ABSTRACT

REZAEI, ARASH. Fault Resilience for Next Generation HPC Systems. (Under the direction of Frank Mueller.)

Fault resilience techniques enable application completion with correct results despite the existence of multiple sources of faults and their possible occurrence in the system. HPC systems are particularly interesting to study as multiple challenges arise from the size (millions of cores) and the programming model (tightly coupled). Resilience is considered a major roadblock in next generation supercomputers due to large systems size and a shorter Mean Time Between Failure (MTBF). A number of state-of-art resilience methods exist: Checkpoint/Restart (CR), process migration, redundant computing, and Application Based Fault Tolerance (ABFT). The aim of this work is to overcome shortcomings that are present in the current software solutions for resilience in HPC.

In CR, the core idea is to save the state of processes on storage periodically and recover from failures by reloading a previous checkpoint. Process migration aims at moving away processes from nodes that are predicted to fail soon. But there are shortcomings in CR and process migration: No CR approach exist for the Intel Xeon Phi coprocessor, a new architecture that allows for fast computation through offloading. We provide Snapify, a set of extensions to Intel software stack that provides three novel features for Xeon Phi applications: CR, process swapping, and process migration.

Redundant computing aims at providing resilience through launching multiple equivalent processes to perform the same task (replicas). The intermediate results can be compared among replicas and soft errors can be detected/corrected. However, current redundancy approaches do not replenish the failed replicas. We present a redundant execution environment that quickly repairs failures via DIvergent NOde cloning (DINO). DINO contributes a novel node cloning service integrated into the MPI runtime system that solves the problem of consolidating divergent states among replicas on-the-fly.

Finally, as the HPC applications become more complex due to a deep memory hierarchy and the large scale of systems, we propose end-to-end resilience, which provides a set of APIs crafted for HPC application developers to express their resilience demands in a modular fashion. This allows the separation of resilience concerns from algorithmic parts of the computation. The compiler and the runtime system support the requested resilience techniques seamlessly. Moreover, end-to-end resilience allows for creating regions with various resilience methods while protecting the data across regions.

Fault Resilience for Next Generation HPC Systems

by
Arash Rezaei

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

Rada Chirkova	Matthias Stallmann
Huiyang Zhou	Frank Mueller Chair of Advisory Committee

BIOGRAPHY

Arash Rezaei attended NC State University from 2010 to 2016.

ACKNOWLEDGEMENTS

I would like to thank Frank Mueller for his continuous support and open-mindedness. I also owe a debt of gratitude to Paul Hargrove, and Eric Roman for their mentorship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 High Performance Computing

High Performance Computing (HPC) is a method of computing that utilizes parallel processing for running applications efficiently, quickly and reliably that would be very time consuming if executed serially. HPC systems enable the execution of computation-intensive applications on a large number of high-end processors connected through a fast network interconnect. HPC systems allow scientists to perform research on climate modeling, protein folding, molecular dynamics, fluid dynamics, medical imaging, and many more domains. Typically, the performance of HPC systems is measured through a metric named Floating-Point Operations Per Second (FLOPS). In scientific calculations, the main focus of HPC, the number of floating point operations best describes the speed of the system and provides a metric to compare various systems. Table 1.1 shows the top 10 world supercomputers providing petaflops (10^{15} FLOPS) performance. The scale of these systems varies from 115K to 3M CPU cores and some of them receive their performance boost from coprocessors.

Table 1.1: Top 10 supercomputers (top500.org – Last Updated on November 2015)

Rank	**Name**	**FLOPS**	**CPU core count**	**Coprocessor**
1	Tianhe-2	33.86 peta	3,120K	Intel Xeon-Phi
2	Titan	17.59 peta	560K	Tesla GPUs
3	Sequoia	17.17 peta	1,572K	none
4	K Computer	10.51 peta	705K	none
5	Mira	8.58 peta	786K	none
6	Trinity	8.1 peta	301K	none
7	Piz Daint	6.27 peta	115K	Tesla GPUs
8	Hazel Hen	5.64 peta	185K	none
9	Shaheen II	5.53 peta	196K	none
10	Stampede	5.16 peta	462K	Intel Xeon-Phi

Coprocessors (or accelerators) are devices designed to further extend the performance of CPU cores. Computation and data are offloaded onto these devices to leverage the parallel architecture to execute embarrassingly parallel kernels of computation. Nvidia GPUs and the Intel Xeon Phi are two examples of such devices that are vastly used in HPC clusters and top supercomputers alike.

The growth in scale of problems and the demand for models with higher resolution necessitates the move toward next-generation supercomputers, which will provide exaflops (10^{18} FLOPS). The performance gap between current and future supercomputers is three orders of magnitude. Thus, innovations in all software/hardware components are required to bridge this gap and conquer the challenges on the path. Table 1.2 shows the Department of Energy (DOE) original timeline for exascale computing. It shows the specification of a 200 petaflops system that should have been deployed by 2015 in order to meet the exascale goal by 2018. As the 2015 deadline was not met, the timeline shifted to a target date of 2020 for exascale. Although the timeline has shifted, the specification still applies. An exascale system should have a Mean Time To Interrupt (MTTI) in the order of one day with a total memory size of 32-64 petabytes distributed across about 100K nodes (10x more memory and 2x more nodes compared to a 200 petaflops system). Note that the system needs to cope with errors such that the MTTI does not change from 200 petaflops to 1 exaflops (third and forth column). Thus, reliability is a major concern.

Table 1.2: DOE Exascale Timeline

System attributes	**2010**	**“2015”**	**“2018”**
System peak FLOPS	2 Peta	200 Peta	1 Exa
Power	6 MW	15 MW	20 MW
System memory	0.3PB	5 PB	32-64PB
Node performance	125 GF	0.5TF or 7 TF	1 TF or 10x
Node memory BW	25GB/s	0.1TB/s or 10x	0.4TB/s or 10x
Node concurrency	12	O(100)	O(1k) or 10x
TotalNode Interconn BW	1.5 GB/s	20 GB/s or 10x	200GB/s or 10x
System size (nodes)	18700	50,000 or 1/10x	O(100,000) or 1/10 x
MTTI	days	O(1day)	O(1 day)

1.1.1 Exascale Challenges

Fig. 1.1 depicts the challenges in the path toward exascale [1]. Failures and their effects on system performance and correctness are a major roadblock. Node failures could happen due to hardware or software faults. Hardware faults could be due to aging, power loss, or operation beyond certain temperature thresholds. Software faults can be due to bit flips due to radiation from

[1]Image borrowed from `http://aics-research.riken.jp/ishikawa_e.html`

space or small fabrication sizes, bugs that seldom materialize, complex software component interactions, and race conditions that only rare interleaving in the execution of tasks trigger.

The performance and scalability of Input/Output (I/O) subsystems are the second concern. Persistent storage is usually provided through the file system abstractions. There are dedicated I/O nodes that provide this service across the whole system. With a larger number of execution threads in the exascale system, higher load with be put on the I/O subsystem. This could potentially create a performance bottleneck.

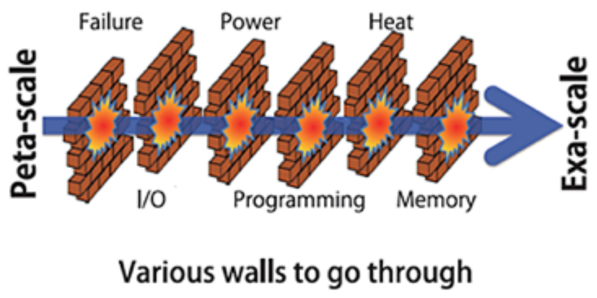


Figure 1.1: Exascale Challenges

The next challenge is power. Current supercomputers consume between 5 to 10 megawatts of electricity. Just to put it into perspective, this amount of energy is enough to supply thousands of houses. The electricity bill of HPC systems is a major concern, especially for exascale, and a growing body of research focuses on analyzing the requirements and proposing new methods for power savings.

Programibility, heat (cooling), and performance of memory systems are the remaining challenges. Exascale systems with billions of application threads potentially require innovations in the programming models such that synchronization, and communication can be performed with low overheads.

In order to create next-generation HPC systems, the above-mentioned challenges must be addressed. This work focuses on the first major challenge: reliability and failures.

1.2 Resilience

Resilience aims at keeping applications running so that they produce correct solution in a timely and efficient manner even in the presence of system degradation and failures. A correct solution can be achieved only through protecting the application from data corruption and errors. Note that degradation and failures include any hardware or software event that impedes application progress [41].

Fault tolerance aims at recovering from transient faults, masking permanently disabled components, and allowing applications to make progress even in the presence of failures. Fault tolerance has a long history in computational systems. Recently, the term fault resilience has been used to describe novel approaches that tackle challenges in system reliability. This includes a wide range of approaches from rollback recovery through preserving previous states to forward recovery via exploiting algorithmic properties for certain types of applications and errors. From a different perspective, it encompasses reactive methods that are only triggered after observing errors to proactive methods that utilize preventive actions to survive predicted failures.

In this dissertation, we use fault tolerance and fault resilience synonymously.

1.2.1 Approaches

Multiple methods to address failures are presented in this section.

Checkpoint-based Methods

These methods consider the system as a collection of processes that are communicating over a network. CR methods periodically take snapshots of the processes related to a job, create a consistent global state and save it to a persistent storage system. In case of a failure, rollback recovery reloads the last checkpoint, and the job is restarted. Common approaches mainly use a single process CR system as the core mechanism and then extend that to a system-wide CR implementation inside or on the top of a Message Passing Interface (MPI) library. MPI is a standardized and portable message-passing system designed by researchers from academia and industry to operate on a wide variety of parallel computing systems. CoCheck [126] is one of the earliest systems that implements CR on top of Condor [88] for the Parallel Virtual Machine (PVM) and MPI [62]. Starfish [3] provides automatic recovery in MPI-2 programs running on a network of workstations. It uses strict atomic group communication protocols to handle state changes. LAM/MPI [115] and subsequently Open MPI provide CR on the top of Berkeley Lab Checkpoint/Restart (BLCR) [49], which is a system-level and transparent CR implementation. BLCR can be used as a standalone system for a single node or as part of a larger system that runs parallel jobs. One of the challenges related to CR is to choose the length of a checkpoint interval. A method to calculate the optimal checkpoint interval has been presented in [40], which minimizes the application runtime considering the overhead of writing checkpoint files, the time required to restart the application in case of failures, and the time lost as a result of a failure. Unfortunately, traditional CR-based approaches do not scale well in terms of performance with increasing job sizes due to I/O contention. Ouyang et al. [100] try to improve the performance of checkpointing by aggregating checkpoint writes into a buffer pool and overlapping the application progress with file writes.

There are two fundamental types of checkpoint protocols: coordinated ones and uncoordinated ones. While coordinated checkpointing involves collaboration between processes to create a system-wide consistent state, uncoordinated checkpointing does not impose any restriction on the checkpoint time. Uncoordinated checkpointing results in less I/O contention but requires message logging and is subject to the *domino effect* [105] discussed next. In case of failure recovery, the inter-process dependencies imposed by message passing may force some of the processes that did not fail to roll back. Such dependencies can create a chain going all the way back to job start time since causal dependencies are transitive. This phenomenon is called the *domino effect*. Methods based on coordinated checkpointing are not affected by the domino effect since a globally consistent state is established at the cost of synchronization overhead [25]. Log-based methods combine checkpointing with logging of all non-deterministic events, which means all messages in the worst case, due to the domino effect. This results in large amounts of overhead and requires significant storage to maintain these logs. Considering the size and the induced overhead of these methods, exascale systems [46] may require uncoordinated checkpointing.

Communication-Induced Checkpointing [4] provides uncoordinated checkpointing without the domino effect. However, their work shows that the approach provides good performance only under a low communication load and it does not scale well with larger numbers of processes. Guermouche et al. [64] address the same problem for send deterministic MPI applications. A given MPI application is said to be send deterministic, if, for a set of input parameters, the sequence of sent messages, for any process, is the same in any correct execution [28]. Focusing on applications with this property, they provide a protocol that needs to log only a small subset of the application messages. They also provided an implementation in MPICH2 with experimental results showing low overhead.

Multiple attempts have been made to improve the scalability and overhead of CR methods. Ropars et al. [110] combine a hierarchical coordinated checkpointing and message logging. They exploit a property called "channel-determinism" that exist in some HPC applications. It states that for a given set of input parameters the sequence of messages sent on communication channels is the same in any valid execution. With channel-determinism, recovery can be performed based on a partial order of events. Their results with 512 MPI ranks show a recovery time of only few seconds. Riesen et al. [109] combine coordinated checkpointing with optimistic message logging and use a protocol that glues them together. This combination eliminates some of the drawbacks of each individual approach. They show the applicability of their approach through simulation.

Sato et al. [117] use a combination of non-blocking and multi-level checkpointing. The main idea is to deploy agents on extra nodes (e.g., I/O nodes) to asynchronously transfer checkpoint files to the Parallel File System (PFS). Using dedicated nodes to create checkpoints gives compute nodes the opportunity to continue their execution.

There are also log-based methods with different policies including optimistic logging [79], casual logging (Manetho [53]), and pessimistic logging (MPI/FT [13] and MPICH-V [19]).

Redundancy

Redundancy improves the reliability of systems by performing the same task multiple times. Dual Modular Redundancy (DMR) allows for detecting errors through comparison while Triple Modular Redundancy (TMR) can further correct the error of a single duplicate task via voting [89].

rMPI [22] is a library that provides transparent redundant computing. rMPI is implemented using PMPI, the profiling layer of MPI. It supports a large subset of MPI communication API calls and relies on the MPI library to implement collective operations. MR-MPI [55] supports partial and full replication and also uses PMPI to intercept MPI calls. Elliott et al. [51] determine the best configuration of a combined approach including redundancy and CR. They propose a cost model to capture the effect of redundancy on the execution time and checkpoint interval. Their result shows the superiority of full redundancy over partial redundancy in terms of execution time and specifically dual redundancy. Ferreira et al. [58] investigate the feasibility of process replication for exascale computing. A combination of modeling, empirical, and simulation experiments are presented in this work. The authors show that replication outperforms traditional CR approaches over a wide range of points in the exascale system design space.

RedMPI [59] allows the execution of MPI applications in a redundant fashion. Each process is replicated r times for a redundancy degree of r. Point-to-point messages and collective operations are performed within each replica set (original nodes and shadow nodes for $r = 2$, or a second shadow node set for $r = 3$). Under dual redundancy, an application with originally N processes now creates twice the number of MPI processes ($2N$). The native rank is the rank assigned by `mpirun` within the range $[0...2N - 1]$. The rank API call, `MPI_Comm_Rank`, returns the virtual rank. The MPI processes of each replica sphere, so-called replica ranks, are numbered $[0...r - 1]$. Fig. 1.2 depicts a dual redundant job and introduces the terminology for virtual, native, and replica ranks, where a virtual rank with its two boxes represents a sphere. The native ranks (assigned by `mpirun`) are in the range of 0 to 7. Only virtual ranks are visible to the application (in the range of 0 to 3). RedMPI supports SDC detection and correction through comparing the exchanged messages. It can be configured to utilize one of the following protocols: *Basic*, *Message-plus-hash*, and *All-to-all*. In the *Basic* protocol, no SDC detection/correction is provided. In *Message-plus-hash*, a message and a hash are sent to different endpoints and endpoints can check a received hash against the corresponding received message. In *All-to-all*, messages are sent/received by all replicas in a sphere. E.g., for $r = 2$, a send becomes two sends and a receive becomes two receives. *Message-plus-hash* has superior performance over *All-to-all* since fewer and shorter messages are exchanged.

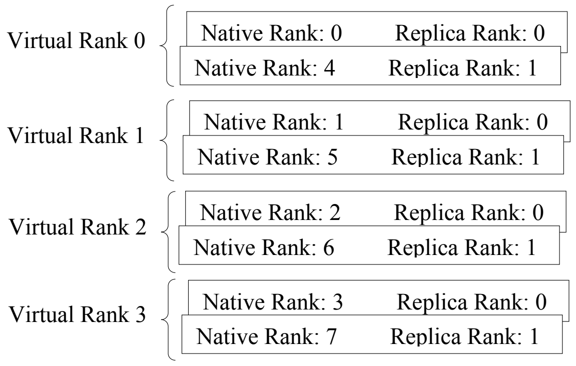

Virtual Rank 0
Native Rank: 0 Replica Rank: 0
Native Rank: 4 Replica Rank: 1
Virtual Rank 1
Native Rank: 1 Replica Rank: 0
Native Rank: 5 Replica Rank: 1
Virtual Rank 2
Native Rank: 2 Replica Rank: 0
Native Rank: 6 Replica Rank: 1
Virtual Rank 3
Native Rank: 3 Replica Rank: 0
Native Rank: 7 Replica Rank: 1

Figure 1.2: Dual Redundancy for a job with 4 tasks

Migration

A process-level proactive live migration approach is presented by Wang et al. [131]. It includes live migration support realized within BLCR, combined with an integration within LAM/MPI. Their experimental results show low overhead. They also compare process-level live migration against operating system migration running on top of Xen virtualization. Engelmann et al. [56] propose a framework and architecture for proactive fault tolerance in HPC, including health monitoring and feedback control-based preventive actuation. This work investigates the challenges in monitoring, aggregating data and analysis.

Rejuvenation

Three rejuvenation techniques for HPC have been introduced by Naksinehaboon et al. [96]. They also provide an overhead estimation of using rejuvenation right after taking checkpoints. Based on their simulation results, they conclude that rejuvenation does not increase the reliability of HPC systems at all times. Another intuitive result is that the computing time lost in case of CR with rejuvenation is larger than that when only CR is used.

1.3 Hypothesis

Checkpoint/Restart, redundancy, and process migration are the common methods for resilience in today's high performance computing systems. However, current approaches suffer from the following shortcomings:

- There is a lack of support for CR and process migration for new architectures such as the Intel Xeon-Phi.
- State-of-art redundancy-based HPC systems do not repair failed replicas, thus errors may remain undetected or a job may even fail altogether.
- Developing resilient HPC applications requires considerable effort. Application developers might need to re-write their application in a specific programming model with GVR, Charm++ to address soft errors.

Hypothesis: As scalability and soft errors increasingly become challenges, resilience for HPC systems require novel capabilities that handle checkpointing on accelerators, reduce overhead of redundant computing, and provide protection spanning across phases of alternating resilience techniques.

1.4 Contributions

We present solutions for these three key challenges. First, we design and implement Snapify, a snapshot system (CR, process migration, and swapping) for Intel Xeon-Phi servers. Second, we design and implement DINO, a system for replenishing failed nodes in redundant computing. Third, we provide a compiler-based approach to separate resilience concerns from algorithmic goals in order to provide maximum code modularity with minimum effort by programmers for implementing resilient HPC applications.

1. **Snapify:** Intel Xeon Phi coprocessors provide excellent performance acceleration for highly parallel applications and have been deployed in several top-ranking supercomputers. One popular approach of programming the Xeon Phi is the offload model, where parallel code is executed on the Xeon Phi, while the host system executes the sequential code. However, Xeon Phi's Many Integrated Core Platform Software Stack (MPSS) lacks fault-tolerance support for offload applications. We introduce Snapify, a set of extensions to MPSS that provides three novel features for Xeon Phi offload applications: checkpoint and restart, process swapping, and process migration. The core technique of Snapify is to take consistent process snapshots of the communicating offload processes and their host processes. To

reduce the PCI latency of storing and retrieving process snapshots, Snapify uses a novel data transfer mechanism based on remote direct memory access (RDMA). Snapify can be used transparently by single-node and MPI applications, or be triggered directly by job schedulers through Snapify's API. Experimental results on OpenMP and MPI offload applications show that Snapify adds a runtime overhead of at most 5%, and this overhead is low enough for most use cases in practice.

2. **DINO:** Redundant computing has been proposed as a solution at extreme scale by allocating two or more processes to perform the same task. However, current redundant computing approaches do not repair failed replicas. Thus, correct execution completion is not guaranteed after a replica failure occurs as a node no longer has a redundant shadow node. Replicas are logically equivalent, yet may have divergent runtime states during job execution, which complicates on-the-fly repairs for forward recovery. In this work, we present a redundant execution environment that quickly repairs hard failures via Divergent Node cloning (DINO) at the MPI task level. DINO contributes a novel task cloning service integrated into the MPI runtime system that solves the problem of consolidating divergent states among replicas on-the-fly. Experimental results indicate that DINO can recover from failures nearly instantaneously, thus retaining the redundancy level throughout job execution. The cloning overhead, depending on the process image size and its transfer rate, ranges from 5.60 to 90.48 seconds. To the best of our knowledge, the design and implementation for repairing failed replicas in redundant MPI computing is unprecedented.

3. **End-to-End Resilience:** We propose an infrastructure that allows for computation with end-to-end resilience that requires minimum programmer effort while providing high code modularity. As an automatic approach for providing resilience throughout the whole application execution, our system lifts the burden from the application programmers, allowing them to focus solely on performance of the computation while resilience means are embedded into source code via compiler/library enhancements and runtime support. This work is motivated by the aspect-oriented programming (AOP) paradigm, which aims to increase modularity by allowing the separation of concerns. AOP includes programming methods that support the modularization of concerns (goals) at the source code level. We further define the vulnerability factor as product of the length of a liveness range of a data structure (in seconds) and the size (in bytes) as a metric that indicates the likelihood for a soft error (such as bit flips) to occur in this data.

1.5 Organization

Chapter 2 presents Snapify and our innovations in providing a snapshot service for Xeon Phi servers. The design, implementation and evaluation of DINO is presented in Chapter 3. End-to-end resilience provides protection during the whole lifetime of data structures and is presented in Chapter 4. Chapter 5, summarizes this work and discusses potential future work.

Chapter 2

Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors

2.1 Introduction

A Xeon Phi coprocessor has up to 61 cores, connected by a high-speed on-chip interconnect. Each core supports four hardware threads, and has a 512-bit wide vector unit to execute SIMD instructions. The coprocessor has its own physical memory of 8/16GB, which can be accessed with an aggregate memory bandwidth of 352GB/s. Both the coprocessor and the host system run their own operating systems. The host system and the coprocessor do not share a common memory space, and the two are physically connected by the PCIe bus.

Xeon Phi's software stack (MPSS) supports two programming models. In the *offload* programming model, highly parallel, code segments are executed on Xeon Phi, while the host system executes the sequential code. On the other hand, the *native* programming model allows users to execute their applications entirely on the Xeon Phi coprocessor. MPSS provides high-level language support and a modified Linux OS on Xeon Phi to facilitate both programming models. As a result of its performance acceleration for highly parallel applications and ease of programming, Xeon Phi coprocessors have been deployed in HPC systems, including several top-ranking supercomputers [66].

Using coprocessors like Xeon Phi in HPC systems, however, compounds the problem of the increasing failure rate due to the system's growing size and complexity [27]. A recent study on a GPU-based supercomputer shows a failure rate of 13 hours on average [116]. It is projected that the mean time between failure of HPC systems will continue to shrink [27, 118]. Therefore, programmers must adopt certain fault tolerance mechanism for their applications.

Checkpoint and restart is a fault-tolerance technique widely used in HPC systems. Such a method periodically takes a snapshot of the application state and saves it on persistent storage. In case of an error, the application can be restored to a former saved state. The popular checkpoint and restart tool BLCR provides application-transparent checkpoint and restart support [49]. It can take a snapshot of the entire process state in both the user and the kernel space, with no modification to the application code. It has been integrated with MPI to provide distributed checkpoint and restart for MPI applications running on a cluster [115].

Although Xeon Phi's software stack is designed to ease the programming effort, its support of fault tolerance is inadequate. MPSS uses BLCR to support checkpoint and restart of native applications. BLCR can either save the snapshot of a native process on Xeon Phi to the host's file system through Network File System (NFS), or to Xeon Phi's own local file system. However, both of these two storage choices have limitations. Saving directly to the host file system through NFS incurs high data transfer latency on PCIe bus. And because Xeon-Phi does not have any directly accessible storage and uses a RAM-based file system, a locally saved snapshot on Xeon Phi's own file system competes the physical memory space with active processes, including the native process whose snapshot is to be saved.

A more severe problem is that MPSS has no fault-tolerance support for offload applications. Given the fact that even a single-node offload application involves the participation of a number of host and coprocessor processes that use proprietary communication libraries in MPSS to exchange messages, it is not surprising that none of the existing distributed checkpoint and restart tools like BLCR or DMTCP [9] can be used. This is because these tools do not consider the communication between the coprocessor processes and the host processes.

In addition MPSS also lacks sufficient support for process migration among Xeon Phi coprocessors. When compute nodes are equipped with multiple coprocessors, process migration can benefit coprocessor load balancing and fault resiliency. For example, a job scheduler may decide to migrate an offload or native process from a heavily-loaded coprocessor to a lightly-loaded one to increase job turnaround time [15]. Moreover, by using fault prediction methods [114], it is possible to avoid imminent coprocessor failures by proactively migrating processes to other healthy coprocessors.

The limited physical memory on the Xeon Phi coprocessor also restricts the sharing of the coprocessor among multiple applications. Previous studies have shown that allowing multiple user applications to share coprocessor like GPU or Xeon Phi can significantly benefit system utilization and job turnaround time [23, 15, 103, 113]. However, Xeon Phi OS's own page swapping mechanism is a poor solution to overcome the capacity limit of Xeon Phi's physical memory for multiprocessing. First, as shown by the study in [23], Xeon Phi OS's swap uses the host's file system as its secondary storage. Swapping in and out memory pages between the host and the coprocessor incurs high data transfer latency. Second, many offload applications use pinned memory buffers to allow fast remote direct memory access (RDMA) between the host and the coprocessor's memory. Pinned memory pages cannot be swapped out by OS. As a result, the size of Xeon Phi's physical memory puts a hard limit on the number of processes that can concurrently run on the coprocessor.

Contributions. To address these shortcomings of MPSS, we created Snapify, a set of extensions to MPSS that captures snapshots of offload applications on Xeon Phi. We make the following specific contributions:

- We propose Snapify, an application-transparent, coordinated approach to take *consistent* snapshots of host and coprocessor processes of an offload application. We believe this is the first proposal that correctly and transparently captures a process-level snapshot of offload applications on many-core processors like the Xeon Phi.

- We use Snapify to implement three new capabilities for offload applications: application-transparent checkpoint and restart, process migration, and process swapping. Again, to the best of our knowledge, this is the first proposal that provides such capabilities for offload applications on Xeon Phi.

- We propose a fast, remote file access service based on RDMA that speeds up the storage and retrieval of snapshots (of both offload and native applications) from the host's file system.

- We evaluate Snapify on several OpenMP and MPI benchmarks. Our results show that Snapify imposes negligible overhead in normal execution of offload applications (less than 5%), and the overheads due to the capture of snapshots in checkpoint and restart, process swap, and process migration are small enough to make it practical to use Snapify for a variety of offload applications.

This chapter is organized as follows. Section 2.2 gives a concise background of Xeon Phi's programming model. Section 2.3 discusses Snapify's design challenges. Section 2.4 describes Snapify's internal design and its API. Section 2.5 shows how Snapify's API can be used to implement checkpoint and restart, process swapping, and process migration. Section 2.6 introduces our RDMA-based remote file access service for snapshot storage. Section 2.7 presents the experimental results. Related work is discussed in Section 2.8. Section 2.9 summerizes our findings.

2.2 Background

In this section, we briefly describe the offload programming model and its implementation in MPSS.

Xeon Phi's software stack MPSS consists of a hierarchy of runtime libraries that execute on the host and the coprocessor [71]. These libraries hide complex, low-level communication mechanisms away and present a simple, high-level offload programming semantics to the programmers. The software stack also has a modified Linux OS that is run on Xeon Phi. The Xeon Phi OS has its own file system (built on RAM disk using Xeon Phi's own physical memory), virtual memory management, and an optional page-swap mechanism that uses the host's file system as secondary storage.

To program an offload application a programmer uses special compiler directives to delineate regions of code to be executed on Xeon Phi coprocessors. In particular, the pragma "offlaod" is used to mark an offload region with language-specific scoping constructs (curly braces in C, and "begin" and "end" in Fortran). In particular, the offload pragma can have additional data transfer clauses "in" and "out", specifying the input and output buffers of the offload region. Input buffers specified in the "in" clauses are transferred from the host memory to the Xeon Phi coprocessor through the PCIe bus prior to the execution of the offload region, while output buffers specified in the "out" clauses are transferred back from the Xeon Phi coprocessor to the host memory following the completion of the offload region. More details of Xeon Phi programming can be found in [77].

The Xeon Phi compiler generates one binary for the host processor and one binary for the Xeon Phi coprocessor for an offload application. The host binary is an executable, while the Xeon Phi binary is a dynamically loadable library. The compiler translates each offload region as a function to be executed on the Xeon Phi coprocessor and saves the generated code in the Xeon Phi binary. For each offload region, in the host binary the compiler also generates a function call to the lower-layer runtime libraries, which coordinate the data transfer between the host and the Xeon Phi coprocessor and initiate the "remote" procedure call of the offload function.

The execution of an offload application to be accelerated by Xeon Phi coprocessors involves a minimum of three processes, as reported in Fig. 2.1. The three processes are a host process (`host_proc`) running on the host processor, an offload process (`offload_proc`) running on the Xeon Phi, and a service named Coprocessor Offload Infrastructure (COI) daemon (`coi_daemon`), also running on the Xeon Phi. After a user launches the application (`host_proc`), the host process requests the COI daemon to launch a process on the Xeon Phi. Then the host process copies the Xeon Phi binary to the coprocessor. The binary contains offload functions to be executed on the coprocessor, and these functions are dynamically loaded into `offload_proc`'s memory space. The execution of an offload function is done by a server thread in the offload

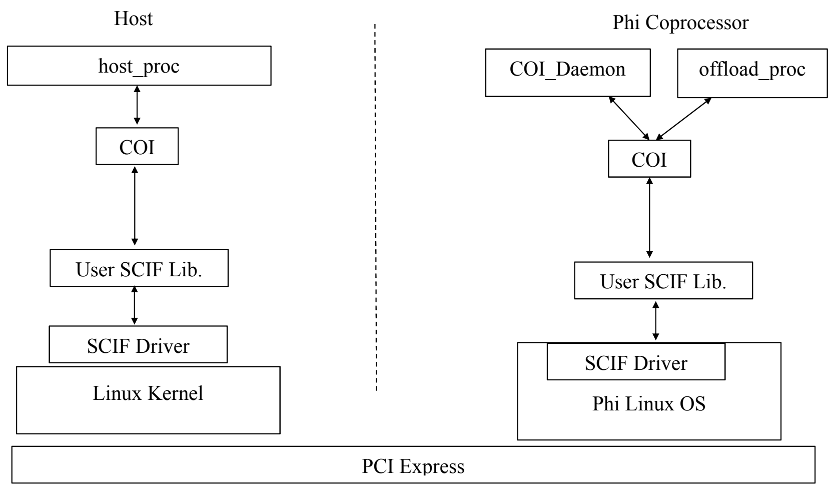


Figure 2.1: The software architecture of host & Xeon Phi.

process. To execute a function, the host process sends a request to the server thread to run the specified function. Once the function completes on Xeon Phi, the server thread will send the function's returned value back to the host process.

Prior to the execution of an offload function the host process also transfers the input data needed by the offload region to the offload process's memory space. The host process also receives data, if any, that is generated by the offload process.

MPSS provides different levels of abstractions to facilitate the communications between the host process and the processes running on the Xeon Phi. The COI library is an upper level library offering APIs for a host process to perform process control and remote function calls on a Xeon Phi coprocessor [70, 71]. It also allows a host process to create buffers, called COI buffers. The host process can use COI's API to transfer data between the host process and the buffers allocated in the offload process. The COI library in turn uses the lower level Symmetric Communications Interface (SCIF) library to accomplish the real message exchanges and data transfers between the host process and the offload process [72].

