ORNL/TM-2016/687

Resilience Design Patterns

A Structured Approach to Resilience at Extreme Scale

ORNL Technical Report - Version 1.0

Saurabh Hukerikar
Christian Engelmann
{hukerikarsr, engelmannc}@ornl.gov

October 2016

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Early Career Research Program under contract number DE-AC05-00OR22725.

Computer Science and Mathematics Division

RESILIENCE DESIGN PATTERNS

A Structured Approach to Resilience at Extreme Scale

Saurabh Hukerikar & Christian Engelmann

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale

Abstract

Reliability is a serious concern for future extreme-scale high-performance computing (HPC) systems. Projections based on the current generation of HPC systems and technology roadmaps suggest that very high fault rates in future systems. The errors resulting from these faults will propagate and generate various kinds of failures, which may result in outcomes ranging from result corruptions to catastrophic application crashes. Practical limits on power consumption in HPC systems will require future systems to embrace innovative architectures, increasing the levels of hardware and software complexities.

The resilience challenge for extreme-scale HPC systems requires management of various hardware and software technologies that are capable of handling a broad set of fault models at accelerated fault rates. These techniques must seek to improve resilience at reasonable overheads to power consumption and performance. While the HPC community has developed various solutions, application-level as well as system-based solutions, the solution space of HPC resilience techniques remains fragmented. There are no formal methods and metrics to investigate and evaluate resilience holistically in HPC systems that consider impact scope, handling coverage, and performance & power efficiency across the system stack. Additionally, few of the current approaches are portable to newer architectures and software ecosystems, which are expected to be deployed on future systems.

In this document, we develop a structured approach to the management of HPC resilience based on the concept of resilience-based design patterns. A design pattern is a general repeatable solution to a commonly occurring problem. We identify the commonly occurring problems and solutions used to deal with faults, errors and failures in HPC systems. The catalog of resilience design patterns provides designers with reusable design elements. We define a design framework that enhances our understanding of the important constraints and opportunities for solutions deployed at various layers of the system stack. The framework may be used to establish mechanisms and interfaces to coordinate flexible fault management across hardware and software components. The framework also enables optimization of the cost-benefit trade-offs among performance, resilience, and power consumption. The overall goal of this work is to enable a systematic methodology for the design and evaluation of resilience technologies in extreme-scale HPC systems that keep scientific applications running to a correct solution in a timely and cost-efficient manner in spite of frequent faults, errors, and failures of various types.

Contents

Listing of figures

Listing of tables

1 Introduction

High-performance computing systems enable transformative scientific research and discovery in various areas of national importance through computational modeling, simulation, data analysis and prediction. The opportunities to address complex challenges that are important for national security, environmental issues, as well as to drive fundamental scientific research are the key motivators behind the HPC community's drive towards extreme-scale high-performance computing systems. Future extreme-scale systems will enable computing at scales in the hundreds of petaflops, exaflops, and beyond, which will provide the computing capability for rapid design and prototyping and big data analysis in a variety of scientific and engineering disciplines. However, to build and effectively operate extreme-scale HPC systems, there are several key challenges, including the management of power, massive concurrency and resilience to the occurrence of faults or defects in system components [25].

In the pursuit of greater computational capabilities, the architectures of future extreme-scale HPC systems are expected to change radically. Traditional HPC system design methodologies have not had to account for power constraints, or parallelism on the level designers must contemplate for future extreme-scale systems [74]. The evolution in the architectures will require changes to the programming models and the software environment to ensure application scalability. Many of the innovations in the architectures are expected to be driven by the continued scaling of transistors made possible by Moore's Law. However, the reliability of these systems will be threatened by a decrease in individual device reliability due to manufacturing defects prevalent at deeply scaled technology nodes, device aging related effects [9]. Additionally, the chips built using these devices will be increasingly susceptible to errors due to the effects of operational and environmental conditions on the reduced noise margins arising from the near-threshold voltage (NTV) operation [26] to meet the limits on power consumption. These effects are expected to increase the rate of transient and hard errors, such that scientific applications running on these future systems will no longer be able to assume correct behavior of the underlying machine.

Managing the resilience of future extreme-scale systems is a multidimensional challenge. As these systems approach exaflops scale, the frequency of faults and errors in these systems will render many of the existing resilience techniques ineffective. Newer modes of failures due to faults and errors that will only emerge in advanced process technologies and complex system architectures will require novel resilience solutions, as well as existing techniques to adapt their respective implementations. Additionally, HPC resilience methodologies, both hardware and software, must optimize for some combination of performance, power consumption and cost while providing effective protection against faults, errors and failures. The HPC research community has developed a number of hardware and software resilience technologies, but there are no formalized, comprehensive methods to investigate and evaluate the protection coverage and efficiency of resilience solutions. The development of HPC resilience solutions no longer relies on the invention of novel methodologies for dealing with extreme rates and a variety of fault types that may occur; rather, it based on the selection of the most appropriate solutions among the well-understood resilience techniques and adapting them to the design concerns and constraints of future extreme-scale systems. Therefore, the designers of HPC hardware and software components have a compelling need for a systematic methodology for designing resilience solutions for HPC systems and their applications.

In this work, we develop a structured approach for resilient extreme-scale high-performance computing systems. In general, resilience solutions provide techniques to manage faults and their consequences in a

system through the detection of errors in the system, and providing the means that ensure the limiting their propagation, and the recovery of the system from the error/failure, or the masking of error/failure. Using the concept of design patterns for resilience, we identify and evaluate repeatedly occurring resilience problems and solutions throughout the hardware/software stack and across various system components. These patterns, which are based on best practices for HPC resilience, are abstracted and codified into a catalog of design patterns. The patterns in this document present solutions for each of the three aspects of resilience, which are the detection, containment, and recovery from an error or a failure. The resilience design patterns describe solutions that are free of implementation details, and these have the potential to shape the design of HPC applications' algorithms, numerical libraries, system software, and hardware architectures, as well as the interfaces between layers of system abstraction.

In addition to the catalog of resilience design patterns, which provides HPC designers with a collection of reusable design elements, we define a discipline that enables designers to combine an essential set of design patterns into productive, efficient resilience solutions. We define a layered hierarchy of these resilience design patterns, which facilitates the design process of combining the various patterns in the catalog. The classification scheme provides guidelines for designers to distinctly solve problems of detection, containment and mitigation/recovery, to stitch these patterns together, and refine the design of the overall system based on the roles of the individual patterns, and how they interact. We define a framework based on *design spaces* that guides hardware and software designers and architects, as well as application developers, in navigating the complexities of developing effective resilience solutions within the constraints of hardware and software implementation challenges, performance and power considerations. The construction of resilience solutions using the design patterns and using the design spaces framework makes the design choices explicit and the critical issues addressed by a solution clear. The structured, pattern-based approach enables:

- Development of resilience solutions with a clear understanding of their protection coverage and efficiency
- Evaluation and comparison of alternative resilience solutions through qualitative and quantitative evaluation of the coverage and handling efficiency of each solution
- Design portable resilience solutions that are effective on different systems, since each HPC system architecture has different resilience features and solutions are not universally applicable

The rest of this document is organizes as follows:

- Section 2 provides a summary of the terminology used in fault tolerance and the basic concepts of resilience to enable HPC designers, as well as system operators and users to understand the essence of the resilience patterns and use them in their designs, whether hardware or software.
- Section 3 describes the challenges in managing the resilience of future extreme-scale HPC systems.
- Section 4 surveys the various HPC resilience solutions, including those used in production HPC systems, as well as research proposals. The aim of this section is to provide a comprehensive overview of the various HPC resilience techniques.
- Section 5 introduces the design pattern concept and discusses the potential for capturing the HPC resilience techniques in the form of patterns.

- Section 6 describes a classification scheme to organize the various resilience techniques in a layered hierarchy to enable designers to understand the capabilities of each resilience solution.
- Section 7 presents the catalog of resilience design patterns that capture well-understood HPC resilience techniques for error detection, recovery and masking in a structured format.
- Section 8 presents a structured methodology to use the resilience design patterns for the construction of effective and efficient resilience solutions. The design framework forms the basis for HPC designers and programmers to use patterns from the catalog to develop complete resilience solutions with specific properties.
- Sections 9, 10, 11 present case studies that demonstrate how these patterns may be used to understand and evaluate existing resilience solutions, as well as develop new solutions using the pattern-based design framework.

2 Resilience Terminology and Concepts

The terminology is largely based on prior work on establishing agreed upon definitions and metrics for HPC Reliability, Availability & Servicability (RAS) [30, 32, 49, 65, 79, 83].

2.1 Reliability

Reliability is the property of a system that characterizes its probability to have an error or failure, *i.e.*, it provides information about the error- or failure-free time period.

2.2 Availability

Availability is the property of a system that defines the proportion of time it provides a correct service, instead of incorrect service.

2.3 Systems

- **System:** An entity that interacts with other entities.
- **Component:** A system that is part of a larger system.
- **State:** A system's information, computation, communication, interconnection, and physical condition.
- **Behavior:** What a system does to implement its function, described by a series of states.
- **Service:** A system's externally perceived behavior.
- **Functional specification:** The description of system functionality, defining the threshold between:
  - **Correct service:** The provided service is acceptable, i.e., within the functional specification.
  - **Incorrect service:** The provided service is unacceptable, i.e., outside the functional specification.
- **Life cycle:** A system has life cycle phases in the following order:
  1. **Development:** A system is in development, including designed, constructed, deployed and tested.
  2. **Operational:** A system is in operation, providing correct or incorrect service.
  3. **Retired:** A system is not in operation anymore.
- **Operational status:** A system has the following operational states:
  1. **Scheduled service outage:** A system is delivering an incorrect service due to a planned outage.
  2. **Unscheduled service outage:** A system is delivering an incorrect service due to an unplanned outage.
  3. **Service delivery:** A system is delivering a correct service.

The terms **fault**, **error** and **failure** are sometimes used interchangeably. However in fault tolerance literature, these terms are associated with distinct formal concepts which are defined as follows [6]:

2.4 Faults

Fault is an underlying flaw or defect in a system that has potential to cause problems. A fault can be dormant and can have no effect, e.g., incorrect program code that lies outside the execution path. When activated during system operation, a fault leads to an error. Fault activation may be due to triggers that are internal or external to the system.

- **Fault classes:** *{benign,dormant,active} {permanent,transient,intermittent} {hard,soft}*

These fault classes have the following categories:

- **Benign:** An inactive fault that does not become active.
- **Dormant:** An inactive fault that does become active at some point in time.
- **Active:** A fault that causes an error at the moment of becoming active.
- **Permanent:** A fault's presence is continuous in time.
- **Transient:** A fault's presence is temporary.
- **Intermittent:** A fault's presence is temporary and recurring.
- **Hard:** A fault that is systematically reproducible.
- **Soft:** A fault that is not systematically reproducible.

The following common terms map to these fault classes:

- **Latent fault:** Any type of *dormant fault*.
- **Solid fault:** Any type of *hard fault*.
- **Elusive fault:** Any type of *soft fault*.

For example, a radiation-induced bit-flip in memory is a *dormant transient soft fault* that becomes an *active transient soft fault* when the memory is read. The fault disappears when the memory is written. A radiation-induced bit-flip in memory is a *dormant permanent soft fault* if the memory is never written. It becomes an *active permanent soft fault* when the memory is read.

2.5 Errors

Errors result from the activation of a fault and cause an illegal system state. For e.g., a faulty assignment to a loop counter variable may result in an error characterized by an illegal value for that variable. When such a variable is used for control of a for-loop's execution, it may lead to incorrect program behavior.

The following error classes exist:

- **Error classes:** *{undetected,detected} {unmasked,masked} {hard,soft}*

These error classes have the following categories:

- **Undetected:** An error that is not indicated.
- **Detected:** An error that is indicated, such as by a message or a signal.
- **Unmasked:** An error that is propagating.
- **Masked:** An error that is not propagating.
- **Hard:** An error caused by a permanent fault.
- **Soft:** An error caused by a transient or intermittent fault.

