execution time with the baseline, in which each application runs with unmodified OpenMPI.

Figure 29 shows the comparison of the execution time for the 8 applications in the absence of faults. All the experiments are conducted with 256 application-visible processes, except that LULESH requires the number of processes to be a cube of an integer. When the baseline uses 256 MPI ranks, rsMPI uses 256 mains together with 256 shadows. The

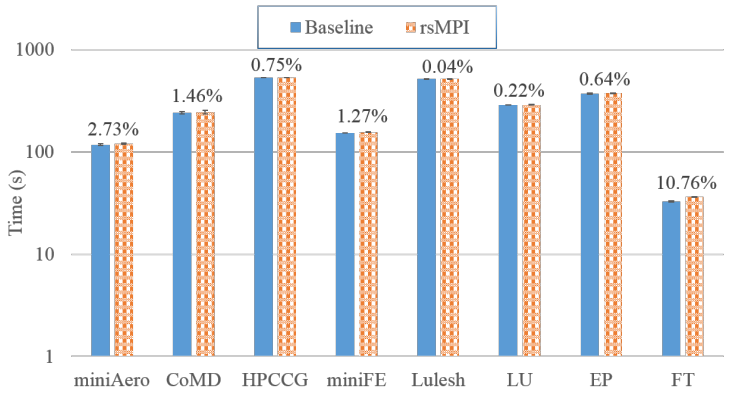
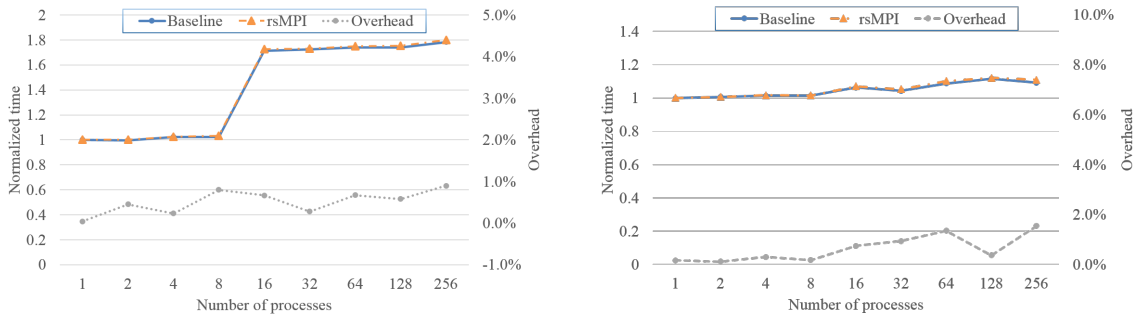


Figure 29: Comparison of execution time between baseline and rsMPI. 256 application-visible processes, except 216 processes for LULESH. Collocation ratio is 2 for rsMPI.

baseline execution time varies from seconds to 15 minutes, so we plotted the time in log-scale. From the figure we can see that rsMPI has comparable execution time to the baseline for all applications except FT. The reason for the exception is that FT uses a lot of broadcast, reduce, and all-to-all communication, and thus is heavily communication-intensive. This is verified by adding fake computation to the application and we can see an immediate drop of the overhead to negligible level. We argue that all-to-all communication applications like FT are not scalable, and as a result, they are not suitable for massively parallel HPC. For all others, the overhead varies from 0.04% (LULESH) to 2.73% (miniAero). Even for HPCCG, which uses MPI_ANY_SOURCE, the overhead is only 0.75%, thanks to the flow control mechanism deployed in rsMPI. Therefore, we conclude that rsMPI's runtime overheads are modest for applications that exhibit a fair communication-to-computation ratio.

7.6.2 Scalability

In addition to measuring the runtime overhead at a fixed process count, we also assessed the applications' weak scalability, which measures how the execution time varies with the number of processes for a fixed problem size per process. Among the eight applications, HPCCG, CoMD, and miniAero allow us to configure the problem size for weak scaling test.



(a) HPCCG weak scalability

(b) CoMD weak scalability

Figure 30: Weak scalability measurement with number of processes from 1 to 256. Collocation ratio is 2 for rsMPI. Time is normalized to that of the baseline with 1 process.