The COI library on the Xeon Phi coprocessor manages the memory space used by COI buffers. A COI buffer is composed of one or more files that are memory mapped into a contiguous region. These files are called local store. COI buffers can be created and destroyed through COI functions, while the files created to be used by COI buffers are persistent until the offload process terminates.

SCIF provides two types of APIs that allow two processes in different memory space to communicate. The first type is message-based. The processes use `scif_send()` and `scif_-receive()` to send and receive data. The second type offers remote direct memory access (RDMA) functions to speed up the data transfer. To use SCIF's RDMA functions to transfer a buffer, a process first registers the buffer's virtual memory address using `scif_register()` function. The function returns an offset address that can be used in SCIF's RDMA functions: `scif_vreadfrom()`, `scif_readfrom()`, `scif_vwriteto()`, and `scif_writeto()`. In fact, the COI library uses SCIF's RDMA functions to copy data in COI buffers between the host and coprocessors.

Each Xeon Phi device runs one COI daemon process to coordinate the execution of offload processes and the corresponding host processes. The COI daemon maintains SCIF connections with each active host process that uses COI library to offload part of its computation to Xeon Phi. The connections are used to communicate the process control messages between host processes and the COI daemon. For example, the COI daemon launches new offload processes upon requests from applications on the host. If the host process exits, the daemon will terminate the offload process and clean up the temporary files used by the offload process.

2.3 Challenges

There are several challenges that must be overcome to capture a process-level snapshot of an offload application, and subsequently restart the application from the snapshot. These challenges arise from the distributed nature of an offload application on a Xeon Phi server, and Xeon Phi's own software stack. Below we summarize these new challenges.

2.3.1 Capturing consistent, distributed snapshots

Since the execution of an offload application on Xeon Phi involves multiple communicating processes that do not share memory or a global clock, it is necessary to ensure that the snapshots of the host and offload processes form a *consistent* global state [30, 84, 115]. In the simplest case, an offload application has three processes: a process on the host and two processes (an offload process, and `coi_daemon` process) on the Xeon Phi. A global state consists of states of the three processes, as well as the state of the communication among these processes. A consistent global state satisfies two properties. First, it is possible to reach this state during the normal operation of the application. Second, it is possible to correctly restart the processes and resume the execution of the application. As a counter example, if the host snapshot is taken before the host process sends a message to the coprocessor, and the snapshot of the offload process is taken after the receipt of the message, then this pair of snapshots does not form a consistent global state, and the global state cannot be applied to resume the application. Due to the distributed nature of Xeon Phi's offload model, snapshots obtained by just using an existing single-process checkpoint tool like BLCR [49] or MTCP [108] cannot form a consistent global state.

The states of communication channels are also part of the global state. Since we cannot take a snapshot of the physical state of a PCIe bus and the internal hardware state of its controllers, we must make sure all communication channels between the involved processes are drained before local snapshots are taken.

2.3.2 Xeon Phi-specific communication libraries

A Xeon Phi offload application uses its own proprietary communication libraries (i.e. COI and SCIF) for inter-process communication. Therefore, the existing cluster checkpoint tools designed for applications based on communication libraries like MPI [115] or TCP/IP [75, 111, 9] cannot be applied to Xeon Phi offload applications.

2.3.3 Dealing with distributed states

The states that are distributed among the processes participating in the execution of an offload application may get disturbed by the action of taking a snapshot, swapping out, or restarting the offload process. For example, the COI daemon is responsible for monitoring the status of both the host and the offload process. If an offload process is terminated due to being swapped out or being migrated, the `coi_daemon` will assume that the offload process has crashed and (incorrectly) mark the process as terminated. On the other hand, when an offload process is restarted from a snapshot, the `coi_daemon` needs to be brought into the picture again to monitor the restarted host and offload process.

2.3.4 Storing and retrieving snapshots

The split of Xeon Phi's physical memory between the file system and system memory puts a serious restriction on how the snapshot of an offload process can be stored. A naive solution that saves a snapshot on Xeon Phi's local file system cannot be applied to any process, either native or offload, whose memory footprint exceeds 50% of the Xeon Phi's physical memory.[1] The same restriction also applies when we attempt to restart a process from a snapshot stored on Xeon Phi's local file system. Even if the snapshot and the process fit in the Xeon Phi's physical memory, the memory used to store the snapshot is unavailable to other offload or native applications on the Xeon Phi, resulting in a decrease in the number of applications that can run concurrently or even a crash of some applications due to lack of memory. Therefore, it is desirable to store and retrieve snapshots from the host's file system.

[1]The actual limit is less than half, since the system files and the OS use a small portion of the memory.

2.3.5 Saving data private to an offload process

A snapshot of an offload process contains the offload process's own private data. Unlike GPU-based coprocessor systems, an offload process on Xeon Phi is a full-blown Linux process. The offload process has host-allocated COI buffers, and its own private data that may not be visible to the host system (for example, stacks of threads or data regions that are either statically or dynamically allocated through standard system calls like `malloc()` by the functions in the offload process are not visible to the host). The offload-private data may persist across several offload regions. Therefore, the strategy of only saving the host-controlled memory regions in a snapshot, as proposed for GPU-accelerated applications in CheCL [128] and CheCUDA [127], is not suitable for Xeon Phi's offload applications.

2.4 Snapify

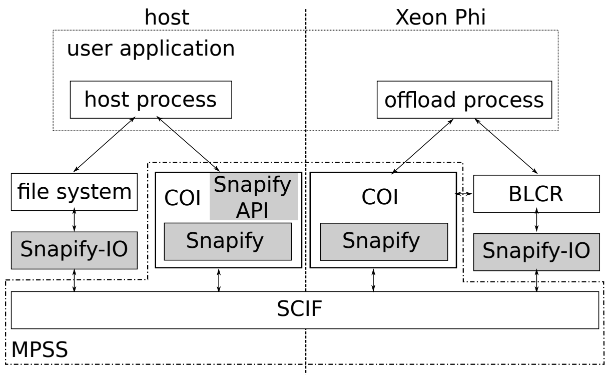


Figure 2.2: The architecture of Snapify.

Fig. 2.2 shows the positioning of Snapify technologies in the software stack of the host and the Xeon Phi. Intel's MPSS includes COI (which contains the COI daemon) and SCIF. The file system is part of the Linux OS on the host, and BLCR is the open-source checkpoint and restart framework. Applications offload computations to the Xeon Phi using the COI library, which has a component that executes on the host and a component that executes on the Xeon Phi. Key technologies in Snapify are implemented as modifications to the COI library and the COI daemon, and as an independent user-level library called *Snapify-IO*. The implementation does not change the COI programming interface, and thus is transparent to user applications. Sections 2.4.1, 2.4.2, and 2.4.3 discuss our techniques to create process-level snapshots of the offload application, restore the execution of the application from a snapshot, and resume the application after a

```
typedef struct {   char* m_snapshot_path;
                   sem_t m_sem;
                   COIProcess* m_process;
                   }snapify_t;

void snapify_pause(snapify_t* snapshot);

void snapify_capture(snapify_t* snapshot,
                   bool terminate);

void snapify_wait(snapify_t* snapshot);

void snapify_resume(snapify_t* snapshot);

void snapify_restore(snapify_t* snapshot,
                   int device);
```

Table 2.1: Snapify API.

snapshot has been taken, respectively. Snapify-IO, to be described in Section 2.6, is a very efficient mechanism that we designed to store and retrieve snapshots from the host file system using SCIF's RDMA functions.

Snapify provides a simple API that is used to capture snapshots of an offload process and restore the offload process from snapshots. The API is summarized in Table 2.1. Snapify's API defines a C structure and five functions. The functions are called by a host process to capture a snapshot of the offload process (`snapify_pause()` and `snapify_capture()`), to resume the communication and the partially-blocked execution of the host process and the offload process after a snapshot is taken (`snapify_resume()`), to restore an offload process from a snapshot (`snapify_restore()`), and to wait for a non-blocking API function to complete (`snapify_-wait()`). All functions except `snapify_capture()` are blocking calls. The structure is used to pass parameters and to receive results of the functions. The API functions are detailed in the rest of this section.

2.4.1 Taking Snapshots

Taking a snapshot of an offload process involves the host process on the host and the COI daemon and the offload process on each of the coprocessors installed in a Xeon Phi server. Although our approach handles multiple Xeon Phi coproessors in a server, for simplicity we assume there is only one Xeon Phi coprocessor in the following discussions. Therefore, we consider the case of three involved processes: the host process, the COI daemon, and the offload process.

The snapshot process is accomplished in two separate steps. In step one, all of the communications between the host process and the offload process are stopped, and the channels are drained. In step two, a snapshot of the offload process is captured and saved in the file system of the host. These two steps are implemented by `snapify_pause()` and `snapify_capture()`, respectively.

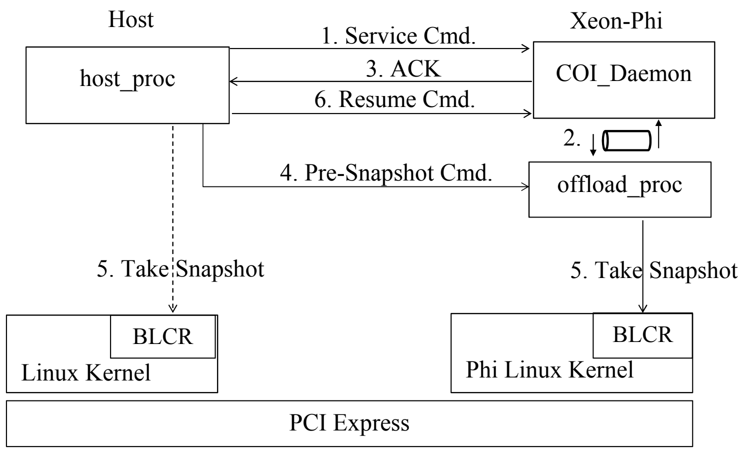


Figure 2.3: Overview of taking a snapshot.

Pause

To pause the communications between the host process and the offload process, the host process calls `snapify_pause()` and passes the handle of the offload process (`COIProcess` in the structure) to `snapify_pause()`. A directory structure in the host's file system for storing the files of a snapshot is also needed by `snapify_pause()` (and `snapify_capture()`). The path to the directory is passed to `snapify_pause()` through the member variable `m_snapshot_path`. In the first step of `snapify_pause()` it saves the copies of the runtime libraries from the host's file system needed by the offload process to the snapshot diirectory.[2]

Fig. 2.3 shows the interactions between the host process, the COI daemon, and the offload process that are triggered by `snapify_pause()`. Function `snapify_pause()` first sends a snapify-service request to the COI daemon (step 1 in Fig. 2.3). The daemon then creates a UNIX pipe to the offload process, and writes the pause request to the offload process. Next the daemon signals the offload process, triggering the signal handler in the offload process to the pipe and send an acknowledgement back to the daemon through the pipe (step 2). The daemon then relays the acknowledgement back to the host process (step 3). At this point all parties (the host

[2]MPSS maintains copies of the runtime libraries on the host file system. Therefore, as an optimization we do not copy the libraries of the offload process from the coprocessor back to the host system.

process, the offload process, and the COI daemon) have agreed to pause the communications and drain the communication channels.

The COI daemon is chosen as the coordinator of Snapify's pause procedure. This is because there is one daemon per coprocessor, and each daemon listens to the same fixed SCIF port number. It services pause requests that may come from different host processes. It also maintains a list of active requests. Upon receiving a new pause request, the daemon adds an entry to the list. The entry is removed after the pause request is serviced.

To avoid any interference with its regular tasks, the daemon uses a dedicated Snapify monitor thread to oversee the progress of the pause procedure. Whenever a request is received and no monitor thread exists, the daemon creates a new monitor thread. The monitor thread keeps polling the pipes to the offload processes on the list of active pause requests for status updates. The monitor thread exits when there is no more active pause request in the list.

Following the initial handshake `snapify_pause()` sends a pause request to the offload process (step 4 in Fig. 2.3) to drain the communication channels. The draining needs the collaboration between the host process, the COI daemon, and the offload process, and will be discussed in more detail shortly. It is a necessary step to ensure that the snapshots form a consistent global state (see Section 2.3). During the draining process some of the threads in the host process and the offload process spawned by the COI library are blocked. The blocking of these threads keeps the SCIF channels from being used until `snapify_resume()` is called. These threads are responsible for sending and receiving COI commands, COI events, and the COI logs.

After the SCIF channels are quiesced, the offload process will save its local store (memory allocated in the offload process's memory space for storing data in COI buffers) to the host's snapshot directory. This operation does not use any existing SCIF channels between the host process and the offload process. Saving the local store and the snapshot will be discussed in detail in Section 2.6.

At the end of `snapify_pause()` all of the SCIF channels between the host process, the COI daemon, and the offload process become empty. To notify the host process that the pause has completed, the offload process sends a message through the pipe to the COI daemon, and the COI daemon informs the host process that the offload process has completed the pause operation. After this the offload process waits on the pipe to wait for the next request from the the host process. The next request is either a capture or a resume request, which will be discussed later.

We now give more details on how `snapify_pause()` drains the SCIF communication channels. We first classify all SCIF communication use instances in the COI runtime to four different cases.

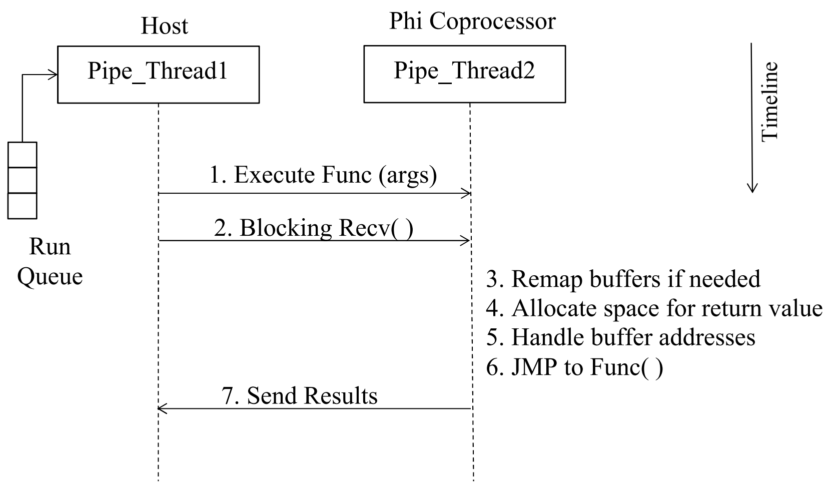


Figure 2.4: Executing an offload function.

1. The host process, the offload process, and the COI daemon exchange messages when an offload process is created and before it is destroyed. These messages carry information regarding process creation, confirmation, request for termination, and etc.

2. The host process and the offload process use one SCIF channel to perform RDMA transfers of the data in COI buffers. The RDMA transfers are carried out by `scif_writeto()` and `scif_readfrom()` functions.

3. The host process, the COI daemon, and the offload process have several pairs of client-server threads. Each server thread serves only one client thread. It handles the incoming commands in a sequential fashion. The commands are sent by the client thread through a dedicated SCIF channel.

4. The execution of an offload function is also implemented by a client-server model. In order to take a snapshot during the execution of an offload function, however, we treat this case separately. Our method handles both synchronous and asynchronous offload executions.

   Fig. 2.4 reports the client-server model that executes offload functions. When a thread *Pipe_Thread1* in the host process enters an offload region, it sends a run function request to the server thread *Pipe_Thread2* in the offload process. After sending this request, *Pipe_Thread1* performs a blocking receive to wait for the result, while *Pipe_Thread2* calls the function and sends the function's return value back to *Pipe_Thread1* of the host process.

For each of the four use cases of SCIF we develop a method to drain the SCIF communication channels. For case 1, we declare the initialization and cleanup code regions of creating and terminating offload processes as critical regions, protected by a mutex lock. When `snapify_-pause()` is called, it will try to acquire the lock. If a thread is executing the code in a critical region, `snapify_pause()` will be blocked until the thread leaves the critical region. On the other hand, once `snapify_pause()` holds the lock, any other thread that attempts to enter these critical regions will be blocked.

For case 2 we delay any snapshot attempt when a RDMA transfer is active. Similar to the case above, we protect the call sites of SCIF's RDMA functions with mutex locks.

To handle a SCIF channel in the client-server model of case 3, we take advantage of the sequential nature of the client-server implementation in COI. We added a new "shutdown" request to the server's request handling routine. This request is only issued by `snapify_pause()`, and is used as a special marker that indicates no more commands will follow until `snapify_resume()` is called. To send the shutdown request, `snapify_pause()` first tries to acquire the lock that is used by the client thread to protect the communication channel. After `snapify_pause()` acquires the lock, the client thread will not be able to send any more requests. The lock that is used by a client thread will only be released in `snapify_resume()`. After acquiring the lock `snapify_pause()` sends the shutdown request to the server. The pause function will not continue until all of the server threads in the host process, the COI daemon, and the offload process receives a shutdown command. This ensures that the SCIF channels used between the client and the server threads stay empty until `snapify_resume()`.

For case 4 to drain the SCIF channel used by *Pipe_Thread1* and *Pipe_Thread2* we made a number of changes to the implementation of the COI pipeline. First we transformed the two send functions in Step 1 and 7 to be blocking calls. We then placed these two send functions in two separate critical regions protected by mutex locks. The thread executing `snapify_pause()` in the host process and in the offload process will acquire these locks. The locks will be released in `snapify_resume()`.

Capture

To capture a snapshot of an offload process the host process calls `snapify_capture()`. Similar to `snapify_pause()`, the caller of `snapify_capture()` passes the handle to the offload process and the path to the directory on the host's file system where the snapshot files should be saved. It also gives a Boolean variable `terminate` to indicate whether the offload process should be terminated after its snapshot is captured. At the beginning `snapify_capture()` sends the capture request first to the COI daemon, which in turn forwards the request to the offload process through the pipe opened in `snapify_pause()`. The snapshot of the offload process can be captured by any application-transparent checkpoint tool.

Our current implementation uses BLCR to capture the snapshot of the offload process. When the offload process receives the capture request from the pipe, it calls BLCR's `cr_request_-checkpoint()`. When the snapshot is captured, the offload process sends back the completion message using the pipe to the COI daemon, which in turn informs the host process.

The snapshot of the offload process is saved on the host file system. The snapshot is written by the checkpoint and restart tool running on the coprocessor. Section 2.6 details several novel techniques of saving a snapshot "on the fly" from the coprocessor to the host file system.

Notice that `snapify_capture()` is a non-blocking function call. It returns immediately with a semaphore `m_sem` in `snapify_t* snapshot`. The caller can thereafter call `snapify_-wait()` with the `snapify_t` structure to wait for the completion of the capturing operation. The semaphore will be signaled when the host process receives the complete message from the COI daemon.

2.4.2 Resume

To resume the execution of the blocked threads of both the host process and the offload process after a snapshot of the offload process is taken, the host process calls `snapify_resume()` with the handle of the offload process. To resume the host process first sends a resume request to the COI daemon. The daemon then forwards the request to the the offload process through the pipe that is created in `snapify_pause()`. In `snapify_resume()`, both the host process and the offload process release all the locks acquired in the pause operation. Once the locks are released in the offload process, the offload process sends an acknowledgement back to the host process through the COI daemon. After the host process receives the acknowledgement and releases the locks, it returns from `snapify_resume()`.

2.4.3 Restore

To restore an offload process from its snapshot the host process calls `snapify_restore()` with the path to the snapshot files. Snapify relies on the COI daemon and Xeon Phi's checkpoint and restart tool (BLCR) that is also used by `snapify_capture()` to restart an offload process. To restore, `snapify_restore()` first sends a restore request to the COI daemon. After receiving the restore request, the COI daemon first copies the local store and the runtime libraries needed by the offload process on the fly to the coprocessor. Then it calls BLCR to restart the offload process from its snapshot. We developed a novel I/O mechanism, called *Snapify-IO*, that allows BLCR to read the snapshot of the offload process "on-the-fly" from the host storage directly without first saving the entire snapshot in the memory file system on Xeon Phi. Snapify-IOwill be discussed in detail in Section 2.6.

After BLCR restores the process image of the offload process, the host process and the offload process will reconnect all of the disconnected SCIF communication channels between them. After SCIF channels are restored, the host process and the offload process re-registers the memory regions used by the COI buffers for RDMA. The re-registration of a buffer may return a new RDMA address different from the original one. Therefore, we keep a lookup table of (old, new) address pairs for conversion.

Since the restored offload process is new, `snapify_restore()` returns a new COI process handle (`COIProcess*`) to the offload process. The new handle can be used by the host process in the subsequent COI function calls. Notice that the offload process, though restored, is not fully active after `snapify_restore()` returns. The caller needs to call `snapify_resume()` so that the blocked threads in the host process and the offload process can continue their executions.

2.5 API Use Scenarios

In this section, we explain how Snapify's API can be used to implement checkpoint and restart, process swapping and migration.

2.5.1 Checkpoint and restart

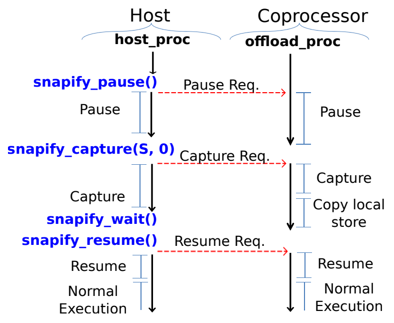
To take a checkpoint of an offload application we need to capture both the snapshots of the host process and of the offload process. To capture a snapshot of the host process, we can use an application-transparent checkpoint and restart tool like BLCR on the host. As to the snapshot of the offload process, we use `snapify_pause()` and `snapify_capture()` in Snapify's API.

The sample code in Fig. 2.5(a) shows how Snapify's API can be combined with the host BLCR to implement checkpoint and restart for offload applications. Fig. 2.5(b) and 2.5(c) reports the timing diagrams of the checkpoint and restart. The function `snapify_blcr_callback()` is a callback function that is registered to BLCR on the host. When BLCR receives a checkpoint request, it will call `snapify_blcr_callback()`. Within `snapify_blcr_callback()`, we call BLCR's `cr_checkpoint()` to take a snapshot (a checkpoint) of the host process. Before `cr_-checkpoint()`, we call `snapify_pause()` and `snapify_capture()` to take a snapshot of the offload process. Notice that `snapify_capture()` is a non-blocking call. Therefore, we need to wait for its return in the "continue" section of the `if` statement after `cr_checkpoint()` returns.

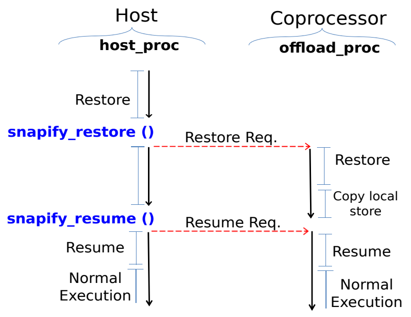
In restarting BLCR first restores the host process. The execution of the restored host process will begin after `cr_checkpoint()` returns with `ret` > 0. The control flow of the execution will go through the "restart" section of the "if" statement. There we call `snapify_restore()` to

```
int snapify_blcr_callback(void* args){
    int ret = 0;
    snapify_t* snapshot = (snapify_t*)args;
    snapify_pause(snapshot);
    snapify_capture(snapshot, false);
    ret = cr_checkpoint(0);
    if ( ret > 0 ) { // Restarting.
       snapify_restore(snapshot,
         GetDeviceId(snapshot->m_process));
       snapify_resume(snapshot);
       // Save snapshot.m_process.
    }
    else { // Continue.
       snapify_wait(snapshot);
       snapify_resume(snapshot);
    }
}
```

(a) Sample code.



(b) Checkpoint timing diagram.



(c) Restart timing diagram.

Figure 2.5: Using Snapify's API to implement checkpoint and restart.

recreate the offload process. In the sample code the offload process will be restored on a Xeon Phi coprocessor whose device ID is extracted from `COIProcess*` by function `GetDeviceID()`.

2.5.2 Process swapping

Fig. 2.6 shows sample process swapping-out and swapping-in functions and their timing diagrams. Process-swapping can be used, for example, by a job scheduler to swap out one offload process and swap in another based on the scheduler's scheduling and resource management policies. Both of the swapping functions are called in the context of the host process. The caller of `snapify_-swapout()` needs to prepare a directory where the snapshot files of the offload process can be stored, and passes the path in parameter `snapshot` to `snapify_swapout()`. The implementation of `snapify_swapout()` is fairly straightforward: we call `snapify_pause()`, `snapify_capture()`, and `snapify_wait()` one by one. Since the offload process is to be swapped out, we set the second parameter of `snapify_capture()` to be true, terminating the offload process after its snapshot is captured and saved. The returned pointer of `snapify_t` structure from `snapify_-swapout()` represents a snapshot of the offload process. It can be used to restore the offload process.

The swapping-in of an offload process reverses the effect of swapping-out. In `snapify_-swapin()`, we use `snapify_restore()` to restore the offload process. The returned Snapify data structure `snapshot` from `snapify_swapout()` is passed to `snapify_swapin()`, which uses the path of the snapshot files in `snapshot` to restart the the offload process on the specified Xeon Phi coprocessor (identified by `device_to` parameter). The new handle to the restored offload process is returned at the end of `snapify_swapin()`.

2.5.3 Process migration

Fig. 2.7 shows the implementation of a process-migration function. Process migration moves an offload process from one coprocessor to another on the same machine. It can be viewed as swapping out the offload process from coprocessor 1 and swapping it in on coprocessor 2. Its implementation simply reuses `snapify_swapout()` and `snapify_swapin()`.

2.5.4 Command-line tools

The offload applications can directly benefit from Snapify without any modifications. Checkpoint and restart can be applied transparently by using BLCR's `cr_checkpoint` command-line tool on the host system. This utility will send a signal to trigger the checkpoint procedure, which calls Snapify's own BLCR callback function as shown in Fig. 2.5(a). If the MPI runtime supports BLCR, MPI applications using Xeon Phi for offload computation will automatically benefit from Snapify.

In order to provide swapping and migration transparently, we provide a command-line utility named `snapify`. Its arguments are the PID of the host process and a command. The commands include swapping-out, swapping-in, and migration. In case of swapping-in and migration, `snapify` also needs an additional parameter indicates the coprocessor number on which the offload process will be launched. This utility signals the host process and submits the command through a pipe. The signal handler provided by Snapify in the host process then calls one of the three functions in Fig. 2.6(a) and 2.7 according to the user-given command.

Remark. Process swapping and migration may lead to resource contentions. E.g., two processes might be swapped into the same Xeon Phi. Such problems are best addressed by a job scheduler like COSMIC in [23], and are beyond the scope of this work.

2.6 Snapify-IO

All of the snapshots taken on the host and on a Xeon Phi coprocessor are saved to a file system mounted in the host OS. Snapify provides three novel "on-the-fly" approaches to store and retrieve snapshots between the host and coprocessors. All of these methods use very little Xeon Phi memory for buffering. In the following we will first describe the most efficient approach based on SCIF's RDMA API, called Snapify-IO. Then we will discuss our NFS-based methods.

2.6.1 Snapify-IO.

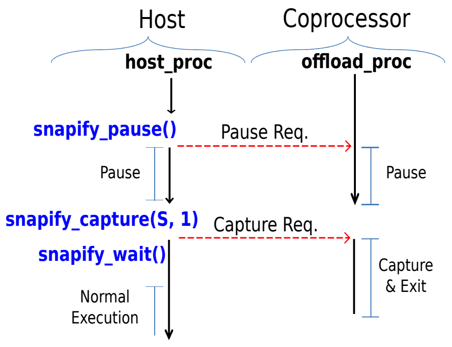
Snapify-IO is a remote file access service that transfers data using RDMA between the host and the Xeon Phi coprocessors on a Xeon Phi server. It provides a simple interface that uses UNIX file descriptors as data access handles. Snapify-IO allows a local process running on a Xeon Phi coprocessor to read from or write to a remote file on the host through standard file I/O functions, as if the file is local. For example, the file descriptor created by Snapify-IO can be directly passed to BLCR for saving and retrieving snapshots. Internally, Snapify-IO transfers the data over the PCIe bus using SCIF's RDMA data transfer functions.

Fig. 2.8 shows Snapify-IO's architecture. Snapify-IO consists of a user-level library providing a simple I/O interface (Snapify-IO library) and a standalone binary called *Snapify-IO daemon*. The Snapify-IO library is linked to the user code that wants to use Snapify-IO for remote file I/O, while each SCIF node (the host and any of the Xeon Phi coprocessors on a Xeon Phi server) runs a Snapify-IO daemon as a long-running process. The Snapify-IO daemon serves I/O requests from both the local user processes using Snapify-IO library and remote Snapify-IO daemons. It can either receive data from a local process, transfer the data to a remote Snapify-IO daemon, which in turn saves the data into a remote file system. Or it can retrieve data from a

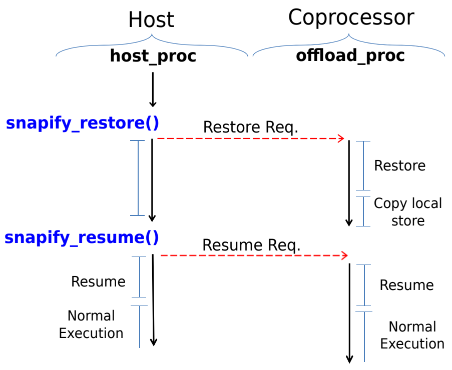
```
snapify_t*  snapify_swapout(const char* path, COIProcess* proc){
    snapify_t* snapshot = (snapify_t*)malloc(sizeof(snapify_t));
    snapshot->m_snapshot_path = path;
    snapshot->m_process = proc;
    snapify_pause(snapshot);
    snapify_capture(snapshot, true);
    snapify_wait(snapshot);
    return snapshot;
}

COIProcess* snapify_swapin(snapify_t* snapshot, int device){
    COIProcess* ret = 0;
    snapify_restore(snapshot, device);
    snapify_resume(snapshot);
    ret = snapshot->m_process;
    free(snapshot);
    return ret;
}
```

(a) Sample code.



(b) Swapping-out timing diagram.



(c) Swapping-in timing diagram

Figure 2.6: Using Snapify's API to implement process swapping.

```
COIProcess* snapify_migration(COIProcess* proc, int device_to){
    const char* path = "/tmp";
    snapify_t* snapshot = snapify_swapout(path, proc);
    return snapify_swapin(snapshot, device_to);
}
```

Figure 2.7: An implementation of process migration.

local file system, transfer the data to a remote Snapify-IO daemon, which feeds the data into a remote user process.

Snapify-IO library is designed for transparent integration with the standard `read()` and `write()` system calls. Its only API function `snapifyio_open()` returns a standard UNIX file descriptor. It accepts three arguments: a SCIF node ID, a path to a file that is valid on the SCIF node, and a file access mode flag indicating either a read or write mode (but not both). The returned file descriptor represents a file on a (remote) SCIF node as specified by the arguments.

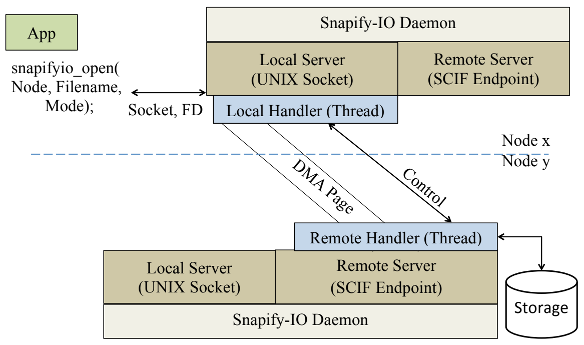

App
snapifyio_open(Node, Filename, Mode);
Socket, FD
Snapify-IO Daemon
Local Server (UNIX Socket)
Remote Server (SCIF Endpoint)
Local Handler (Thread)
Node x
Node y
DMA Page
Control
Remote Handler (Thread)
Local Server (UNIX Socket)
Remote Server (SCIF Endpoint)
Snapify-IO Daemon
Storage

Figure 2.8: The architecture of Snapify-IO.