The following common terms map to these error classes:

- **UE:** Any type of *undetected error*.
- **Latent error:** Any type of *undetected error*.
- **Silent error:** Any type of *undetected error*.
- **SDC:** An *undetected unmasked hard* or *soft error*.
- **DE:** Any type of *detected error*.

For example, an *active transient soft fault*, created by a radiation-induced bit-flip in memory being read, causes an *undetected masked soft error*, when the read value is used in a multiplication with another value that happens to be 0. It causes an *undetected unmasked soft error*, or SDC, when the read value is used as an index in a memory address calculation.

A detectable correctable error is often transparently handled by hardware, such as a single bit flip in memory that is protected with single-error correction double-error detection (SECDED) error correcting code (ECC) [60]. A detectable uncorrectable error (DUE) typically results in a failure, such as multiple bit flips in the same addressable word that escape SECDED ECC correction, but not detection, and ultimately cause an application abort. An undetectable error may result in silent data corruption (SDC), e.g., an incorrect application output.

2.6 Failures

Failure occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with the system's specification. For e.g., a faulty assignment to a pointer variable leads to erroneous accesses to a data structure or buffer overflow, which in turn may cause the program to crash due to an attempt to access an out-of-bound memory location.

The following failure classes exist:

- **Failure classes:** *{undetected,detected} {permanent,transient,intermittent} {complete,partial,Byzantine}*

These failure classes have the following categories:

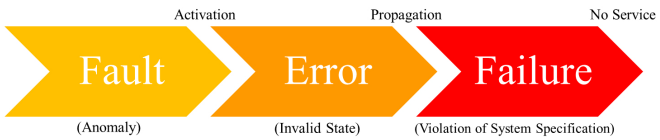

Figure 1: Relationship between fault, error and failure

- **Undetected:** A failure that is not indicated.
- **Detected:** A failure that is indicated, such as by a message or a signal.
- **Permanent:** A failure's presence is continuous in time.
- **Transient:** A failure's presence is temporary.
- **Intermittent:** A failure's presence is temporary and recurring.
- **Complete:** A failure causing a service outage.
- **Partial:** A failure causing a degraded service within the functional specification.
- **Byzantine:** A failure causing an arbitrary deviation from the functional specification.

The following common terms map to these error classes:

- **Fail-stop:** An *undetected* or *detected permanent complete failure*.

For example, an *active transient soft fault*, created by a radiation-induced bit-flip in memory being read, causes an *undetected unmasked soft error*, when the read value is used as an index in a memory address calculation. A memory access violation caused by using a corrupted calculated address results in a *detected permanent complete failure*, as the executing process is killed by the operating system (OS), and a message is provided to the user. However, if using the corrupted calculated address results in an incorrect service that is not indicated, such as erroneous output, an *undetected intermittent Byzantine failure* occured.

2.7 The Relationship between Faults, Errors and Failures

While a fault (e.g., a bug or stuck bit) is the cause of an error, its manifestation as a state change is considered an error (e.g., a bad value or incorrect execution), and the transition to an incorrect service is observed as a failure (e.g., an application abort or system crash) [79]. A failure in a HPC system is typically observed through an application abort or a full/partial system outage. There is a causality relationship between fault, error and failure, as shown in Figure 1. A **Fault-error-failure chain** is a DAG representation with faults, errors and failures as vertices. When the system is composed of multiple components, the failure of a single component causes a permanent or transient external fault for the other components that receive service from the failed component. Therefore, errors may be transformed into other errors and propagate through the system, generating further errors. A **failure cascade** is a failure of component B that cascades to component C if it causes a failure of C. For example, a faulty procedure argument leads to erroneous computation and may manifest as a failure in the form of an illegal procedure return value. To the caller of the function, this may activate a chain of errors that propagate until service failure occurs, i.e., a program crash.

2.8 Resilience Capabilities

There are three key components to designing a resilience strategy:

2.8.1 Detection

Detection entails the discovery of an error in the state of the system, either in the data, or in the instructions. It is typically accomplished with the help of redundancy; the extra information enables the verification of correct values.

Errors are detected by identifying the corresponding state change. Failures are detected by identifying the corresponding transition to an incorrect service. An error or a failure is indicated by a detector. This detector can fail as well.

The following detection classes exist:

- **Detection classes:** *{true,false} {positive,negative}*

These detection classes have the following categories:

- **True:** A correct detection.
- **False:** An incorrect detection.
- **Positive:** An indication, such as a message or a signal.
- **Negative:** No indication.

2.8.2 Containment

A containment capability enables limiting the effects of an error from propagating. Containment is achieved by reasoning about the modularity of components or sub-systems that make up the system. In terms of resilience of the system, a containment module is an unit that fails independently of other units and it is also the unit of repair or replacement.

2.8.3 Masking

Masking enatils recovery or mitigation, which ensures correct operation despite the occurrence of an error. Masking is usually accomplished by providing additional redundant in order to discover correct, or at least acceptably close, values of the erroneous state. When the masking involves the change of incorrect state into correct state, it is called error correction.

A detected error may be masked by an error correction method, such as using ECC. The following error classes exist that are equivalent to already defined error classes:

- **Error classes:** *{uncorrected,corrected}*

These error classes have the following categories:

- **Uncorrected:** An error that is not corrected, i.e., an *undetected* or *detected unmasked error*.
- **Corrected:** An error that is detected and corrected, i.e., a *detected masked error*.

The following common terms map to these error classes:

- **DUE:** An *uncorrected error*, i.e., a *detected unmasked error*.
- **DCE:** A *corrected error*, i.e., a *detected masked error*.

In practice, a resilience mechanism may merge the implementation of two or even all three of the capabilities to provide a complete solution.

2.9 Resilience Metrics

2.9.1 Reliability Metrics

The following reliability metrics exist:

- **Error** or **failure reliability:** A system's probability not to have an error or failure during $0 \leq t$, $R(t)$.
- **Error** or **failure distribution:** A system's probability to have an error or failure during $0 \leq t$, $F(t)$.
- **PDF:** The relative likelihood of an error or failure, $f(t)$.
- **Error** or **failure rate:** A system's error or failure frequency, $\lambda(t)$.
- **MTTE:** A system's expected time to error, $MTTE$.
- **MTTF:** A system's expected time to failure, $MTTF$.
- **FIT rate:** A system's number of expected failures in 10^9 hours of operation, FIT.
- **Serial reliability:** The reliability of a system with n dependent components, $R(n,t)_s$.
- **Parallel reliability:** The reliability of a system with n redundant components, $R(n,t)_p$.
- **Identical serial reliability:** The serial reliability with n identical components, $R(n,t)_{is}$.
- **Identical parallel reliability:** The parallel reliability with n identical components, $R(n,t)_{ip}$.

$$R(t) = 1 - F(t) = \int_t^\infty f(t)dt \tag{1}$$

$$F(t) = 1 - R(t) = \int_0^t f(t)dt \tag{2}$$

$$\lambda(t) = \frac{f(t)}{R(t)} \tag{3}$$

$$MTTE \ or \ MTTF = \int_0^\infty R(t)dt \tag{4}$$

$$FIT = \frac{10^9}{MTTF} \tag{5}$$

$$R(n,t)_s = \prod_{i=1}^{n} R_i(t) \tag{6}$$

$$R(n,t)_p = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{7}$$

$$R(n,t)_{is} = R(t)^n \tag{8}$$

$$R(n,t)_{ip} = 1 - (1 - R(t))^n \tag{9}$$

2.9.2 Availability Metrics

Availability is the property of a system that defines the proportion of time it provides a correct service, instead of incorrect service. The following availability metrics exist:

- **Availability:** A system's proportion of time it provides a correct service, instead of incorrect service, A.
- **PU:** A system's *service delivery* time, t_{pu}.
- **UD:** A system's *unscheduled service outage* time, t_{ud}.
- **SD:** A system's *scheduled service outage* time, t_{sd}.
- **MTTR:** A system's expected time to repair/replace, $MTTR$.
- **MTBF:** A system's expected time between failures, $MTBF$.
- **Serial availability:** The availability of a system with n dependent components, A_s.
- **Parallel availability:** The availability of a system with n redundant components, A_p.
- **Identical serial availability:** The serial availability with n identical components, A_{is}.
- **Identical parallel availability:** The parallel availability with n identical components, A_{ip}.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{10}$$

$$= \frac{MTTF}{MTTF + MTTR} \tag{11}$$

$$= \frac{MTTF}{MTBF} \tag{12}$$

$$MTBF = MTTF + MTTR \tag{13}$$

$$A_s = \prod_{i=1}^{n} A_i \tag{14}$$

$$A_p = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{15}$$

$$A_{is} = A^n \tag{16}$$

$$A_{ip} = 1 - (1 - A)^n \tag{17}$$

A system can also be rated by the number of 9s in its availability figure (Table 1). For example, a system with a five-nines availability rating has 99.999% availability and an annual UD of 5 minutes and 15.4 seconds.

Table 1: Availability measured by the "nines"

9s	Availability	Annual Downtime
1	90%	36 days, 12 hours
2	99%	87 hours, 36 minutes
3	99.9%	8 hours, 45.6 minutes
4	99.99%	52 minutes, 33.6 seconds
5	99.999%	5 minutes, 15.4 seconds
6	99.9999%	31.5 seconds

2.9.3 Error and Failure Detection Metrics

- **Precision:** The fraction of indicated errors or failures that were actual errors or failures.
- **Recall:** The fraction of errors or failures that were detected and indicated.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} = \frac{True\ Positives}{Indicated\ Errors\ or\ Failures} \tag{18}$$

$$= 1 - \frac{False\ Positives}{True\ Positives + False\ Positives} = 1 - \frac{False\ Positives}{Indicated\ Errors\ or\ Failures} \tag{19}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} = \frac{True\ Positives}{Errors\ or\ Failures} \tag{20}$$

$$= 1 - \frac{False\ Negatives}{True\ Positives + False\ Negatives} = 1 - \frac{False\ Negatives}{Errors\ or\ Failures} \tag{21}$$

For example, a *true positive* detection corresponds to an existing error or failure being indicated, while a *false positive detection* corresponds a non-existing error or failure being indicated. A *true negative* detection corresponds to a non-existing error or failure not being indicated, while a *false negative* detection corresponds to an existing error or failure not being indicated.

2.9.4 Mean Time to Failure

Resilience is measured by vendors and operators from the system perspective, e.g., by system mean-time to failure (SMTTF) and system mean-time to repair (SMTTR). Users measure resilience from the application perspective, e.g., by application mean-time to failure (AMTTF) and application mean-time to repair (AMTTR) [79]. Both perspectives are quite different [17]. For example, an application abort caused by a main memory DUE does not require the system to recover, i.e., the SMTTR is 0. However, the aborted application needs to recover its lost state after it has been restarted, i.e., the AMTTR may be hours. Conversely, a failure of a parallel file system server may only impact a subset of the running applications, as the other ones access a different server. In this case, the server failure is counted toward the SMTTF, while the AMTTF differs by application.

3 The Resilience Challenge for Extreme-Scale HPC Systems

Various studies that analyze faults, errors and failures in HPC systems indicate that faults are not rare events in large-scale systems and that the distribution of failure root cause is dominated by faults that originate in hardware. These may include faults due to radiation-induced effects such as particle strikes from cosmic radiation, circuit aging related effects, and faults due to chip manufacturing defects and design bugs that remain undetected during post-silicon validation and manifest themselves during system operation. With aggressive scaling of CMOS devices, the amount of charge required to upset a gate or memory cell is decreasing with every process shrink. For very fine transistor feature sizes, the lithography used in patterning transistors causes variations in transistor geometries such as line-edge roughness, body thickness variations and random dopant fluctuations. These lead to variations in the electrical behavior of individual transistor devices, and this manifests itself at the circuit-level in the form of variations in circuit delay, power, and robustness [4]. The challenge of maintaining resilience continues to evolve as process technology continues to shrink and system designers will use components that operate at lower threshold voltages. The shrinking noise margins makes the components inherently less reliable and leads to a greater number of manufacturing defects, as well as device aging-related effects. The use of system-level performance and power modulation techniques, such as dynamic voltage/frequency scaling, also tend to induce higher fault rates. It is expected that future exascale-capability systems will use components that have transistor feature sizes between 5 nm and 7 nm, and that these effects will become more prevalent, thereby causing the system components to be increasingly unreliable [16]. The modeling and mitigation of these effects through improved manufacturing processes and circuit-level techniques might prove too difficult or too expensive.