The results for miniAero are similar to those of CoMD, so we only show the results for HPCCG and CoMD in Figure 30.

Comparing between Figure 30(a) and Figure 30(b), it is obvious that HPCCG and CoMD have different weak scaling characteristics. While the execution time for CoMD increases by 9.3% from 1 process to 256 processes, the execution time is almost doubled for HPCCG (1.78X). However, further analysis shows that the execution time increases by 67.4% for HPCCG from 8 to 16 processes. We suspect that the results are not only affected by the scalability of the application, but also impacted by other factors, such as cache and memory contention on the same node, and network interference from other jobs running on the cluster. Note that each node in the cluster has 20 cores and we always use all the cores of a node before adding another node. Therefore, it is very likely that the node level contention leads to the substantial increase in execution time for HPCCG. The results from 16 to 256 processes show that both HPCCG and CoMD are weak scaling applications.

Similar to the results of the previous section, the runtime overhead for rsMPI is modest. The maximum overhead observed is 1.6% when running CoMD with 256 processes. Excluding this case, the overhead is always below 1.0%. To predict the overhead at exascale, we applied curve fitting to derive the correlation between runtime overhead and the number of

processes. At 2^{20} processes, it is projected that the runtime overhead is 9.4% for CoMD and 4.5% for HPCCG.

7.6.3 Performance under Failures

As one main goal of this work is to achieve fault tolerance, an integrated fault injector is required to evaluate the effectiveness and efficiency of rsMPI to tolerate failures during execution. To produce failures in a manner similar to naturally occurring process failures, the failure injector is designed to be distributed and co-exist with all rsMPI processes. Failure is injected by sending a specific signal to a randomly picked target process.

We assume that the underlying hardware platform has a Reliability, Availability and Serviceability (RAS) system that provides failure detection. In our test system, we emulate the RAS functionality by associating a signal handler with every process. The signal handler catches failure signals sent from the failure injector, and uses a rsMPI defined failure message via a dedicated communicator to notify all other processes of the failure[4]. To detect failure of another process, rsMPI receiving operation checks for failure messages before performing the actual receiving. Similar to ULFM [16], a process in rsMPI can only detect failure when it posts an MPI receive operation.

The first step was to test the effectiveness of leaping. Figure 31 shows the execution time of HPCCG with a single failure injected at a specific time, measured as a proportion of the total execution of the application, at an increment of 10%. The execution time is normalized to that of the failure-free baseline. The blue solid line and red dashed line represent rsMPI with collocation ratio of 2 and 4, respectively. For simplicity, they are referred to as rsMPI_2 and rsMPI_4 in the following text.

As shown in Figure 31, rsMPI's execution time increases with the failure occurrence time, regardless of the collocation ratio. The reason is that recovery time in rsMPI is proportional to the amount of divergence between mains and shadows, which grows with the execution. Another factor that determines the divergence is the shadow's execution rate. The slower the shadows execute, the faster the divergence grows. As a result, rsMPI_2 can recover faster

[4]A similar idea has been tested in [52]. By using GDB to intercept OS signals generated after a fault, a signal handler allows the failing process to continue work before crashing.

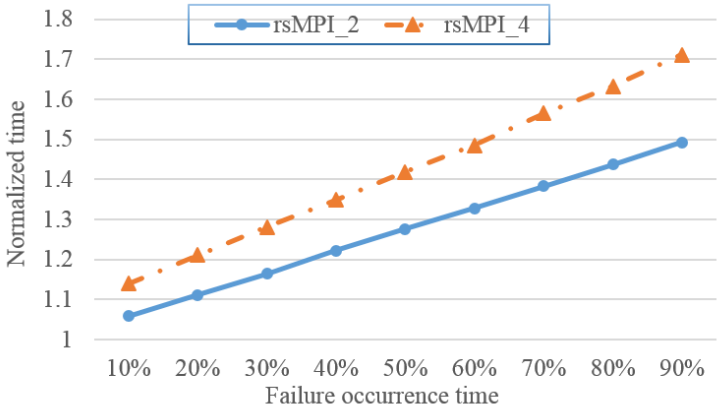


Figure 31: Execution time of HPCCG under rsMPI with a single failure injected at various time, normalized to that of the failure-free baseline.