Snapify-IO uses a UNIX socket as the local communication channel between the Snapify-IO library and the Snapify-IO daemon. When `snapifyio_open()` is called, the Snapify-IO library creates a UNIX socket that connects to the local Snapify-IO daemon. Once the socket is established, `snapifyio_open()` sends the SCIF node ID, the file path, and the access mode to the Snapify-IO daemon. It then returns the file descriptor of the socket to the caller. To serve the local socket connections, the Snapify-IO daemon has a local server thread listening on a designated port. Once the server thread accepts the socket connection from the Snapify-IO library, it spawns a local handler thread to handle further I/O activities coming from the user process, which may either write to or read from the socket, depending on the access mode.

Notice that the file descriptor returned by `snapifyio_open()` is a UNIX file descriptor, so the user code can call `close()` to release the resources associated with the file descriptor.

The communication channel between two Snapify-IO daemons is a SCIF connection. After receiving the SCIF node ID, the file path, and the access mode from a local user process, the Snapify-IO daemon's local handler thread will create a new SCIF connection to the Snapify-IO daemon on the specified SCIF node. Once the SCIF connection is established, the local handler thread will forward the path and the access mode of the file to the remote Snapify-IO daemon, and register an internal buffer to the SCIF library for RDMA transfer. The buffer size is configurable. To balance between the requirement of minimizing memory footprint and the need of shorter transfer latency, the buffer size is set at 4MB. To handle incoming SCIF connections, the Snapify-IO daemon employs a remote server thread, which listens to a predetermined SCIF port. Once the remote server thread accepts a SCIF connection from a remote Snapify-IO daemon, it spawns a handler thread to handle communications over the newly established SCIF channel.

Once the communication channels are established, the local handler thread will start to direct the data flow between the user application and the remote file. In write access mode, the local handler will copy the data written to the socket by the user application to the registered RDMA buffer. After the buffer is filled, the local handler will send a SCIF message to notify the remote Snapify-IO daemon (i.e. the daemon's remote handler thread) using `scif_send()`. Subsequently the remote handler thread will use SCIF's RDMA function `scif_vreadfrom()` to read the data from the registered RDMA buffer, and saves the data to the file system at the specified location. After the RDMA completes, the local handler thread will reuse the RDMA buffer and repeat the above process until all data from the user process are saved to the remote file. In the read access mode, the data flow in the reverse direction. The remote handler thread in the remote Snapify-IO daemon will read data from the specified file, and copy the data to the registered RDMA buffer using `scif_vwriteto()`. Once the buffer is filled, it will notify the local handler thread in the local Snapify-IO daemon, which in turn will copy the data from the RDMA buffer to the socket.

Snapify uses Snapify-IO to save and retrieve snapshots. The COI library with Snapify implementations is linked with the Snapify-IO library. To take a snapshot of an offload process, Snapify calls `snapifyio_open()` in write mode in the pre-snapshot phase. The returned file descriptor is then passed to BLCR, which uses it to write the context file. Similarly, to restart an offload process Snapify calls `snapifyio_open()` to open a remote context file in read mode. The returned file descriptor is then used by BLCR to load the process context from the remote file to the local Xeon Phi's memory. Thanks to Snapify-IO, the data transfer between a Xeon Phi coprocessor and the host's file system is completely transparent to BLCR. In addition, the

Table 2.2: Characteristics of our Xeon Phi server.

	Host Processor	**Coprocessor**
CPU	Intel E5-2630 @ 2.30GHz	Intel Xeon Phi 5110P
Cores	6 physical cores (12 threads) per socket	60 physical cores (240 threads) per coprocessor
Memory	32GB	16GB per coprocessor
OS	Linux RHEL 6.2, kernel 2.6.32-220	Linux kernel 2.6.38.8 MPSS 2.1.6720-130
Number	2 CPU sockets	2 coprocessors

Snapify-IO library does not introduce extra SCIF connections. Therefore, it does not complicate the process of taking a snapshot.

2.6.2 NFS.

We also implemented two optimizations to speedup NFS-based snapshot storage. In these approaches Snapify uses NFS to mount the host file system on a Xeon Phi coprocessor. A snapshot is stored or retrieved through the NFS. To overcome the problem of high latency of small writes in NFS, we developed two new approaches based on buffering. In the first approach we modified BLCR's kernel module such that it accumulates write data to a larger chunk before the data is written to the file system. Since using a modified BLCR's kernel module may not always be feasible, our second approach uses the same concept but applies it in the user level. In this approach the BLCR writes are redirected to our user-space utility through the standard output and input. This utility buffers data from its standard input and writes out the data in the buffer to NFS at larger granularity.

2.7 Experimental Results

We used micro benchmarks and a suite of MPI and OpenMP applications to evaluate Snapify-IO and Snapify on a Xeon-Phi cluster. This section reports our experimental results.

2.7.1 Setup

Table 2.2 shows the hardware and software configuration of our computing system. For Snapify-IO evaluation, we used a single node that has one Xeon Phi (8GB of physical memory). For MPI applications, we used a 4-node cluster, where each node in the cluster has one Xeon Phi many-core processor with 8GB of memory.

Table 2.3: File copy performance (seconds).

Size	**Device to Host**			**Host to Device**		
(MB)	scp	NFS	Snapify-IO	scp	NFS	Snapify-IO
1	0.97	0.01	0.03	0.48	0.01	0.07
64	15.07	3.23	0.43	14.08	1.94	0.53
128	29.88	6.36	0.85	28.30	3.96	1.82
256	57.02	11.81	1.67	55.50	7.95	3.68
512	113.71	20.91	3.34	111.07	16.07	6.17
1024	224.68	40.10	6.76	221.59	31.06	9.57

Table 2.4: Comparing Snapify-IO with NFS-based I/O in BLCR (seconds).

	Checkpoint time (Snapshot + Write)					**Restart Time (Read + Restore)**		
Size (MB)	Local (RAM)	NFS	NFS-Buffered (kernel)	NFS-Buffered (user)	Snapify-IO	Local (RAM)	NFS	Snapify-IO
1	2.16	67.64	1.70	2.71	**2.76**	0.577	1.392	**0.979**
64	2.38	71.21	3.25	4.81	**2.28**	0.829	3.547	**1.377**
128	2.62	70.97	4.30	7.64	**3.71**	0.832	5.834	**2.066**
256	3.15	75.67	7.30	12.48	**4.59**	1.255	8.975	**3.397**
512	4.12	80.98	12.60	20.98	**6.44**	2.034	17.613	**4.934**
1024	6.34	87.27	21.44	30.47	**9.94**	3.732	32.550	**7.373**
2048	10.14	118.65	38.48	53.82	**17.41**	6.736	62.217	**13.506**
4096	NA	155.92	93.05	109.06	**32.84**	NA	121.452	**22.935**

2.7.2 Snapify-IO performance

To evaluate the performance of Snapify-IO, we used a micro-benchmark that copies files of various sizes between the host and the Xeon Phi. The micro-benchmark runs natively on the Xeon Phi. We compared the time taken by our Snapify-IO to move files between the host and the Xeon Phi with the time taken by two methods natively supported by Xeon Phi's OS, i.e. `scp` and read/write from NFS mounted directories.

Table 2.3 shows the time taken to copy files of different sizes (file-size ranged from 1MB to 1GB). We observed that Snapify-IO consistently performs better than NFS and `scp` (except for the 1 MB file-size case, where NFS outperforms others by buffering data). As the file size increases, Snapify-IO's advantage is more pronounced. For a 1GB file, Snapify-IO has about 6x better write performance and 3x better read performance when compared with NFS. For the same file, Snapify-IO has 30x faster write performance and 22x faster read performance when compared with `scp`. We also observed that transfer of a file from the Xeon Phi to the host by using Snapify-IO is generally faster than moving the same file from the host to Xeon Phi. This is because Snapify-IO daemon on the host flushes the file to the secondary storage asynchronously. Thus the write operation on the host runs parallel to the data transfer.

We also evaluated the impact of using Snapify-IO for storing and restoring BLCR's checkpoints of processes on the Xeon Phi. We ran a second micro-benchmark as a native application on the coprocessor and captured the snapshots using BLCR. Our micro-benchmark performed a `malloc()` call and it had a long loop in an OpenMP region (240 threads). We used different malloc sizes (ranging from 1MB to 4GB) to control the file-size of snapshots. Table 2.4 compares the performance of Snapify-IO with three variants of read/write from NFS mounted directories, as well as a method (labeled as *Local* in Table 2.4) that saves the application snapshot in the physical memory of the Xeon Phi. The BLCR checkpoint time in Table 2.4 is the end-to-end latency of capturing and saving the process snapshot. BLCR restart time is the end-to-end time to read and restore the snapshot. Note that in most cases the snapshot is written to and read from the host file system (except the *Local* case). As expected, storing and restoring snapshots from the physical memory of the Xeon Phi (the *Local* case) takes the least time. However, when the checkpoint file-size increases to 4GB, it is impossible to store the checkpoint file in the physical memory of the Xeon Phi (memory limit on the Xeon Phi card is 8GB, and 4GB is already used by the micro-benchmark). In practice, it is not feasible to save checkpoint files locally because, more often than not, several other processes on the Xeon Phi are already using the limited physical memory on the Xeon Phi.

The performance of all three variants of NFS was poor (when compared with Snapify-IO) for storing checkpoints. BLCR performs multiple small writes before reaching the loop where it actually takes snapshots of the application's memory pages, and these small writes lead to poor performance for the NFS variants. Our method of "NFS-Buffered in kernel" boosts the performance of NFS to a large degree while our buffering in user-space does so to a lesser degree (but it still provides significant improvements). Finally, Snapify-IO performance is a large improvement compared to NFS and NFS-Buffered (both modes).

Note that the buffering solutions do not apply to the cases of restarting or restoring. Again, restarting from checkpoint files in the physical memory of the Xeon Phi is very fast, but this is generally not possible due to the limited physical memory on the Xeon Phi. We observed that Snapify-IO performs 1.4x, 2.6x and 5.9x faster than NFS for 1MB, 256MB and 4GB snapshots, respectively.

Table 2.5: Description of benchmarks.

Name	**Description**	**Problem Size**
MD	Molecular dynamics simulation	25000 particles, 10 time steps
MC	Monte Carlo simulation of N paths and T time steps	N = 32M, T = 2000
SG	A series of matrix-matrix multiplications (SGEMM)	8Kx8K matrices, 10 iterations
SS	Supervised semantic search indexing computing top K for each of the Q queries	256K documents, K=32, Q=512
KM	Computing K-means using Lloyd clustering algorithm	4M points, 3 dimensions, 32 means
LU-MZ	A CFD application using lower-upper Gauss-Seidel solver [12]	Grid: 162x162x162, 250 iterations
BT-MZ	Computation fluid dynamics (CFD) using block tri-diagonal solver [12]	Grid: 162x162x162, 200 iterations
SP-MZ	A CFD application using scalar penta-diagonal solver [12]	Grid: 162x162x162, 400 iterations

2.7.3 Snapify overhead

We evaluated Snapify on 8 OpenMP and 3 NAS MPI benchmarks (LU-MZ, SP-MZ, BT-MZ in [43]). The benchmarks are described in Table 2.5. All benchmarks were modified to offload computations to the Xeon Phi. All time measurements were made on the host, unless mentioned otherwise.

Fig. 2.9 compares the runtime of the normal executions (no snapshot) of the OpenMP benchmarks with and without Snapify support. Each experimental run was repeated 20 times. The average runtime is reported as bars, and the runtime overhead (in percentage) added by Snapify is shown in line graph on the right y-axis. In average Snapify adds a 1.5% overhead to the application runtime, and in the worst case the overhead is less than 5% (MD). We used the Linux command-line tool `time` to measure the end-to-end execution time of an offload application.

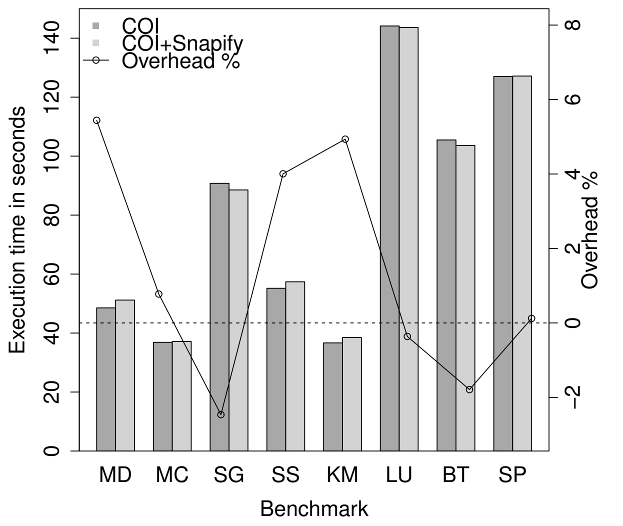
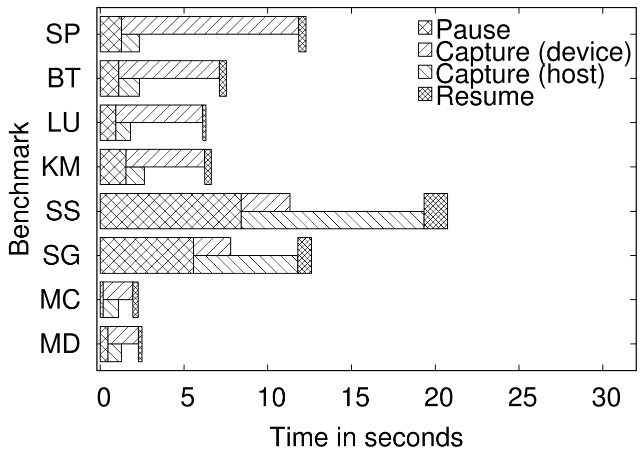


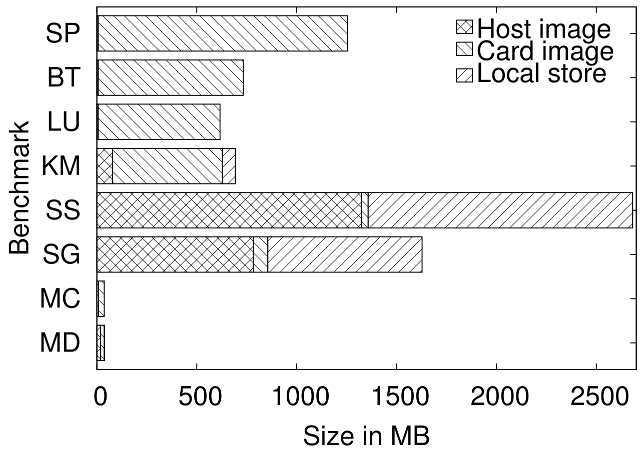
Figure 2.9: Runtime overhead of Snapify.

2.7.4 Checkpoint and restart

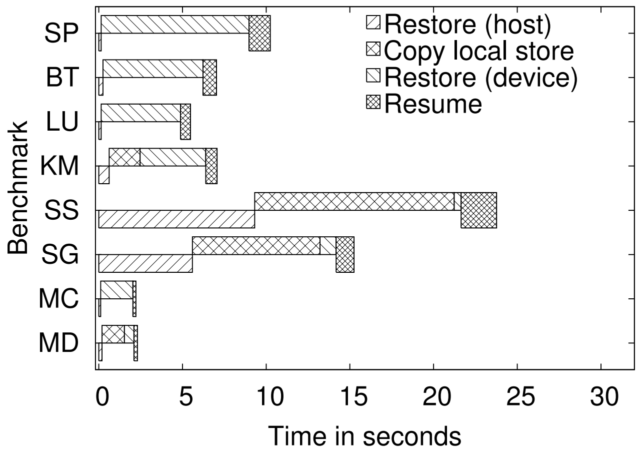
Fig. 2.10(a) shows the checkpoint time and 2.10(b) reports the size of the files generated by the checkpoint procedure. As detailed in Section 2.4.1, during pause the local store (files on Xeon Phi for COI buffers) of the offload process was stored in the snapshot directory on the



(a) Checkpoint



(b) File size



(c) Restart

Figure 2.10: Performance evaluation on OpenMP benchmarks.

host system as a file. Thus for benchmarks with a large local store (SS and SG in Fig. 2.10(b)), the pause is longer. The time that BLCR on the host and BLCR on the coprocessor take to capture and save snapshots are the bars labeled as “Snapshot + Write (host)” and “Snapshot + Write (device)”, respectively. The host BLCR finishes early in all cases except for SS and SG. In these two cases, Fig. 2.10(b) indicates a large host-process snapshot while the offload-process snapshot is fairly small. Thus the offload process finishes early. The checkpoint time ranges from 3 to 21 seconds in time, shorter for small files (8.4 MB) and longer for large ones (1.3 GB).

Fig. 2.10(c) reports the restart time of the OpenMP benchmarks. The total restart time ranges from 3 to 24 seconds across the benchmarks. Fig. 2.10(c) also reports the breakdown of the time spent in each stage of the restart. The host-restart time varies based on the size of the host-process snapshot. Benchmarks SS and SG have larger host snapshots, and thus longer host-restart time. The time of restoring an offload process strongly depends on the size of local store, which is copied from the host to the coprocessor when the offload process is restored.

2.7.5 Process migration

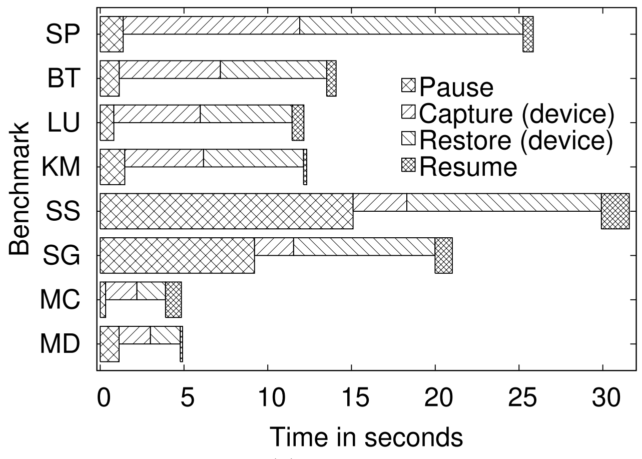
Fig. 2.11(a) shows the runtime overhead of process migration. The migration time varies from 4.9 seconds (MC) to 31.6 seconds (SS). As expected, it is strongly correlated with the size of the local store and the snapshot of an offload process. In process migration, the offload process copies its local store directly from its current coprocessor to another coprocessor using Snapify-IO. Thus the pause time in process migration is different from the one in the checkpoint procedure. In all but one benchmarks the time of capturing and saving the snapshot of an offload process is shorter than the time of reading the snapshot and restoring the offload process. This is because Snapify-IO is faster when writing to the host from a coprocessor, as explained earlier.

2.7.6 Process swapping

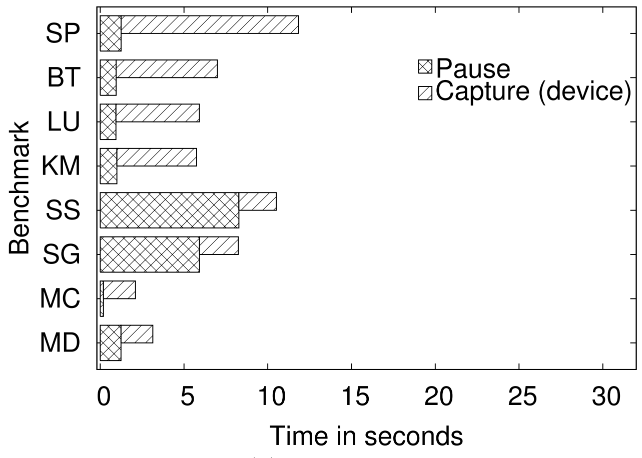
Fig. 2.11(b) and 2.11(c) show the runtime of swapping-out and swapping-in, respectively. The time of swapping out the offload processes ranges from 2.1 seconds to 11.8 seconds, and the time of swapping-in takes between 2 seconds and 14.8 seconds. Except in the case of SS and SG, the pause of swapping-out is much shorter than the time of the capturing phase. Again, this is because the local stores of SS and SG are larger than their snapshots.

2.7.7 Checkpoint and restart for MPI

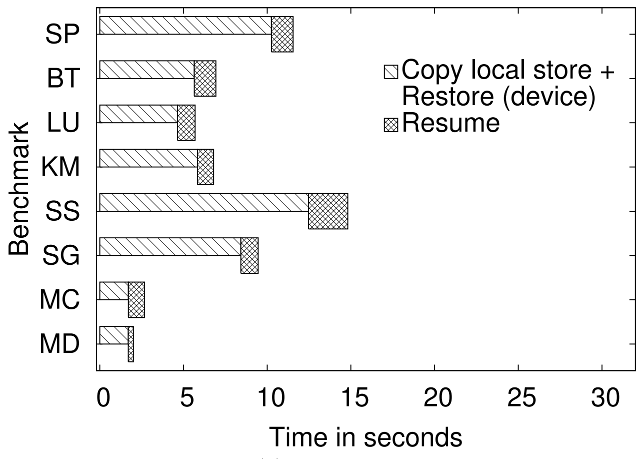
We use three MPI benchmarks LU-MZ, SP-MZ, and BT-MZ, to evaluate checkpoint and restart of MPI applications. For all three benchmarks, we choose the class C input size, and run the benchmarks with 1, 2 and 4 MPI tasks (ranks). Each rank is executed on one node. Fig. 2.12(a) and 2.12(b) report the time of taking a checkpoint and restarting from a checkpoint, respectively.



(a) Migration



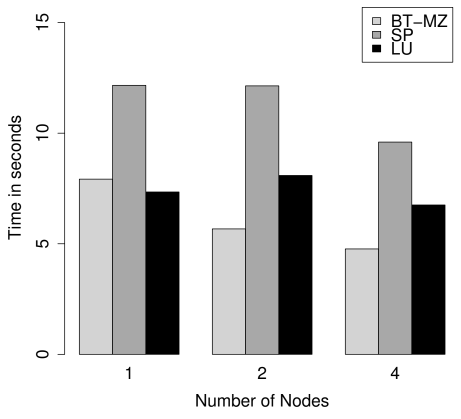
(b) Swapping-out



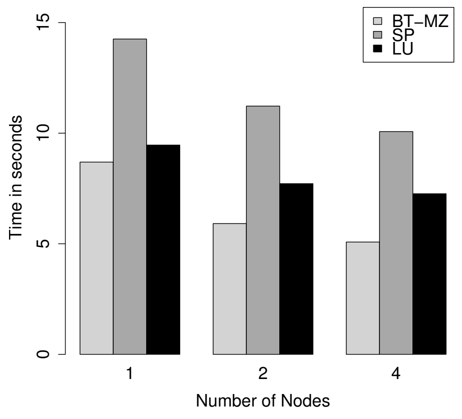
(c) Swapping-in

Figure 2.11: Performance evaluation on OpenMP benchmarks.

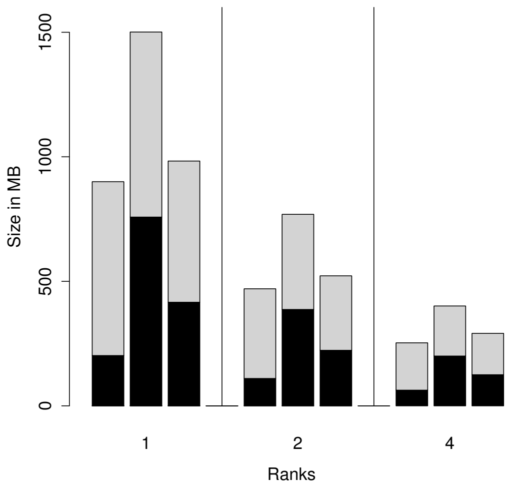
Fig. 2.12(c) shows the checkpoint size of a single rank. We observe that as the number of nodes increases, CR time decreases at various degrees. This is because the checkpoint size of each MPI rank decreases as the total number of MPI ranks increases. CR time ranges between 4 and 14 seconds for a single checkpoint, depending on the respective benchmarks and the number of the MPI ranks. When no checkpoint is performed, the runtime of the benchmarks ranges from 2-3 minutes for the selected input size. This indicates the feasibility to taking frequent checkpoints, particularly for larger inputs and longer runtime.



(a) Checkpoint



(b) Restart



(c) File size

Figure 2.12: Performance evaluation of checkpoint and restart on MPI benchmarks.

2.8 Related Work

Checkpoint and restart (CR) has a long history in computing systems. Libckpt [104] was one of the first UNIX implementations. Condor [129] provides CR and process migration for load balancing. These libraries provide process-level checkpointing. There are also user-level checkpointing libraries [45, 65, 75], and recent studies suggest using nonvolatile memory to improve CR performance [81]. Rollback-recovery protocols in message-passing systems is a classical research field [54]. Multiple MPI libraries, including Open MPI [69] and MPICH-V [21], provide distributed CR for MPI applications.

Several previous studies proposed CR and process migration for GPUs. CheCUDA supports a part of basic CUDA APIs [127]. It copies all the user data from a GPU to the host system during checkpointing. Then it destroys the CUDA context before taking a checkpoint. The data and context are copied back from the host to the GPU at post-checkpoint (restart) time. NVCR keeps a database of the memory allocations in GPUs [98]. Before checkpointing it releases all the memory contents and replays the log at the restart time. The replay is necessary to avoid invalid memory addresses at restart time. This imposes overhead during restart time and specially normal execution of application. CheCL provides CR and migration for OpenCL [128]. CheCL synchronizes the host and command queues by waiting for all commands to complete. A command queue is used to schedule the execution of kernels and perform memory operations in OpenCL context. CheCL benefits from a proxy mechanism to decouple the process from OpenCL implementation. However, each kernel execution involves an additional step of inter-process communication, incurring extra communication latency.

Process migration has been extensively studied in the past. Zap is a system that performs process-group migration [99], while Wang et al. studied live migration of processes in HPC environment [131]. In addition, live migration of entire virtual machines is a very useful tool for data center and cluster administrations [38].

2.9 Conclusion

To conclude, in this chapter we presented Snapify, a set of extensions to Xeon Phi's software stack that captures process snapshots of offload applications. Using Snapify we implemented application-transparent checkpoint and restart, process migration, and process swapping for offload applications. Experimental results on OpenMP and MPI offload applications show that Snapify added negligible runtime overhead (1.5% in average) and is very efficient in taking snapshots and restoring processes.

We also created Snapify-IO, a remote file access service based on RDMA to transfer process snapshots between the host system and coprocessors. Snapify-IO benefits both Snapify *and* the default checkpoint and restart tool for native applications. For native applications our experimental results show that Snapify-IO achieves 4.7x to 8.8x speedup in checkpoint, and 4.4x to 5.3x speedup in restart over NFS, for snapshot size between 1GB to 4GB.

Chapter 3

DINO: Divergent Node Cloning for Sustained Redundancy in HPC

3.1 Introduction

Reliability has been highlighted as a key challenge for next generation supercomputers [16, 27, 46]. Node failures are commonly due to hardware or software faults. Hardware faults may result from aging, loss of power, and operation beyond temperature thresholds. Software faults can be due to bugs (some of which may only materialize at scale), complex software component interactions and race conditions that surface only for rare parallel execution interleavings of tasks [39].

One resilience method is redundant computing [22, 57, 55, 59]. It aims at improving reliability and availability of systems by allocating two or more components to perform the same work. A recent study [91] showed that redundancy could be a very viable and even cost-effective approach for HPC on the cloud. Their approach to combine checkpointing and redundancy on Amazon EC2 using a variable-cost spot market provides up to 7 times cheaper execution compared to the on demand default market. Redundancy provides tolerance not only against hard faults but also soft errors, such as Silent Data Corruptions (SDCs), which do not stop application execution as they are undetectable. SDCs may manifest at application completion by producing wrong results or, prior to that, wrong interim results. A study at CERN raised concerns over the significance of SDC in memory, disk and RAID [101]. Their results indicate that SDC rates are orders of magnitude larger than manufacture specifications. Schroeder et al.'s study [119] of the DRAM errors on a large scale over the course of 2.5 years concludes that more than 8% of DIMMs are affected by errors per year. SDCs can be detected by Dual Modular Redundancy (DMR) and can be corrected with voting under Triple Modular Redundancy (TMR) [59].

Current approaches for redundant computing do not provide a sustained redundancy level during job execution when processes are hit by failures. When a replica fails, either the application deadlocks (RedMPI [59]) or other replicas ensure that the application can progress in execution [22]. Note that after a replica failure, even if the job can continue its execution, the SDC detection module cannot guarantee application correctness (e.g., an undetected SDC might occur). Checkpoint/Restart (CR) is another popular method for tolerating hard errors but cannot handle soft errors. CR takes snapshots of all processes and saves them to storage. Should a hard error occur, all processes re-load the last snapshot into memory and the application continues.

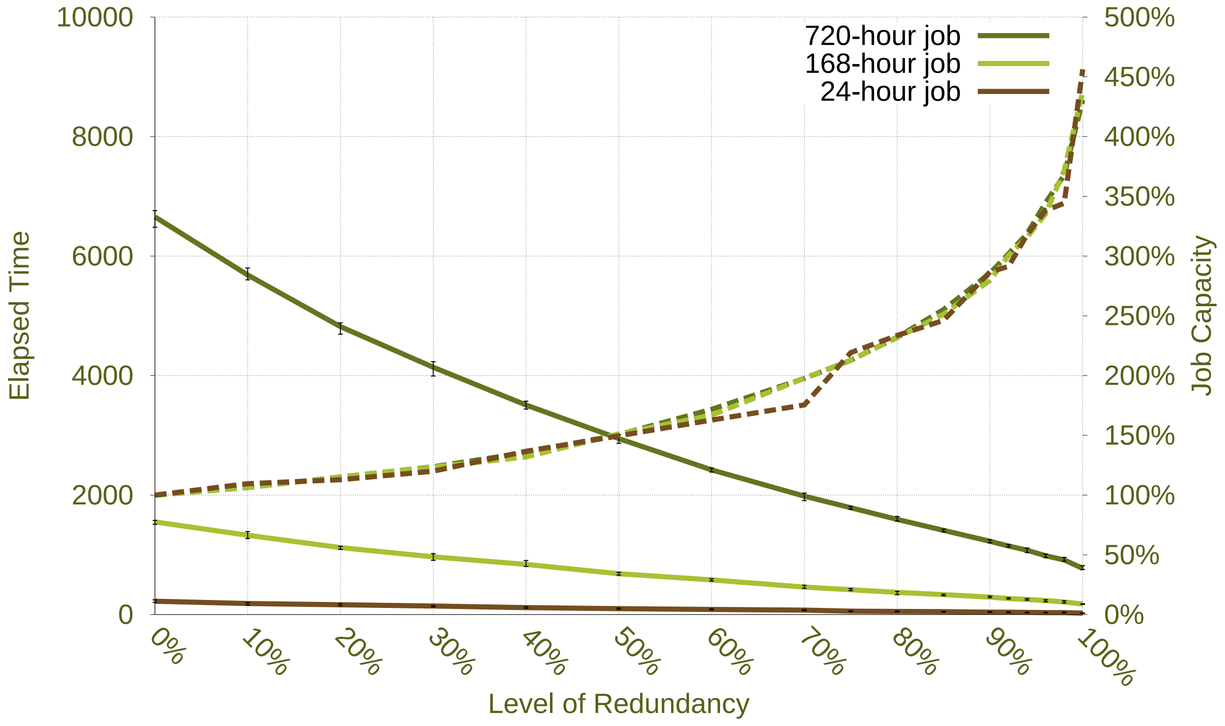


Figure 3.1: Modeled application runtime and job capacity per redundancy level for 100,000 nodes

Using the same simulation parameters as Ferreira et al. [57], we plotted in Fig. 3.1 the elapsed times (left x-axis) of three jobs of 24/168/720 hours over different redundancy levels (y-axis) ranging from no redundancy (0%) over half of the nodes replicated (50%) to dual redundancy for all nodes (100%). The results (dashed lines) show that for a 720-hour job applications runtime without replication is more than six times higher than under dual redundancy. Due to this, job capacity (2nd x-axis, dashed lines) increases up to a factor of 4.5 under dual redundancy, i.e., 4.5 times more dual redundant jobs of the same size can finish execution using the same resources in the time it would have taken to finish a non-redundant job due to checkpoint and (mostly) restart overheads. This shows that replication has the potential to outperform CR around the exascale range.