Today's petascale-class HPC systems already employ millions of processor cores and memory chips to drive HPC application performance. The recent trends in system architectures suggest that future exascale-class HPC systems will be built from hundreds of millions of components organized in complex hierarchies. However, with the growing number of components, the overall reliability of the system decreases proportionally. If p is the probability of failure of an individual component and the system consists of N components, the probability that the complete system works is $(1 - p)^N$ when the component failures are independent. It may therefore be expected that some part of an exascale class supercomputing system will always be experiencing failures or operating in a degraded state. The drop in MTTF of the system is expected to be dramatic based on the projected system features [48]. In future exascale-class systems, the unreliability of chips due to transistor scaling issues will be amplified by the large number of components. For long running scientific simulations and analysis applications that will run on these systems, the accelerated rates of system failures will mean that their executions will often terminate abnormally, or in many cases, complete with incorrect results. Finding solutions to these challenges will therefore require a concerted and collaborative effort on the part of all the layers of the system stack.

Resilience is an approach to fault tolerance for high-end computing (HEC) systems that seeks to keep the application workloads running to correct solutions in a timely and efficient manner in spite of frequent errors [21]. The emphasis is on the application's outcome and the reliability of application level information in place of or even at the expense of reliability of the system. Resilience technologies in HPC embrace the fact that the underlying fabric of hardware and system software will be unreliable and seek to enable effective and resource efficient use of the platform in the presence of system degradations and failures [16]. A complete resilience solution consists of detection, containment and mitigation strategies.

Performance, resilience, and power consumption are interdependent key system design factors. An increase in resilience (e.g., though redundancy) can result in higher performance (as less work is wasted) and in higher power consumption (as more hardware is being used). Similarly, a decrease in power consumption (e.g., through NTV operation) can result in lower resilience (due to higher soft error vulnerability) and lower performance (due to lower clock frequencies and more wasted work). The performance, resilience, and power consumption cost/benefit trade-off between different resilience solutions depends on individual system and application properties. Understanding this trade-off at system design time is a complex problem due to uncertainties in future system hardware and software reliability. It is also difficult due to a needed comprehensive methodology for design space exploration that accounts for performance, resilience, and power consumption aspects across the stack and the system. Similarly, runtime adaptation to changing resilience demands, while staying within a fixed power budget and achieving maximum performance, is currently limited to checkpoint placement strategies. While resilience technologies seek to provide efficient and correct operation despite the frequent occurrence of faults and defects in components that lead to errors and failures in HPC systems, there is no methodology for optimizing the trade-off, at design time or runtime, between the key system design factors: performance, resilience, and power consumption.

4 Survey of HPC Resilience Techniques

This section surveys various fault-tolerance techniques used in practical computing systems, as well as research proposals.

4.1 Hardware-based Techniques

Dual-modular redundancy (DMR) and triple-modular redundancy (TMR) approaches have been successfully used in mission-critical systems using hardware-based replication. Examples of fault-tolerant servers include the Tandem Non-Stop [56] and the HP NonStop [7] that use two redundant processors running in locked step. The IBM G5 [76] employs two fully duplicated lock-step pipelines to enable low-latency detection and rapid recovery. While these solutions are transparent to the supervisor software and application programmer, they require specialized hardware.

Error correction codes (ECC) use a flavor of redundancy in memory structures that typically add additional bits to enable detection and correction of errors. Single bit-error correction and double bit-error detection (SECDED) is the most widely used variant of ECC, while researchers have also explored Bose-Chaudhuri-Hocquenghem (BCH) and double-bit error correction and triple-bit error detection (DECTED) [62] for multi-bit detection and correction. Chipkill [23] is a stronger memory protection scheme that is widely used in production HPC systems. It accommodates single DRAM memory chip failure as well as multi-bit errors from any portion of a single memory chip by interleaving bit error-correcting codes across multiple memory chips.

HPC vendors have also developed a number of hardware resilience technologies, including: SECDED ECC for main memory, caches, registers and architectural state, as well as, Chipkill [22] for main memory, redundant power supplies and voltage regulators [43], and reliability, availability and serviceability management systems for monitoring and control.

4.2 Software-implemented Techniques

Software-based redundancy promises to offer more flexibility and tends to be less expensive in terms of silicon area as well as chip development and verification costs; it also eliminates the need for modifications of architectural specifications.

4.2.1 Operating System & Runtime-based Solutions

The most widely used strategies in production HPC systems are predominantly based on checkpoint and restart (C/R). In general, C/R approaches are based on the concept of capturing the state of the application at key points of the execution, which is then saved to persistent storage. Upon detection of a failure, the application state is restored from the latest disk committed checkpoint, and execution resumes from that point. The Condor standalone checkpoint library [54] was developed to provide checkpointing for UNIX processes, while the Berkeley Labs C/R library [28] was developed as an extension to the Linux OS. The libckpt [66] provided similar OS-level process checkpointing, albeit based on programmer annotations.

In the context of parallel distributed computing systems, checkpointing requires global coordination, i.e., all processes on all nodes are paused until all messages in-flight and those in-queue are delivered, at which point all the processes' address spaces, register states, etc., are written to stable storage, generally a parallel file system, through dedicated I/O nodes. The significant challenge in these efforts is the coordination among processes so that later recovery restores the system to a consistent state. These approaches typically launch daemons on every node that form and maintain communication groups that allow tracking and managing recovery by maintaining the configuration of the communication system. The failure of any given node in the group is handled by restarting the failed process on a different node, by restructuring the computation, or through transparent migration to another node [2] [13] [51].

Much work has also been done to optimize the process of C/R. A two-level recovery scheme proposed optimization of the recovery process for more probable failures, so that these incur a lower performance overhead while the less probable failures incur a higher overhead [81]. The scalable checkpoint/restart (SCR) library [59] proposes multilevel checkpointing where checkpoints are written to storage that use RAM, flash, or local disk drive, in addition to the parallel file system, to achieve much higher I/O bandwidth. Oliner *et al.* propose an opportunistic checkpointing scheme that writes checkpoints that are predicted to be useful - for example, when a failure in the near future is likely [64]. Incremental checkpointing dynamically identifies the changed blocks of memory since the last checkpoint through a hash function [1] in order to limit the amount of state required to be captured per checkpoint. Data aggregation and compression also help reduce the bandwidth requirements when committing the checkpoint to disk [44]. Plank *et al.* eliminate the overhead of writing checkpoints to disk altogether with a diskless in-memory checkpointing approach [68].

Process-level redundancy (PLR) [75] creates a set of redundant application processes whose output values are compared. The scheduling of the redundant processes is left to the operating system (OS). The RedThreads API [41] provides directives that support error detection and correction semantics through the adaptive use of redundant multithreading.

4.2.2 Message Passing Library-based Solutions

In general, automatic application-oblivious checkpointing approaches suffer from scaling issues due to the considerable I/O bandwidth for writing to persistent storage. Also, practical implementations tend to be fragile [29]. Therefore, several MPI libraries have been enabled with capabilities for C/R [52]. The CoCheck MPI [80], based on the Condor library, uses synchronous checkpointing in which all MPI processes commit their message queues to disk to prevent messages in flight from getting lost. The FT-MPI [33], Open MPI [42], MPICH-V [10] and LAM/MPI [73] implementations followed suit by incorporating similar capabilities for C/R. In these implementations, the application developers do not need to concern themselves with failure handling; the failure detection and application recovery are handled transparently by the MPI library, in collaboration with the OS.

The process-level redundancy approach has also been evaluated in the context of a MPI library implementation [34], where each MPI rank in the application is replicated and the replica takes the place of a failed rank, allowing the application to continue. The RedMPI library [31] [35] replicates MPI tasks and compares the received messages between the replicas in order to detect corruptions in the communication data. Studies have also proposed the use of proactive fault tolerance in MPI [61] [85]. However, with the growing complexity of long running scientific applications, complete multi-modular redundancy, whether through hardware

or software-based approaches, will incur exorbitant overhead to costs, performance and energy, and is not a scalable solution to be widely used in future exascale-class HPC systems.

4.2.3 Compiler-based Solutions

SWIFT [70] is a compiler-based transformation which duplicates all program instructions and inserts comparison instructions during code generation so that the duplicated instructions fill the scheduling slack. The DAFT [90] approach uses a compiler transformation that duplicates the entire program in a redundant thread that trails the main thread and inserts instructions for error checking. The SRMT [86] uses compiler analysis to generate redundant threads mapped to different cores in a chip multi-processor and optimizes performance by minimizing data communication between the main thread and trailing redundant thread. Similarly, EDDI [63] duplicates all instructions and inserts "compare" instructions to validate the program correctness at appropriate locations in the program code. The ROSE::FTTransform [53] applies source-to-source translation to duplicate individual source-level statements to detect transient processor faults.

4.2.4 Programming Model Techniques

Most programming model approaches advocate a collaborative management of the reliability requirements of applications through a programmer interface in conjunction with compiler transformations, a runtime framework and/or library support. Each approach requires different levels of programmer involvement, which has an impact on amount of effort to re-factor the application code, as well as on the portability of the application code to different platforms.

HPC programs usually deploy a large number of nodes to implement a single computation and use MPI with a flat model of message exchange in which any node can communicate with another. Every node that participates in a computation acquires dependencies on the states of the other nodes. Therefore, the failure of a single node results in the failure of the entire computation since the message passing model lacks well-defined failure containment capabilities [29]. User-level failure mitigation (ULFM) [8] extends MPI by encouraging programmer involvement in the failure detection and recovery by providing a fault-tolerance API for MPI programs. The error handling of the communicator has changed from MPI_ERRORS_ARE_FATAL to MPI_ERRORS_RETURN so that error recovery may be handled by the user. The proposed API includes MPI_COMM_REVOKE, MPI_COMM_SHRINK to enable reconstruction of the MPI communicator after process failure and the MPI_COMM_AGREE as a consistency check to detect failures when the programmer deems such a sanity check necessary in the application code.

The abstraction of the *transaction* has also been proposed to capture a programmer's fault-tolerance knowledge. This entails division of the application code into blocks of code whose results are checked for correctness before proceeding. If the code block execution's correctness criteria are not met, the results are discarded and the block can be re-executed. Such an approach was explored for HPC applications through a programming construct called *Containment Domains* by Sullivan *et al.* [15] which is based on weak transactional semantics. It enforces the check for correctness of the data value generated within the containment domain before it is communicated to other domains. These containment domains can be hierarchical and provide the means to locally recover from an error within that domain. A compiler technique that, through static analysis, discovers regions that can be freely re-executed without checkpointed state or side-effects, called idempotent regions, was proposed by de Kruijf *et al.* [20]. Their original proposal [19], however, was based

on language-level support for C/C++ that allowed the application developer to define idempotent regions through specification of *relax* blocks and *recover* blocks that perform recovery when a fault occurs. The FaultTM scheme adapts the concept of hardware-based transactional memory where atomicity of computation is guaranteed. The approach requires an application programmer to define *vulnerable* sections of code. For such sections, a backup thread is created. The original and the backup thread are executed as an atomic transaction, and their respective committed result values are compared [89].

Complementary to approaches that focus on resiliency of computational blocks, the Global View Resilience (GVR) project [36] concentrates on application data and guarantees resilience through multiple snapshot versions of the data whose creation is controlled by the programmer through application annotations. Bridges *et al.* [11] proposed a `malloc_failable` that uses a callback mechanism to handle memory failures on dynamically allocated memory, so that the application programmer can specify recovery actions. The Global Arrays implementation of the Partitioned Global Address Space (PGAS) model presents a global view of multidimensional arrays that are physically distributed among the memories of processes. Through a set of library API for checkpoint and restart with bindings for C/C++/FORTRAN, the application programmer can create checkpoints of array structures. The library guarantees that updates to the global shared data are fully completed and any partial updates are prevented or undone [24]. Rolex [40] provides various resilience semantics for error tolerance and amelioration through language-based extensions that enable these capabilities to be embedded within standard C/C++ programs.