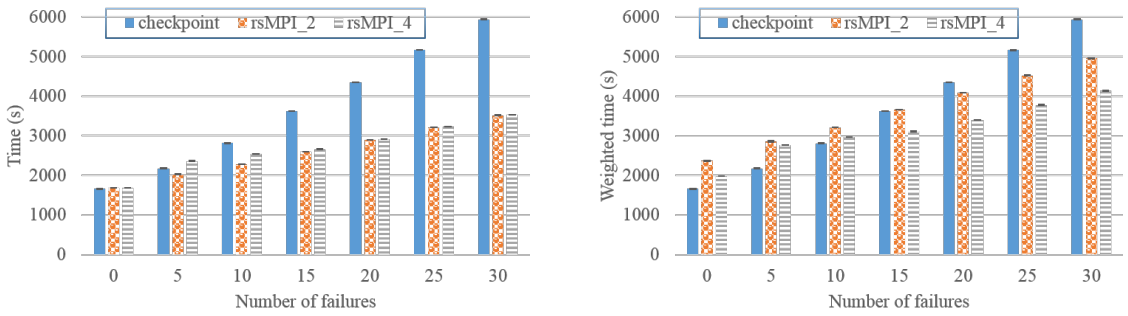
than rsMPI_4, and therefore achieves better execution time.

The results in Figure 31 suggests that rsMPI is better suited to environments where failures are frequent. This stems from the fact that, due to leaping, the divergence between mains and shadows is eliminated after every failure recovery. As the number of failure increases, the interval between failures decreases, thereby reducing the recovery time per failure. To demonstrate the above analysis, we compare rsMPI with checkpoint/restart under various failure rates. To run the same number of application-visible processes, rsMPI needs more nodes than checkpoint/restart to host the shadow processes. For fairness, we take into account the extra hardware cost for rsMPI by defining the weighted execution time:

$$T_{weight} = T_e \times S_p,$$

where T_e is the wall-clock execution time and S_p is the projected speedup. For example, we measured that the speedup of HPCCG from 128 processes to 256 processes is 1.88, and rsMPI_2 needs 1.5 times more nodes than checkpoint/restart, so the projected speedup is $1.5 \times \frac{1.88}{2} = 1.41$. Similarly, the projected speedup for rsMPI_4 is $1.25 \times \frac{1.88}{2} = 1.17$.

In this analysis, we set the checkpointing interval to $0.1T$, where T is the total execution time. To both checkpoint/restart and rsMPI, we randomly inject over T a number of faults,



(a) Wall-clock execution time

(b) Weighted execution time

Figure 32: Comparison between checkpoint/restart and rsMPI with various number of failures injected to HPCCG. 256 application-visible processes, 10% checkpointing interval.

K, ranging from 1 to 30. This fault rate corresponds to a processor's MTBF of NT/K, where N is the number of processors. That is, the processor's MTBF is proportional to the total execution time and the number of processors. For example, when using 256 processors and executing for 1700 seconds, injecting 10 faults corresponds to a processor's MTBF of 12 hours. However, when using a system of $64,000$ processors and executing over 4 hours, injecting 10 faults corresponds to a processor's MTBF of 3 years.

Figure 32 compares checkpoint/restart and rsMPI, based on both wall-clock and weighted execution time. Ignoring the hardware overhead, Figure 32(a) shows that, in the failure-free case, checkpoint/restart slightly outperforms rsMPI. As the number of failures increases, however, rsMPI achieves significantly higher performance than checkpoint/restart. For example, when the number of failures is 20, rsMPI_2 saves 33.4% in time compared to checkpoint/restart. The saving rises up to 40.8%, when the number of failures is increased to 30. Compared to checkpoint/restart, rsMPI_4 reduces the execution time by 33.1% and 40.5%, when the number of failures are 20 and 30, respectively.

Careful analysis of Figure 32(a) reveals that, as the number of failures increases, checkpoint/restart and rsMPI exhibit different performance degradation. As expected, the execution time for checkpoint/restart increases proportionally with the number of failures. For

rsMPI, however, the increase is sub-linear. This is due to fact that as more failures occur, the interval between failures is reduced, and as a result, the recovery time per failure is also reduced. Eventually, the constant rebooting time dominates the recovery overhead. This is the reason why the execution time of rsMPI_2 and rsMPI_4 tends to converge as the number of failure increases.