Elliott et al. [51] showed in a more refined model that CR will eventually take longer than redundancy due to recomputation, restart and I/O cost. At scale, this makes capacity computing (maximizing the throughput of smaller jobs) more efficient than capability computing (using all nodes of an exascale machine). E.g., at 80,000 CPU sockets, dual redundancy will finish twice the number of jobs that can be handled without redundancy. This includes a redundancy overhead of 0-30% longer time (due to additional messages) with hashing protocols [59], which has no impact on bandwidth for Dragonfly networks since original and replica sphere exchange full messages independently. As hash messages are small, they add latency but do not impact bandwidth. Since twice the jobs finished under dual redundancy, this amounts to the *same* energy.

This work targets tightly-coupled parallel applications/jobs executing on HPC platforms using MPI-style message passing [63]. We use the term *rank* to refer to an MPI task/process. In MPI-style programming, each MPI process is associated with a unique integer value identifying the rank. Suppose there is a job with n ranks that requires t hours to complete without any failures. This is called plain execution time. We consider systems with r levels of redundancy (at the rank level). Our system then consists of $r \times n$ ranks, where n logical MPI tasks are seen by the user while redundant replicas remain transparent. There is no difference between replicas of the same task in terms of functionality as they perform the same operations. We also assume the availability of a small pool of spare nodes. Spare nodes are in a powered state but initially do not execute any jobs. We assume that a fault detector is provided by the system. We focus on the recovery phase and consider works in failure detection [133, 90] orthogonal to our work.

We introduce node cloning as a means to sustain a given redundancy level. (We use the terms node / MPI task cloning synonymously.) The core idea is to recover from hard errors with the assistance of a healthy replica. A healthy replica is cloned onto a spare node to take over the role of the failed process in "mid-flight". To address shortcomings in current redundant systems, we provide the following contributions:

- We devise a generic high performance node cloning service under divergent node execution (DINO) for recovery. DINO clones a process onto a spare node in a live fashion. We integrate DINO into the MPI runtime under redundancy as a reactive method that is triggered by the MPI runtime to forward recover from hard errors, e.g., node crash or hardware failure.

- We propose a novel Quiesce algorithm to overcome divergence in execution without excessive message logging. Execution of replicas is not in a lock-step fashion, i.e., can diverge. Our approach establishes consistency through a novel, scalable multicast variant of the traditional (non-scalable) bookmark protocol [115] and resolves inconsistencies through exploiting the symmetry property of redundant computing.

- We evaluate DINO's performance for MPI benchmarks. The time to regain dual redundancy after a hard error varies from 5.60 seconds to 90.48 seconds depending on process image size and cross-node transfer bandwidth, which is short enough to make our approach practical.

- We provide a model with low error (at most 6%) for estimating job execution time under redundancy and validate our model on the Stampede supercomputer [2]. We extrapolate results under this model to extreme scale with a node MTTF of 50 years [60] and show that dual redundancy+cloning outperforms triple redundancy within the exascale node count range, yet at 33% lower power requirement.

- We show that 25 spare nodes suffice for a 256K node system (when nodes can be repaired) independent of communication overhead of the applications.

The chapter is structured as follows: Challenges are discussed in Section 3.2. Section 3.3 presents a brief background on RedMPI. Section 3.4 introduces the design of DINO. Our Quiesce algorithm is presented in Section 3.5. Section 3.6 details the implementation of DINO. Analysis of job completion times are presented in Section 3.7. The experimental evaluation is provided in Section 3.8. Simulation results are presented in Section 3.9. Section 3.10 compares our work to related work. Section 3.11 summarizes the chapter.

3.2 Challenges

High performance node cloning service. With increasing memory capacity and the applications' tendency to load larger data into memory, a cloning service that limits the interference with the process execution is more preferable. A generic service that clones a process (including the runtime/kernel state, memory content, and CPU state) onto a spare node is the first challenge. Off-the-shelf checkpointing libraries like BLCR [49] or MTCP [107] provide the needed functionality. However, the downtime to take the snapshot, save it to storage and retrieve it from storage on the other node followed by restoring the snapshot is quite high. We compare "CR-based" process cloning with our "live" process cloning in Section 3.8.
Communication consistency and divergent states. The execution of replica ranks does not occur in a lock-step fashion, i.e., they tend to diverge not within computational regions

but also by advancing or falling behind relative to one another in terms of communication events. This provides a performance boost to redundant computing as it allows the replicas to execute freely to some extent. However, this divergence introduces complications for failure recovery. Lack of a proper approach that guarantees the consistency, results in deadlock. This would generally require message logging over large periods of time. Instead, we devised a novel algorithm to establish communication consistency that tolerates divergence due to asymmetric message progression and region-constrained divergence in execution.

Extending a job to a spare node. The MPI runtime leverages helper daemons on the compute nodes to facilitate job execution. The helper daemons launch the MPI ranks, monitor their liveness and play a role in application termination. The spare nodes are not part of a specific job, and consequently, no daemon is running on them. A spare node might be assigned to any job that has a failed replica as a part of the recovery phase in DINO. Thus, extending the job includes modifying the internal data structures like routing information, task map and daemon map information besides spawning a new daemon.

Integration into the runtime system. Cloning a process without carefully resuming its communication state results in inconsistency and job failure. After the process is cloned onto a spare node, it should exchange its communication endpoint information with the rest of the processes in order to make normal application progress.

3.3 Background (RedMPI)

We use RedMPI [59] for redundant and transparent MPI execution. RedMPI detects SDCs through analyzing the content of messages. Data corruption may occur in CPU, memory or cache, either due to multi-bit flits under ECC or even single-bit flips without ECC. Such corruption either does not affect the output or eventually materializes in the transmitted message (or in disk/std I/O, which is addressed elsewhere [18]).

It supports linear collectives where all the collective operations are mapped to point-to-point communication. These collectives do not provide high performance, especially not at large scale. Another mode, named *optimized*, directly calls the collective module of the MPI library and provides native performance (parallelized collectives). RedMPI benefits from the interpositioning layer of MPI known as the MPI profiling layer.[1] The current implementation of RedMPI is not capable of detecting hard errors. As a result, an MPI process failure leads to a deadlock where the processes wait indefinitely for progress in communication with the failed process (or until the time-out from the communication layer terminates the application with an error).

[1] PMPI is the MPI standard profiling interface. Each MPI function can be called with a MPI_ or PMPI_ prefix. This feature of MPI allows one to interpose functions with MPI_ prefix while using functions with PMPI_ prefix to implement the required functionality.

In the following, we describe how basic MPI functions are implemented in RedMPI under dual redundancy. `MPI_Send` and `MPI_Recv` are blocking calls, and `MPI_Isend` and `MPI_Irecv` are non-blocking. `MPI_Send` is implemented with two non-blocking send calls to rank x and its corresponding replica x' followed by a call to `MPI_Waitall` (see Table 3.1). `PMPI_Isend` provides the actual send functionality. Similarly, `MPI_Recv` is implemented with two `PMPI_Irecv` calls, then a `MPI_Waitall` call. RedMPI extends the `MPI_Request` data structure to contain the extra requests that it creates. For the sake of brevity, this is omitted. `MPI_Isend` and `MPI_Irecv` are similar to `MPI_Send` and `MPI_Recv` but there is no `MPI_Waitall` call. The `MPI_Wait` is implemented with a call to the actual function (`PMPI_Wait`), it then verifies the integrity of the messages in case of a receive request. The implementation of `MPI_Waitall` performs a `PMPI_Waitall` over all requests followed by the integrity check for received messages.

Table 3.1: Implementation of 6 basic MPI functions in RedMPI

MPI_Send(x,msg){
 PMPI_Isend(x,msg);
 PMPI_Isend(x',msg);
 MPI_Waitall();
}
	MPI_Recv(x,msg){
	 MSG msg';
	 PMPI_Irecv(x,msg);
	 PMPI_Irecv(x',msg');
	 MPI_Waitall();
	}

MPI_Isend(x,msg){
 PMPI_Isend(x,msg);
 PMPI_Isend(x',msg);
}
	MPI_Irecv(x,msg){
	 MSG msg';
	 PMPI_Irecv(x,msg);
	 PMPI_Irecv(x',msg');
	}

MPI_Wait(req){
 PMPI_Wait(req);
 if(req.type=="Recv")
  verify_integrity(req);
}
	MPI_Waitall(req){
	 PMPI_Waitall(req);
	 if(req.type=="Recv")
	  verify_integrity(req);
	}

There is no lock-step execution among the replicas and they only communicate directly to resolve certain calls to avoid non-determinism. Wildcard values in source or tag (`MPI_ANY_TAG`, `MPI_ANY_SOURCE`) and `MPI_Wtime` are examples of the latter case (not shown here), which are supported (see [59]). In redundant computing, send (or receive) calls are always posted in pairs. In other words, the number of posted send (or receive) requests to (or from) any two replicas are always equal. We call this the *symmetry* property and exploit it in the Quiesce algorithm (Section 3.5).

3.4 Design of DINO

DINO has a generic process cloning service at its core. Node cloning creates a copy of a given running process onto a *spare* node. The cloning mechanism itself is MPI agnostic and is applied to processes encapsulating MPI tasks in this work. DINO considers the effect of cloning on the MPI runtime system, as detailed later. Fig. 3.2 shows how the system retains dual redundancy in case of a failure. A and A' are logically equivalent and both perform the same computation. They run on nodes 0 and 1, respectively, and comprise *sphere 1* of redundant nodes. Ranks B, B' on nodes $2, 3$ are also replicas and shape sphere 2. If node 2 (B) fails, its replica (B') on node 3 (*source* node) is cloned onto node 4 (a *spare* node) on-the-fly. The newly created rank B'' takes over the role of failed rank B and the application recovers from the loss of redundancy. At the end of node cloning, B' and B'' are in the same state from the viewpoint of the application, but not necessarily from another rank's point of view due to stale B references. The Quiesce algorithm resolves such inconsistencies.

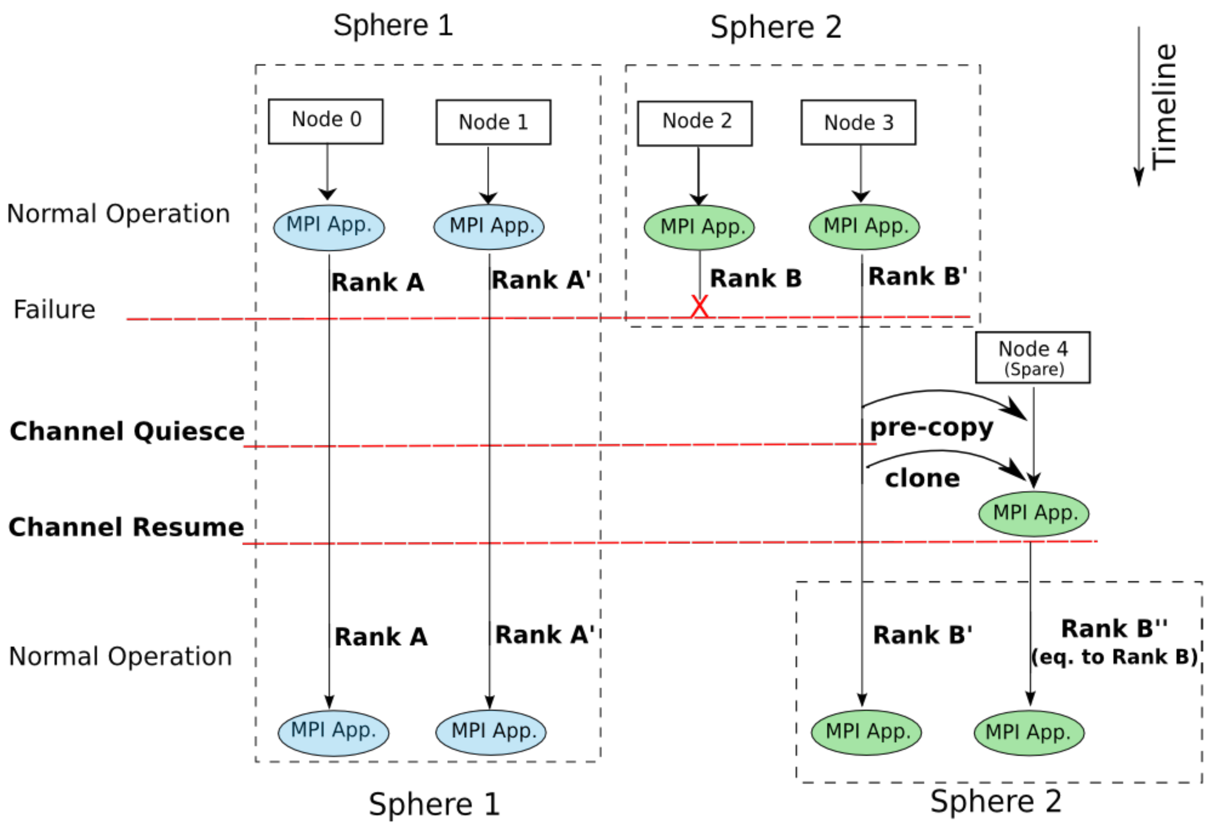


Figure 3.2: Node cloning: Application w/ 2 ranks under dual redundancy

The process B'' is created on node 4 as follows. While B' performs its normal execution, its memory is "live copied" page by page to B''. This happens in an iterative manner (detailed in Section 3.6). When we reach a state where few changes in dirty pages (detailed in the implementation) remain to be sent, the communication channels are drained. This is necessary to keep the system of all communication processes in a consistent state. After this, the execution of rank B' is briefly paused so that the last dirty pages, linkage information, and credentials are sent to node 4. Rank B'' receives and restores this information and then is ready to take over the role of failed rank B. Then, communication channels are resumed and execution continues normally. Between channel draining and channel resumption, no communication may proceed. This is also necessary for system consistency with respect to message passing.

The time interval between error detection and the end of DINO recovery is a "vulnerability window" where undetected SDCs may occur. The vulnerability window depends on the process image size and is evaluated experimentally in Section 3.8 and projected for large scale in Section 3.9.

3.5 Quiesce Algorithm

The purpose of the Quiesce algorithm is to resolve the communication inconsistencies inside DINO at the library level and provide transparent and consistent recovery to the application layer. The inconsistencies are rooted in the state divergence of replicas. In Section 3.3, we described basic MPI functions and their implementation inside RedMPI. Blocking operations impose limited divergence. But non-blocking operations can easily create scenarios where the state of replicas differs largely as there is no enforced state synchronization among replicas. Only application *barriers* are true synchronization points and RedMPI does not introduce additional barriers. However, the application-specific inter process dependencies caused by *wait* operations limit the divergence among replicas. Thus, divergence of replicas is application-dependent. If an application only uses Send/Recv (blocking calls), the divergence is bounded by 3 MPI calls (see Fig. 3.3-A). But if it has "n" Isend/Irecv calls followed by a Waitall, then the divergence bound is $2n + 1$ (see Fig. 3.3-B). Note that the bound is an indirect result of communication dependencies.

Algorithm 1 shows the steps for Quiesce. Let us assume that rank B has failed and rank B' is cloned to create B'', which takes over the work of B. Ranks B and B'' are in the same state, but any other ranks may still assume B'' to have the state of B.

Stage 1 and 2 of the Algorithm 1 clear the outgoing channels of B' and of any other ranks that have initiated a send to B'. We modified the bookmark exchange protocol [115] to equalize these differences. The original bookmark protocol creates a consistent global snapshot of the MPI job, which requires an all-to-all communication and is not scalable due to its high overhead.

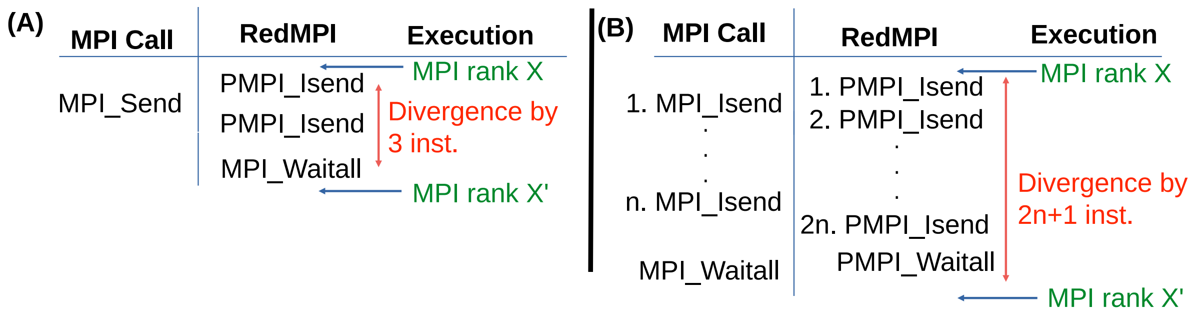


Figure 3.3: Divergence of replicas: (A) one MPI_Send (B) “n” MPI_Isends

In our modified bookmark protocol, each process informs B' of the number of messages it has sent to B' and receives how many messages are sent by B'. Then, the following question can be answered: Have I received all the messages that B' put on the wire? If not, some messages (buffered or in transit) remain in the MPI communication channel and should be drained. Rank B' performs a similar task to drain all messages that other ranks put on the wire to reach B'. In stage 2, receive requests are posted to drain the outstanding messages identified in stage 1. They are saved in temporary buffers. When the application execution later (during normal execution) reaches a Recv call, these drain lists are first consulted to service the request. At the end of the drain phase, no more outstanding send requests to/from B' exist in the system.

Due to the symmetry property of redundant computing, every rank receives the same number of messages from members of a given sphere (e.g., B and B'). The same rule applies to the number of messages sent to a given sphere. This property is the basis for resolving the inconsistencies in stages 3 and 4. The goal of these two stages is to prepare all ranks for communicating with B'' by resolving any state inconsistency between their view of B and B''. Every rank keeps a vector of the number of messages that are sent to other ranks ($Sent[]$) and received from other ranks ($Received[]$) along with the message signatures.

In stage 3, each rank X (other than B') resolves its possible communication inconsistency due to sends to B. Three cases are distinguished: (1) Bookmarks match ($Sent[B] == Sent[B']$): Then B, B', and B'' are in the same state from the point of view of X, and no action is needed. (2) B lagged behind B' ($Sent[B] < Sent[B']$): Then sends from X to B are in transit/will be issued (Fig. 3.4 part 3.2). Since B has been removed and B'' is ahead (has already seen these messages), they are silently suppressed (skipped). (3) B was ahead of B' ($Sent[B] > Sent[B']$): Then there exist messages in transit/to be sent from X to B' (Fig. 3.4 part 3.3). Since B'' is in the same state as B', these messages need to be sent to B'' as well.

In stage 4, the same procedure is performed on the Received counters. The 3 cases are symmetric to the prior stage: (1) Bookmarks match ($Received[B] == Received[B']$): Then B,

B', and B'' are in the same state from the point of view of X, and no further action is needed. (2) B' was ahead of B ($Received[B] < Received[B']$): Then X is expecting messages from B (Fig. 3.4 part 4.2). Since B does not exist anymore and B'' will not send them (as it is ahead), these receives silently complete (skipped). Instead, X will provide the corresponding message from B' to the user level. (3) B' lagged behind B ($Received[B] > Received[B']$): Then X is expecting messages from B' (Fig. 3.4 part 4.3). Since B'' and B' are in the same state, both will send those messages, even though X has already received a copy from B. Thus, messages from B'' are silently absorbed (up to the equalization point).

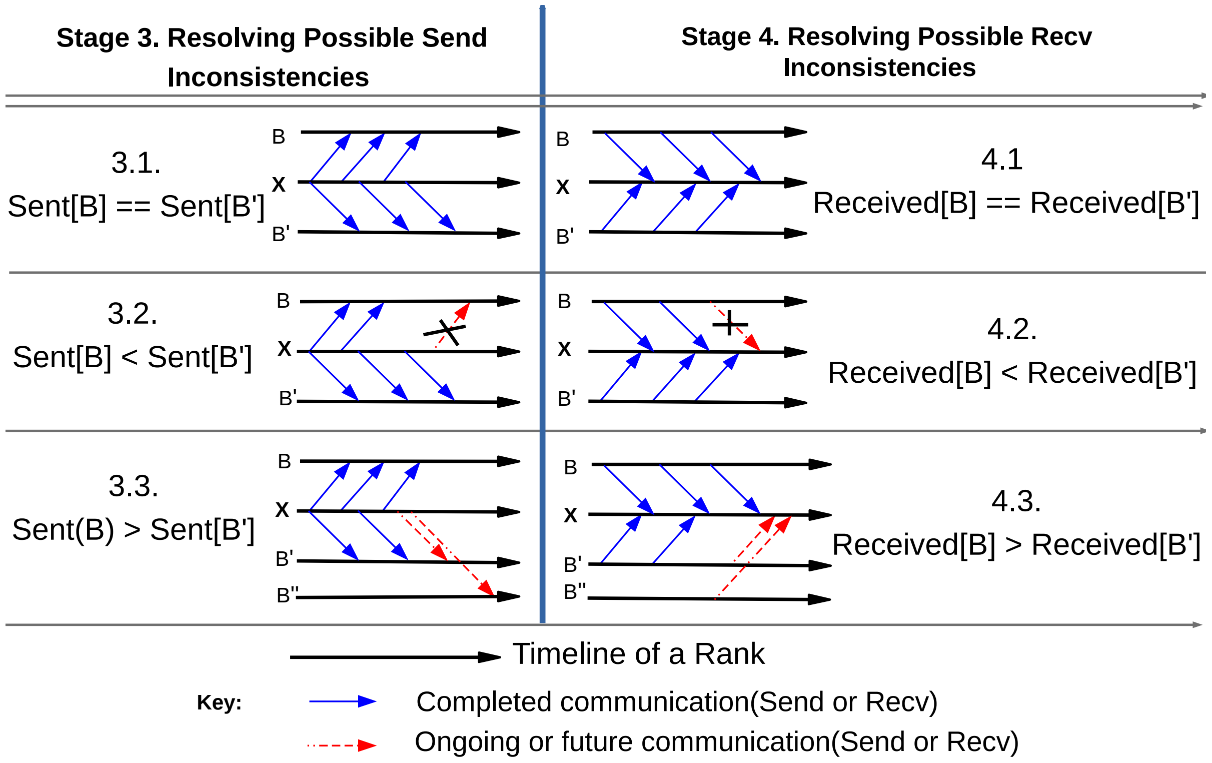

Stage 3. Resolving Possible Send Inconsistencies
Stage 4. Resolving Possible Recv Inconsistencies
3.1.
Sent[B] == Sent[B']
B
X
B'
4.1
Received[B] == Received[B']
3.2.
Sent[B] < Sent[B']
4.2.
Received[B] < Received[B']
3.3.
Sent(B) > Sent[B']
B''
4.3.
Received[B] > Received[B']
Timeline of a Rank
Key:
Completed communication(Send or Recv)
Ongoing or future communication(Send or Recv)

Figure 3.4: A view of Steps 3 and 4 of the Quiesce algorithm

Fig. 3.5 (left side) depicts a program fragment with two ranks, A and B. Two computation sections are separated by two message exchanges from rank B to A (for a predefined type MSG). A failure scenario and our recovery approach with dual redundancy are as follows. Four ranks are created by the MPI runtime, namely A, A', B, B'. `MPI_Recv` and `MPI_Send` calls are redirected to the RedMPI library and implemented as described in Section 3.3. Fig. 3.5 (right side) describes the execution per rank at the levels of the application and DINO. Suppose rank

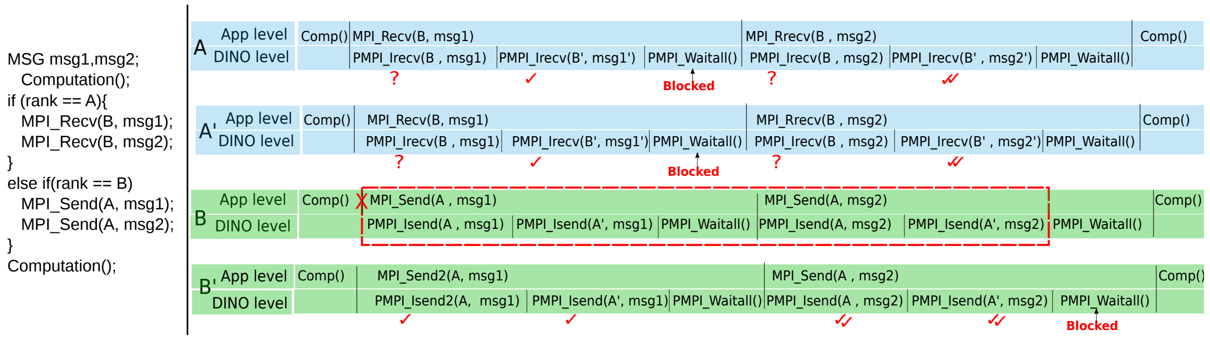
Algorithm 1 Quiesce Algorithm

```
/* 1. Exchange communication state with $B'$ */
if (rank = B') then
    bookmarks *array;
    for $i = 0$ to $nprocs$ do
        if $i \neq B' \wedge i \neq B$ then
            /* send bookmark status, then receive into */
            /* appropriate location in bookmarks array */
            send_bookmarks(i);
            recv_bookmarks(i, array[i]);
else
    bookmark bkmrk;
    /* Receive remote bookmark into bkmrk then send */
    recv_bookmarks($B'$, bkmrk);
    send_bookmarks($B'$);

/* 2. Calculate in-flight msg(s) and drain them */
    Cal_and_Drain();

if ( rank $\neq B'$) then
    /* 3. Resolve possible Send inconsistencies */
    if (Sent[$B$] = Sent[$B'$]) then
        noop;
    else if $Sent[B] < Sent[B']$ then
        Skip the diff to $B''$;
    else if $Sent[B] > Sent[B']$ then
        Repeat the diff to $B''$;
    /* 4. Resolve possible Recv inconsistencies */
    if (Received[$B$] = Received[$B'$]) then
        noop;
    else if (Received[$B$] < Received[$B'$]) then
        Skip the diff from $B''$;
    else if Received[$B$] > Received[$B'$] then
        Repeat the diff from $B''$;
```

B fails at mark "X" in Fig. 3.5. Then ranks A and A' receive the first message from B' and are blocked to receive a message from B. Rank B', after sending all of its messages, continues execution and reaches the waitall. At this point, only communications with single check-marks have finished, and all processes are blocked so that recovery is required. During recovery, B' is cloned to a spare node to create B'' (equivalent to B). Process B'' starts executing the application from where B' was blocked (the *waitall*). Therefore, neither B nor B'' ever execute the `PMPI_Isend` calls shown in the dotted area. Consequently, ranks A and A' do not receive



```
MSG msg1,msg2;
  Computation();
if (rank == A){
  MPI_Recv(B, msg1);
  MPI_Recv(B, msg2);
}
else if(rank == B)
  MPI_Send(A, msg1);
  MPI_Send(A, msg2);
}
Computation();
```

Figure 3.5: Program fragment with ranks A and B (left); its application-/DINO-level execution for dual redundancy under 1 failure (right)

these messages from B (or B''). In stage 1 of the Quiesce algorithm, all outstanding sends to B' are drained. Receive requests are posted by A and A', and these messages are logged in temporary buffers (indicated by double check-marks in Fig. 3.5). Receives from B, highlighted by question marks, remain a unanswered. Quiesce then cancels the first receive (from B) in both A and A' and skips the next receive from B. Subsequently, ranks have consistent states and rank B'' may join the communication of the entire job.

This algorithm does not support wild-cards and assumes collective operations are implemented over point-to-point messages. Other implementations of collectives require a more advanced coordination among ranks during Quiesce. This includes distinguishing the missing messages due to failure along with the topology to correctly determine the destination of messages and to issue cancel/skip operations. Quiesce assumes that the applications issue `MPI_{Wait/Waitall}` calls at least once after a sequence of n non-blocking operations.

3.6 Implementation

3.6.1 Architecture with Open MPI

Fig. 3.6 shows the system architecture where novel DINO components are depicted with shaded boxes. RedMPI provides a transparent interpositioning layer for MPI calls between the application and the MPI runtime system (Open MPI). Open MPI has 3 layers: the Open MPI (OMPI) layer, the Open Run-Time Environment (ORTE) and the Open Portability Access Layer (OPAL). OMPI provides the top-level MPI API, ORTE is the interface to the runtime system, and OPAL provides a utility layer and interfaces to the operating system. The `mpirun` process interacts with the cloning APIs to launch tools on source/spare nodes. The node cloning service provides generic process-level cloning functionality via our extensions to BLCR [49]. This service includes three new tools and functionalities named `restore`, `pre-copy` and `clone`. DINO recovery starts with `mpirun` initiating the `restore` tool on the spare node (Step 1). Then `mpirun` runs `pre-copy` on the source node where the healthy replica exists (Step 2). A communication channel between these two processes is created to transfer the process image. Then `mpirun` invokes the daemons on every node for the Quiesce phase. The daemons send a Quiesce signal to the corresponding ranks, and all ranks enter the Quiesce phase (Step 3). This phase includes the Quiesce algorithm that is described in the previous section and ends by pausing the communication channels. The `mpirun` process runs the `clone` tool on the source node to copy the last portion of process information (Step 4). As the last step (Step 5), `mpirun` communicates with the daemons again to update internal data structures of all processes and resumes the communication among ranks.

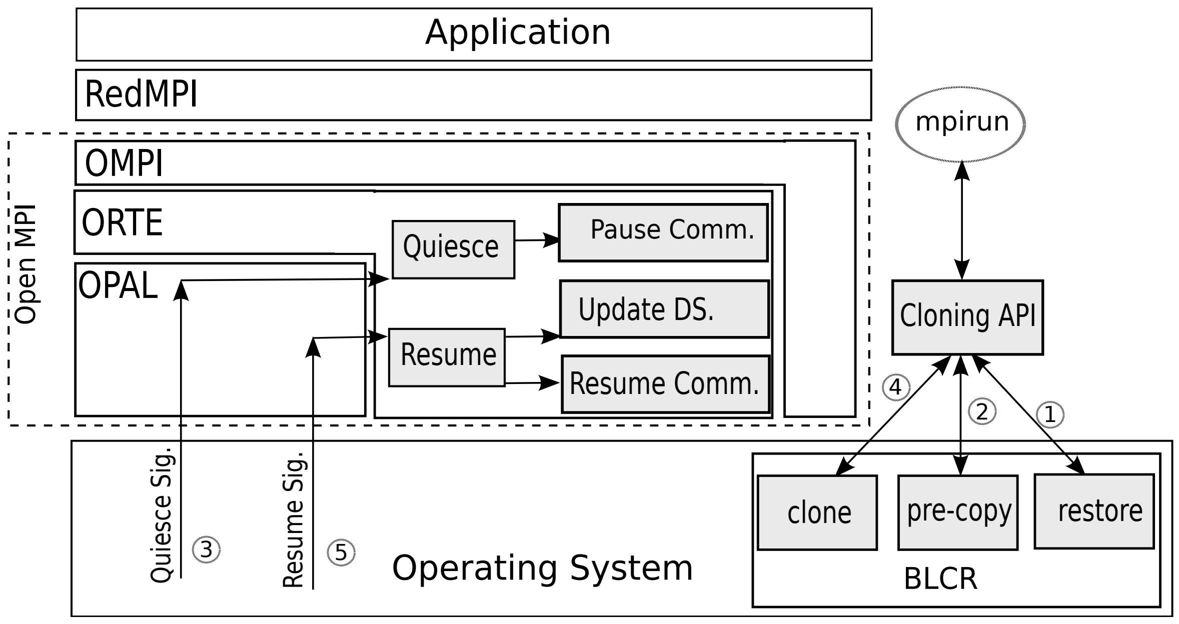


Figure 3.6: DINO Architecture

We next discuss the steps taken during DINO recovery:

1. Launch restore tool The procedure starts with executing `restore` on the spare node. It listens on a predefined port for incoming information and memory pages later sent by the `pre-copy` and `clone` tools and builds the process state, both at kernel and application level, on the spare node.