4.2.5 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance (ABFT) schemes encode the application data to detect and correct errors, e.g., the use of checksums on dense matrix structures. The algorithms are modified to operate on the encoded data structures. ABFT was shown to be an effective method for application-layer detection and correction by Huang and Abraham [39] for a range of basic matrix operations including addition, multiplication, scalar product, transposition. Such techniques were also proven effective for LU factorization [18], Cholesky factorization [38] and QR factorization [46]. Several papers propose improvements for better scalability in the context of parallel systems, that provide better error detection and correction coverage with lower application overheads [71] [67] [72]. The checksum-based detection and correction methods tend to incur very high overheads to performance in sparse matrix-based applications. Sloan *et al.* [77] have proposed techniques for fault detection that employ approximate random checking and approximate clustered checking by leveraging the diagonal, banded diagonal, and block diagonal structures of sparse problems. Algorithm-based recovery for sparse matrix problems has been demonstrated through error localization and re-computation [78] [14].

Various studies have evaluated the fault resilience of solvers of linear algebra problems [12]. Iterative methods including Jacobi, Gauss-Seidel and its variants, the conjugate gradient, the preconditioned conjugate gradient, and the multi-grid begin with an initial guess of the solution and iteratively approach a solution by reducing the error in the current guess of the answer until a convergence criterion is satisfied. Such algorithms have proved to be tolerant to errors, on a limited basis, since the calculations typically require a larger number of iterations to converge, based on magnitude of the perturbation, but eventual convergence to a correct solution is possible. Algorithm-based error detection in the multigrid method shown by Mishra *et al.* [57], uses invariants that enable checking for errors in the relaxation, restriction and the interpolation operators.

For fast Fourier transform (FFT) algorithms, an error-detection technique called the sum-of-squares (SOS) was presented by Reddy *et al.* [69]. This method is effective for a broader class of problems called orthogonal transforms and therefore applicable to QR factorization, singular-value decomposition, and least-squares minimization. Error detection in the result of the FFT is possible using weighted checksums on the input and output [87].

While the previously discussed methods are primarily for numerical algorithms, fault tolerance for other scientific application areas has also been explored. In molecular dynamics (MD) simulations, the property that pairwise interactions are anti-symmetric (F_{ij} = - F_{ji}) may be leveraged to detect errors in the force calculations [88]. The resilience of the Hartree-Fock algorithm, which is widely used in computational chemistry, can be significantly enhanced through checksum-based detection and correction for the geometry and basis set objects. For the two-electron integrals and Fock matrix elements, knowing their respective value bounds allows for identifying outliers and correcting them with reasonable values from a range of known correct values. The iterative nature of the Hartree-Fock algorithm helps to eliminate the errors introduced by the interpolated values [82]. The fault-tolerant version of the 3D- protein reconstruction algorithm (FT-COMAR) proposed by Vassura *et al.* [84] is able to recover from errors in as many as 75% of the entries of the contact map.

4.3 Cooperative Hardware/Software Approaches

Cross-layer resilience techniques [58] employ multiple error resilience techniques from different layers of the system stack to collaboratively achieve error resilience. These frameworks combine selective circuit-level hardening and logic-level parity checking with algorithm-based fault tolerance methods to provide resilient operation.

5 Design Patterns for Resilience

5.1 Introduction to Design Patterns

A *design pattern* describes a generalizable solution to a recurring problem that occurs within a well-defined context. Patterns are often derived from best practices used by designers and they contain essential elements of the problems and their solutions. They provide designers with a template for how to solve a problem that can be used in many different situations. The patterns may also be used to describe design alternatives to a specific problem.

The original concept of design patterns was developed in the context of civil architecture and engineering problems [3]. The patterns were defined with the goal of identification and cataloging solutions to recurrent problems codify and solutions in the building and planning of neighborhoods, towns and cities, as well as in the construction of individual rooms and buildings. In the domain of software architecture, the intent of design patterns isn't to provide a finished design that may be transformed directly into code. Rather, design patterns are used to enhance the software development process by providing proven development paradigms. With the use of design patterns, there is sufficient flexibility for software developers to adapt their implementation to accommodate any constraints, or issues that may be unique to specific programming paradigms, or the target platform for the software.

In the context of object-oriented programming design patterns provide a catalog of methods for defining class interfaces and inheritance hierarchies, and establish key relationships among them [37]. In many object-oriented systems, reusable patterns of class relationships and communicating objects are used to create flexible, elegant, and ultimately reusable software design. In the pursuit of quality and scalable parallel software, patterns for parallel programming were developed [55] and a pattern language, called Our Pattern Language (OPL) [47], was used as the means to systematically describe these parallel computation patterns and use them to architect parallel software. In each of these domains of design and engineering, design patterns capture the essence of effective solutions in a succinct form that may be easily applied in similar form to other contexts and problems.

5.2 Design Patterns for HPC Resilience Solutions

Since the early days of computing, designers of computer systems repeatedly used well-known techniques to increase reliability: redundant structures to mask failed components, error-control codes and duplication or triplication with voting to detect or correct information errors, diagnostic techniques to locate failed components, and automatic switchovers to replace failed subsystems [5]. In general, every resilience solution consists of the following capabilities:

- **Detection:** Detecting the presence of errors or failures in the data or control value is an important aspect of any resilience management strategy. The presence of errors in the system are typically detected using redundant information.
- **Containment:** When an error is detected in a system, its propagation must be limited. The containment of an error requires specification of well-defined modular structures and interfaces. Containment strategies assist in limiting the impact of errors on other modules of the system.

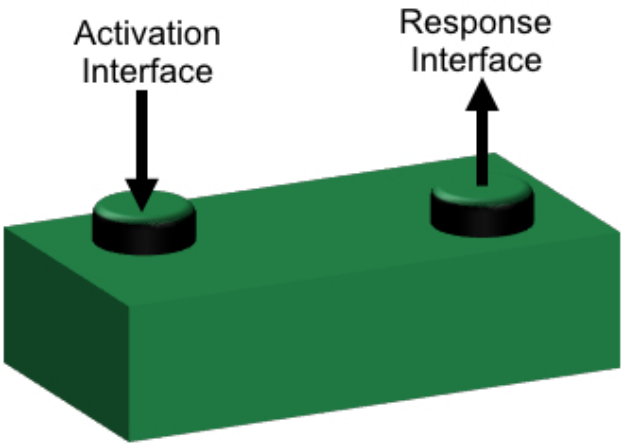

Figure 2: Anatomy of a Resilient Design Pattern

- **Recovery:** The recovery aspect of any resilience strategy is needed to ensure that the application outcome is correct in spite of the presence of the error. Recovery may entail preventing faults from resurfacing or eliminating the error completely. Rollback and roll-forward are used to position the system state to a previous or forward known correct state. Alternatively, the error may be compensated through redundancy. Recovery may also include system reconfiguration or reinitialization where the system is reset to a set of known parameters which guarantees correct state.

Every resilience strategy must contain these core capabilities. However, many design decisions in HPC resilience are unique and the approach to designing a solution may vary considerably based on the layer of system abstraction and the optimization constraints. Each resilience technique provides different guarantees regarding the properties associated to the system qualities such as the time or the space overhead introduced to the normal execution of the system, the efficiency of the reaction to a failure, the design complexity added to the system.

5.3 Anatomy of a Resilience Design Pattern

The basic template of a resilience design pattern is defined in an event-driven paradigm based on the insight that any resilience solution is necessary in the presence of, or sometimes in the anticipation of an anomalous event, such as a fault, error, or failure. The abstract resilience design pattern consists of a *behavior* and a set of *activation* and *response interfaces* (Figure 2). The appeal of defining the resilience design patterns in such an abstract manner is that they are universal. The abstract definition of the resilience design pattern behavior enables description of solutions that are free of implementation details. The instantiation of pattern behaviors may cover combinations of detection, containment and mitigation capabilities. The individual implementations of the same pattern may have different levels of performance, resilience, and power consumption. Also, the resilience pattern definition abstracts a pattern's interfaces from the implementation of these interfaces.

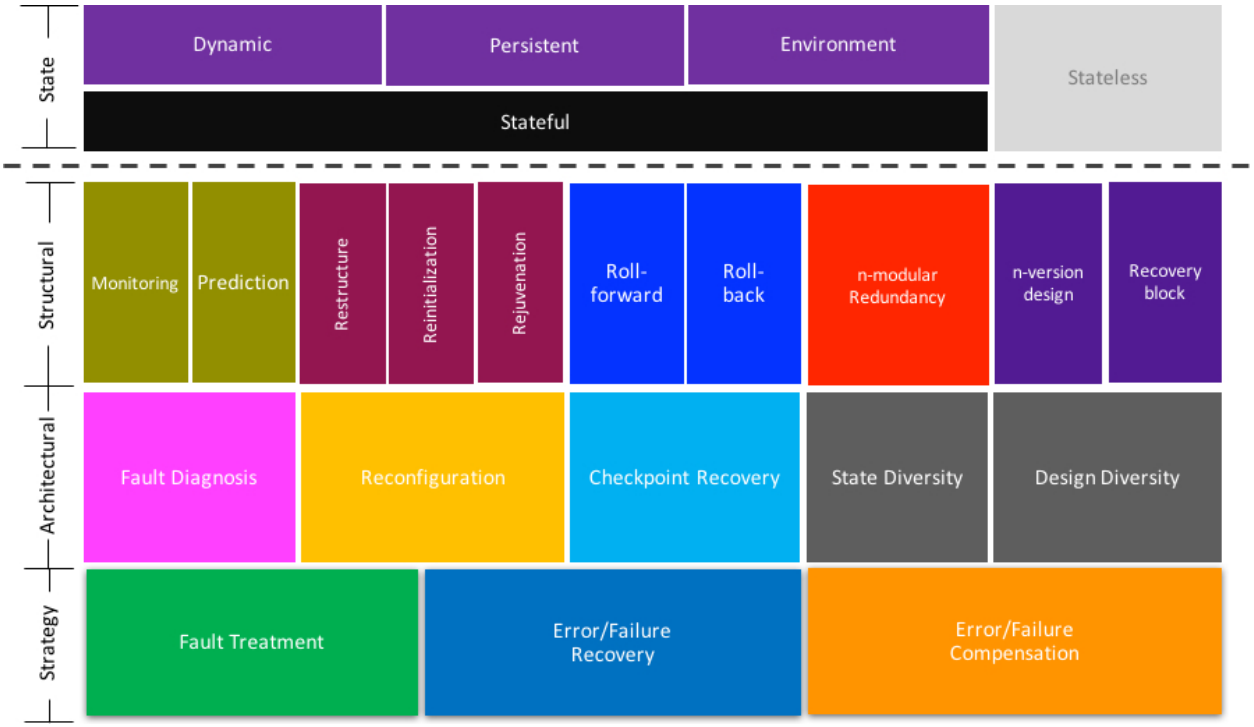

Figure 3: Classification of resilience design patterns

6 Classification of Resilience Design Patterns

Architecting a HPC system and its software ecosystem is a complex process. In order to make the incorporation of resilience capabilities an essential part of the design process, the resilience design patterns are presented in a layered hierarchy. The hierarchy enables different stakeholders to reason about resilience capabilities based on their view of the system and their core expertise — system architects may analyze protection coverage for the various hardware and software components that make up the system based on the patterns applied by each component; the designers of individual components may operate within a single layer of system abstraction and focus on instantiation of patterns based on local constraints and without the need to understand the overall system organization.

Resilience has two dimensions: (1) forward progress of the system and (2) data consistency in the system. Therefore, the design patterns are broadly classified into:

- **State Patterns:** These patterns describe all aspects of the system structure that are relevant to the forward progress of the system. The correctness and consistency of the system state ensures that the correct operation of the system. The state implicitly defines the scope of the protection domain that must be covered by a resilience mechanism.
- **Behavior Patterns:** These design patterns identify common detection, containment, or mitigation actions that enable the components that realize these patterns to cope with the presence of a fault, error, or failure event.