Incorporating hardware overhead, Figure 32(b) compares the weighted execution time between checkpoint/restart and rsMPI. As expected, checkpoint/restart is better when the number of failures is small (e.g., 5 failures). When the number of failures increases, however, checkpoint/restart loses its advantage quickly. At 30 failures, for example, rsMPI_2 and rsMPI_4 are 16.5% and 30.4% more efficient than checkpoint/restart, respectively. Note that, when comparing rsMPI_2 and rsMPI_4, the former shows higher performance with respect to wall-clock execution time, while the latter is better with respect to weighted execution time.

7.7 SUMMARY

This chapter presents the details of a proof-of-concept implementation of Leaping Shadows in MPI, referred to as rsMPI. To ensure correct execution while maximizing performance, rsMPI consists of five building blocks: wrappers for MPI functions, consistency protocols to maintain sequential consistency and resolve non-determinism, main and shadow coordination mechanism, flow control algorithm with customized data structure, and an application-level leaping protocol. Based on shadow collocation, this implementation associates one shadow process with each main process to tolerate crash failures. By capturing the leaping (include failure-induced leaping, buffer-forced leaping, and rejuvenation-induced leaping) and rejuvenation techniques, rsMPI enables MPI applications to tolerate multiple failures with minimized hardware and power costs.

With rsMPI, extensive experiments have been done to evaluate the performance of the Leaping Shadows fault tolerance model in real systems. To cover key HPC simulation workloads, we use benchmark applications that represent both different communication patterns

and different computation-to-communication ratios. By comparing with the original Open-MPI library, we demonstrate that rsMPI has negligible runtime overheads when there is no failure. Also, we implement and compare to in-memory checkpoint/restart under various failure rates. Experiment results show that rsMPI has a high potential in outperforming in-memory checkpoint/restart in both execution time and resource utilization, especially in failure-prone environments.

8.0 CONCLUSION

As our reliance on IT continues to increase, the complexity and urgency of the problems our society will face in the future will increase much faster than are our abilities to understand and deal with them. Future IT systems are likely to exhibit a level of interconnected complexity that makes it prone to failure and exceptional behaviors. The high risk of relying on IT systems that are failure-prone calls for new approaches to enhance their performance and resiliency to failure.

HPC and Cloud are two ecosystems that are designed for different applications and with disparate design principles. However, Big data technologies, such as Hadoop, clustered storage, and data visualization, are now merging with traditional HPC technologies. On the one hand, an increasing portion of HPC workloads is becoming data intensive. On the other hand, Big data applications are requiring more and more computing power. As the boundaries between Cloud and HPC continue to blur, it is clear that there is an urgent demand for a systematic computational model that adapts to the computing platform and accommodates the underlying workloads.

This thesis presents Leaping Shadows as a novel fault-tolerant computational model that unifies HPC and Big Data analytics and scales to future extreme-scale computing systems. The flexibility in the model allows it to embrace different execution strategies in accordance with the underlying workloads, whether it is compute-intensive or data-intensive. Leaping Shadows takes advantage of the unique design in the Shadow Replication model that original main processes are associated with "lazy" shadows. The differential and dynamic execution rates control enables Leaping Shadows to achieve fault tolerance with power awareness, as well as adaptivity to trade-offs among performance, resilience, and energy costs. Furthermore, by incorporating creative optimization techniques, Leaping Shadows is able to

maintain a consistent level of resilience across high rate and diverse types of failures, with improved performance and reduced resource requirements.

This thesis systematically studies the viability of Leaping Shadows to enhance system resilience in emerging extreme-scale, failure-prone computing environments. As a first step, customized execution dynamics are designed to deal with different types of failures. Then, analytical models and optimization frameworks are built to derive the optimal process execution rates, while at the same time multiple mechanisms are explored to effectively achieve the desired execution rates. To further verify Leaping Shadows and to validate the analytical models, a prototype implementation is provided in the context of HPC. Empirical evaluation with various benchmark applications confirms that Leaping Shadows is able to outperform state-of-the art fault tolerance approaches.