2. Pre-copy. This phase transfers a snapshot of the memory pages in the process address space communicated to the spare node while normal execution of the process continues on the source node. We use TCP sockets to create a communication channel between *source* and *spare* nodes. The pre-copy approach borrows concepts from [131] (under Linux 2.4), but adapted to Linux to 2.6 (see related work for a comparison). Vital meta data, including the number of threads, is transferred.[2] The spare node receives the memory map from the pre-copy thread. All non-zero pages are transferred and respective page dirty bits are cleared in the first iteration. In subsequent iterations, only dirty pages are transferred after consulting the dirty bit. We apply a patch to the Linux kernel to shadow the dirty bit inside page table entry (PTE) and keep track of the transferred memory pages. The pre-copy phase terminates when the number of transferred pages reaches a threshold (1MB in our current setting).

3. Channel Quiesce. The purpose of this phase is to create a settle point with the shadow process. This includes draining all in-flight MPI messages. The runtime system also needs to stop posting new send/receive requests. We build this phase on top of the functionality for message draining provided by the CR module of Open MPI [69]. The equalization stage described in Section 3.5 is implemented in this step.

4. Clone. This phase stops the process for a short time to transfer a consistent image of its recent changes to the `restore` tool. The memory map and updated memory pages are transferred and stored at the corresponding location in the address space in B''. Then, credentials are transferred and permissions are set. Restoration of CPU-specific registers is performed in the next phase. The signal stack is sent next and the sets of blocked and pending signals are installed. Inside the kernel, we use a barrier at this point to ensure that all threads have received their register values before any file recovery commences. In short, different pieces of information are transferred to fully create the state of the process.

5. Resume. In this phase, processes re-establish their communication channels with the recovered sphere. All processes receive updated job mapping information, reinitialize their Infiniband driver and publish their endpoint information.

3.6.2 MPI System Runtime

The off-the-shelf Open MPI runtime does not allow to dynamically add nodes (e.g., patch in spare nodes to a running MPI job) and, subsequently, to add daemons to a given job. We implemented this missing functionality, including manipulation of job data structures, creation of a daemon, redirection of I/O and exchange of contact information with the `mpirun` process. Finally, there are communication calls issued by RedMPI that violate the symmetry property. These control messages are required to ensure correct MPI semantics under redundancy, e.g., for MPI calls like `MPI_Comm_split` and wildcard receives. We distinguish them and only consider application-initiated calls when identifying messages to *skip* and *repeat* in Algorithm 1.

[2]We assume that applications maintain a constant sized thread pool after initialization, e.g., OpenMP implementations. Cloning applies to the execution phase after such thread pool creation.

3.6.3 OS Runtime and constraints

Many modern Linux distributions support *prelinking*, which enables applications with large numbers of shared libraries to load faster. Prelinking is a method of assigning fixed addresses to and wrapping shared libraries around executables at load time. However, these addresses of shared libraries differ across nodes due to randomization. We assume prelinking to be disabled on compute nodes to facilitate cloning onto spare nodes. Files opened in write mode on a globally shared directory (e.g., NFS) can cause problems (due to access by both replicas), a problem considered in orthogonal work [18].

3.7 Job Completion Time Analysis

The objective of this section is to provide a qualitative job completion analysis for redundant computing to assess the effect of our resilience technique. We make the following assumptions in the mathematical analysis. (1) Node failures follow a Poisson process. Subsequently, the time between two failures follows an exponential distribution. (2) Failures occur independently. A failure does not affect or increase the failure probability of other ranks. (4) Failures do not strike the two nodes involved in a cloning operation (the source and the spare node) while DINO recovery is in progress. The short time required for cloning (seconds) relative to time between failures (hours) allows us to make this assumption. Let us denote:

- t: failure-free execution time of the application (without redundancy)
- r: level of Redundancy
- α: fraction of total application time spent on communication (without redundancy)
- β: fraction of total application time spent on serial communication (with redundancy)
- θ: MTTF of a node ($\lambda = 1/\theta$: failure rate of a node)
- t_{clone}: time spent on a DINO recovery

Estimating redundancy overhead requires measuring the application's communication overhead. We use mpiP [95] to collect such information. The mpiP tool computes "AppTime" and "MPITime". "AppTime" is the wall-clock time from the end of `MPI_Init` until the beginning of `MPI_Finalize`. "MPITime" is the wall-clock time for all MPI calls within "AppTime". It also reports the aggregate time spent on every MPI call. "SendTime", "IsendTime", "RecvTime", "IrecvTime" and "WaitTime" are the aggregate times spent on `MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Irecv` and `MPI_{Wait,Waitall}`, respectively.

We use mpiP with the 1x (no redundancy) execution of the application, collect the information on communication overhead and then try to estimate the execution time with rx ("r" level of redundancy). We define α as the fraction of time spent on communication under 1x. The

α ratio is application specific and also depends on the input data size. α is defined as: $\alpha = MPITime/AppTime$.

Next we consider a redundant computing environment and reason about different types of communication. Point-to-point MPI calls are always processed serially as r messages are required to be transmitted. E.g., `MPI_Send` takes 2 times longer under 2x. However, this depends on the message size and the transfer protocol deployed dynamically by the MPI runtime (r times overhead may not be the best estimate under some scenarios). Collective operations, in contrast, are performed in parallel as we use RedMPI in its `optimized` collective mode. Under this mode, collectives are called directly from RedMPI and are internally parallelized. As a result, only the point-to-point (P2P) segment of the application is performed serially. We define F_{Send} and F_{Recv} fractions as follows:

$$F_{Send} = \frac{SendTime + IsendTime + C_1 \cdot WaitTime}{MPITime} \tag{3.1}$$

$$F_{Recv} = \frac{RecvTime + IrecvTime + C_2 \cdot WaitTime}{MPITime} \tag{3.2}$$

Without non-blocking receives, we fold "WaitTime" into the F_{Send} fraction ($C_1 = 1, C_2 = 0$). Similarly, without non-blocking sends, we fold wait time in F_{Recv} fraction ($C_1 = 0, C_2 = 1$). With both `MPI_Isend` and `MPI_Irecv`, we divide the wait time equally between the F_{Send} and F_{Recv} fractions. ($C_1 = 0.5, C_2 = 0.5$). Finally, without non-blocking sends and receives, we assign $C_1 = 0, C_2 = 0$.

We define β as portion of the communication that is performed serially with redundancy: β is estimating the message transfer overhead. The F_{Send} and F_{Recv} fractions have an inherent wait (e.g., until the other side's execution reaches the communication point). A good estimation is the minimum of the two values as it captures their overlap:

$$\beta = min(F_{Send}, F_{Recv}) \cdot \alpha \tag{3.3}$$

Under redundancy, the serial segment of the communication ($\beta \cdot t$) is multiplied by the redundancy factor (r) while the rest due to computation and parallel communication ($(1-\beta) \cdot t$) remains the same. In other words, the extra point-to-point communication due to redundancy is performed serially, but the replicas are performing their computation and collective segments in parallel. As a result, the execution time with redundancy is estimated as:

$$T_{red} = \beta \cdot t \cdot r + (1-\beta)t \tag{3.4}$$

Table 3.2 shows the model validation experiment performed on the Stampede supercomputer. Stampede nodes are Dell C8220z running CentOS 6.3 with the 2.6.32 Linux kernel. Each node contains two Xeon Intel 8-Core 2.7GHz E5-processors (16 total cores) with 32GB of memory

(2GB/core). We choose 5 benchmarks each in 4 configurations: 3 of NPB programs (CG, LU, MG), Sweep3D (S3D), and LULESH. Sweep3D represents an ASC application that solves a neutron transport problem. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements [82]. We used 1024 and 2048 ranks under dual redundancy for CG, LU, MG and Sweep3D. As LULESH requires cubic values for the number of ranks, we used 1000 and 1728 ranks. We are interested in studying the overhead of redundancy in large number of ranks, so we launch one rank per core (16 ranks per node). Each benchmark is tested with two input sizes, Classes D and E, for NAS benchmarks. Two configurations of 100×40×400 and 320×40×400 are used for Sweep3D and size 25 and 27 are used for LULESH. Table 3.2 depicts execution times without redundancy (1x), α, *Send*, and *Recv* fractions from mpiP profiling. Eq. 3.3 is used to compute the β column. Experimental execution time under redundancy (2x Real), and modeled execution time under redundancy using Eq. 3.4 (2x Model) are the next two columns.

CG, LU and MG do not have any `MPI_Isend` call, so we assign $C_1 = 0, C_2 = 1$ in Equations 3.1 and 3.2. Sweep3D has no non-blocking calls ($C_1 = 0, C_2 = 0$) and LULESH has both non-blocking send and receive calls ($C_1 = 0.5, C_2 = 0.5$). The benchmarks cover 3 of 4 possible cases. The fourth case ($C_1 = 1, C_2 = 0$) is rather uncommon due to penalties associated with it.

Fig. 3.7 compares the real overhead and modeled overhead for the experiments in Table 3.2. The overhead percentage of redundancy is derived from the experiment via $\frac{2xReal-1x}{1x}$ (Real Overhead) and by modeling using $\frac{2xModeled-1x}{1x}$ (Modeled Overhead). The x axis shows 4 configurations of each benchmark numbered from 1 to 4 in the same order shown in Table 3.2. The model error is also shown on the top of the modeled overhead. The results show overestimation in CG between 1.56% and 6.01%. Our model underestimates the LU benchmark in all 4 cases between 2.4% to 6.18%. Configurations 1 and 2 of MG are underestimated (3.24% and 5.75% error) while the other two (3, 4) are overestimated (2.61% and 2.51% error). Sweep3D, which has no non-blocking calls, has been estimated more accurately (with error between 0.57% to 2.1%). Finally, overhead for LULESH has been underestimated (between 2.78% and 4.79%).

3.7.1 Discussion

Overall, the presented results show a low model error of at most 6.18%. It also shows that capturing the overhead of non-blocking calls is more challenging as Sweep3D has no non-blocking calls and results in the least model error. In the current model, we give equal weight to *wait* time in Equations 3.1 and 3.2 when both non-blocking sends and receives exist. A more advanced profiling technique could separate the *wait* time spent on *Send* from *Recv* and might provide a more accurate *Send* and *Recv* fraction. One could extract such information from the `MPI_-Request` data structure and extend mpiP to retrieve fine-grained *wait* information. One other

Table 3.2: Experiment vs. modeled execution time under redundancy

	# of ranks (problem size)	1x (seconds)	α	F_{Send}	F_{Recv}	β	2x Real (sec)	2x Model (sec)
CG	1024 (Class D)	22.75	.5351	.6755	.3092	.1654	25.5	26.52
	2048 (Class D)	13.85	.6507	.6027	.3725	.2423	16.38	17.21
	1024 (Class E)	232.62	.2985	.8012	.196	.0585	242.62	246.23
	2048 (Class E)	109.04	.4657	.6967	.2987	.1391	119.36	124.21
LU	1024 (Class D)	30.28	.3298	.2159	.7553	.0712	33.79	32.43
	2048 (Class D)	26.55	.4621	.1857	.7702	.0858	30.42	28.83
	1024 (Class E)	376.80	.1131	.1979	.7942	.0223	394.28	385.24
	2048 (Class E)	206.08	.1738	.1887	.7992	.0327	222.39	212.84
MG	1024 (Class D)	2.28	.3357	.4332	.317	.1064	2.59	2.52
	2048 (Class D)	1.45	.5116	.3055	.3594	.1562	1.76	1.67
	1024 (Class E)	17.18	.1458	.4903	.4162	.0606	17.77	18.22
	2048 (Class E)	9.06	.2342	.3884	.462	.0909	9.65	9.88
S3D	1024 ($100 \times 40 \times 400$)	28.06	.286	.1139	.7345	.0325	28.81	28.97
	2048 ($100 \times 40 \times 400$)	35.39	.3592	.1054	.6893	.0378	35.99	36.73
	1024 ($320 \times 40 \times 400$)	88.04	.2821	.1036	.7217	.0292	89.74	90.61
	2048 ($320 \times 40 \times 400$)	117.81	.3742	.1289	.6804	.0482	121.39	123.49
LULESH	1000 (Size=25)	270.82	.2855	.679	.599	.171	288.20	275.45
	1728 (Size=25)	328.34	.2885	.72	.631	.182	350.07	334.32
	1000 (Size=27)	351.39	.244	.669	.576	.140	371.63	356.33
	1728 (Size=27)	448.60	.2533	.719	.689	.174	468.89	456.43

model enhancement is to extract the overhead of each P2P call individually (O_i). Then, the overall P2P overhead can be computed as $\beta = \sum_{i=1}^{n} O_i$.

3.7.2 Job Completion time with DINO failure recovery

The t_{clone} parameter depends on the application as it is directly related to the process image size. Moreover, increasing the input size of the application leads to larger data in memory and eventually larger t_{clone}. The total job completion time is the sum of the time to perform actual computation (t) and the time to recover from failures.

$$T_{total} = T_{red} + T_{recovery} \tag{3.5}$$

Let n_f be the number of failures that occur till the application completes. On average, a node failure occurs every $\frac{\theta}{n \cdot r}$. Therefore, n_f is calculated as $n_f = T_{total} \cdot \frac{n \cdot r}{\theta}$ or $n_f = T_{total} \cdot \lambda \cdot n \cdot r$. Then, $T_{recovery} = n_f \cdot t_{clone}$. In summary,

$$T_{recovery} = T_{total} \cdot \lambda \cdot n \cdot r \cdot t_{clone} \tag{3.6}$$

Plugging Eq. 3.6 into Eq. 3.5, the job completion time under redundancy and DINO failure recovery is estimated as:

$$T_{total} = \frac{T_{red}}{1 - \lambda \cdot n \cdot r \cdot t_{clone}} \tag{3.7}$$

We use Eq. 3.7 in Section 3.9 for large scale simulations.

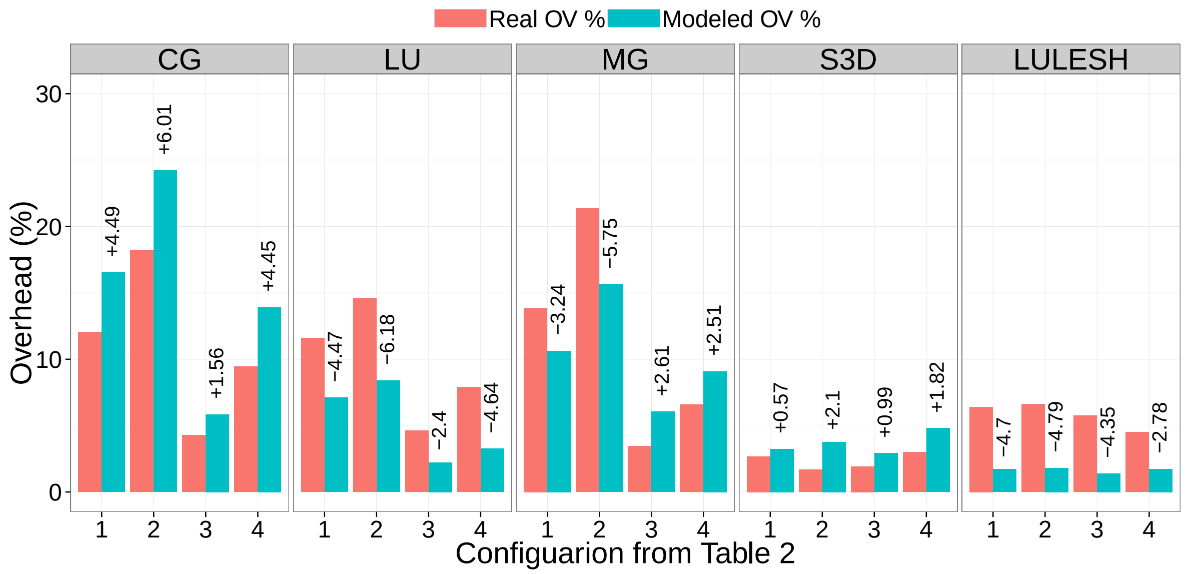


Figure 3.7: Experimental vs. modeled overhead for redundancy – The model error is depicted on top of the each bar of modeled overhead

3.8 Experimental Results

The node cloning experiments require insertion of our kernel module into the Linux kernel. This permission is not granted on large-scale supercomputers maintained by NSF or DOE. Thus, we conducted the experiments on a 108-node cluster with QDR Infiniband. Each node is equipped with two AMD Opteron 6128 processors (16 cores total) and 32GB RAM running CentOS 5.5, Linux kernel 2.6.32 and Open MPI 1.6.1. The experiments are demonstrating failure recovery rather than exploring compute capability for extreme scale due to the limitations of our hardware platform. Hence, we exploit one process per node in all experiments. Experiments were repeated five times and average values of metrics are reported.

3.8.1 Live Node Cloning Service.

In this section, we evaluate the node cloning performance at process-level. We compare "CR-based" node cloning (checkpoint, transfer, restart) with our live cloning approach. We created a microbenchmark consisting of `malloc` calls (`sbrk` system calls) to control the size of the process image. It has an `OpenMP` loop with 16 threads long enough to compare the performance of our node cloning mechanism with Checkpoint/Restart(CR). In CR, we checkpoint the process locally, transfer the image to another node and then restart the snapshot (using off-the-shelf BLCR). We omit the file transfer overhead and consider the best case for CR (checkpoint and restart locally).

Table 3.3 shows the results for different image sizes ranging from 1GB to 8GB. Our cloning approach takes less time than just checkpointing only without restart. It is more than two times faster than CR in all cases (2.24x for 1GB and 2.17x for 8GB). Cloning is performed via TCP over QDR Infiniband with an effective bandwidth of 300 MB/s. The speedup of our approach is due to creating the process while copying the process image. As a result, we parallelize the two steps that are serial in CR. Furthermore, we do not use the disk but instead transfer memory pages over the network.

Table 3.3: Microbenchmark performance (in seconds) of process cloning vs CR (Process with 16 threads)

Process Image Size	**Cloning**	**Checkpoint**	**Restart**	**Total CR**	**Speedup ($\frac{CR}{Cloning}$)**
1GB	**12.42**	14.98	12.85	**27.83**	2.24
2GB	**26.11**	29.79	25.63	**55.42**	2.12
4GB	**49.58**	59.79	50.39	**110.36**	2.22
8GB	**100.94**	119.10	100.20	**219.3**	2.17

3.8.2 Overhead of Failure Recovery.

In this section, we analyze the performance of DINO. We consider 9 MPI benchmarks: (BT, CG, FT, IS, LU, MG, SP) from the NAS Parallel Benchmarks (NPB) plus Sweep3D (S3D) and LULESH. We use input class D for NPB, size $320 \times 100 \times 500$ for Sweep3D and size 250 for LULESH. We present results for 4, 8, 16 and 32 processes under dual redundancy (CG, IS, LU, MG). We use 4, 9, 16, 25 processes for BT and SP (square numbers are required by the benchmark). FT with 4 and 8 processes could not be executed due to memory limitations. LULESH only runs with cubic numbers of processes, so we run it with 9 and 27 processes. Due to lack of support from the Infiniband driver to cancel outstanding requests without invalidating the whole work queue and lack of safe re-initialization, current experiments are performed with marker messages. Every process receives a message indicating the fault injection and acts accordingly. One rank mimics the failure by performing a `SIGSTOP`. Then the Cloning APIs are used to start the clone procedure and the discussed steps in Section 3.4 are performed: Pre-copy, Quiesce, Clone, Resume.

Fig. 3.8(a) and 3.8(b) depict the overhead and transferred memory size, respectively. NPB are strong scaling applications and the problem size is constant in a given class. Therefore, the transferred memory size and consequently time decreases when the number of processes increases. In contrast, Sweep3D and LULESH are weak scaling and the problem size remains constant for each process, solving a larger overall problem when the number of processes increases. Thus, weak scaling benchmarks show negligible difference in overhead and transferred process image size over different number of processes.

FT has the largest process image. The size of memory for FT with 16 processes is 7GB and takes 90.48 sec to transfer, while it takes 46.75 seconds with 32 processes to recover from a failure when transferring 3.52GB of memory. LU has the smallest process image among NPB, its memory size ranges from 2.64GB to 0.36GB with transfer times of 32.51 to 5.60 seconds for 4 to 32 processes, respectively. For Sweep3D, the overhead is almost constant at 23.5 seconds over different numbers of processes when transferring a 1.8GB image. The same applies to LULESH with a constant process image size of 2.75GB and an overhead of 38.51 seconds.[3] The relative standard deviation in these experiments is less 7% in all cases.

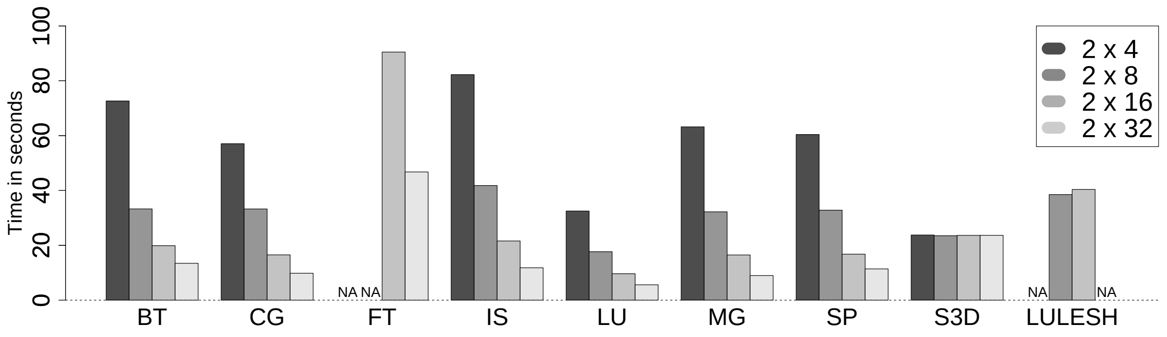
We also measure the time spent in each phase: pre-copy, quiesce, clone and resume for 32 processes except for LULESH, where 27 processes are used (see Fig. 3.8(c)). The pre-copy phase is shown on the left axis, and the rest of phases are shown on the right axis which is an order of magnitude smaller. The majority of time is spent in the pre-copy phase and the remaining three phases take about 1 second combined. These three phases take almost similar time across all the benchmarks with only small variations. In our experiments, we observed that shortly after the fault injection, a deadlock occurs. Other processes wait for the completion of a communication with the failed process and this will eventually create a chain of dependencies among all processes. The Quiesce phase of the DINO recovery resolved this problem successfully. Due to early deadlock occurrence, the overlap of the normal operation with the pre-copy phase was negligible for the chosen benchmarks.

3.9 Simulation Results

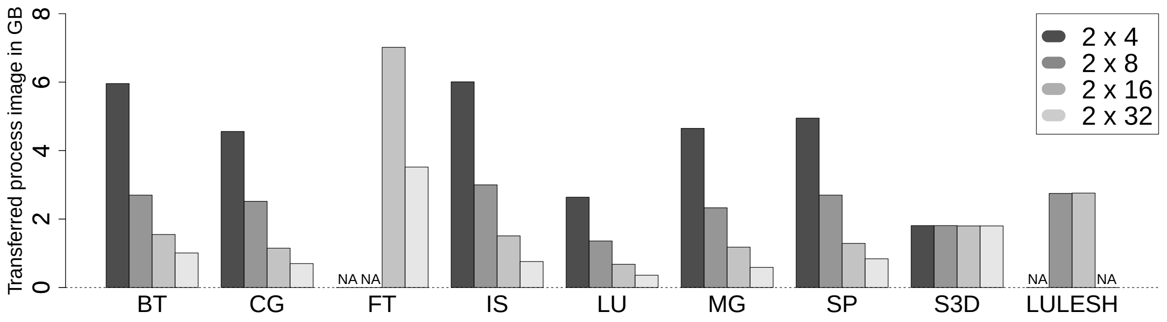
3.9.1 Size of Spare Node Pool.

This section analyzes the effect of cloning on the average number of required spare nodes to still complete a job. Assume each node has a $MTTF$ of 50 years. On average, every $MTTF/(r \times n)$, a node fails in the system. Further, assume $r = 2$ and $T = 200$ hours. The time to complete the job with $\beta = 0.2$, 0.4 and 0.6 can be calculated using Eq. 3.7. We consider large values for β to study the node pool size in more extreme cases where redundancy has larger overhead. Table 3.4 indicates the number of spare nodes required for successful job completion for different values of n ranging from 18 ($n = 16K$, $\beta = 0.2$) to 465 nodes ($n = 256K$, $\beta = 0.6$). In this case, we did not consider any repair for the system (MTTR $=\infty$).

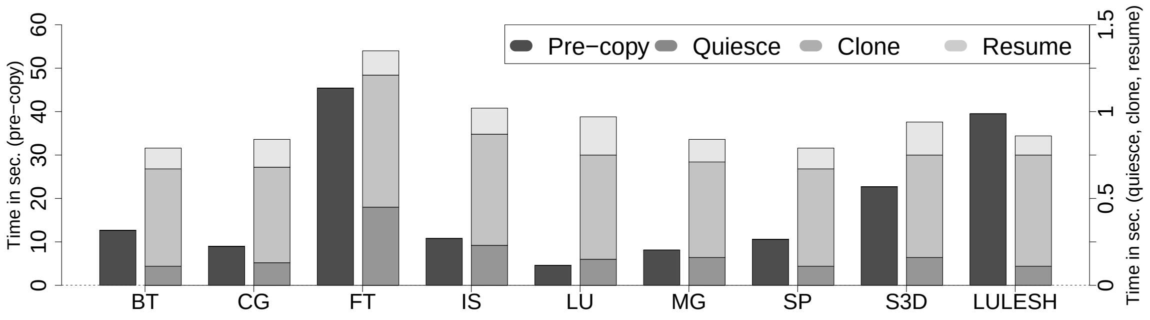
[3] In the current implementation, the memory is copied one page at a time. This lowers the performance of the cloning operation. A larger buffer size might increase the performance as the effective bandwidth could be increased.



(a) Overhead (vulnerability window)



(b) Transferred process image



(c) Overhead: Step-wise with 2 × 32 ranks

Figure 3.8: DINO recovery from 1 fault injection for different MPI benchmarks (1 rank per physical compute node)

If we consider a Mean Time to Repair (MTTR) of 20 hours, the required number of spare nodes is shown in the last column of Table 3.4. We observe that the average number of spare nodes is ranging from 2 to 25, which is only a small fraction of total number of nodes. Assuming that nodes are repairable, the average number of required spare nodes turns out to be independent of the β value. For example, consider $n = 64K$. When $\beta = 0.2$, the job takes 252.28 hours, and we have $\lfloor 252.28/20 \rfloor = 12$ repair intervals. Similarly, for $\beta = 0.4$, there are $\lfloor 283.45/20 \rfloor = 14$ repair intervals, and for $\beta = 0.6$, there are $\lfloor 336.38/20 \rfloor = 16$ repair intervals. If we divide the number of required spare nodes by the number of repair intervals, we obtain a bound on the number of required spare nodes. This value is $\lceil 74/12 \rceil$, $\lceil 86/14 \rceil$ and $\lceil 99/16 \rceil$ for $\beta = 0.2$, 0.4 and 0.6, respectively. In all three cases, 7 spare nodes are required. Similar conditions hold for the rest, meaning that results are independent of β.

Table 3.4: Avg. number of required spare nodes

	MTTR = ∞			**MTTR = 20h**
Size (n)	$\beta = 0.2$	$\beta = 0.4$	$\beta = 0.6$	$\beta = 0.2, 0.4, 0.6$
16000	18	21	24	2
32000	36	42	48	3
64000	74	86	99	7
128000	156	182	208	13
256000	349	407	465	25

3.9.2 Job Completion Time.

Next, we study the behavior of different methods at extreme scale. Plain job execution time (t) is 256 hours and β varies from 0.1 to 0.4. We use Eq. 3.7 to extrapolate the job completion time with DINO recovery. Under redundancy, we only re-execute the job upon job failure (when both replicas fail) and use the following equation [48]: $T_{total} = (D + \frac{1}{\Lambda})(e^{\Lambda T_{red}} - 1)$ where D is time to re-launch the job (assumed to be 2 minutes), and Λ is the system MTTF ($\Lambda = -\ln(R_{sys})/T_{red}$). R_{sys} is the overall system reliability and can be computed as: $R_{sys} = (1 - (T_{red}/\theta)^r)^{n \times r}$ [51].

Three solutions are studied: dual redundancy (2x), 2x with DINO recovery (2xD), and triple redundancy (3x). The cloning overhead is 2 minutes since only a single node pair is involved in cloning. We choose a node MTTF of $\theta = 50$ years. Fig. 3.9 shows the job completion time for systems with different numbers of nodes ranging from 10K to 2M. 2x is the dashed line, 2xD is the solid line, and 3x is the dotted line. Based on the results from Table 3.2, we choose $\beta = 0.1$ and 0.2 in this experiment. Fig. 3.9(A+B) depict the results for $\beta = 0.1$ and 0.2. A zoomed view of the area of interest is shown on the top of each plot. This area is 10K to 500K in Fig. 3.9(A) and 10K to 1M in Fig. 3.9(B). 2xD outperforms 3x for up to 540K and 940K nodes for a β of

0.1 and 0.2, respectively. After this point, 3x provides shorter job execution times due to DINO recovery overhead for every failure, but this range is likely beyond the size of exascale systems. For $\beta = 0.1$, 2x is between 282 to 719 hours. This changes to 308.72 to 960.99 hours for $\beta = 0.2$.