This classification enables designers to separately reason about scope of the protection domain, and the

semantics of the pattern behavior. The notion of state may be classified into three aspects [50]:

- *Persistent/Static State*, which represents the data that is computed once in the initialization phase of the application and is unchanged thereafter.
- *Volatile/Dynamic State*, which includes all the system state whose value may change during the computation.
- *Operating Environment State*, which includes the data needed to perform the computation, i.e., the program code, environment variables, libraries, etc.

While certain behavior patterns may be applied to individual aspects of the system state, the state patterns may also be fused in order to enable the application of a single behavior pattern to more than one state pattern. Certain resilience behaviors may be applied without regard for state; such patterns are concerned with only the forward progress of the system. The classification of state patterns also includes a *stateless* pattern to enable designers to create solutions that define behavior without state.

The behavior patterns are presented in a layered hierarchy (Figure 3) to provide designers with the flexibility to organize the patterns in well-defined and effective solutions:

- **Strategy Patterns:** These patterns define high-level polices for a resilience solution. Their descriptions are deliberately abstract to enable hardware and software architects to reason about the overall organization of the techniques used and their implications on the full system design.
- **Architecture Patterns:** These patterns convey specific methods of construction of a resilience solution. They explicitly convey the type of fault/error/failure event that they handle and provide detail about the key components and connectors that make up the solution.
- **Structure Patterns:** These patterns provide concrete descriptions of implementation strategies. They comprise of recipes that may be directly realized in hardware/software components. The implementation of these patterns is closely related to the state patterns.

Implementation patterns bridge the gap between the design principles and the concrete details. These are compound patterns, i.e., patterns of patterns, and consist of a structure pattern and a state pattern. The specification also defines a *complete* resilience solution as one which provides detection/diagnosis of a fault, error or failure event, containment of its effects, and capabilities for mitigation to enable the affected system to continue intended operation.

7 The Resilience Pattern Catalog

7.1 Describing Design Patterns

Patterns are expressed in a written form in a highly structured format to enable HPC architects and designers to quickly discover whether the pattern is relevant to the problem being solved. For convenience and clarity, each resilience pattern follows the same prescribed format. There are three key reasons behind this pattern format: (1) to present the pattern solution in a manner that simplifies comparison of the capabilities of patterns and their use in developing complete resilience solutions, (2) to present the solution in a sufficiently abstract manner that designers may modify the solution depending on the context and other optimization parameters, and (3) to enable these patterns to be instantiated at different layers in the system.

Name:
Identifies the pattern and provides a convenient way to refer to it, typically using a short phrase.

Problem:
A description of the problem indicating the intent behind applying the pattern. This describes the intended goals and objectives that will accomplished with the use of this specific pattern.

Context:
The preconditions under which the pattern is relevant, including a description of the system before the pattern is applied.

Forces:
A description of the relevant forces and constraints, and how they interact or conflict with each other and with the intended goals and objectives. This description clarifies the intricacies of the problem and make explicit the trade-offs that must be considered.

Solution:
A description of the solution that includes specifics of how to achieve the intended goals and objectives. This description identifies the core structure of the solution and its dynamic behavior, including its collaborations with other patterns. The description may include guidelines for implementing the solution as well as descriptions of variants or specializations of the solution.

Capability:
The resilience management capabilities provided by this pattern, which may include detection, containment, mitigation, or a combination of these capabilities. The listing of capabilities enables designers to determine whether other patterns must be employed to compose a complete resilience solution.

Protection Domain:
The resiliency behavior provided by the pattern extends over a certain scope, which may not always be explicit. Also, a solution may be suitable for a specific fault model. The description of scope and nature of fault model that is supported by the pattern enables designers to reason about the coverage scope in terms of the complete system.

Resulting Context:
A brief description of the post-conditions arising from the application of the pattern. There may be trade-offs between competing optimization parameters that arise due to the implementation of a solution using this pattern. The resulting context describes what aspects of the systems have been provided with protection, and which remain unprotected. This indicates other patterns that may be applied for supplementing the protection domain.

Examples:
One or more sample applications of the pattern, which illustrate the use of the pattern for a specific problem, the context, and set of forces. This also includes a description of how the pattern is applied, and the resulting context.

Rationale:
An explanation of the pattern as a whole, with an elaborate description of how the pattern actually works for specific situations. This provides insight into its internal workings of a resilience pattern, including details on how the pattern accomplishes the desired goals and objectives.

Related Patterns:
The relationships between this pattern and other relevant patterns. These patterns may be predecessor or successor patterns in the hierarchical classification. The pattern may collaborate to complement, or enhance the resilience capabilities of other patterns. There may also be dependency relationships between patterns, which necessitate the use of co-dependent patterns in order to develop complete resilience solutions.

Known Uses:
Known applications of the pattern in existing HPC systems, including any practical considerations and limitations that arise due to the use of the pattern at scale in production HPC environments.

Terminology: The aim of defining a catalog of resilience design patterns is to provide reusable solutions to specific problems in a way that they may be instantiated in various ways, in hardware and software. Hardware design covers design of microarchitecture blocks, processor architecture, memory hierarchy design, network interface design, as well as design of racks, cabinet and system-level design. The scope of software design spans the spectrum of operating systems; runtimes for scheduling, memory management, communication frameworks, performance monitoring tools, computational libraries; compilers; programming languages; and application frameworks. In order to make the resilience pattern relevant to these diverse domains of computer system design, we describe solutions in a generic manner. The desrciptions use *system* to refer to an entity that has the notion of structure and behavior. A *subsystem* is a set of elements, which is a system itself, and is a component of a larger system, i.e., a system is composed of multiple sub-systems or components. For a HPC system architect, the scope of system may include compute nodes, I/O nodes, network interfaces, disks, etc., while an application developer may refer to a library interface, a function, or even a single variable as a system. The instantiation of the pattern descriptions interpret the notion of *system* to refer to any of these hardware or software-level component. A *full system* refers to the HPC system as a whole, and which is capable of running an application.

7.2 Strategy Patterns

7.2.1 Fault Treatment Pattern

Name: Fault Treatment Pattern

Problem: The `Fault Treatment` pattern solves the problem of discovering and treating the presence of defects or anomalies in the system that have the potential to activate, leading to error or failure conditions in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring sub-system to discover the presence of anomalies in the behavior of a monitored sub-system.
- The interaction between the monitored and monitoring sub-systems is bounded in terms of time.
- The monitoring sub-system may have the capability modify the parameters of the monitored sub-system that enables the anomaly or defect to be removed before it results in an error or a failure.

Forces:

- The interactions of the monitoring and monitored sub-systems may interfere with the operation of the monitored sub-system. The frequency of these interactions must be bounded.
- The time interval for the monitoring sub-system to gather data about the monitored sub-system and infer the presence of an anomaly or a defect incurs overhead to the operation of the monitored sub-system.

Solution: The `Fault Treatment` pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system, and creates conditions that prevents the activation of the fault into an error or failed state.

Capability: The pattern provides fault mitigation semantics, which supports the following capabilities:

- **Fault detection**, to detect anomalies during operation before they impact the correctness of the system state, and cause errors and failures.
- **Fault mitigation**, which includes methods to enable an imminent error or failure to be prevented, or the defect to be removed.

The `Fault Treatment` pattern may support either one, or both of these capabilities.

Protection Domain: The protection domain extends to the scope of the monitored sub-system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other sub-systems that are interfaced to the monitored sub-system.

Resulting Context: The `Fault Treatment` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring sub-system, which introduces design complexity in the overall system design. When the monitoring sub-system is extrinsic to the monitored sub-system, the design effort may be simplified, and the interference between the sub-systems may be kept minimal. However, when the monitoring sub-system is intrinsic to the design of the monitored sub-system, the complexity of the design process increases, as well as the interaction between the sub-systems.

Examples: Various hardware-based solutions for fault detection observe the attributes of a system, such as thermal state, timing violations in order to determine the presence of a defect in the behavior of the system that may potentially cause an error or failure. Similarly, software-based solutions detect the anomalies in the behavior of a system's data or control flow attributes to determine the presence of a fault.

Rationale: The key benefit of incorporating fault mitigation patterns in a design, or deploying it during system operation is to preemptively recognize faults in the system, before they are activated and result in errors or partial/complete failures of the subsystem. The preventive actions avoid the need for expensive recovery and/or compensation actions.

Related Patterns: The recovery and compensation patterns are complementary to the fault treatment patterns. Those patterns are necessary only the fault has been activated and an error or failure state exists in the system.

Known Uses:

- Processor chips used in HPC systems contain thermal sensors that detect anomalous thermal conditions in the processor cores. When the temperature reaches a preset level, the sensor trips and processor execution is halted to prevent failure of the chip.
- Software-based heartbeat monitoring for liveness checking of MPI processes enables detection of faults in a processor rank, before it may activate to result in failure of the MPI communicator.
- The Cray RAS and Management System (CRMS) supports real-time fault monitoring of the status of Cray XT series system components, including the cabinets, blades, CPUs, Seastar processors and Seastar links.

7.2.2 Recovery Pattern

Name: Recovery Pattern

Problem: The `Recovery` pattern solves the problem of errors in the system, or failure of the system leading to catastrophic failure, which results in fail-stop behavior. In an HPC environment, the occurrence of errors or failures in the system results in catastrophic crashes, or incorrect results.

Context: The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The system must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected; the pattern offers no implicit error/failure detection.
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.

Forces:

- The pattern requires stable storage to capture system state, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The amount of state captured during each creation of a recovery point incurs space and time overheads.
- The frequency of creation of system state snapshots determines overhead: more frequent creation of recovery points increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The interval between recovery from consecutive errors/failures must be less than the interval to create a stable recovery point from the present state of the system to enable system to make forward progress.

Solution: The solution suggested by the `Recovery` pattern is based on the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system. Upon detection of an error or a failure, the checkpoints are used to recreate last known error/failure-free state of the system. Based on a temporal view of the system's progress, the error recovery may be either forward or backward.

Capability: The `Recovery` pattern periodically preserves the essential aspects of the system state that may be subsequently used to resume operation from a known stable and correct version. The pattern handles an error or a failure by substituting an error-free state in place of the erroneous state. The pattern enables a system to tolerate errors/failures by resuming operation from a stable checkpointed version of the system that is free of the effects of the error/failure. The solution offered by this pattern is not dependent on

the precise semantics of the error/failure propagation. The pattern does not offer error/failure detection capabilities.

Protection Domain: The protection domain for a `Recovery` pattern is determined by the extent of state that is captured during checkpoint operation, which accurately representatives the complete execution of the system. The broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event.

Resulting Context: With a recovery pattern, a system is capable of tolerating failures by substitution of erroneous/failed system state to a known previous stable state (backward recovery), or to an inferred future stable state (forward recovery). The frequency of creation of checkpoints determines the overhead to system operation; frequent checkpointing incurs proportionally greater overheads during error/failure-free operation, but reduces the amount of lost work when an error/failure event does occur. The latency of saving and restoring state influence the overhead during error/failure-free operation and the overhead of recovering from an error/failure respectively.

Examples:

- Various checkpoint and restart libraries enable HPC applications to capture program state and commit the checkpoint files to parallel file systems. C/R capabilities in the OS such as BLCR [28] and libckpt [66] enables checkpointing the process state.

Rationale: Since the solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low.

Related Patterns: The `Compensation` pattern is complementary to the recovery pattern, although they both seek to create conditions to recreate correct state. The key difference between the `Recovery` and the `Compensation` patterns is the method used to maintain any additional state that is used for error/failure processing. While the compensation pattern use replication of the system, a recovery pattern relies on committing error/failure-free versions of the system to stable storage.

Known Uses:

- The Berkeley Labs C/R library (BLCR) [28] is an extension to the Linux OS that supports creation checkpoint & restart capabilities for Linux processes and also provides an interface for programmers to checkpoint application program state.