The study of the Leaping Shadows model in this thesis is not meant to be complete. The flexibility and diversity in the model point to many future directions. In current design of Leaping Shadows, each main is associated with the same number of shadows. This is ignorant of the variance in the underlying hardware reliability and above application criticality. Previous studies have shown that failure rates both vary across systems and vary from node to node within the same system [124, 41]. According to [41], 19% of the nodes account for 92% of the machine check errors on Blue Waters. At the same time, within a system processes may have different criticality. For example, in the master-slave execution model the master process is a single point of failure, making the failure of the master process much more severe than that of a slave process. In fact, *heterogeneous shadowing* techniques can be explored to dynamically harness all available resources to achieve the highest level of QoS. Within the resource limitation, more shadows would be allocated for more critical mains or mains that are more likely to fail.

Failure-induced leaping has proven to be critical in reducing the divergence between a main and its shadow, thus reducing the recovery time for subsequent failures. Consequently, the time to recover from a failure increases with failure intervals. Based on this observation, a proactive approach is to "force" leaping when the divergence between a main and its shadow exceeds a specified threshold. This is analogous to checkpoint/restart in that checkpoints are periodically taken to minimize the cost of lost work due to a failure. Thus, it is worth studying

this approach to determine what behavior triggers forced leaping in order to optimize the average recovery time.

Another future direction is topology-aware process mapping [140]. Process mapping is of vital importance in Leaping Shadows, since it not only determines the failure isolation degree, but also impacts communication performance. For the main and shadow(s) of the same task, we would like to place them far away so that they are unlikely to be victims of a single failure. On the other hand, placing mains and shadows close to each other tends to minimize message forwarding cost, especially under the receiver-forwarding protocol. Therefore, a process mapping algorithm needs to be developed to balance the trade-offs, while considering the interconnect topology.

In the extreme cases where hardware resources are quite limited, there may be no redundant hardware to support the execution of the shadows. If this is the case, one might still apply Leaping Shadows with every main collocated with a shadow, which is associated with a different main. Furthermore, to prevent shadows from taking too much resources and extensively slowing down the mains, shadows may only be allowed to "steal" CPU cycles when mains are blocked. It is expected that Leaping Shadows with such collocation should be able to achieve fault tolerance with comparable performance under the given limitation on resources. However, it remains a question whether there is an efficient mechanism to precisely control the CPU sharing while ensuring performance isolation. Also, process mapping is an important problem to study.

Lastly, the idea of approximate computing can be applied to shadows. Instead of having shadows as exact replicas of the mains, one can assign a reduced version of the computation to the shadows or let them process a portion of the input data. Many workloads today, such as HPC simulation and large-scale machine learning, often have the flexibility in tuning the fidelity of their results, such as the granularity of a simulation or the precision of convergence. Energy and performance gains may be achieved, when relaxing the precision constraints in the case of a failure.

BIBLIOGRAPHY

[1] Top500 list. https://www.top500.org.

[2] What is hdfs (hadoop distributed file system)? https://www.ibm.com/analytics/us/en/technology/hadoop/hdfs/.

[3] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS 04*, St. Malo, France.

[4] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu. *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*. 2011.

[5] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010.

[6] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing*, 1999.

[7] L. Alvisi and K. Marzullo. Trade-offs in implementing causal message logging protocols. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 58–67. ACM, 1996.

[8] E. Anderson, L.-l. Lam, C. Eschinger, S. Cournoyer, J. Correia, L. Wurster, R. Contu, F. Biscotti, V. Liu, T. Pang, et al. Forecast overview: Public cloud services, worldwide, 2011-2016, 4q12 update. *Gartner Inc., February*, 2013.

[9] S. Ashby, B. Pete, C. Jackie, and C. Phil. The opportunities and challenges of exascale computing, 2010.

[10] C. Balding. A question of integrity: To md5 or not to md5. *cloudsecurity. org (June 2008)*, 2009.

[11] R. Baldoni, S. Cimmino, and C. Marchetti. Total order communications: A practical analysis. In *EDCC*, volume 3463, pages 38–54. Springer, 2005.