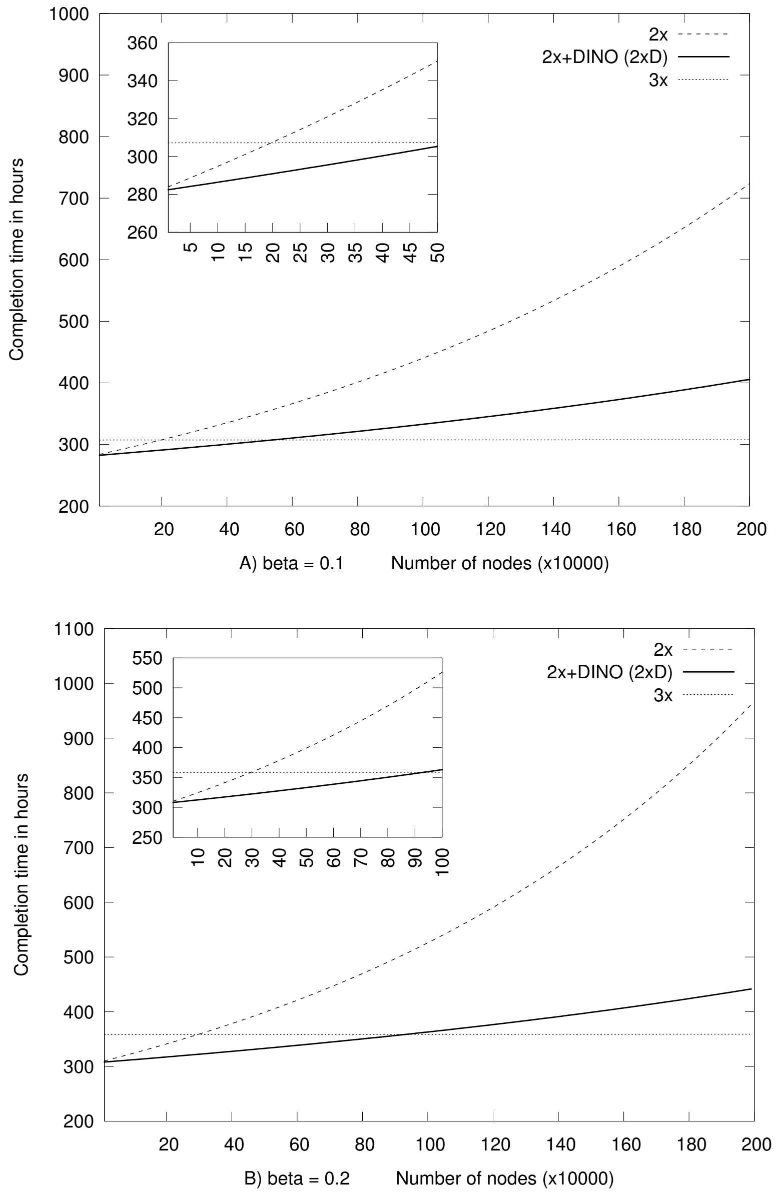


Figure 3.9: Modeled job completion time

3.9.3 Vulnerability Window.

Our experimental evaluation in Section 3.8 showed that for typical HPC application, the DINO recovery time is in order of seconds to less than 2 minutes (depending of the application footprint per process). In this section, we reason about the probability of experiencing a soft error during the vulnerability window under dual redundancy. Suppose the vulnerability window is w, and the execution time per rank is T_p (total: $2 \times N \times T_p$). Assume that at some point of time, a process fails and there are currently $2 \times N - 1$ active process. The probability of a soft error hitting the application during w is $w/2 \times N \times T_p$. Assuming $w = 2$ minutes, $T_p = 100$ hours and $N = 1$ million nodes, this probability is very small ($1.65e^{-10}$). As a result, the effect of the vulnerability window on the correctness of application result is negligible.

3.10 Related Work

Our earlier work [106] provides stochastic modeling and simulation results for a system based on redundancy and process cloning. This chapter presents the design and implementation of a node cloning service, the Quiesce algorithm, an experimental evaluation with fault injection and a model for job completion time under redundancy. rMPI [22] and MR-MPI [55] provide transparent redundant MPI computing. Elliott et al. [51] determines the best configuration of a combined approach including redundancy and CR. They propose a model to capture the effect of redundancy on the execution time and checkpoint interval. Ferreira et al. [57] investigate the feasibility of process replication for exascale computing. A combination of modeling, empirical and simulation experiments is presented in this work. PLR [121] provides transparent process redundancy and is capable of detecting soft errors. None of the above works deal with replica failures and only focus on providing replication capability. Our work addresses hard replica failures and solves the consistency problem in the MPI environment during failure recovery.

A process-level proactive live migration approach is presented in [131] with an integration within LAM/MPI. VMware Workstation [130] provides virtual machine migration and cloning. In cloning, a snapshot of the VM is created. The cloned VM is independent from the main VM. Clones are useful when one must deploy many identical virtual machines in a group. Cloning in VMs is mainly to avoid the time-consuming installation of operating systems and application software. Unlike DINO, prior work does not handle (a) new process/thread creation with the same progress but a *different* identity, (b) compensation for divergence in progress, and (c) techniques to limit divergent scopes.

3.11 Conclusion

We introduced DINO, a quick forward recovery method from failures in redundant computing. DINO contributes a novel live node cloning service with a *scalable* multicast variant of the bookmark algorithm and a corresponding Quiesce algorithm for consistency among *diverging* communicating tasks. With its integration into the MPI runtime system, DINO allows a redundant job to retain its redundancy level via cloning throughout job execution. Experimental results with multiple MPI benchmarks indicate low overhead for failure recovery. A model for job for redundant computing indicates that dual redundancy with cloning (DINO-style) outperforms triple redundancy up to 0.5-1M nodes depending on the communication to computation ratio, which is the range relevant to exascale computing. This means node failures can be tolerated by forward recovery with DINO with sustained resilience levels due to live cloning. In future work, SDC could also be detected (without correction) by DINO at 33% lower power than triple redundancy.

Chapter 4

End-to-End Application Resilience to Protect Against Soft Errors

4.1 Introduction

Fault resilience remains a fundamental problem in computing, and especially parallel computing. Bit flips, operating system (OS) errors, hardware faults, and their manifestation as application failures are an inherent part of computing systems at scale. One can try to alleviate the consequences by compensating with re-computation or redundancy. However, the traditional approaches need improvements and innovations as the future exascale systems are projected to more frequent failures. It has been estimated that an increase in the frequency of faults in exascale systems can be prevented at the cost of 20% more circuitry and energy consumption [122]. However, the hardware vendors do not tend to design and build specific hardware due to manufacturing costs. As a result, the future systems will be likely built with off-the-shelf components while delegating a significant part of the resilience responsibility to the software layer.

Fault resilience techniques enable application completion with correct results despite the existence of multiple sources of faults and their possible occurrence in the system. The significance of resilience in future HPC systems has been emphasized in prior research [122, 26, 42, 29]. HPC systems are particularly interesting to study as multiple challenges arise from the size (millions of cores) and the programming model (tightly coupled). Intuitively, larger numbers of components result in a higher probability of failures. Moreover, a tightly coupled programming model may result in fast fault propagation after just one node has been hit [59]. On the top of that, the software layer (concurrency, bugs) is anticipated to become even more complex [10].

As a result, resilience is considered a major roadblock on the path toward next-generation HPC systems.

Recent innovations in the hardware, such as deep memory hierarchies and heterogeneous cores, provide a performance boost at the price of adding to system complexity. Persistent memory technologies like Phase Change Memory (PCM), magnetic RAM (MRAM), and Memristors are positioned to be placed as burst buffers between DRAM and storage to alleviate the speed gap using a non-volatile abstraction. Application programmers are required to take new parameters into account to fully exploit the performance of novel hardware. As a result, multiple performance metrics need to be taken into account while developing software.

Application programmers often spend a considerable amount of time on choosing and incorporating one or multiple resilience methods into their application. They might need to re-write their application in a specific programming model with Containment Domains [37], GVR [137], Charm++ [80], GMRES inner/outer solver [112], or PERM [132]. Thus, the trend is to add resilience on a per application basis, i.e., vector dot product with GMRES [52], LULESH [82] and NAMD [102] with Charm++, or LULESH and LMAT [5] with PERM. One of the problems in such approaches is code complexity due to interleaving the algorithmic and resilience concerns such that these disjoint goals are hard to identify inside the program, let along to maintain such programs from there on.

As performance and resilience requirements are pushed to the software layer, the software complexity and consequently portability/maintainability become problematic. We argue that a systematic resilience approach is much more effective, especially for future complex systems. This allows the application developer to focus entirely on algorithms and performance and delegate the resilience requirements for seamless protection to the system (library/ runtime). Thus, devising a unified API specifically crafted for end-to-end resilience is the objective of this work, which is motivated by the aspect-oriented programming (AOP) paradigm [83], an approach that aims to increase modularity by allowing the separation of concerns. AOP includes programming methods that support the modularization of concerns (goals) at the source code level.

Hukerikar et al. [68] promote a redundant multithreading integrated as an OpenMP pragma that allows for soft error detection. Our work not only addresses redundant computing but also supports a broader range of resilience methods with an end-to-end resilience goal. Chen et al. [34] propose a tool named ErrorSight to pinpoint data structures that are most vulnerable to soft errors. They use a model to generate error profiles and discover propagation patterns. The overall goal is to help the application developers to focus their efforts on most vulnerable code regions. Their work is orthogonal to ours in the sense that after detecting such regions, our API could be used to protect the data.

This work makes the following contributions:

- We present a systematic approach for end-to-end resilience with minimal assistance from the application developer. We separate the resilience aspects from the algorithmic side of computation aiming for portability and modularity of HPC codes.
- In end-to-end resilience, checks and appropriate recovery methods are automatically inserted into the application source code and a computationally equivalent code is generated that is fault resilient. In case studies, we demonstrate the applicability of our approach and its benefit over conventional resilience techniques.
- We show that the overhead for check/recover methods in a sequentially composed numeric kernel is negligible while the overall space overhead is less than 1% for per matrix checksums and less than 7% for per-block checksums.
- Our experimental results show that check/recover methods in a code that combines MPI with end-to-end resilience results in less than 7% overhead resulting in an overall overhead of $14 - 16\%$ compared to no resilience.

The chapter is organized as follows: Section 4.2 discusses the background and related work. Section 4.3 states the assumptions. Section 4.4 introduces our end-to-end resilience using a pragma-based and a task-based implementation. Section 4.5 presents case studies and experimental results. Section 4.6 summarizes the work.

4.2 Background and related work

4.2.1 Failures

Let us present a brief overview of the terminology used in this Chapter. Avizienis et al. [11] define faults as the cause of errors. Errors, in turn, may lead to failures. Faults occur at the system level, i.e., a physical phenomenon may adversely affect the hardware or its interactions between various components. If the erroneous component is part of a service and there is no built-in correctness approach, an error leads to a failure, which manifests itself as an incorrect result, crash or interruption in service. Failures are further distinguished by their origin into hardware and software failures.

Hardware Failures

Hardware faults could be persistent or transient. Persistent faults are typically due to aging or operation beyond temperature thresholds and maybe mitigated by replacement of the failed component or marking, e.g., of pages with sticky bits. Failures due to persistent faults interrupt the user application. They may render a HPC job of thousands of processes useless.

Transient hardware errors are often called soft errors. Cosmic radiation is traditionally considered to be a major cause for transient hardware errors. When high energy neutrons interact with the silicon die, a secondary cascade of charged particles emerges. The resulting pulse is powerful enough to change the stored bits in memory (DRAM or SRAM). However, this depends on the size of transistors. Smaller transistors carry smaller charges. Consequently, multiple bit flips become more likely to manifest [10]. Transient errors need not stop the application from making progress. (In fact, fail-stop on double bit flips was deliberately disabled in the BIOS os some HPC machines, at least for Linpack runs.) They allow the application to continue execution, albeit with tainted data. Such faults manifest as bit flips in the data in memory or anywhere in the data path (e.g., caches, data bus). Although CPU registers, caches, and main memory are often equipped with ECC, only single bit flips are correctable (by ECC or chipkill) while double-flips generally are not (by SEC-DED ECC while chipkill can correct some multi-bit errors depending on their device locality). [1] Jaguar's 360TB of DRAM experienced a double bit flip every 24 hours [61]. With ECC/chipkill, end-to-end resilience complements hardware protection. Some soft errors may remain undetectable and may result in so-called Silent Data Corruption (SDC). SDCs may manifest at application completion by producing wrong results or, prior to that, wrong interim results. A study at CERN raised concerns over the significance of SDC in memory, disk and RAID [101]. Their results indicated that SDC rates are orders of magnitude larger than manufacture specifications. Schroeder et al.'s study [119] of the DRAM errors on large-scale platforms over the course of 2.5 years concludes that more than 8% of DIMMs are affected by errors per year. Schroeder's findings are refined by Sridharan et al. [125] in a detailed analysis on the effectiveness of hardware techniques, namely ECC and chipkill. A study by Microsoft over 1 million consumer PCs also confirms that CPU faults are frequent [97]. There is a lack of extensive experiments and estimations of the SDC rate in current systems. Their future effect on exascale systems remains unknown. A recent report [122] recommends for more research to be conducted on SDC rates and considers refinements in hardware to keep SDC rates low or exploit software methods to detect and correct SDCs. Li et al. [87] protect resources (e.g., registers), instructions, and control flow in GPUs using JIT technqiues. In contrast, we protect an entire data structure (could be several GB of memory), which they cannot.

Software Failures

Software failures can be due to bugs [39] (some of which may only materialize at scale), complex software component interactions, and race conditions that surface only for rare parallel execution interleavings of tasks. Moreover, application codes are also becoming ever so more complex with

[1] Bit flips in code (instruction bits) create unpredictable outcomes (most probably segmentation faults or crashes) and are out of the scope of this work.

the need to reduce communications and improve asynchrony. The resulting algorithms have higher performance but often become more error-prone [10]. In fact, it has even been predicted that large parallel jobs may experience a failure every 30 minutes on exascale platforms [122].

4.2.2 Resilience Methods

Resilience methods usually compensate for the computation/state loss by performing a backward or forward recovery. Backward recovery recreates an older state of an application through classic rollback recovery methods, such as system-level or application-level checkpoint/restart. Forward recovery typically handles errors by repairing the affected data structures. A correction procedure is invoked that may recover the intended values from a peer replica (redundant computing) or from checksums.

Application Based Fault Tolerance (ABFT)

ABFT methods can be used to tolerate transient hardware errors (bit flips). These methods rely on a deep understanding of the problem and exploit an inherent property/invariant specific to an algorithm/a data structure to recover from errors. Methods for matrix multiplication [67, 35], matrix factorization [47], Cholesky decomposition [36], and sparse linear solver [120] have been proposed. Results indicate that each method may have its merits based on the algorithm/data that it protects, which implies that future systems may use a mix of alternating methods per phase throughout application execution.

Checkpoint/Restart (CR)

Applications are periodically checkpointed, and their state are written to persistent storage. Upon failure, the application is rolled back to the last checkpoint (backward recovery), its state is retrieved from storage in all tasks, and execution resumes from this point. Multiple improvements have been proposed to alleviate the overhead of CR. Data aggregation [73] and data deduplication [17] shrink the checkpoint size, while in memory [136] and multi-level checkpointing [94, 14] improve the scalability and speed of CR.

Redundancy

Another resilience method is redundant computing [22, 57, 55, 59]. It aims at improving reliability and availability of systems by allocating two or more components to perform the same work. A recent study [91] showed that redundancy could be a very viable and even cost-effective approach for HPC on the cloud. Their approach to combine checkpointing and redundancy on Amazon

EC2 using a variable-cost spot market provides up to 7 times cheaper execution compared to the on demand default market.

Many HPC applications are comprised of multiple kernels that form a multi-phase pipeline. The above-mentioned methods are resilient to one or multiple types of faults with different overhead. Intuitively, there is no single solution that fits all scenarios while providing the best performance. Thus, a combination of methods enables the selection of the best resilience mechanism per application phase considering factors such as computation time and size of data that needs protection.

Vulnerabilities

Multiple metrics are commonly used to evaluate the failure of systems. The Failures in Time (FIT) rate is defined as a failure rate of 1 per billion hours. FIT is inverse proportional to MTBF (Mean Time Between Failures). The Architectural Vulnerability Factor (AVF) [124] is the probability that a fault (in microprocessor architecture) leads to a failure (in the program), defined over the fraction of time that data is vulnerable. The Program Vulnerability Factor (PVF) [123] allows insight into the vulnerability of a software resource with respect to hardware faults in a micro-architecture independent way by providing a comparison among programs with relative reliability. The Data Vulnerability Factor (DVF) [134] considers data spaces and the fraction of time that a fault in data will result in a failure. DVF takes into account the number of accesses as a contributor to fault occurrence. We introduce the term Live Vulnerability Factor (LVF) defined as:

$$LVF = L_v \times S_v,$$

where L_v is the (dynamic) length of the live range of an arbitrary data structure/variable v (vulnerability window), and S_v is the space required to hold the related data in memory. Length is measured as wall-clock time from the first set to the last use (live range) of a variable during execution. LVF takes into account time $\times$ space, which covers the effect of soft errors. Our metric is agnostic to architectural aspects of a processor (covered by AVF) and their impacts on programs (see PVF). It is also agnostic of the number of references (unlike DVF) as it considers both (a) written, incorrect results and (b) SDCs that may occur, even in the absence of write instructions (which others do not).

4.2.3 Fine-grained Task-based Scheduling

Task-based scheduling is an old concept in computing and has been extensively studied for multi-cores, e.g., in [33, 6]. The base idea is to assign resources to tasks until they complete. Multiple metrics have been taken into account to assess the performance of schedulers, e.g., maximizing throughput [92, 86], minimizing response time (or latency) [31, 32], guaranteeing deadlines [7, 44, 76], and maximizing fairness [8, 85]. Almost all of these metrics are performance-driven.

A Directed Acyclic Graph (DAG) is a generic model of a parallel program consisting of a set of tasks and their dependencies. A scheduler could traverse the DAG with a depth-first or breadth-first policy, or a combination or them. Generally, a high performance scheduler traverses a DAG in a way to minimize memory demands and cross-thread communication. Depth-first best exploits the cache locality and reduces memory demands, while breadth-first maximizes parallelism assuming an infinite number of available physical cores [1]. Breadth-first scheduling has the potential to lengthen the vulnerability window if it triggers new loads. As an example, consider a wide DAG, where task dependencies are rooted in data pipelines extending from a parent to a child. In a breadth-first schedule, the data of all parents across a level is live and vulnerable, which can be significant in size and may increase fast. In a depth-first schedule, only the data along the currently executing subtree branches is live, which is often significantly smaller.

Work stealing is a scheduling strategy where a thread grabs a task from the bottom of another thread's work stack (the oldest task). Multiple scenarios could occur with regard to the vulnerability window. If data is no longer needed beyond the "stolen" task, the window may shrink. If new loads are triggered, the actual impact depends on the subsequent actions, i.e., how long a task will be queued until its next execution (for temporal locality). We discuss the integration of end-to-end resilience into (a) OpenMP-based tasking and (b) task-based runtime systems before contrasting both.

4.3 Assumptions

We assume that the correctness of a *data structure* can be verified (through a *Checker* method) and the stored values can be recovered through a *Recover* method should an inconsistency be detected. Many algorithms commonly used in HPC, such as numeric solvers, have approximation methods based on convergence tests. These convergence tests could be used as the *Checker*. If an algorithm lacks a simple checking method or invariant, the *Checker* can be provided through comparison with a checksum over the data that was computed beforehand and stored in a safe region.[2] The *Recover* method can be supplied through the forward recovery phase in ABFT methods, or simply by restoring a light-weight deduplicated [17] or compressed [73] checkpoint of the data.

[2]Extra checks are added to guarantee the correctness of data stored in a safe region. A safe region is assumed to neither be subject to bit flips nor data corruption from the application viewpoint — yet, the the techniques to make the region safe remains transparent to the programmer. In other words, a safe region is simply one subject to data protection/verification via checking.

We further assume that the computation is idempotent *with respect to the encapsulated region*, i.e., if global data structures are changed inside the region, they have to be restored by the recovery method. In other words, if a method/region is called twice in a row, the result would be the same as the inputs (or global variables) remain unmodified by the computation (no side effects). This programming model does not restrict the application programmer significantly. (After all, any algorithm can be expressed in a functional language without side effects.) Yet, it allows the design of algorithmic solutions that are easier to read, debug and extend. This assumption ensures that we can retry a computation if needed, i.e., when no other recovery method exist (or if the other recovery methods have failed). We also allow communication inside regions (see Section 4.4.1).

4.4 End-to-End Resilience

Applications are typically composed of phases during which different algorithmic computations are being performed. Intermediate results are created and passed from phase to phase before the final result is generated. Our core idea is to exploit the live range of variables within and across phases, and to immediately perform a correctness check after the last use of any given variable. Live range analysis is a well-understood technique employed by compilers during code optimization, such as register allocation (among others). Fig. 4.1 depicts the overall idea. Based on the result of the check, either no action is taken (correct results), or the correct values are recovered (if detected as erroneous), or re-computation is performed (if erroneous but direct recovery has failed). The intuition here is to avoid the high overhead of frequent checks (e.g., after every use) while providing a guaranteed end-to-end correctness of the computation.

Assume a simple computation as follows:

$$\{x_1, ..., x_k\} \leftarrow compute(z_1, ..., z_n),$$

where `compute` is the last *use* of $z_1, ..., z_n$. The transformed resilient code is depicted in Fig. 4.2. A boolean array of size n is defined to hold the result of Checker methods on $z_1, ..., z_n$. After the *compute* call, each of the inputs $z_1, ..., z_n$ is checked and recovered if necessary. For simplicity, we assume that $z_1, ..., z_n$ cannot be recomputed in this example.[3] Thus, if the recovery fails, an exception will be thrown.

4.4.1 The Resilience Pragma

We propose a pragma-based resilience scheme for different types of control-flow constructs and show how the corresponding code is expanded to provide the extra end-to-end protection. Pragmas are introduced to indicate which boundaries should be protected and how protection

[3]In the next section, the recomputation case is addressed.

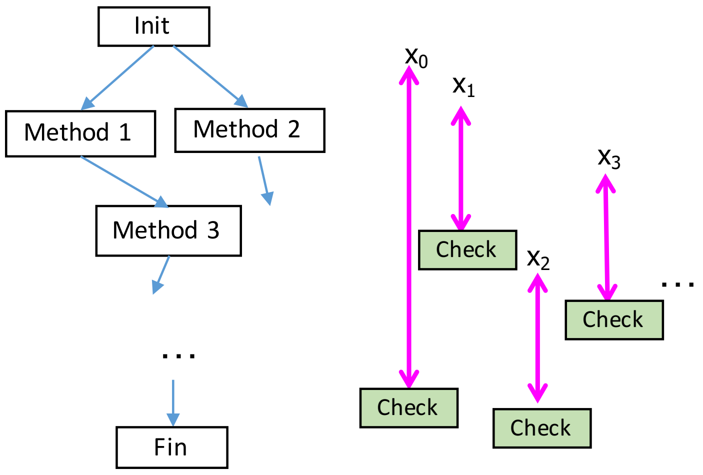


Figure 4.1: The control flow of an application with different variables in the source code and their live ranges is indicated by the arrows. A check is performed at the end of each live range to verify the correctness of the variable's value.

```
bool check[n];
do{
  {x_1, ..., x_k} = compute(z_1, ..., z_n);
  if(!(check[0] = Checker(z_1)))
    if(!Recover(z_1))
      throw unrecoverable;
  ...
}while(!check[0] || ... || !check[n]);
```

Figure 4.2: Code with checks

shall be realized. This allows us to cover the vulnerability window of different variables by automatically expanding codes through the compiler. The expanded code performs check and recovery actions on the vulnerable data. The recovery pragma has a simple, yet powerful and extendable interface with the following syntax:

$$\#\textbf{pragma}\ \textit{protect}\ [M]\ [\textit{Check}(f_1, ..., f_n)]\ [\textit{Recover}(g_1, ..., g_m)]$$
$$[\textit{Comm}]\ [\textit{Continue}]$$

where the resilience method, `M`, `CR` or `Redundancy (2/3)` (dual/triple), is an optional argument. `Check` and `Recover` receive a list of functions parameterized by each variable that needs protection. We use f to denote a checker, and g for a recovery method. A region that contains communication is annotated with the `Comm` keyword. The `Continue` keyword indicates that data is live beyond the current region, i.e., crossing phases, and requires end-to-end protection. At first, we focus on the control flows for the `Continue` keyword. We later discuss the resilience methods (`M`) in Section 4.4.1 and `Comm` in Section 4.4.1.

Fig. 4.3 (upper part) depicts three types of conventional flows: sequential (pipelined), data merge (join), and split (conditional) composition. Fig. 4.3(a) is a pipeline of data dependent compute phases. A set of independent computes is followed by a reduction dependent on previous outputs depicted in Fig. 4.3(b), and a set of conditional assignments based on predicates is shown in Fig. 4.3(c). All of these cases feature data flow across pragma regions (as indicated by the `Continue` keyword).

If an error strikes during the lifetime of phase-dependent variables, conventional resilience methods cannot provide any assistance as they are locally constrained to region boundaries.

The snippet in the lower part of Fig. 4.3 outlines the high-level representation of the final code. First, each `compute` method is transformed following the approach discussed in Fig. 4.2. These examples illustrate how the dependencies (`Continue` keyword) are considered in the code transformation. Sequential composition (Fig. 4.3(a) is turned into a nested structure. At the end of every compute, a corresponding `completed` flag is set. If a check fails on a data structure and the recovery also fails, then the related `completed` flag will be reset to repeat that computation. Note that the check/recover methods are not shown here as they are part of the generated code for each compute region. Fig. 4.3(b) (lower part) shows the composition for joining data flows. The code is organized in two levels of nesting. The `completed` flags are set after every compute method and in the generated code for $compute_n$. If check and recover for any of the $x_1, ...x_n - 1$ cases fail, then the respective `completed` flag is reset to trigger a recomputation of that specific data. Fig. 4.3(c) (lower part) depicts the transformed conditional composition. The conditions are directly expanded in the generated code to select the computation based on the respective predicate.

Integration with Resilience Methods

The protect pragma also supports resilience methods like CR and Redundancy. We perform the code transformation in two steps. At first, the pragma region with CR or Redundancy is transformed to an intermediate code that still contains the pragma, but the CR or Redundancy part is already expanded.

Fig. 4.4-(top) depicts a pragma with `CR`. The intermediate pragma code is shown in Fig. 4.4-(bottom). `Create_ckpt` takes a checkpoint of data and `Restore` recovers x_0 from the checkpoint.

Fig. 4.5-(top) depicts a pragma with the `Redundancy` keyword. The intermediate code is depicted in Fig. 4.5-(bottom). `Init` and `Fin` initialize and finalize a shadow process, respectively, for dual redundancy. [4] The routines `wake_up_shadow` and `sleep_shadow` indicate to the shadow

[4]A pool of shadow processes is dynamically maintained to avoid re-spawn costs.

```
User code
#pragma protect Check($f_0(x_0)$) Recover($g_0(x_0)$) Continue
    $x_1$ = compute$_1(x_0)$;
#pragma protect Check($f_1(x_1)$) Recover($g_1(x_1)$) Continue
    $x_2$ = compute$_2(x_1)$;
...
#pragma protect Check($f_{n-1}(x_{n-1})$,        \
                Recover($g_n(x_{n-1})$)
    $x_n$ = compute$_n(x_{n-1})$;
Generated pragma expanded code
bool completed[n];
while(!completed[n − 1]){
  ...
      while(!completed[1]){
        while(!completed[0]){
          //generated code for compute$_1$
          completed[0] = true;
        }
       //generated code for compute$_2$
       completed[1] = true;
       }
  ...
}
```

(a) Sequential

```
User code
#pragma protect Check($f_1(z_1)$) Recover($g_1(z_1)$) Continue
    $x_1$ = compute$_1(z_1)$;
#pragma protect Check($f_2(z_2)$) Recover($g_2(z_2)$) Continue
    $x_2$ = compute$_2(z_2)$;
...
#pragma protect Check($f_1(x_1)$, .., $f_{n-1}(x_{n-1})$), \
                Recover($g_1(x_1)$, ..., $g_{n-1}(x_{n-1})$)
    $x_n$ = compute$_n(x_1$, ..., $x_{n-1})$;
Generated pragma expanded code
while(!done){
  bool completed[n];
  while(!completed[0]){
    //generated code for compute$_1$
    completed[0] = true;
  }
  while(!completed[1]){
    //generated code for compute$_2$
    completed[1] = true;
  }
  ...
  //generated code for compute$_n$
}
```

(b) Join

```
User code
#pragma protect Check($f_1(z_1)$, ...),\
                Recover($g_1(z_1)$,...)
    if cond$_1$
        $x_1$ = compute$_{11}(z_1$, ...);
    else if cond$_2$
        $x_1$ = compute$_{12}(z_1$, ...);
    ...
    else
        $x_1$ = compute$_{1k}(z_1$, ...);
Generated pragma expanded code
if(cond$_1$)
  //generated code for compute$_{11}$
else if(cond$_2$)
  //generated code for compute$_{12}$
...
else
  //generated code for compute$_{1k}$
```

(c) Split

Figure 4.3: Resilience pragma for end-to-end protection of kernels: (a) Sequential, (b) Join, and (c) Split. The pragma-based user code is depicted in upper part and the code generated from pragma expansion is shown in lower part

```
#pragma protect CR Check($f_0(x_0)$)
    $x_1$ = compute$_1(x_0)$;
Create_ckpt($x_0$);
#pragma protect Check($f_0(x_0)$),Recover(Restore($x_0$))
    $x_1$ = compute$_1(x_0)$;
```

Figure 4.4: Pragma with Checkpoint/Restart (top) and its intermediate code generated from the pragma (bottom).

process that it shall start / stop the computation, respectively. .[5] Execution between primaries and shadows is coordinated using RedMPI [59].

[5]The former sends a wake_up signal from the the main process, the latter blocks on a port awaiting this signal while a shadow process is inactive.

```
#pragma protect Redundancy(2) Check(f_0(x_0)),\
                                         Recover(g_0(x_0))
    x_1 = compute_1(x_0);
---------------------------------------------------------
Init();     // creates a shadow process
            // sets the communicator,...
...
wake_up_shadow();    // primary wakes up shadow
#pragma protect Check(f_0(x_0)), Recover(g_0(x_0))
    x_1 = compute_1(x_0);
sleep_shadow();       // primary puts shadow to sleep
...
Fin();     // finalizes the shadow process
```

Figure 4.5: Pragma with Redundancy on top, generated (expanded) code with its pragma is shown on bottom

Regions with Communication

Recovery from regions that involve communication with other processes requires special care. Two approaches are feasible: (1) For message logging [78], multiple protocols have been proposed to log sent/received messages and to perform recovery without coordination with peer MPI tasks. (2) By aggregating the check results, if any of the checks failed, a recovery with recomputation is required, where all the required peer MPI tasks participate in the recomputation. To this end, a `Comm` keyword is added to the pragma to indicate that communication exists inside that pragma region.

4.4.2 Task-Based Resilience

An alternative to the pragma approach is to design a task-based programming scheme that implicitly provides end-to-end resilience. Tasking libraries are becoming more popular in the HPC community due to their more graceful load balancing and potentially asynchronous execution models, e.g., PaRSEC [20], OmpSs [50], the UPC Task library [74], and Cilk. Attempts have been made to add resilience to PaRSEC [24] and OmpSs [93]. PaRSEC focuses on soft errors, i.e., they take advantage of the algorithmic properties of ABFT methods to detect and recover from failures at a fine grain (task level) and utilize periodic checkpointing at a coarse grain (application). OmpSs uses CR and message logging at the task granularity to tolerate faults with re-execution.

Instead of focusing on a specific resilience approach, we target a more complex problem. We propose a tasking system that allows for different resilience methods to interact in an easily understandable and extendable manner. Fig. 4.6 depicts the overall idea where a `Resilience_-pre()` and a `Resilience_post()` are inserted before and after the task code. These methods implement CR, redundancy, and the necessary checks after the last use of a data structure.

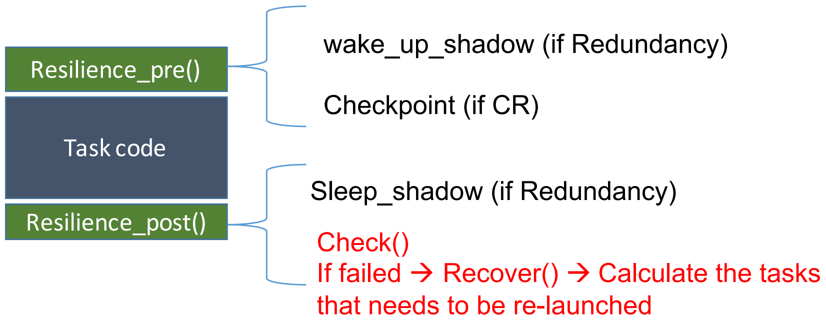

Resilience_pre()
Task code
Resilience_post()
wake_up_shadow (if Redundancy)
Checkpoint (if CR)
Sleep_shadow (if Redundancy)
Check()
If failed → Recover() → Calculate the tasks that needs to be re-launched

Figure 4.6: A task with the resilience code before and after

Resilient Task Objects

Fig. 4.7 depicts the class declaration of a resilient task. Resilience is provided through two methods that are called before and after the actual execution of a task, namely `resilience_pre`, `resilience_post`. In `resilience_pre`, depending on the resilience type of the task, CR or Redundancy, the `checkpoint` method or `wakeup_shadow` is called, respectively. In `resilience_post`, first the shadow process is put to sleep for under redundant execution. Then data structures that have their last use in the task are checked and corrected if needed. If the correction fails, a list of tasks are put into the scheduling queue to recompute the tainted data structures.