7.2.3 Compensation Pattern

Name: Compensation Pattern

Problem: The `Compensation` pattern solves the problem of errors in the system, or failure of the system leading to catastrophic failure, which results in fail-stop behavior. In an HPC environment, the occurrence of errors or failures in the system results in catastrophic crashes, or incorrect results.

Context: The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The system must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected, although the pattern may offer implicit error/failure detection.
- The error/failure must not be in the inputs provided to the system.

Forces:

- The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs.
- The error/failure-free overhead penalty must be minimized.

Solution: The `Compensation` pattern is based on creation of a group of system replicas. The replicas are functionally identical and each replica receives identical inputs.

Capability: The pattern provides error/failure detection and correction, depending on the level of replication. The replicas of the system permit the system to continue operation despite the occurrence of a fail-stop failure by substituting a failed system replica with another. In order to recover from 2N failures in the system, there must be 2N + 1 distinct replicas. For the detection and correction errors, the outputs of the replicas of the system are compared by a monitor sub-system. When there are at least two replicas of the system, the monitor system compares the outputs to determine the presence of an error or failure in either replica. When the number of replicas is greater than two, and an odd number, the monitor performs majority voting on the outputs produced by the replicas, which enables incorrect outputs from replicas in error/failed state to be filtered out.

Protection Domain: The protection domain of the `Compensation` pattern extends to the scope of the system that is replicated.

Resulting Context: The pattern requires the replication of the system and its inputs. The design effort and complexity of replication of the system depends on the replication method. A naive replication requires low design effort; the design of functionally identical but independently designed versions of the replicas requires higher design effort. The preparedness of the replica during system operation to compensate for the error/failure state determines the level of overhead: The replica may *active* or *passive*; the replica state may be classified *hot*, *warm* and *cold* are the forms of replication configurations based on the levels of intervention required for compensating for an error/failure.

Examples:

- Dual-modular redundancy for error detection; triple-modular redundancy for error detection and correction
- Redundant information for compensation of data errors

Rationale: The `Compensation` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system to substitute a failed system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Related Patterns: The `Recovery` pattern is complementary to the compensation pattern, although they both seek to create conditions to recreate correct state. The key difference between the `Recovery` and the

`Compensation` patterns is the method used to maintain any additional state that is used for error/failure processing. Unlike `Recovery` pattern, which uses a temporally forward or backward error/failure-free version of the system, a compensation pattern utilizes some form of redundancy to tolerate errors/failures in the system.

Known Uses:

- Production HPC systems use memory modules that contain SECDED ECC, which maintain redundant bits per memory line. These redundant bits compensate for bit flip errors within the memory lines and enables detection and correction of certain errors.
- The MR-MPI is an implementation of the MPI that transparently replicates and detects errors in MPI messages through active comparison between redundant execution instances of an application.

7.3 Architectural Patterns

7.3.1 Fault Diagnosis Pattern

Pattern Name: Fault Diagnosis Pattern

Problem: The `Fault Diagnosis` pattern solves the problem of identifying the presence of a defect or anomaly in the system. A fault in the system has the potential to activate, leading to the occurrence of an error or a failure.

Context: The pattern applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring sub-system to discover the presence of anomalies in the behavior of a monitored sub-system.
- The interaction between the monitored and monitoring sub-systems is bounded in terms of time.
- The monitoring sub-system has the capability to analyze the behavior of the monitored sub-system.

Forces:

- The interactions of the monitoring and monitored sub-systems may interfere with the operation of the monitored sub-system. The frequency of these interactions must be bounded.
- The time interval for the monitoring sub-system to gather data about the monitored sub-system and infer the presence of an anomaly or a defect incurs overhead to the operation of the monitored sub-system.

Solution: The `Fault Diagnosis` pattern contains a monitoring sub-system that observes specific parameters of a monitored sub-system. The monitoring sub-system contains a range of acceptable values for the observed parameter to establish the notion of *normal* operation of the monitored sub-system. The monitoring sub-system observes deviations in the parameters to determine the presence of a fault, and the location of the fault.

Capability: The pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system and identifies the fault location and type.

Protection Domain: The protection domain extends to the scope of the monitored sub-system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other sub-systems that are interfaced to the monitored sub-system.

Resulting Context: The `Fault Diagnosis` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring sub-system, which introduces design complexity in the overall system design. When the monitoring sub-system is extrinsic to the monitored sub-system, the design effort may be simplified, and the interference between the sub-systems may be kept minimal. However, when the monitoring sub-system is intrinsic to the design of the monitored sub-system, the complexity of the design process increases, as well as the interaction between the sub-systems.

Rationale: The fault diagnosis patterns enable error/failure avoidance by detecting an anomaly in the system before it results in an error/failure.

Examples: Various hardware faults are detected through analysis of software symptoms, such as observation of latency of operations.

Related Patterns: In contrast to the error/detection recovery and compensation patterns, the fault diagnosis pattern is a passive pattern that observes system behavior and infers the presence of a fault based on the deviation from specified *normal* behavior of the system.

Known Uses:

- SMART (Self-Monitoring and Reporting Technology) is used in disk systems.

7.3.2 Reconfiguration Pattern

Pattern Name: Reconfiguration Pattern

Problem: The `Reconfiguration` pattern prevents a fault, error or failure in the system state affecting the correct operation of a system.

Context: The pattern applies to a system that has the following characteristics:

- The system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.

- The reconfiguration may require the system to operate in degraded state using fewer than N sub-systems. The performance degradation of the system must be minimized.

Solution: The `Reconfiguration` pattern is based on isolation of the part of a system affected by a fault, error or failure, excluding the affected sub-system from interaction with other sub-systems, and restructuring the system to remain functionally equivalent to the system before the occurrence of the fault, error or failure event.

Capability: The pattern enables systems to tolerate the impact of a fault, error or failure by enabling the system to continue operation by preventing the affected part of the system from affecting the correctness of the system.

Protection Domain: The protection domain of the `Reconfiguration` pattern spans the part of system whose state may be reconfigured, and yet is able to continue operating in a functionally equivalent operating state.

Resulting Context:

- The reconfiguration of the system may result in the operation of the system in degraded condition. This incurs additional time overhead to the system.
- The pattern introduces additional design complexity since the system must remain functionally correct in multiple configurations.

Examples:

- Cluster management systems dynamically adapt the cluster configuration based on the health of various compute nodes in the system.

Rationale: The `Reconfiguration` pattern enables a system to tolerate to a fault, error or failure by adapting itself to the impact of the event and continuing to operate. The pattern enables systems to make forward progress by relying on the reconfigured version of the system.

Related Patterns: Like the `Checkpoint-Recovery` pattern, the `Reconfiguration` pattern is also supports recovery of the system from the impact of a fault, error or failure event. While the `Checkpoint-Recovery` pattern maintains snapshots of the system to stable storage to perform forward or backward recovery, the `Reconfiguration` pattern requires the system to adapt itself to isolate the impact of the event.

Known Uses: The ULFM implementation of the MPI interface [8] supports enables parallel applications to survive process failures during application execution by isolating the failed process, establishing agreement between the remaining processes in a MPI communicator and reconfiguring the communicator to include the remaining process ranks.

7.3.3 Checkpoint Recovery Pattern

Pattern Name: Checkpoint Recovery Pattern

Problem: The `Checkpoint-Recovery` pattern solves the problem of errors in the system, or failure of the system leading to catastrophic failure, which results in fail-stop behavior.

Context: The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The system must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected; the pattern offers no implicit error/failure detection.
- The error or failure that the pattern handles must be transient, i.e., the error/failure must not repeatedly occur post-recovery
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.

Forces:

- The pattern requires stable storage to capture system state, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The amount of state captured during each creation of a recovery point incurs space and time overheads.
- The frequency of creation of system state snapshots determines overhead: more frequent creation of recovery points increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The interval between recovery from consecutive errors/failures must be less than the interval to create a stable recovery point from the present state of the system to enable system to make forward progress.

Solution: The solution suggested by the `Checkpoint-Recovery` pattern is based on the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system. Upon detection of an error or a failure, the checkpoints are used to recreate last known error/failure-free state of the system. Based on a temporal view of the system's progress, the error recovery may be either forward or backward.

Capability: The `Checkpoint-Recovery` pattern periodically preserves the essential aspects of the system state that may be subsequently used to resume operation from a known stable and correct version. The pattern handles an error or a failure by substituting an error-free state in place of the erroneous state. The pattern enables a system to tolerate errors/failures by resuming operation from a stable checkpointed version of the system that is free of the effects of the error/failure. The solution offered by this pattern is not dependent on the precise semantics of the error/failure propagation. The pattern does not offer error/failure detection capabilities.

Protection Domain: The protection domain for a `Checkpoint-Recovery` pattern is determined by the extent of state that is captured during checkpoint operation, which accurately representatives the complete

execution of the system. The broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event.

Resulting Context: With a recovery pattern, a system is capable of tolerating failures by substitution of erroneous/failed system state to a known previous stable state (backward recovery), or to an inferred future stable state (forward recovery). The frequency of creation of checkpoints determines the overhead to system operation; frequent checkpointing incurs proportionally greater overheads during error/failure-free operation, but reduces the amount of lost work when an error/failure event does occur. The latency of saving and restoring state influence the overhead during error/failure-free operation and the overhead of recovering from an error/failure respectively.

Rationale: Since the solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low.

Examples: Application and system-level libraries that provide interfaces for creating checkpoints and restoring state are examples of the checkpoint-recovery pattern.

Related Patterns: The `State Diversity` and `Design Diversity` patterns are complementary to the `Checkpoint-Recovery` pattern, although they both seek to create conditions to recreate correct state. The key difference between these classes of patterns is the method used to maintain any additional state that is used for error/failure processing. While the diversity patterns use replication of the system, a recovery pattern relies on committing error/failure-free versions of the system to stable storage.

Known Uses: In order to tolerate fail-stop errors, current production-quality technologies rely on the classic rollback recovery approach using checkpoint restart (application-level or system-level) such BLCR, SCR.

7.3.4 State Diversity Pattern

Pattern Name: State Diversity Pattern

Problem: The `State Diversity` pattern solves the problem of detecting and correcting errors or failures in the system state.

Context: The pattern applies to a system that has the following characteristics:

- The system must be deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs.
- The error/failure-free overhead penalty introduced by the replication of state must be minimized.

Solution: The `State Diversity` pattern creates a group of N replicas of a system's state. The redundancy may include replication of the system's operation and/or the inputs to the system. Each of the N copies of

the system state exist simultaneously. The redundant state versions of the systems are provided with the identical inputs, and their respective outputs are compared in order to detect and potentially correct the impact of an error or a failure in either replica of the systems.

Capability: The availability of replicated versions of the system state enable the following capabilities:

- Fail-over, which entails substitution of a replica in error or failed state with another replica that is error/failure-free.
- Comparison, which entails observing the likeness of each replica's outputs as means to detect the presence of an error or failure in either replica.
- Majority voting on the outputs produced by each replica system enables the detection of errors and failures, and filtering out the outputs that fall outside the majority.

Protection Domain: The protection domain of the pattern extends to the scope of the system state that is replicated.

Resulting Context: The design effort and complexity of replication of the system state requires low design effort since the replication entails creation of identical copies of the system state.

Rationale: The `State Diversity` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system state to substitute a failed system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Examples:

- Dual-modular redundancy for error detection; triple-modular redundancy for error detection and correction
- Redundant information for compensation of data errors

Related Patterns: The `State Diversity` and `Design Diversity` patterns are based on inclusion of redundancy in order to compensate for errors or failures. The diversity in the `State Diversity` pattern stems from the replication of the system's state unlike the `Design Diversity` pattern, which uses independently implemented versions of the system's design to tolerate errors or failures.

Known Uses: The use of ECC memory and Chipkill in production HPC systems are known uses of the `State Diversity` pattern.

7.3.5 Design Diversity Pattern

Pattern Name: Design Diversity Pattern

Problem: The `Design Diversity` pattern solves the problem of detecting and correcting errors or failures in the behavior of the system that may occur due design faults in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has a well-defined specification for which multiple implementation variants may be designed.