[12] L. A. Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[13] J. F. Bartlett. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 22–29, New York, NY, USA, 1981. ACM.

[14] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[15] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[16] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 2013.

[17] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra. Failure detection and propagation in hpc systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 312–322. IEEE, 2016.

[18] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[19] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina. Compiler-enhanced incremental checkpointing for openmp applications. In *IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, 2009.

[20] J. A. Butts and G. S. Sohi. A static power model for architects. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 191–201. IEEE, 2000.

[21] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp. Towards a more complete understanding of sdc propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 131–142. ACM, 2017.

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

[23] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[24] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 86–97. ACM, 2003.

[25] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Rapport de recherche RR-7951, INRIA, May 2012.

[26] S. Chakravorty, C. Mendes, and L. Kalé. Proactive fault tolerance in mpi applications via task migration. *High Performance Computing-HiPC 2006*, pages 485–496, 2006.

[27] 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.

[28] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[29] H. chang Nam, J. Kim, S. Lee, and S. Lee. Probabilistic checkpointing. In *In Proceedings of Intl. Symposium on Fault-Tolerant Computing*, pages 153–160, 1997.

[30] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[31] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127. ACM, 2006.

[32] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[33] X. Cui, B. Mills, T. Znati, and R. Melhem. Shadow replication: An energy-aware, fault-tolerant computational model for green cloud computing. *Energies*, 7(8):5151–5176, 2014.

[34] 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.

[35] X. Cui, T. Znati, and R. Melhem. Adaptive and power-aware resilience for extreme-scale computing. In *16th IEEE International Conference on Scalable Computing and Communications*, July 18-21 2016.

[36] J. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006.

[37] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 31–40. ACM, 2011.

[38] G. Deconinck, J. Vounckx, R. Lauwereins, and J. A. Peperstraete. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. *International Journal of Modeling and Simulation*, 18:262–265, 1993.

[39] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[40] S. Di and F. Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823, 2016.

[41] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621. IEEE, 2014.

[42] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[43] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *ICDCS '12*, Washington, DC, US.

[44] E. Elnozahy and J. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *DSC*, 1(2):97 – 108, april-june 2004.

[45] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *TC*, 41:526–531, 1992.

[46] E. N. Elnozahy. How safe is probabilistic checkpointing? In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 358–363. IEEE, 1998.

[47] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[48] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems-Volume 4*, pages 8–8. USENIX Association, 2003.

[49] C. Engelmann and S. Böhm. Redundant execution of hpc applications with mr-mpi. In *PDCN*, pages 15–17, 2011.

[50] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 252–257. IEEE, 2009.

[51] S. Eyerman and L. Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Trans. Archit. Code Optim.*, 8(1):1:1–1:24, Feb. 2011.

[52] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu. Letgo: A lightweight continuous framework for hpc applications under failures. 2017.

[53] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. of the* 17^{th} *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 37–48, New York, NY, USA, 2012. ACM.

[54] 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*, SC '11, pages 44:1–44:12, 2011.

[55] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. SC, Los Alamitos, CA, USA, 2012.

[56] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 260–271. ACM, 2001.

[57] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *Journal of Parallel and Distributed Computing*, 68(9):1175–1185, 2008.

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

[59] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[60] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron. CPU miser: A performance-directed, run-time system for power-aware clusters. In *International Conference on Parallel Processing (ICPP)*, pages 18–18. IEEE, 2007.

[61] A. Geist. What is the monster in the closet? In *Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*, volume 2, 2011.

[62] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *IPDPS*, pages 989–1000, May 2011.

[63] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Drpm: dynamic speed control for power management in server class disks. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169–179. IEEE, 2003.

[64] P. Hargrove and J. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494, 2006.

[65] J.-M. Helary, A. Mostefaoui, R. H. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *RDS*, 1997.

[66] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra. Practical scalable consensus for pseudo-synchronous distributed systems. In *SC*, 2015.

[67] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[68] K.-H. Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[69] S. Huang and W. Feng. Energy-efficient cluster computing via accurate workload characterization. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 68–75. IEEE Computer Society, 2009.

[70] Y. Huang and Y.-M. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 459–463. IEEE, 1995.