Fig. 4.8 shows the code skeleton for the resilient task. The `resilience_pre` and `resilience_post` methods are called before and after running the task and the results of execution are returned.

4.5 Case Studies and Results

We present examples of pragma- and task-based end-to-end resilience for two programs, matrix multiplication and a page ranking program, followed by experimental results.

4.5.1 Sequentially Composed Matrix Multiply

Computing a series of matrix multiplications is a problem that involves long ranges of live data and computational dependencies. This is an interesting test case to study conventional resilience properties and demonstrate how end-to-end resilience can improve on conventional methods. Fig. 4.9 shows the multiplication of three matrices ($A \times B \times D$), with the final result stored in matrix E.

The vulnerability windows for each variable are shown on right as ranges with arrows. Matrix A and B are vulnerable during the first multiplication, D is vulnerable during the second

```

class resilient_task_t{
  struct result_t{
    bool status; // SUCCESS or FAIL:
    vector <void *>failed_list;
  };

  task_t* task;
  vector <void (*)()> Check; //list of Checker funcs.
  vector <void (*)()> Recover;//list of Recover funcs.
  vector <void *> check_list; //list of vars to check
  int resilience_type;         //CR/Redundancy/none
public:
  bool resilience_pre(){
    if (resilience_type == CR){
      return checkpoint(task->input_list);
    }else if (resilience_type == REDUNDANCY){
      return wakeup_shadow();
    }
  }
  result_t* resilience_post(){
    result_t *result;
    if (resilience_type == REDUNDANCY)
      sleep_shadow();

    result = check_all(check_list);
    if (result->status == FAIL){ // failed check
      result = recover_all(result->failed_list);
      if (result->status == FAIL) // failed recovery
        put_tasks(result->failed_list);
    }
    return result;
  }
  run(){ task->run(); }
};
```

Figure 4.7: Resilient Task Class

multiplication, and matrix C is vulnerable during both. Matrix C has the longest window as it spans from when it is populated in the first multiplication to the end of the second multiplication where its *last use* occurs.

Check and Recover Methods

Huang and Abraham [67] first introduced an ABFT method for matrix operations based on adding a checksum row and column. Suppose we want to compute $C = A \times B$, where A is an $m \times n$ and B is a $n \times p$ matrix. A fully checksummed matrix A_{fc} is expanded as follows:

$$A_{fc} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} & \mathbf{a_{1,n+1}} \\ \vdots & \ddots & \vdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} & \mathbf{a_{m,n+1}} \\ \mathbf{a_{m+1,1}} & \cdots & \mathbf{a_{m+1,n}} & \mathbf{a_{m+1,n+1}} \end{bmatrix}$$

```
bool task_run(){
 this->resilience_pre();
 this->run();
 result_t *result = this->resilience_post();
 return result->status;
}
```

Figure 4.8: Resilient Task Skeleton

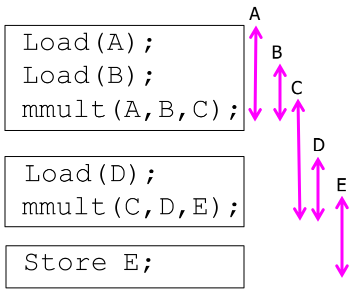


Figure 4.9: Sequentially composed matrix multiplication and the range of live variable

Elements in the last column are the checksums of each row and the elements in the last row are the checksums of each column. Should a soft error strike an element, it can be both detected and corrected. Similarly, a fully checksummed matrix can be created for B. Moreover, multiplying A_{fc} (without last column) by B_{fc} (without last row) results in C_{fc}. The result C_{fc} is also a fully checksummed matrix. The checksum property can be verified on matrices to detect and correct errors. With these checksums, single errors can be successfully detected and corrected. However, more than one error cannot be deterministically corrected [67].

In the next two sections we provide pragma-based and task-based solutions for sequentially composed matrix multiplication ($A \times B \times D$). The pragma-based approach stores per matrix checksums and uses OpenMP parallelism. The task-based approach computes per block checksums and is implemented using POSIX threads.

Pragma Expansion

We next present our end-to-end resilience approach for sequentially composed matrix multiplication for the pragma approach. Fig. 4.10 depicts the code for matrix multiplication with pragmas. There are three regions: computing the first multiplication, the second multiplication, and storing the results. Each region is highlighted with a pragma with check/recover methods on the inputs of subsequent computations. For example, consider `mmult(A,B,C)`, which takes A and B as inputs and calculates C (OpenMP parallelism). The Check function is parameterized with both checkers. There are two ways to recover A (or B): through checksum-based correction, or by loading A (or B) from non-volatile storage. The recovery procedure first attempts to `Correct(A)`; and if that fails, it will again (`load(A)`) from storage.

```
Matrix A, B, C, D, E;
Load(A);
Load(B);
#pragma protect Check(Checker(A),Checker(B)) \
                Recover(Correct(A),Load(A),  \
                        Correct(B),Load(B))  \
                 Continue

  mmult(A,B,C); // OpenMP threads
Load(D);
#pragma protect Check(Checker(C),Checker(D)) \
                Recover(Correct(C),Correct(D), \
                Load(D)) Continue
  mmult(C,D,E); //OpenMP threads
#pragma protect Check(Checker(E))            \
                Recover(Correct(E))
  Store (E);
```

Figure 4.10: Sequentially composed matrix multiplication with resilience pragma

Since C is live between the first and second pragmas, the keyword `Continue` is used in the first pragma. The same applies to matrix E in the second pragma.

This information is summarized in Table 4.1, where each pragma is numbered by the corresponding (sequential) region. The input variable(s) of each region are shown along with their corresponding Check and Recover methods. Note that matrix C can be recomputed using the code in region 0. Hence, another way to recover C is through executing region 0. Similarly, E can be recalculated via region 1. Thus, we chain two regions of code.

Table 4.1: Compiler-derived resilience information for sequentially composed matrix multiplication

Region	**Variable Name**	**Check method**	**Recover method**
0	A	Checker(A)	Correct(A), Load(A)
	B	Checker(B)	Correct(B), Load(B)
1	C	Checker(C)	Correct(C), Region(1)
	D	Checker(D)	Correct(D), Load(D)
2	E	Checker(E)	Correct(E), Region(2)

Fig. 4.11 depicts the final code. Every region is transformed into a while loop with checks after the computation. Lines 8 to 19 represent region 0. Since there exist two variables that need to be checked, a boolean array of size 2 is defined. After `mmult(A,B,D)`, in the loop, the Check

(function pointer) is always called and the Recover (function pointer) is called if the check fails. If the recovery fails, we do not have any other means to recover since we tried both methods, `Correct(A)` and `Load(A)`.

A boolean array of size 3 named `completed` is maintained for the 3 chained regions in this code, which indicates the correct completion of regions 0, 1, and 2. At the end of region 0/1/2, the corresponding flag is set (lines 21, 38, and 50).

The chaining shown in Table 4.1 is realized through the highlighted lines of code. If `Recover(C)` at line 29 fails, then the flag for region 0 will reset so that the computation of that region is repeated (similarly, for E in line 45).

Task-Based Matrix Multiply

We further devised a resilience-aware tasking system (see task-based resilience class/capabilities in Section 4.4.2) combined with a task-based runtime system and then implemented a blocked matrix multiplication on top of it that utilizes POSIX threads. We add checksums per block of a matrix. The checksum elements are colored in the 2 examples of Fig. 4.12. For a matrix of size 4×4, if the block size k is 2, then 20 extra elements are needed to hold the checksums. For a 6×6 matrix, 45 extra elements are needed. In practice, the size of a block (configured to fit into L1 cache with other data) is much larger than the extra space overhead for checksums.

Fig. 4.13 depicts an error scenario on the left and the recomputation performed after correcting the error. On the left side, an error in matrix A is shown in red. This error is propagated to the second row of C during the matrix multiplication ($C = A \times B$). Using per block checksums, we can correct the error in A. Next, parts of C needs to be recalculated. Parts of A and B involved in this computation are shown in green on the right side. In this case, we need to repeat 4 block multiplications, while in per matrix case, 8 blocks of multiplication are required (to create whole C matrix). Thus, per block checksums result in faster recovery as the recovery is of O(N^2), instead of O(N^3).

4.5.2 Experimental Results for Matrix Multiply

All experiments were conducted on a cluster of 108 nodes, each with two AMD Opteron 6128 processors (16 cores total) and 32GB RAM running CentOS 5.5, Linux kernel 2.6.32 and Open MPI 1.6.1.

We use 5 input sizes for square matrices from 512×512 to 2560×2560. The size of last level cache (L3) is 12MB, and only the first experiment ($N = 512$) completely fits in the L3 data cache. Thus, data is repeatedly loaded from main memory (DRAM) in all other experiments. We use 16 OpenMP threads that perform matrix multiplications in a blocked manner with a tile/block size of 32×32. Each thread needs 3 blocks to perform the multiplication. Thus, the

```
Matrix A, B, C, D, E;
Load(A);
Load(B);
bool completed[3]={false};
while(!completed[2]){
  while(!completed[1]){
    while(!completed[0]){
      bool check[2];      //2 check flags
      do{
        mmult(A,B,C);  //OpenMP threads
        if (!(check[0] = Check(A)))
          if (!Recover(A)){  //Correct(A)/Load(A)
            throw unrecoverable;
          }
        if (!(check[1] = Check(B)))
          if (!Recover(B)){  //Correct(B)/Load(B)
            throw unrecoverable;
          }
      }while(!check[0] || !check[1]);
      if (check[0] && check[1])
        completed[0]=true;
    }
    Load(D);
    bool check[2];     // 2 check flags
    do{
      mmult(C,D,E); //OpenMP threads
      if (!(check[0] = Check(C)))
        if (!Recover(C)){  //Correct(C)
          completed[0]=false;
          break;
        }
      if (!(check[1] = Check(D)))
        if (!Recover(D)){  //Correct(D)/Load(D)
          throw unrecoverable;
        }
    }while(!check[0] || !check[1]);
    if (check[0] && check[1])
    completed[1]=true;
  }
   bool check[1]; // 1 check flag
   do{
     store(E);
     if (!(check[0] = Check(E)))
       if (!(Recover(E))){ //Correct(E)
         completed[1]=false;
         break;
       }
   }while(!check[0]);
   if (check[0])
    completed[2] = true;
}
```

Figure 4.11: Code generated from pragmas with end-to-end resilience for sequentially composed matrix multiplication (mmult is parallelized with OpenMP)

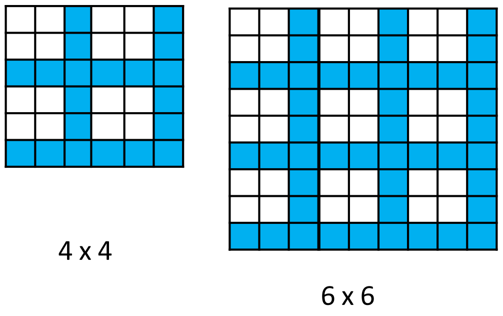


Figure 4.12: Examples of per block checksums for task-based approach. Different matrices: 4×4 and 6×6 with blocks of size 2×2 and checksums per block shown in blue.

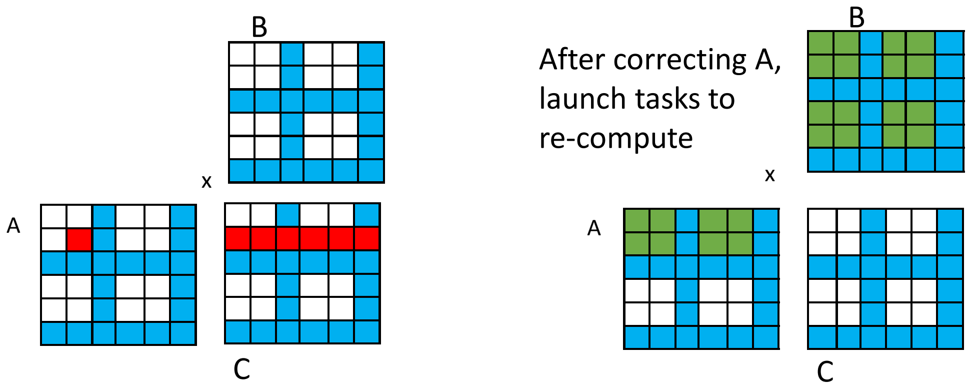


Figure 4.13: An error scenario in A propagated in C after $C = A \times B$ (left). Recocomputation as a part of the correction process is shown on the right side.

block size is selected as number of elements that can be accommodated in $\frac{1}{4}$th of the L1 data cache size (L1 is 64KB).

Table 4.2 depicts the overhead of basic resilience operations for a blocked matrix multiply under OpenMP: `Checksum`, `Checker`, and `Correct`. `Checksum` calculates the checksum value over the rows and columns of a given matrix. `Checker` calculates the checksum value for every row and compares it with the saved checksum. `Correct` is called when a check fails over a given row. It calculates the checksums over the columns to pinpoint the faulty element and corrects its value. Next, we present the overhead of these three resilience operations.

The time spent on computing checksums for an entire matrix (for pragma-based) changes from 0.007 to 0.19 seconds, which accounts for 0.62% to 0.06% of total application time (overhead). The checking time increases from 0.002 to 0.05 seconds while the overhead decreases from 0.17% to 0.01% when increasing the input size. The correction time increases from 0.005 to 0.23 seconds while the overhead decreases from 0.44% to 0.07% when increasing the input size.

The `Check` operation benefits from spatial locality (row-wise calculation) while the `Correct` operates on columns and does not benefit from caching. The gap between check and correct overhead becomes larger as we increase the input size.

In contrast, the checksum overhead per block (for task-based) is higher at 9.08% to 4.96%, and so is the checking cost for larger matrices (up to 0.57%) while checking smaller matrices results in less overhead (down to 0.23%). Correction on per-block basis is always considerable cheaper (0.0113%-0.0002%).

Next, we evaluate the space overhead due to checksums. The matrices' elements are double-precision floating-points. The size of extra elements per matrix can be calculated as

$$(n+1)^2 - n^2 = 2 \times n + 1,$$

which is of linear complexity. Fig. 4.14 depicts the space overhead. When using per-matrix checksums, the extra space for an input size of 512×512 is 40KB, which constitutes for 0.39% of all data. For an input size of 2560×2560, it is 200KB, which is equivalent to 0.07% space overhead. The overhead (both time and space) proportionally decreases when the input size is increased. This property makes these checksums a viable approach for resilience at large scale.

In task-based version, for an arbitrary block size of k, number of checksum elements is:
$Count_{checksum} = (n + \frac{n}{k})^2 - n^2 = \frac{n^2}{k^2} + 2 \times \frac{n}{k}$

The second bar in Fig. 4.14 depicts the space required to accommodate the five matrices (A, B, C, D, E) with end-to-end task-based resilience relative to no resilience. Here, the overhead remains constant at 6.34% when the matrix sizes increase.

Table 4.2: Time overhead of basic resilience operations on the whole matrix: Checksum, Check, Correct for square matrices of different sizes from 512×512 to 2560×2560.

	Per matrix checksum			**Per block checksum**		
size	**Checksum**	**Check**	**Correct**	**Checksum**	**Check**	**Correct**
512	0.62%	0.17%	0.44%	9.08%	0.23%	0.0113%
1024	0.17%	0.06%	0.18%	7.1%	0.37%	0.0023%
1536	0.15%	0.04%	0.17%	5.88%	0.46%	0.0008%
2048	0.06%	0.02%	0.08%	5.02%	0.51%	0.0004%
2560	0.06%	0.01%	0.07%	4.96%	0.57%	0.0002%

Fig. 4.15 contrasts the performance evaluation of sequentially composed matrix multiplication without resilience (left bar) with our end-to-end resilience (right bar). For the pragma-based solution (left half), fault-free execution changes from 1.14 ($n = 512$) to 42.11 seconds ($n = 2560$) when no correction needs to be triggered. In this case, end-to-end resilience has a 16% overhead at $n = 512$; for larger matrix sizes, this overhead is negligible (around 5%). Task-based execution (right half) results in slightly higher execution times and overheads between 6.3% and 15.71%. This can be attributed to scheduling and synchronization overhead, where the latter is required

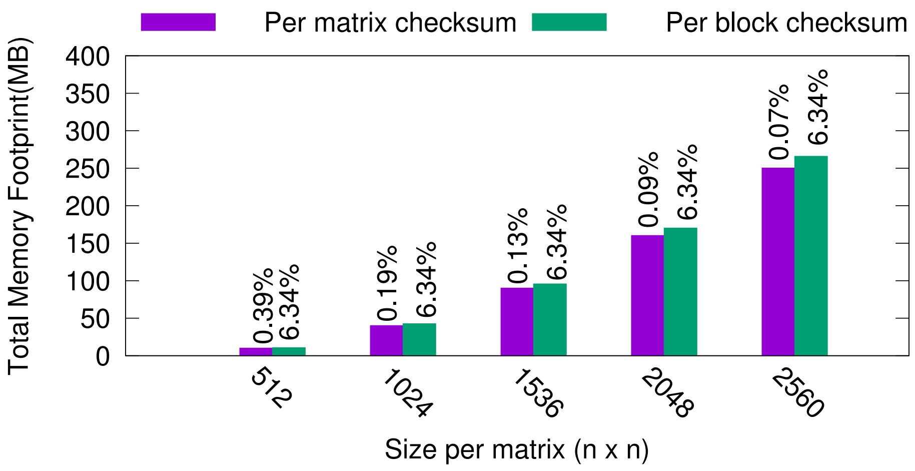


Figure 4.14: Space overhead of end-to-end resilience due to checksum for square matrices of different sizes from 512 × 512 to 2560 × 2560.

to establish task dependencies: A latter task may only start after its inputs become available, which are produced as outputs of a prior task. Such dependencies are enforced through lock-based synchronization by the tasking runtime systems.

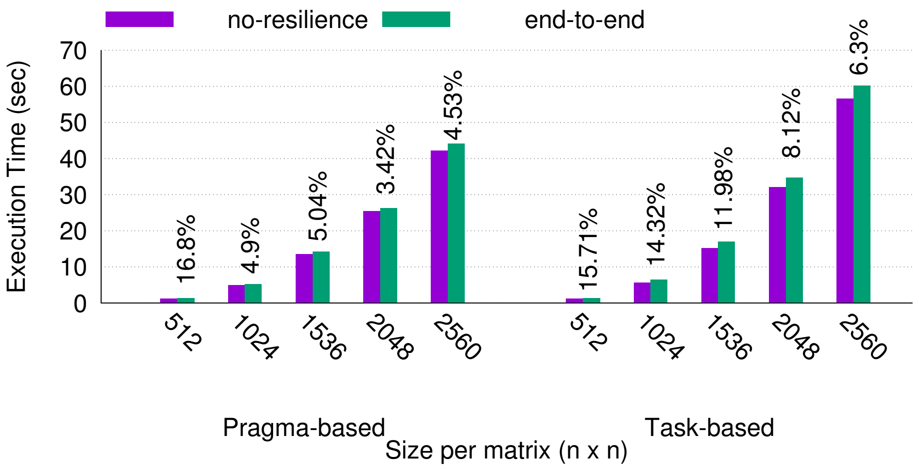


Figure 4.15: Failure-free execution time of end-to-end resilience (in seconds) for pragma based (16 OpenMP threads) on the left and task-based matrix multiplication (with 16 threads), overhead depicted on top of bars.

Performance under Faults and Resulting Failures

We next investigate the correlation between LVF and the likelihood of failures in matrices. The LVF can be computed from the vulnerability window of data structures (see Section 4.2.2). Fig. 4.16 depicts the LVF of each matrix under the failure-free execution of the application. The vulnerability size is 50.03MB and the vulnerability window depends on the live range of each matrix. C has the highest LVF, next comes E and then A. B and D have the same LVF, the smallest among the 5 matrices. This reflects the live ranges of the respective data structures during program execution.

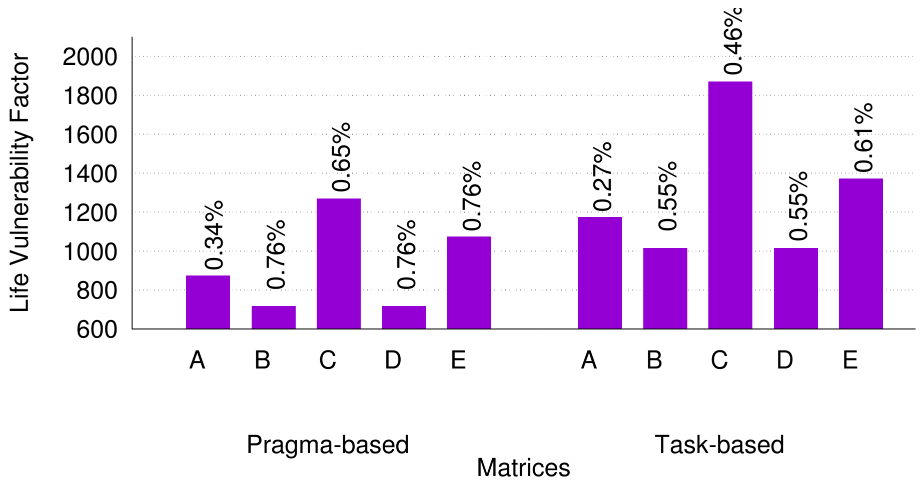


Figure 4.16: Live Vulnerability Factor

We also developed a model based on the actual runtime results to study the effect of fault injections in the system. Our simulation program uniformly injects faults across all data (all 5 matrices, each sized at 2560×2560).

We randomly inject faults during runtime based on different fault rates (λ)from 1 per 25 seconds to 1 per 45 seconds using a uniform distribution. We compare the robustness of ABFT (see Fig. 4.17) with our end-to-end resilience. We capture the result of fault injections by assigning the number of errors that were detected-corrected, undetected, and those faults that cause no error. Fig. 4.18 depicts the ABFT results where the matrix calculations are parallelized with OpenMP threads. We see that ABFT mostly but not always protects C and E (except one case in $\lambda = \frac{1}{25}$). However, since no checks are performed on the inputs, those errors remain undetected and, hence, result in incorrect final outputs (failure cases).

```
Matrix A, B, C, D, E;
Load(A);
Load(B);
mmult(A,B,C);
if (!Check(C))
  if(!Recover(C))
      throw unrecoverable;
Load(D);
mmult(C,D,E)
if (!Check(E))
  if(!Recover(E))
      throw unrecoverable;
```

Figure 4.17: Sequentially composed matrix multiplication with ABFT

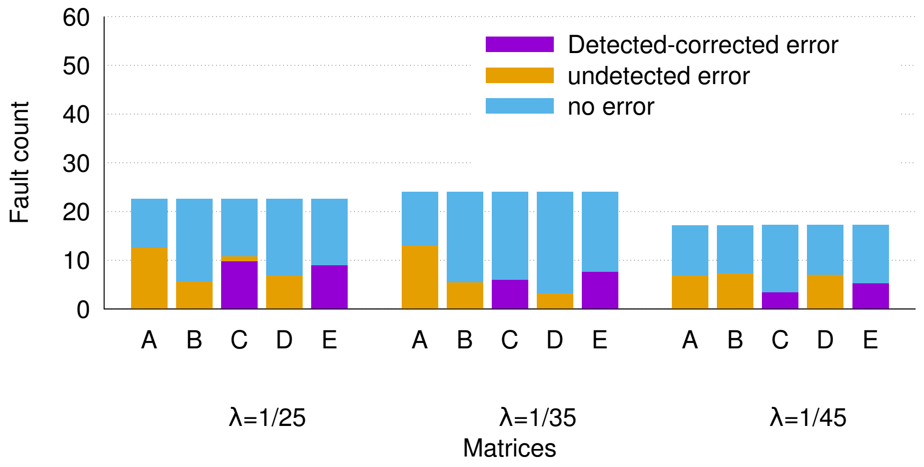


Figure 4.18: ABFT with fault injection

Fig. 4.19 depicts different fault injection scenarios for pragma-based execution. The y axis shows the number of faults. For end-to-end resilience, some of the faults result in detectable errors that are all subsequently corrected by end-to-end resilience (i.e., they hit the data when its *live*), depicted as the lower part of the stacked bar. Others do not result in failure as they hit stale data (depicted by the upper portion of each bar). In other words, end-to-end resilience never resulted in erroneous results. Furthermore, the distribution of corrected inject counts over matrices loosely resemble the distribution of the LVF across matrices in Fig. 4.21. This is significant as injection experiments and LVF analysis thus validate each other. Slight differences can be attributed to the fact the LVF is based on failure-free execution while Fig. 4.19 is based on repeated executions for some corrections for certain detected errors (e.g., in the input matrices).

Without end-to-end resilience, many errors would remain undetected, even if the checksum-based ABFT version of matrix multiply were utilized, since inputs A, B, and D remain unchecked,

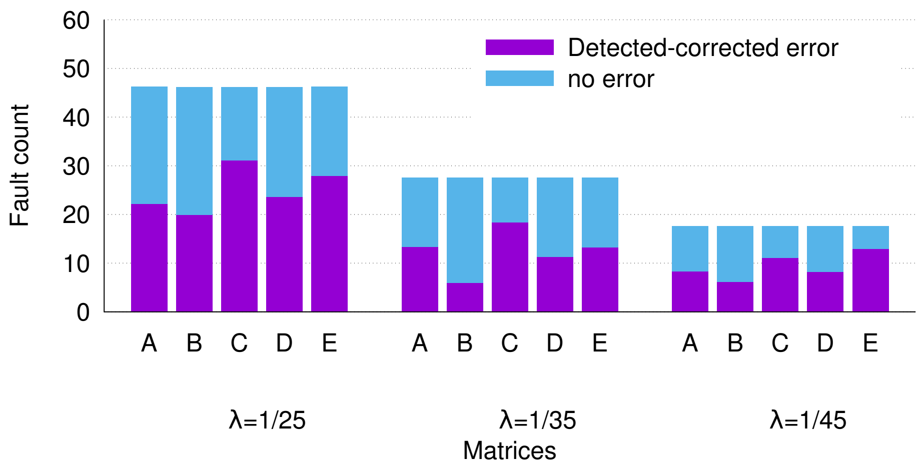


Figure 4.19: Fault injection scenario with different fault rates (λ per seconds) over 100 simulated runs (pragma-based)

and the result of the first multiply, C, also remains unchecked in the second multiply. Furthermore, faults in any input matrix may cause many errors in the result matrix, which cannot be repaired.

Fig. 4.20 depicts the corresponding results for task-based end-to-end resilience. We observe a similar distribution across matrices to Fig. 4.19, yet the number of faults in slightly higher since the task-based approach requires longer to execute. Consequently, more faults are injected at the same fault rate. In order to better understand the fault behavior, we calculated the LVF for the 5 matrices under various fault rates. Fig. 4.21 depicts the results. The first bar shows the baseline ($\lambda = 0$). Fault injection assesses the vulnerability window of data structures, yet only our end-to-end resilience can successfully contain the errors and produce the correct final results.

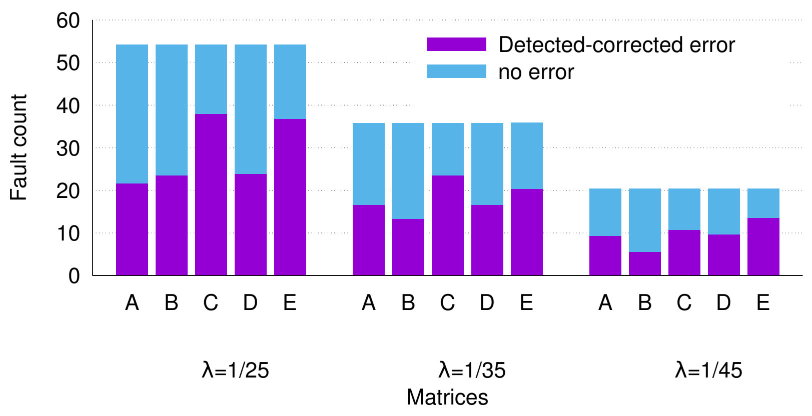


Figure 4.20: Fault injection scenario with different fault rates (λ per seconds) over 100 simulated runs (task-based)

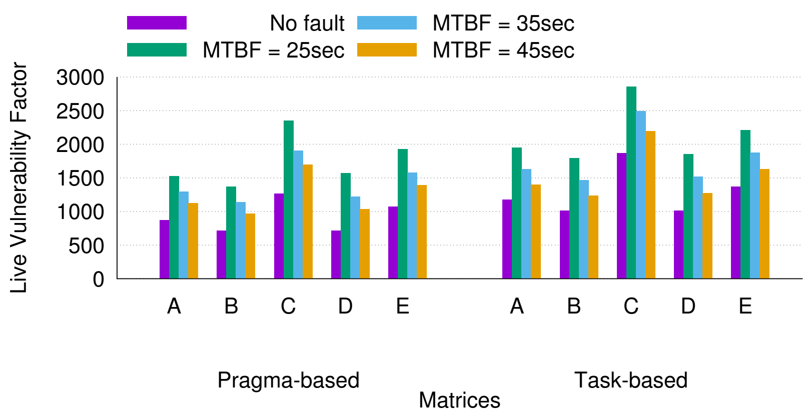


Figure 4.21: LVF with fault injection

Fig. 4.22 depicts the average completion times after fault injection, where all faults that resulted in an error were detected and corrected by end-to-end resilience. The pragma-based approach (left half) resulted in 20%-51% overhead for fault rate from 1 per 45 seconds to 1 per 25 seconds. Notice that such a high fault rate results in one or more faults per execution, some of which result in detectable errors that are subsequently corrected at the expense of executing recovery code. For the task-based approach, the overhead ranged from 7.25%-30.60%, which is noticeably lower than pragma-based since the former requires corrections of entire matrices while the latter can more frequently correct values locally within a block.

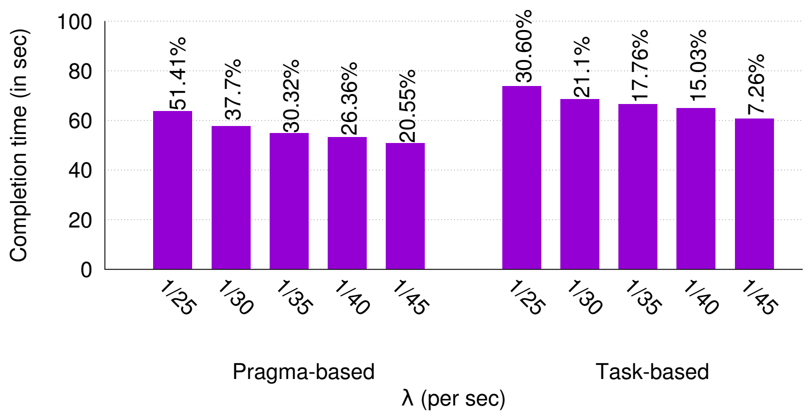
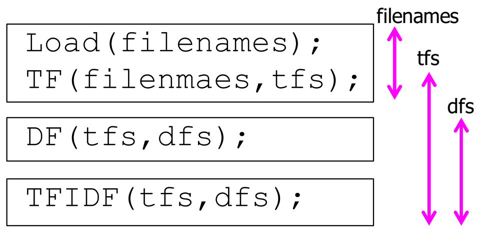


Figure 4.22: Completion time under fault injection scenario with different fault rates (λ per seconds) over 100 simulated runs

4.5.3 TF-IDF

We further assessed the resilience capabilities for an MPI-based benchmark. We ported a term frequency/inverse document frequency (TF-IDF) benchmark for document clustering based on prior work [135]. TF-IDF is a classification technique designed to distinguish important terms in a large collection of text documents, which is the basis for page ranking with applications in data mining and search engines. The classification is broken into two steps. (1) TF calculates the frequency of a term on a per document basis. (2) DF counts the number of occurrences of a given term (document frequency). The final result is $tfidf = TF \times log\frac{N}{DF}$. Note that TF is a local computation while DF is global across all documents. As a result, the DFs need to be aggregated.