- There is an implicit assumption of independence of between multiple variants of the implementation.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern requires distinct implementations of the same design specification, which are created by different individuals or teams.
- The pattern increases the system complexity due to the need additional design and verification effort required to create multiple implementations.
- The error/failure-free overhead penalty due to disparity in the implementation variants must be minimized.

Solution: The pattern enables systems to tolerate design faults in that may arise out of and incorrect interpretation of the specifications, or due to mistakes during implementation. The design diversity pattern entails partitioning the system into N replica sub-systems that are variants of a system design. These replicas are developed separately but are designed to a common specification. These design variants are applied in a time or space redundant manner. The redundant systems are provided with the identical inputs and their respective outputs are compared in order to detect and potentially correct the impact of an error or a failure in either replica of the systems.

Capability: The pattern relies on independently created, but functionally equivalent sub-system versions of a system specification. Since the sub-systems operate in parallel, in a time or space redundant manner, they are able to account for errors or failures caused by design flaws.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which multiple implementation variants are created.

Resulting Context: Although the primary intent of the `Design Diversity` pattern is to enable systems to tolerate errors and failures due to design faults, the pattern also supports resilience transient errors/failures. Since the replica sub-systems are functionally identical and are deployed in a redundant manner, the likelihood that each replica is affected by the same transient error or failure is small.

Rationale: The intent behind applying this pattern is to eliminate the impact of human error during the implementation of a system. Due the low likelihood that different individuals or teams introduce identical bugs in their respective implementations, the pattern enables compensating for errors or failures caused by a bug in any one implementation of the same design.

Examples: The concept of N-modular (NMR) programming is used in developing software development in which variants of a software developed by different teams, but to a common specification. These implementations may be applied in a time or space redundant manner to detect and correct errors or failures that are caused by bugs in the implementation.

Related Patterns: The `Design Diversity` and `State Diversity` patterns are based on inclusion of redundancy in order to compensate for errors or failures. In contrast to the `State Diversity` pattern, which replicates the system state, the `Design Diversity` pattern typically uses multiple versions of the system that are only functionally equivalent.

Known Uses: Applications that require high precision floating point arithmetic, particularly application that require multiple-precision floating-point computations use multiple alternative compiler toolchains and library implementations that are functionally equivalent to ensure high precision in the computations.

7.4 Structural Patterns

7.4.1 Monitoring Pattern

Pattern Name: Monitoring Pattern

Problem: The `Monitoring` pattern solves the problem of analyzing the behavior of a system that indicates the immediate presence of a defect or anomaly in the system that has the potential to cause errors or failures in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring sub-system to discover the presence of anomalies in the behavior of a monitored sub-system.
- The interaction between the monitored and monitoring sub-systems is bounded in terms of time.
- The monitoring sub-system has the capability to readily analyze the behavior of the monitored sub-system in order to identify anomalous behavior.

Forces:

- The interactions of the monitoring and monitored sub-systems may interfere with the operation of the monitored sub-system. The frequency of these interactions must be bounded.
- The time interval for the monitoring sub-system to gather data about the monitored sub-system and infer the presence of an anomaly or a defect incurs overhead to the operation of the monitored sub-system.

Solution: The `Monitoring` pattern contains a monitoring sub-system that observes specific parameters of a monitored sub-system. The monitoring sub-system contains a range of acceptable values for the observed parameter to establish the notion of *normal* operation of the monitored sub-system. The monitoring sub-system observes deviations in the parameters and the location of the anomaly to determine the root cause, type and precise location of a fault.

Capability: The pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system and identifies the fault location and type.

Protection Domain: The protection domain extends to the scope of the monitored sub-system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other sub-systems that are interfaced to the monitored sub-system.

Resulting Context: The `Monitoring` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring sub-system, which introduces design

complexity in the overall system design. When the monitoring sub-system is extrinsic to the monitored sub-system, the design effort may be simplified, and the interference between the sub-systems may be kept minimal. However, when the monitoring sub-system is intrinsic to the design of the monitored sub-system, the complexity of the design process increases, as well as the interaction between the sub-systems.

Rationale: The pattern enables the monitored sub-system to determine the presence of a fault and to analyze its root cause and location. The pattern enables the system to take precise corrective actions to prevent the activation of the fault to cause an error or failure event in the system.

Examples: Various hardware and software systems provide the capabilities to observe system behavior for the purpose of inferring the presence of faults, such as online self-tests.

Related Patterns: The structure of the `Monitoring` pattern is closely related to the `Prediction` pattern since they both contain monitoring and monitored sub-system entities. The key difference between these patterns is the amount of temporal information used by the patterns to assess the presence of a defect or anomaly in the system. The `Monitoring` pattern uses presently observed system parameters in contrast to the `Prediction` pattern, which uses historical trend information to forecast future fault events.

Known Uses: The Intelligent Platform Management Interface (IPMI) provides message-based interface to collect sensors readings for health monitoring, including the data on temperature, fan speed, and voltage for the purpose of monitoring the hardware components in the system.

7.4.2 Prediction Pattern

Pattern Name: Prediction Pattern

Problem: The `Prediction` pattern solves the problem of identifying patterns of behavior that indicate the potential for future errors or failures in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring sub-system to discover the presence of anomalies in the behavior of a monitored sub-system.
- The interaction between the monitored and monitoring sub-systems is bounded in terms of time.
- The monitoring sub-system has the capability to store historical data about the behavior of the monitored sub-system in order to analyze trends in fault occurrences.

Forces:

- The interactions of the monitoring and monitored sub-systems may interfere with the operation of the monitored sub-system. The frequency of these interactions must be bounded.
- The time interval for the monitoring sub-system to gather data about the monitored sub-system and infer the presence of an anomaly or a defect incurs overhead to the operation of the monitored sub-system.

Solution: The `Prediction` pattern contains a monitoring sub-system that observes specific parameters of a monitored sub-system. The monitoring sub-system contains storage of the history of observed parameter values and fault events in the monitored sub-system. The monitoring sub-system uses past experiences of correlating the parameter values and fault occurrences to establish trends. These trends are used to predict occurrence of future faults based on observed deviations in the parameters.

Capability: The pattern provides a method that anticipates the occurrence of fault events based on patterns of behavior of the monitored sub-system that attempts to recognize the potential for future occurrences of an anomaly or a defect within a system.

Protection Domain: The protection domain extends to the scope of the monitored sub-system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other sub-systems that are interfaced to the monitored sub-system.

Resulting Context: The `Prediction` pattern requires the designer to identify parameters that may be used to forecast the occurrence of fault events. The system design must include a monitoring sub-system, which introduces design complexity in the overall system design. When the monitoring sub-system is extrinsic to the monitored sub-system, the design effort may be simplified, and the interference between the sub-systems may be kept minimal. However, when the monitoring sub-system is intrinsic to the design of the monitored sub-system, the complexity of the design process increases, as well as the interaction between the sub-systems.

Rationale: The pattern enables the monitored sub-system to use historical trends in system behavior before and during fault events to predict future fault events. If future fault events are predicted with high precision, then avoidance or preventive actions may be used.

Examples: Hardware and software systems use correlations between past behaviors to predict the future occurrences of fault events, such as a memory device tends to show, for a given address, multiple repetitive correctable errors before showing an uncorrectable error.

Related Patterns: The structure of the `Prediction` pattern is closely related to the `Monitoring` pattern since they both contain monitoring and monitored sub-system entities. The key difference between these patterns is the amount of temporal information used by the patterns to assess the presence of a defect or anomaly in the system. The `Prediction` pattern uses historical trend information to forecast future fault events in contrast to the `Monitoring` pattern, which uses presently observed system parameters.

Known Uses: IPMI compliant servers have a System Event Log (SEL) which is a centralized, nonvolatile repository for all events generated. The trends in the SEL are used to make predictions of future events in the system.

7.4.3 Restructure Pattern

Pattern Name: Restructure Pattern

Problem: The `Restructure` pattern solves the problem of a fault, error, or failure event affecting the correct operation of a system.

Context: The pattern applies to a system that has the following characteristics:

- The system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The reconfiguration may require the system to operate in degraded state using fewer than N sub-systems. The performance degradation of the system must be minimized.

Solution: The `Restructure` pattern is based on modifying the configuration between the N interconnected sub-systems to isolate the sub-system affected by a fault, error or failure. This reconfiguration excludes the affected sub-system from interaction with other sub-systems. The restructured system includes N-1 sub-systems, and yet seeks to remain functionally equivalent to the system before the occurrence of the fault, error or failure event.

Capability: The pattern enables systems to tolerate the impact of a fault, error or failure by enabling the system to continue operation by preventing the affected part of the system from affecting the correctness of the system.

Protection Domain: The protection domain of the `Restructure` pattern spans the part of system whose state may be reconfigured, and yet is able to continue operating in a functionally equivalent operating state.

Resulting Context:

- The reconfiguration of the system may result in the operation of the system in degraded condition. This incurs additional time overhead to the system.
- The pattern introduces additional design complexity since the system must remain functionally correct in multiple configurations.

Rationale: The `Restructure` pattern enables a system to tolerate to a fault, error or failure by adapting itself to the impact of the event and continuing to operate. The pattern enables systems to make forward progress by relying on the reconfigured version of the system.

Examples: Dynamic page retirement schemes are an example of the restructure pattern, in which pages that have an history of frequent memory errors are removed from the pool of available pages.

Related Patterns: The remaining reconfiguration patterns - the rejuvenation and reinitialization patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses: Chipkill memory provides the capabilities to reduces the amount of available system memory when a DIMM experiences an escalated sequence of ECC memory errors. The reconfiguration enables HPC system operation to continue with degraded performance until the defective memory module is replaced.

7.4.4 Rejuvenation Pattern

Pattern Name: Rejuvenation Pattern

Problem: The `Rejuvenation` pattern solves the problem of a fault, error, or failure event affecting the correct operation of a system.

Context: The pattern applies to a system that has the following characteristics:

- The system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.
- The fault, error, or failure must not be persistent.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The rejuvenation is often a slow process that requires substantial additional overhead to identify the part of the system affected by the fault, error or failure, and to selectively reinitialize the system, in addition to overhead incurred due to any lost work.

Solution: The `Rejuvenation` pattern requires isolating the specific part of the system affected by an error/failure and restoring or recreating the affected state such that the system may resume normal operation.

Capability: The pattern requires the system operation to be halted and identifying the part of the system affected by the error/failure. Only the affected part of the system is restored to ensure correct operation of the system.

Protection Domain: The protection domain of the `Restructure` pattern spans the part of system whose state may be reconfigured, and yet is able to continue operating in a functionally equivalent operating state.

Resulting Context:

- The rejuvenation of the system expects the result in the operation of the system in degraded condition. This incurs additional time overhead to the system.
- The overhead in terms of time to identify the specific state affected by the fault, error or failure, and restore the it to known correct state may be considerable.

Rationale: The `Rejuvenation` pattern enables a system to tolerate to a fault, error or failure by restoring the affect part of the system to known state that will ensure correct operation. Such targeted recovery prevents complete reset, or restructuring the system, both of which carry considerable overhead to the system operation.

Examples: The targeted recovery of data structures in system software, such as kernel modules, permits recovery without the need to reinitialize the complete system.

Related Patterns: The remaining reconfiguration patterns - the reinitialization and restructure patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses: Algorithm-based recovery methods for data corruptions in data structures that are used in numerical analysis problems use interpolation of neighboring data values to rejuvenate a data structure in error state.

7.4.5 Reinitialization Pattern

Pattern Name: Reinitialization Pattern

Problem: The `Reinitialization` pattern solves the problem of a fault, error, or failure event affecting the correct operation of a system.

Context: The pattern applies to a system that has the following characteristics:

- The system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.
- The fault, error, or failure must not be persistent.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The reinitialization is often a slow process that requires substantial additional overhead to reinitialize the system, in addition to overhead incurred due to lost work.