[71] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[72] D. B. Johnson and W. Zwaenepoel. *Sender-based message logging*. Rice University, Department of Computer Science, 1987.

[73] T.-Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 454–461. IEEE, 1991.

[74] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000.

[75] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[76] B. Keller. Opportunities for fine-grained adaptive voltage scaling to improve system-level energy efficiency. Master's thesis, EECS Department, University of California, Berkeley, Dec 2015.

[77] W. Kim, M. S. Gupta, G. Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 123–134, Feb 2008.

[78] I. Koren and C. M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[79] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[80] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[81] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[82] J.-C. Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[83] B. Lee, T. Park, H. Y. Yeom, and Y. Cho. An efficient algorithm for causal message logging. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 19–25. IEEE, 1998.

[84] A. Lefray, T. Ropars, and A. Schiper. Replication for send-deterministic MPI HPC applications. FTXS '13, pages 33–40, New York, NY, USA, 2013. ACM.

[85] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, Aug. 1994.

[86] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 425–434. IEEE, 2006.

[87] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006.

[88] J. Lin and C. Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[89] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice and Experience*, 40(11):943–964, 2010.

[90] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, G. Grider, R. Haring, J. Hittinger, A. Hoisie, D. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, and R. Stevens. *The Top Ten Exascale System Research Challenges*. 2014.

[91] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5(2):172–184, 1988.

[92] E. Meneses, G. Bronevetsky, and L. V. Kale. Evaluation of simple causal message logging for large-scale fault tolerant hpc systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1533–1540. IEEE, 2011.

[93] E. Meneses, O. Sarood, and L. V. Kale. Assessing energy efficiency of fault tolerance protocols for HPC systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 35–42. IEEE, 2012.

[94] E. Meneses, O. Sarood, and L. V. Kalé. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Computing*, 40(9):536–547, 2014.

[95] B. Mills. *Power-Aware Resilience for Exascale Computing*. PhD thesis, University of Pittsburgh, 2014.

[96] B. Mills, T. Znati, and R. Melhem. Shadow computing: An energy-aware fault tolerant computing model. In *ICNC*, 2014.

[97] B. Mills, T. Znati, R. Melhem, K. B. Ferreira, and R. E. Grant. Energy consumption of resilience mechanisms in large scale systems. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 528–535. IEEE, 2014.

[98] B. Mills, T. Znati, R. Melhem, R. E. Grant, and K. B. Ferreira. Energy consumption of resilience mechanisms in large scale systems. In *Parallel, Distributed and Network-Based Processing (PDP), 22st Euromicro International Conference*, Feb 2014.

[99] S. Mittal. Power management techniques for data centers: A survey. *arXiv preprint arXiv:1404.6681*, 2014.

[100] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, pages 1–11, 2010.

[101] H. Nam, J. Kim, S. J. Hong, and S. Lee. Probabilistic checkpointing. *IEICE TRANSACTIONS on Information and Systems*, 85(7):1093–1104, 2002.

[102] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. Acr: Automatic checkpoint/restart for soft and hard error protection. SC, pages 7:1–7:12, New York, NY, USA, 2013. ACM.

[103] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 30–46, Sept 2007.

[104] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. Dma-aware memory energy management. In *HPCA*, volume 6, pages 133–144, 2006.

[105] D. A. Patterson, G. Gibson, and R. H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[106] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 89–102. ACM, 2001.

[107] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 15–26. ACM, 2006.

[108] J. Plank and K. Li. Faster checkpointing with n+1 parity. In *Fault-Tolerant Computing*, pages 288–297, June 1994.

[109] X. Qi, D. Zhu, and H. Aydin. Global reliability-aware power management for multiprocessor real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 183–192, Aug 2010.

[110] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 48–59. ACM, 2008.

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

[112] S. K. Reinhardt and S. S. Mukherjee. *Transient fault detection via simultaneous multithreading*, volume 28. ACM, 2000.

[113] R. Riesen, K. Ferreira, J. R. Stearley, R. Oldfield, J. H. L. III, K. T. Pedretti, and R. Brightwell. Redundant computing for exascale systems, December 2010.

[114] B. Rountree, D. K. Lownenthal, B. R. De Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469. ACM, 2009.