Fig. 4.23 depicts the steps in the TF-IDF benchmark. At first, the names of files are loaded. Then the term frequency (TF) method is called with `filenames` as input and `tfs` as output. Next, the document frequency (DF) is called with `tfs` as input and `dfs` as output. Finally, the $tfidf$ value is computed for every term with a `TFIDF` call with `tfs` and `dfs` as input parameters. Fig. 4.23 also depicts the vulnerability windows (live ranges) of variables shown by the arrows on the right. The DF method contains MPI communication for the aggregation of document frequencies across all MPI ranks.



```
Load(filenames);
TF(filenmaes,tfs);

DF(tfs,dfs);

TFIDF(tfs,dfs);
```

Figure 4.23: Vulnerability windows in TF-IDF

Check and Recover Methods

TF-IDF does not have any check provided by the algorithm. Thus, we compute a SHA-1 hash over the data as the checksum. To demonstrate the capabilities of end-to-end resilience, we use a combination of redundancy and CR in this case study. CR provides a restore function, which we use as a recovery method.

Pragma Expansion

End-to-end resilience for TF-IDF can be provided by augmenting the code with three pragmas over as many regions (see Fig. 4.24). The first region is executed under redundancy while the second region is protected with CR. The data of `tfs` is live across all three regions, while `dfs` is live across the last two pragma regions (`Continue` keyword). Inside the DF method, MPI communication is used and, consequently, the `Comm` keyword is added to the second pragma. Table 4.3 depicts the regions, the input variable(s) to each region and the check and recover method per variable. Note that `tfs` is still live in region 2. Thus, no check should be carried on `tfs` in region 1. Thus, region 1 does not have check/recover methods. The chaining of regions is also shown in Table 4.3. In region 2, `tfs` can be recovered by recomputing region 0. Similarly, `dfs` can be calculated from region 1.

Table 4.3: Compiler-derived resilience information for TF-IDF

Region	**Variable Name**	**Check method**	**Recover method**
0	fn	Checker(fn)	Load(fn)
1	tfs	–	–
2	tfs	Checker(tfs)	Recover(tfs), Region(0)
	dfs	Checker(dfs)	Region(1)

```
vector<string> filenames;// input
vector<Dictionary> tfs;  // output of Region 1
map <string,int> dfs;   // output of Region 2

Load(filenames);
#pragma protect Redundancy Check(Checker(filenames))\
                           Recover(Load(filenames)) \
                           Continue
  TF(filenames, tfs);
#pragma protect CR Check(Checker(tfs)) Comm Continue
  DF(tfs, dfs); //contains MPI calls
#pragma protect Check(Checker(tfs), Checker(dfs))
  TFIDF(tfs,dfs);
```

Figure 4.24: TF-IDF with resilience pragma

Fig. 4.25 depicts the final code. Since there are three chainings, a boolean array of size 3 is defined to show the completion of regions. The code related to chaining is highlighted.

```
MPI_Init();
Init();          // creates a shadow process
                 // sets the communicator,âĂę
vector<string> filenames;// input
vector<Dictionary> tfs;  // output of Region 0
map <string,int> dfs;// output of Region 1
Load(filenames);
bool completed[3]={false};
while(!completed[2]){
  while(!completed[1]){
    while(!completed[0]){
      wake_up_shadow();  // primary wakes up shadow
      bool check[1];    // 1 check flag
      do{
        TF(filenames, tfs);
        if (!(check[0] = Check(filenames)))
          if (!Recover(filenames))
            throw unrecoverable;
      }while(!check[0]);
      sleep_shadow();// primary puts shadow to sleep
      if (check[0])
        completed[0]=true;
    }
    Create_ckpt(tfs); // checkpoint "tfs"
    DF(tfs, dfs); // contains MPI calls
    if (1)
      completed[1]=true;
  }
   bool check[2]; // 2 check flags
   do{
    TFIDF(tfs,dfs);
    if (!(check[0] = Check(tfs)))
      if (!(Recover(tfs))){
        completed[0]=false;//recompute region 0
        break;
      }
      if (!(check[1] = Check(dfs)))
        if (1){
          completed[1]=false;//recompute region 1
          break;
        }
   }while(!check[0] && !check[1]);
   if (check[0] && check[1])
     completed[2] = true;
}
Fin();                // finalizes the shadow process
MPI_Finalize();
```

Figure 4.25: Code generated with end-to-end resilience for TF-IDF

Region 0 extends from line 12 to 20, where the shadow rank is signaled to engage in redundant execution. Lines 24 to 25 comprise region 1, and the final region is represented by lines 29 to 42.

4.5.4 Experimental Results of TF-IDF

We used 750 text books with a total size of 500MB for the TF-IDF benchmark with 4 MPI ranks. We performed the evaluation with 4 input sizes: 125MB, 250MB, 375MB, and 500MB. Table 4.4 shows the overhead of different resilience operations. `Checksum(fn)` and `Check(fn)` each take between 0.8ms to 20ms when increasing the input size from 125MB to 500MB. This corresponds to an overhead of 0.008% to 0.005%. A `load(fn)` includes reading the input directory structure to find filenames and takes between 0.047 (0.47%) to 0.19 seconds (0.5% overhead) for input sizes from 125MB to 500MB.

The `tfs` values are calculated on a per-file basis. Notice that `checksum(tfs)` computes the SHA-1 hash over the `tfs` of every file. This allows us to perform fine-grained checks and, hence, much faster recovery. Calls to `ckecksum(tfs)` take between 0.54 (5.46% overhead) and 2.38 seconds (6.19% overhead). The `check(tfs)` call computes a SHA-1 hash over the data and compares it with the in-memory stored checksum. It takes between 0.6 (6% overhead) and 2.56 seconds (6.65% overhead). The recovery for `tfs` includes restoring the values from the previous checkpoint. As the check is performed per file, recovery is fast and takes around 7.4 micro seconds on average. Note that this is independent of the input size. The last two rows in Table 4.4 depict overheads for `checksum(dfs)` and `check(dfs)`, which range from 1.19% to 0.73% when increasing the input size from 125MB to 500MB, respectively.

Table 4.4: Time overhead (%) of basic resilience operations in TF-IDF: Checksum, Check, Recover (for 4 MPI ranks)

	125MB	**250MB**	**375MB**	**500MB**
checksum(fn)	0.008%	0.006%	0.005%	0.005%
check(fn)	0.008%	0.006%	0.005%	0.005%
load(fn)	0.48%	0.47%	0.46%	0.50%
checksum(tfs)	5.46%	5.75%	5.92%	6.19%
check(tfs)	6.09%	6.43%	6.28%	6.66%
restore(tfs)	7.52E-05%	3.64E-05%	2.30E-05%	1.79E-05%
checksum(dfs)	1.19%	0.95%	0.8%	0.73%
check(dfs)	1.13%	0.90%	0.75%	0.73%

Fig. 4.26 depicts the time for failure-free execution of TF-IDF without resilience and compares that to our end-to-end resilience. The overhead increases from 14% for an input size of 125MB to 16.2% for 500MB. Although we used SHA-1 hash as the checksum values over the data, the overhead is still small considering the benefits of end-to-end resilience provided by our approach.

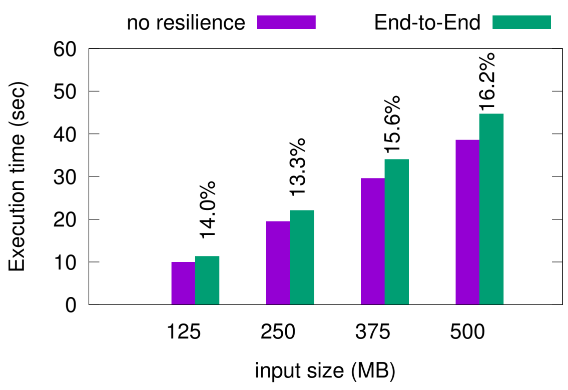


Figure 4.26: Failure-free execution time of TF-IDF: without resilience vs. end-to-end resilience (for 4 MPI ranks)

4.6 Conclusion

In this chapter, we proposed an automatic approach for building highly modular and resilient applications such that resilience concerns are separated from algorithms. Our approach requires a minimal effort by application programmers and is highly portable.

We introduced and investigated the significance of the live vulnerability factor, which takes into account the live range of a data structure and its storage space to provide insight into the likelihood of failures. We introduced an effective set of building blocks for detection and correction of soft errors through *Check* and *Recover* methods for arbitrary data structures. We provided two approaches, pragma- and task-based, to implement end-to-end resilience. We demonstrated the effectiveness of end-to-end resilience for sequentially composed matrix multiplications and TF-IDF under failure-free execution and fault scenarios.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this section, we summarize our efforts and present conclusions drawn from this research.

Chapter 2 details the shortcomings in the software stack for Xeon Phi manycore processors and presents our innovations in this area. We propose Snapify, an application-transparent, coordinated approach to take consistent snapshots of host and coprocessor processes of an offload application. We then use Snapify to implement three new capabilities for offload applications: application-transparent checkpoint and restart, process migration, and process swapping. To the best of our knowledge, this is the first work that provides such capabilities for offload applications on the Xeon Phi. In order to transfer the snapshots in a high performance fashion, we propose a fast, remote file access service based on RDMA that speeds up the storage and retrieval of snapshots from the host's file system. We further evaluate Snapify on several OpenMP and MPI benchmarks. Our results show that Snapify imposes negligible overhead in normal execution of offload applications (less than 5%), and the overheads due to the capture of snapshots in checkpoint and restart, process swap, and process migration are small enough to make it practical to use Snapify for a variety of offload applications

Chapter 3 details shortcomings in redundant computing and presents our innovations in replenishing failed replica processes. We devise a generic high performance node cloning service under divergent node execution (DINO) for recovery. DINO clones a process onto a spare node in a live fashion. We integrate DINO into the MPI runtime under redundancy as a reactive method that is triggered by the MPI runtime to forward recover from hard errors, e.g., a node crash or other hardware failures. Next, we propose a novel Quiesce algorithm to overcome divergence in execution without excessive message logging. Execution of replicas is not in a lock-step fashion, i.e., can diverge. Our approach establishes consistency through a novel, scalable multicast variant of the traditional (non-scalable) bookmark protocol [108] and resolves inconsistencies through exploiting the symmetry property of redundant computing. To evaluate this work, we use the NAS parallel benchmark suite besides Sweep3D and LULESH. We use kernels from BT, CG, FT, IS, LU, MG, SP, and LU to evaluate our implementation. The time to regain dual redundancy after a hard error varies from 5.60 seconds to 90.48 seconds depending on process image size and cross-node transfer bandwidth, which is short enough to make our approach practical. In order to better understand the performance of redundant computing at a very large scale, we provide a model with low error (at most 6%). Our model estimates job execution time under redundancy. We validate our model on the Stampede supercomputer and extrapolate results under this model to extreme scale with a node MTTF of 50 years and show that dual redundancy+cloning outperforms triple redundancy within the exascale node count range, yet at 33% lower power and compute requirements.

Chapter 4 presents our efforts in providing an infrastructure that allows for computation with end-to-end resilience across an entire application run that requires minimal programming effort while providing maximal code modularity. We present a systematic approach for end-to-end resilience with minimal assistance from the application developer. We separate the resilience aspects from the algorithmic side of computation aiming for portability and modularity of HPC codes. In end-to-end resilience, checks and appropriate recovery methods are automatically inserted into the application source code and a computationally equivalent code is generated that is fault resilient. In case studies, we demonstrate the applicability of our approach and its benefit over conventional resilience techniques. We show that the overhead for check/recover methods in a sequentially composed numeric kernel is negligible while the overall space overhead is less than 1% for per matrix checksums and less than 7% for per-block checksums. Our experimental results show that check/recover methods in a code that combines MPI with end-to-end resilience results in less than 7% overhead or an overall overhead of 14 - 16% compared to a baseline without resilience.

5.2 Future Work

In this section, we present future research directions.

- Task scheduling is one of the oldest concepts in computing systems and has been extensively studied for multi-cores. The base idea is to assign resources to tasks until they complete. Multiple metrics have been taken into account to assess the performance of schedulers, e.g., maximizing throughput, minimizing response time (or latency), guaranteeing deadlines, and maximizing fairness. Almost all of these metrics are performance-driven. A Directed Acyclic Graph (DAG) is a generic model of a parallel program consisting of a set of tasks and their dependencies. A scheduler could traverse the DAG with a depth-first or breadth-first policy or a combination or them. Generally, a high performance scheduler traverses a DAG in a way to minimize memory demands and cross-thread communication. Depth-first best exploits the cache locality and reduces memory demands, while breadth-first maximizes parallelism assuming an infinite number of available physical cores. Breadth-first scheduling has the potential to lengthen the vulnerability window if it triggers new loads. As an example, consider a wide DAG where in the top level tasks some arbitrary data are loaded. In a breadth-first scheduling, after the execution of the first level of tasks, all of the data is live and vulnerable. Work stealing is a scheduling strategy where a thread grabs a task from the bottom of another thread's work stack (the oldest task). Multiple scenarios could occur with regard to the vulnerability window. It can potentially shrink the window if data is no longer needed beyond the "stolen" task. If it triggers new loads, the actual impact depends on the subsequent actions, i.e., how long it will be queued until next execution. We want to investigate the effects of scheduling policies on the liveness and further design a data-centric resilience-aware task scheduling system based on the our recent work. Such a system should provide a performance boost while improving the vulnerability window, and hence, increases resilience.

- Multiple metrics are commonly used to evaluate the failure of systems. The Failures in Time (FIT) rate is defined as a failure rate of 1 per billion hours. FIT is inverse proportional to MTBF (Mean Time Between Failures). The Architectural Vulnerability Factor (AVF) is the probability that a fault (in architecture) leads to a failure (in the program), defined over the fraction of time that data is vulnerable. The Program Vulnerability Factor (PVF) allows insight into the vulnerability of a software resource with respect to hardware faults in a micro-architecture independent way by providing a comparison among programs with relative reliability. The Data Vulnerability Factor (DVF) considers data spaces and the fraction of time that a fault in data will result in a failure. DVF takes into account the number of accesses as a contributor to fault occurrence. One common problem in these metrics is that they are abstract numbers that only capture a high-level aspect of the system and do not capture the details of recent trends in architecture. Recent innovations in hardware, such as deep memory hierarchies and heterogeneous cores, provide a performance boost at the price of adding to system complexity. Persistent memory technologies like Phase Change Memory (PCM), magnetic RAM (MRAM), and Memristors are positioned to be placed as burst buffers between DRAM and storage to alleviate the speed gap using a non-volatile abstraction. Thus, a comprehensive resilience metric that captures the architectural properties of all layers of a system is missing. The fault propagation probability is another aspect of a deep memory hierarchy that should be captured by such metric.

REFERENCES

[1] Intel threading building blocks documentation. https://software.intel.com/en-us/tbb-documentation, .

[2] Stampede supercomputer. tacc.utexas.edu/stampede.

[3] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 167–176. IEEE Computer Society, 1999.

[4] L. Alvisi, S. Rao, S. A. Husain, A. de Mel, and E. Elnozahy. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 242–249. IEEE Computer Society, 1999.

[5] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen. Scalable metagenomic taxonomy classification using a reference genome database. *Bioinformatics (Oxford, England)*, 29(18):2253–2260, Sept. 2013. ISSN 1367-4811. doi: 10.1093/bioinformatics/btt389. URL `http://dx.doi.org/10.1093/bioinformatics/btt389`.

[6] J. H. Anderson and J. M. Calandrino. Parallel task scheduling on multicore platforms. *SIGBED Rev.*, 3(1):1–6, Jan. 2006. ISSN 1551-3688. doi: 10.1145/1279711.1279713. URL `http://doi.acm.org/10.1145/1279711.1279713`.

[7] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2400-1. doi: 10.1109/ECRTS.2005.6. URL `http://dx.doi.org/10.1109/ECRTS.2005.6`.

[8] M. Andrews. Instability of the proportional fair scheduling algorithm for hdr. *Wireless Communications, IEEE Transactions on*, 3(5):1422–1426, Sept 2004. ISSN 1536-1276. doi: 10.1109/TWC.2004.833419.

[9] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. 2009.

[10] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, and Others. The Opportunities and Challenges of Exascale Computing. Technical report, U.S. Deptartment of Energy, 2010.

[11] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[12] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing Division, Mar. 1994.

[13] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte. MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:26–33, 2001.

[14] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063427. URL `http://doi.acm.org/10.1145/2063384.2063427`.

[15] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. pages 97–108, June 2012.

[16] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[17] S. Biswas, B. R. d. Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong. Exploiting data similarity to reduce memory footprints. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 152–163, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.24. URL `http://dx.doi.org/10.1109/IPDPS.2011.24`.

[18] S. Böhm and C. Engelmann. File I/O for MPI Applications in Redundant Execution Scenarios. In *Euromicro International Conference on Parallel, Distributed, and network-based Processing*, Feb. 2012.

[19] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6): 36–45, Nov 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2013.98.

[21] G. Bosilca et al. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. 2002.

[22] R. Brightwell, K. Kurt Ferreira, and R. Riesen. Transparent redundant computing with MPI. EuroMPI'10, pages 208–218, Berlin, Heidelberg, 2010. Springer-Verlag.

[23] S. Cadambi, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar. COSMIC: Middleware for high performance and reliable multiprocessing on Xeon Phi coprocessors. pages 215–226, June 2013.

[24] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. Design for a soft error resilient dynamic task-based runtime. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 765–774, May 2015. doi: 10.1109/IPDPS.2015.81.

[25] G. Cao and M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(12):1213–1225, Dec. 1998.

[26] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, Aug. 2009. ISSN 1094-3420. doi: 10.1177/1094342009106189. URL `http://dx.doi.org/10.1177/1094342009106189`.

[27] 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.

[28] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *Proceedings of the 19th International Conference on Computer Communications and Networks, IEEE ICCCN 2010, Switzerland, August 2-5, 2010*, pages 1–8. IEEE, 2010. ISBN 978-1-4244-7115-7.

[29] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014. ISSN 2313-8734. URL `http://superfri.org/superfri/article/view/14`.

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

[31] C. Chekuri, S. Im, and B. Moseley. Minimizing maximum response time and delay factor in broadcast scheduling. In A. Fiat and P. Sanders, editors, *Algorithms - ESA*

2009, volume 5757 of *Lecture Notes in Computer Science*, pages 444–455. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04127-3. doi: 10.1007/978-3-642-04128-0_40. URL http://dx.doi.org/10.1007/978-3-642-04128-0_40.

[32] C. Chekuri, S. Im, and B. Moseley. Online scheduling to minimize maximum response time and maximum delay factor. *Theory of Computing*, 8(7):165–195, 2012. doi: 10.4086/toc.2012.v008a007. URL http://www.theoryofcomputing.org/articles/v008a007.

[33] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: 10.1145/1248377.1248396. URL http://doi.acm.org/10.1145/1248377.1248396.

[34] S. Chen, G. Bronevetsky, L. Peng, B. Li, and X. Fu. Soft error resilience in Big Data kernels through modular analysis. *The Journal of Supercomputing*, pages 1–27, 2016. ISSN 1573-0484. doi: 10.1007/s11227-016-1682-2. URL http://dx.doi.org/10.1007/s11227-016-1682-2.

[35] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1628–1641, Dec 2008. ISSN 1045-9219. doi: 10.1109/TPDS.2008.58.

[36] Z. Chen and P. Wu. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2014. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2320502.

[37] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis, SC '12, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389075`.

[38] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. pages 273–286, 2005.

[39] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. Software Aging Analysis of the Linux Operating System. ISSRE '10, pages 71–80. IEEE Computer Society, 2010.

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

[41] J. T. Daly. Resilience panel, 3 2013. URL `http://www.cs.sandia.gov/Conferences/SOS13/presentations/Daly_Session2.pdf`. 13th Workshop on Distributed Supercomputing.

[42] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Whitepaper, Dec. 2009. URL `http://www.csm.ornl.gov/~engelman/publications/debardeleben09high-end.pdf`.

[43] R. F. V. der Wijngaart and H. Jin. NAS parallel benchmarks, multi-zone versions. Technical Report NAS-03-010, NASA Advanced Supercomputing Division, July 2003.

[44] U. Devi and J. Anderson. Tardiness bounds under global edf scheduling onÂăaÂă-multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008. ISSN 0922-6443. doi: 10.1007/s11241-007-9042-1. URL `http://dx.doi.org/10.1007/s11241-007-9042-1`.

[45] W. R. Dieter. User-level checkpointing for linuxthreads programs. In *USENIX Annual Technical Conf. (FREENIX Track)*, pages 81–92, 2001.

[46] J. Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[47] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 225–234, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145845. URL `http://doi.acm.org/10.1145/2145816.2145845`.

[48] A. Duda. The effects of checkpointing on program execution time . *Information Processing Letters*, 16(5):221 – 229, 1983.

[49] J. Duell. The design and implementation of Berkeley Labs Linux Checkpoint/Restart. Technical report, Lawrence Berkeley National Laboratory, 2003.

[50] A. Duran, E. AyguadÃľ, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. URL `http://dblp.uni-trier.de/db/journals/ppl/ppl21.html#DuranABLMMP11`.

[51] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS) 2012*, Macau, China, June 18-21 2012.

[52] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of sdc on the gmres iterative solver. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1193–1202, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3800-1. doi: 10.1109/IPDPS.2014.123. URL `http://dx.doi.org/10.1109/IPDPS.2014.123`.

[53] E. Elnozahy and W. Zwaenepoel. Replicated distributed processes in Manetho. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 18–27, jul 1992.

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

[55] C. Engelmann and S. Böhm. Redundant Execution of HPC Applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, PDCN' 11, pages 31–38. ACTA Press, Calgary, AB, Canada, Feb. 15-17, 2011.

[56] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive Fault Tolerance Using Preemptive Migration. PDP '09, pages 252–257. IEEE Computer Society, 2009.

[57] 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. SC '11, pages 44:1–44:12. ACM, 2011.

[58] 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, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063443. URL `http://doi.acm.org/10.1145/2063384.2063443`.

[59] D. Fiala, F. Mueller, C. Engelmann, K. Ferreira, and R. Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *Proceedings of the 2012 IEEE conference on Supercomputing*, SC '12, 2012.

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

[61] A. Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*, Feb. 2016.

[62] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[63] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[64] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 989–1000. IEEE Computer Society, 2011.

[65] http://www.criu.org/.

[66] http://www.top500.org/.

[67] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518–528, June 1984. ISSN 0018-9340. doi: 10.1109/TC.1984.1676475.

[68] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas. An evaluation of lazy fault detection based on adaptive redundant multithreading. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014. doi: 10.1109/HPEC.2014.7040999.

[69] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

[70] *MIC COI API Reference Manual.* Intel Corporation, 0.65 edition, .

[71] *Intel Manycore Platform Software Stack (Intel MPSS): User's Guide.* Intel Corporation, 2013.

[72] *Symmetric Communications Interface (SCIF) for Intel Xeon Phi Product Family Users Guide.* Intel Corporation, 2013.

[73] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 17:1–17:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389020`.

[74] S. jai Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *In: Fifth Conference on Partitioned Global Address Space Programming Models. Galveston Island*, 2011.

[75] G. J. Janakiraman, J. Renato, S. D. Subhraveti, and Y. Turner. Cruz: Application-transparent distributed checkpoint-restart on standard operating systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 260–269, June 2005. doi: 10.1109/DSN.2005.33.

[76] P. G. Jansen, S. J. Mullender, P. J. Havinga, and H. Scholten. Lightweight edf scheduling with deadline inheritance. Technical report, University of Twente. May, 2003.

[77] J. Jeffers and J. Reindeer. *Intel Xeon Phi Coprocessor High-Performance Programming.* Morgan Kaufmann Publishers, 2013.

[78] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Houston, TX, USA, 1990. AAI9110983.

[79] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. Technical report, Department of Computer Science, Rice University, Houston, Texas, 1987.

[80] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[81] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. pages 29–40, 2013.

[82] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, LLNL, December 2012.

[83] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. marc Loingtier, J. Irwin, G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.

[84] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. SE-13 (1):23–31, Jan. 1987.

[85] S.-B. Lee, I. Pefkianakis, A. Meyerson, S. Xu, and S. Lu. Proportional fair frequency-domain packet scheduling for 3gpp lte uplink. In *INFOCOM 2009, IEEE*, pages 2611–2615, April 2009. doi: 10.1109/INFCOM.2009.5062197.

[86] E. Leonardi, M. Mellia, M. Ajmone Marsan, and F. Neri. Joint optimal scheduling and routing for maximum network throughput. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 819–830 vol. 2, March 2005. doi: 10.1109/INFCOM.2005.1498313.

[87] S. Li, V. Sridharany, S. Gurumurthiz, and S. Yalamanchili. Software-based dynamic reliability management for gpu applications. In *Workshop on Silicon Errors in Logic - System Effects*, Apr. 2015.

[88] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the COndor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[89] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962. ISSN 0018-8646. doi: 10.1147/rd.62.0200.

[90] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, 37(2):160–174, Feb 1988.

[91] A. Marathe, R. Harris, D. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz. Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2. HPDC '14, pages 279–290. ACM, 2014.

[92] M. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri. Multicast traffic in input-queued switches: optimal scheduling and maximum throughput. *Networking, IEEE/ACM Transactions on*, 11(3):465–477, June 2003. ISSN 1063-6692. doi: 10.1109/TNET.2003.813048.

[93] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 563–570, Sept 2015. doi: 10.1109/CLUSTER.2015.104.

[94] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.18. URL `http://dx.doi.org/10.1109/SC.2010.18`.

[95] mpiP: lightweight, scalable MPI profiling. mpip.sourceforge.net.

[96] N. Naksinehaboon, N. Taerat, C. Leangsuksun, C. F. Chandler, and S. L. Scott. Benefits of Software Rejuvenation on HPC Systems. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, ISPA '10, pages 499–506. IEEE Computer Society, 2010.

[97] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. EuroSys '11, pages 343–356. ACM, 2011.

[98] A. Nukada, H. Takizawa, and S. Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. pages 104–113, 2011.

[99] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. pages 361–376, Dec. 2002.

[100] X. Ouyang, K. Gopalakrishnan, D. K. Panda, and et al. Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture, 2009.

[101] B. Panzer-Steindel. Data Integrity. Technical Report 1.3, CERN, 2007.

[102] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with NAMD. *J. Comput. Chem.*, 26(16):1781–1802, Dec. 2005. ISSN 0192-8651. doi: 10.1002/jcc.20289. URL `http://dx.doi.org/10.1002/jcc.20289`.

[103] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar. Interference-driven resource management for GPU-based heterogeneous clusters. pages 109–120, June 2012.

[104] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *USENIX Technical Conf. Proc.*, 1995.

[105] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[106] A. Rezaei and F. Mueller. Sustained resilience via live process cloning. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1498–1507, May 2013. doi: 10.1109/IPDPSW.2013.224.

[107] M. Rieker and J. Ansel. Transparent user-level checkpointing for the native posix thread library for linux. In *In Proc. of PDPTA-06*, pages 492–498, 2006.

[108] M. Rieker, J. Ansel, and G. Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. pages 492–498, July 2006.

[109] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges. Alleviating Scalability Issues of Checkpointing Protocols. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 18:1–18:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389021`.

[110] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello. SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 8:1–8:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503271. URL `http://doi.acm.org/10.1145/2503210.2503271`.

[111] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems. Mar. 2007.

[112] Y. Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, Mar. 1993. ISSN 1064-8275. doi: 10.1137/0914028. URL `http://dx.doi.org/10.1137/0914028`.

[113] K. Sajjapongse, X. Wang, and M. Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with GPUs. pages 179–190, June 2013.

[114] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3), Mar. 2010.

[115] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[116] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and modeling of a non-blocking checkpointing system. Nov. 2012.

[117] K. Sato, A. Moody, K. Mohror, T. Gamlin, B. R. de Supinski, N. Maruyama, and S. Matsuoka. Design and Modeling of a Non-Blocking Checkpointing System. In *Proceedings of the 2012 IEEE conference on Supercomputing*, SC '12, 2012.

[118] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *J. of Physics: Conf. Series*, 78(1), 2007.

[119] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009.

[120] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 69–78, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304588. URL `http://doi.acm.org/10.1145/2304576.2304588`.

[121] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. Dependable Secur. Comput.*, 6(2):135–148, Apr. 2009. ISSN 1545-5971.

[122] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing*, 2013.

[123] V. Sridharan and D. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 117–128, Feb 2009. doi: 10.1109/HPCA.2009.4798243.

[124] V. Sridharan and D. R. Kaeli. Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, WREFT '08, pages 323–328, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-092-0. doi: 10.1145/1366224.1366225. URL `http://doi.acm.org/10.1145/1366224.1366225`.

[125] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. *ACM SIGARCH Computer Architecture News*, 43(1):297–310, Mar. 2015. ISSN 0163-5964. doi: 10.1145/2786763.2694348. URL `http://doi.acm.org/10.1145/2786763.2694348`.

[126] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proc. of 10th International Parallel Processing Symposium (IPPS âĂŹ96)*, pages 526–531. IEEE CS Press, 1996.

[127] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. pages 408–413, Dec. 2009.

[128] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. pages 864–876, May 2011.

[129] T. Tannenbaum and M. Litzkow. Checkpointing and migration of unix processes in the Condor distributed processing system. *Dr Dobbs Journal*, Feb. 1995.

[130] VMware. Getting Started with VMware Workstation 10. Technical Report EN-001199-00, VMware Inc, 2013.

[131] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[132] D. Wong, G. S. Lloyd, and M. B. Gokhale. A Memory-mapped Approach to Checkpointing. Technical Report LLNL-TR-635611, LLNL, 2013.

[133] K. S. Yim, Z. Kalbarczyk, and R. Iyer. Pluggable Watchdog: Transparent Failure Detection for MPI Programs. IPDPS '13, pages 489–500, May 2013.

[134] L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resilience with the data vulnerability factor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 695–706, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.62. URL `http://dx.doi.org/10.1109/SC.2014.62`.

[135] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-scale multi-dimensional document clustering on gpu clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010. doi: 10.1109/IPDPS.2010.5470429.

[136] G. Zheng, X. Ni, and L. V. KaleÌĄ. A scalable double in-memory checkpoint and restart scheme towards exascale. In *in Proceedings of the 2nd Workshop on FaultTolerance for HPC at Extreme Scale (FTXS)*, 2012.

[137] Z. Zheng, A. Chien, and K. Teranishi. Fault tolerance in an inner-outer solver: A gvr-enabled case study. In Michel, O. Marques, and K. Nakajima, editors, *High Performance Computing for Computational Science – VECPAR 2014*, volume 8969 of *Lecture Notes*

in Computer Science, pages 124–132. Springer International Publishing, 2014. ISBN 978-3-319-17352-8. doi: 10.1007/978-3-319-17353-5_11. URL `http://dx.doi.org/10.1007/978-3-319-17353-5_11`.