[115] C. Rusu, R. Melhem, and D. Mossé. Maximizing rewards for real-time applications with energy constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(4):537–559, 2003.

[116] T. Saito, K. Sato, H. Sato, and S. Matsuoka. Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 41–48. ACM, 2013.

[117] A. Sangroya, D. Serrano, and S. Bouchenak. Benchmarking dependability of mapreduce systems. In *Reliable Distributed Systems (SRDS), IEEE 31st Symp. on*, pages 21–30, 2012.

[118] O. Sarood, E. Meneses, and L. V. Kale. A 'cool' way of improving the reliability of HPC machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 58:1–58:12, New York, NY, USA, 2013. ACM.

[119] J. Scaramella. Worldwide server, power and cooling, and management and administration spending 20142018 forecast. *Market analysis, IDC Inc*, 2014.

[120] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[121] F. B. Schneider. What good are models and what models are good. *Distributed systems*, 2:17–26, 1993.

[122] B. Schroeder and G. A. Gibson. The computer failure data repository.

[123] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[124] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[125] L. M. Silva and J. G. Silva. Using two-level stable storage for efficient checkpointing. *IEE Proceedings-Software*, 145(6):198–202, 1998.

[126] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238. ACM, 1989.

[127] D. B. Skillicorn, J. Hill, and W. F. McColl. Questions and answers about bsp. *Scientific Programming*, 6(3):249–274, 1997.

[128] S. W. Smith and D. B. Johnson. Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback. In *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on*, pages 66–75. IEEE, 1996.

[129] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, page 1094342014522573, 2014.

[130] P. Sousa, N. Neves, and P. Verissimo. Resilient state machine replication. In *Dependable Computing. Proc. 11th Pacific Rim Int. Symp. on*, pages 305–309, 2005.

[131] J. Stearley, K. Ferreira, D. Robinson, J. Laros, K. Pedretti, D. Arnold, P. Bridges, and R. Riesen. Does partial replication pay off? In *DSN-W*, pages 1–6, June 2012.

[132] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[133] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *In 1984 International Conference on Parallel Processing*. Citeseer, 1984.

[134] A. Tchana, L. Broto, and D. Hagimont. Approaches to cloud computing fault tolerance. In *Comp., Infor. & Tele. Sys., Int. Conf. on*, pages 1–6, 2012.

[135] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[136] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *2012 Second International Conference on Cloud and Green Computing (CGC)*, pages 260–267. IEEE, 2012.

[137] W.-T. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen. Service replication strategies with mapreduce in clouds. In *Autonomous Decentralized Systems, 10th Int. Symp. on*, pages 381–388, 2011.

[138] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, 2005.

[139] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, et al. Scaling the power wall: A path to exascale. SC '14, Piscataway, NJ, USA.

[140] S. von Alfthan, I. Honkonen, and M. Palmroth. Topology aware process mapping. In *PARA*, pages 297–308. Springer, 2012.

[141] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *ISCA '14*, Piscataway, NJ, USA. IEEE Press.

[142] P. Wang, D. J. Dean, and X. Gu. Understanding real world data corruptions in cloud systems. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 116–125. IEEE, 2015.

[143] Q. Wang, Y. Kanemasa, J. Li, C. A. Lai, M. Matsubara, and C. Pu. Impact of dvfs on n-tier application performance. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, page 5. ACM, 2013.

[144] T. Woodall, G. Shipman, G. Bosilca, R. Graham, and A. Maccabe. High performance rdma protocols in hpc. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 76–85, 2006.

[145] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. In *Proceedings of the 41st annual Design Automation Conference*, pages 868–873. ACM, 2004.

[146] B. Zhao, H. Aydin, and D. Zhu. Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 633–639. IEEE, 2008.

[147] B. Zhao, H. Aydin, and D. Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Proceedings of the 48th Design Automation Conference*, pages 381–386. ACM, 2011.

[148] W. Zhao, P. Melliar-Smith, and L. E. Moser. Fault tolerance middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 67–74. IEEE, 2010.

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

[150] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing*, pages 93–103, 2004.

[151] Q. Zheng. Improving mapreduce fault tolerance in the cloud. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6, April 2010.