Solution: To recover the error/failure, the pattern restores the system to its initial state. This causes system operation to *restart* and a pristine reset of state, which implicitly cleans up the effects of the error/failure.

Capability: The `Reinitialization` pattern performs a reset of the system state to restore pristine state before system operation is resumed.

Protection Domain: Since the reinitialization causes reset of the system state, the protection domain of the `Reinitialization` pattern spans the complete system.

Resulting Context: The restoral of the system state to the initial state causes lost work, but guarantees the impact of the fault/error/failure is completely removed before service is resumed.

Rationale: The `Reinitialization` pattern is applied in conditions in which the recovery from the fault/error/failure instance is deemed impossible, or excessively expensive in terms of overhead to performance.

Examples: A system reboot is an instance of the `Reinitialization` pattern.

Related Patterns: The remaining reconfiguration patterns - the rejuvenation and restructure patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses: Various cluster management software systems enable malfunctioning nodes in the cluster to be reset by initiating reboot sequence for a specific node without disrupting the remaining nodes in the system.

7.4.6 Roll-back Pattern

Pattern Name: Roll-back Pattern

Problem: The `Roll-back` pattern solves the problem of errors in the system, or failure of the system leading to catastrophic failure, which results in fail-stop behavior.

Context: The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The system must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected; the pattern offers no implicit error/failure detection.
- The error or failure that the pattern handles must be transient, i.e., the error/failure must not repeatedly occur post-recovery
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.

Forces:

- The pattern requires stable storage to capture system state, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The amount of state captured during each creation of a recovery point incurs space and time overheads.
- The frequency of creation of system state snapshots determines overhead: more frequent creation of recovery points increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The interval between recovery from consecutive errors/failures must be less than the interval to create a stable recovery point from the present state of the system to enable system to make forward progress.

Solution: The solution suggested by the `Roll-back` pattern is based on the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system. Upon detection of an error or a failure, the checkpoints are used to recreate last known error/failure-free state of the system. Based on a temporal view of the system's progress, the error/failure recovery is backward, i.e., the restored system state is a previous correct state of the system.

Capability: The `Roll-back` pattern enables the system to recovery and resume operation from the point of occurrence of an error or a failure. However, the recovery of the system state typically leverages previously captured checkpointed state and repeats operations from the last stable known system state.

Protection Domain: The protection domain for a `Roll-back` pattern is determined by the extent of state that is captured during checkpoint operation, which accurately representatives the complete execution of the system. The broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event.

Resulting Context: The time overhead introduced by the application of the `Roll-back` pattern during error-free operation depends on the frequency of taking checkpoints. For recovery, the amount of work lost also correlates with the frequency of the checkpoint operations. The worst-case scenario for recovery using this pattern is rolling-back to the start-up of the system.

Rationale: The solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low.

Examples: The classic rollback recovery approach is implemented using various application-level or system-level checkpoint and restart frameworks.

Related Patterns: The roll-forward pattern is closely related to the roll-back pattern. The key difference between the two patterns is the temporal relation between the recovered state and the error/failure state. In the roll-back pattern the recovered state is based on a previous stable version of the system state.

Known Uses:

- System-level checkpoint used in production HPC systems uses BLCR. Recent research in this domain focuses on the integration of incremental checkpointing in BLCR

7.4.7 Roll-forward Pattern

Pattern Name: Roll-forward Pattern

Problem: The `Roll-forward` pattern solves the problem of errors in the system, or failure of the system leading to catastrophic failure, which results in fail-stop behavior.

Context: The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The system must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected; the pattern offers no implicit error/failure detection.

- The error or failure that the pattern handles must be transient, i.e., the error/failure must not repeatedly occur post-recovery
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.
- The time to recover from an error or a failure must be minimized.

Forces:

- The pattern requires stable storage to capture system state, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The amount of state captured during each creation of a recovery point incurs space and time overheads.
- The frequency of creation of system state snapshots determines overhead: more frequent creation of recovery points increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The interval between recovery from consecutive errors/failures must be less than the interval to create a stable recovery point from the present state of the system to enable system to make forward progress.

Solution: The solution suggested by the `Roll-forward` pattern is based on the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system. Upon detection of an error or a failure, the checkpoints are used to create a new correct error/failure-free state of the system, which enables the system to move forward.

Capability: The pattern enables forward recovery, which entails steps that may involve access to checkpoints from previous stable checkpoints or external state information to recover from the impact of the error/failure. However, based on a temporal view of the system's progress, the error/failure recovery is forward, i.e., the restored system state enables forward progress from the point of occurrence of the error/failure in the system.

Protection Domain: The protection domain for a `Roll-forward` pattern is determined by the extent of state that is captured during checkpoint operation, which accurately representatives the complete execution of the system. The broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event.

Resulting Context: The `Roll-forward` pattern enables the system to recovery and resume operation from the point of occurrence of an error or a failure. However, the recovery of the system state may leverage previously captured checkpointed state but does not require the system to repeat operations from the last stable checkpoint.

Rationale: The solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low.

Examples: Applications that contain some form of algorithmic fault tolerance are capable of forward recovery provided an error keeps the application process alive. The application is able to compensate for the presence of an error and resume operation without the need to roll-back execution.

Related Patterns: The roll-back pattern is closely related to the roll-forward pattern. The key difference between the two patterns is the temporal relation between the recovered state and the error/failure state. In the roll-forward pattern the recovered state is based on a stable version of the system state that represents forward progress from the point of error/failure occurrence.

Known Uses:

- The fault-tolerant MPI (FT-MPI) library supports roll-forward recovery of MPI applications.
- The GVR system versioning of these distributed arrays for resilience through which application-specified forward error recovery is made possible.

7.4.8 N-modular Redundancy Pattern

Pattern Name: N-modular Redundancy Pattern

Problem: The `N-modular Redundancy` pattern solves the problem of detecting and correcting errors or failures in the system state.

Context: The pattern applies to a system that has the following characteristics:

- The system must be deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs.
- The error/failure-free overhead penalty introduced by the replication of state must be minimized.

Solution: The `N-modular redundancy` pattern creates a group of N replicas of a system's state. The redundancy may include replication of the system's operation and/or the inputs to the system. Each of the N copies of the system state exist simultaneously. The redundant state versions of the systems are provided with the identical inputs, and their respective outputs are compared in order to detect and potentially correct the impact of an error or a failure in either replica of the systems.

Capability: The availability of replicated versions of the system state enable the following capabilities:

- Fail-over, which entails substitution of a replica in error or failed state with another replica that is error/failure-free.
- Comparison, which entails observing the likeness of each replica's outputs as means to detect the presence of an error or failure in either replica.
- Majority voting on the outputs produced by each replica system enables the detection of errors and failures, and filtering out the outputs that fall outside the majority.

Protection Domain: The protection domain of the pattern extends to the scope of the system state that is replicated.

Resulting Context: The design effort and complexity of replication of the system state requires low design effort since the replication entails creation of identical copies of the system state.

Rationale: The `N-modular redundancy` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system state to substitute a failed system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Examples:

- Dual-modular redundancy for error detection; triple-modular redundancy for error detection and correction
- Redundant information for compensation of data errors

Related Patterns: The `N-modular redundancy` and the `N-version` patterns are based on inclusion of redundancy in order to compensate for errors or failures. The diversity in the `N-modular` pattern stems from the replication of the system's state unlike the `N-version` pattern, which uses independently implemented versions of the system's design to tolerate errors or failures.

Known Uses:

- Implementations of the MPI interface, such as rMPI, MR-MPI, RedMPI support various forms of n-modular redundancy through replication of processes, the messages between MPI processes
- Charm++ prototypes also offer process-level replication
- Production HPC systems also contain built-in n-modular redundancy for critical components, such as power supply modules, fans, etc.

7.4.9 N-version Design Pattern

Pattern Name: N-version Design Pattern

Problem: The `N-version Design` pattern solves the problem of detecting and correcting errors or failures in the behavior of the system that may occur due design faults in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has a well-defined specification for which multiple implementation variants may be designed.
- There is an implicit assumption of independence of between multiple variants of the implementation.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern requires distinct implementations of the same design specification, which are created by different individuals or teams.
- The pattern increases the system complexity due to the need additional design and verification effort required to create multiple implementations.
- The error/failure-free overhead penalty due to disparity in the implementation variants must be minimized.

Solution: The `N-version design` pattern enables dealing with errors or failures due to Bohrbugs, although Heisenbugs may also be treated using this design pattern. The pattern entails creations of N independent versions of the system that are functionally identical, but designed independently. A majority voting logic is used to compare the results produced by each design version.

Capability: In this pattern, each of the N (N >= 2) versions of the designs are independently implemented, but the versions are functionally equivalent systems. The versions are operated independently and the critical aspects of the system state are compared in order to detect and correct errors/failures due to Bohrbugs or Heisenbugs.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which multiple implementation variants are created.

Resulting Context: The extent to which each of the n design versions are different affects the ability of the pattern to tolerate errors/failures in the system. The use of the n-version design pattern requires significant design overhead in implementing and testing independent versions of a specification. Differences in the design may cause differences in timing in generating output values for comparison and majority voting - these differences incur overhead to the overall system operation.

Rationale: The intent behind applying this pattern is to eliminate the impact of human error during the implementation of a system. Due the low likelihood that different individuals or teams introduce identical bugs in their respective implementations, the pattern enables compensating for errors or failures caused by a bug in any one implementation of the same design.

Examples: Various versions of the same software are used for the detection of errors due to bugs in the implementation of either version.

Related Patterns: The pattern is similar to the n-modular redundancy pattern, which entails creating replica versions of the state associated with the pattern and accounting for the presence of errors/failures through majority voting. The key difference between the patterns is the independence of design between the replica versions of the system.

Known Uses:

- The DIVA processor architecture includes an out-of-order core as well as a simple in-order pipelined core. The in-order pipeline is functionally equivalent to the primary processor core and is used to detect errors in the design of the out-of-order processor core.

7.4.10 Recovery Block Pattern

Pattern Name: Recovery Block Pattern

Recovery Block Problem: The `Recovery Block` pattern solves the problem of detecting and correcting errors or failures in the behavior of the system that may occur due design faults in the system.

Context: The pattern applies to a system that has the following characteristics:

- The system has a well-defined specification for which multiple implementation variants may be designed.
- There is an implicit assumption of independence of between multiple variants of the implementation.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern requires distinct implementations of the same design specification, which are created by different individuals or teams.
- The pattern increases the system complexity due to the need additional design and verification effort required to create multiple implementations.
- The error/failure-free overhead penalty due to disparity in the implementation variants must be minimized.

Solution: The `Recovery block` pattern is a flavor of the N-version design pattern in which a recovery block is invoked when the result from the primary version of the system fails an acceptance test. The recovery block is another implementation version of the same design specification based on which the primary system is implemented.

Capability: With the use of the `Recovery block` pattern, the system is composed of functional blocks. Each block contains at least a primary design and exceptional case handler along with an adjudicator. If the adjudicator does not accept the results of the primary system, it invokes the exception handler subsystem. This indicates that the primary system could not perform the requested service operation. An acceptance test is used to test the validity of the result produced by the primary version. If the result from the primary version passes the acceptance test, this result is reported and execution stops. If, on the other hand, the result from the primary version fails the acceptance test, another version from among the multiple versions is invoked and the result produced is checked by the acceptance test.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which the recovery block implementation variant is created.

Resulting Context: The extent which the primary design and recovery block versions of the system specification are different affects the ability of the pattern to tolerate errors/failures in the system. The

use of the `Recovery block` design pattern requires significant design overhead in implementing and testing independent versions of a specification. Differences in the design may cause differences in timing in generating output values for comparison and majority voting - these differences incur overhead to the overall system operation.

Rationale: This pattern relies on multiple variants of a design which are functionally equivalent but designed independently. The secondary recovery block design is used to perform recovery, if the system implementation of the primary design produces an output that suggests the presence of an error/failure of the primary system. This determination is made by the adjudicator sub-system.

Examples: Various application-based fault tolerance methods include verification routines that check for the validity of a computation.

Related Patterns: The significant differences in the recovery block approach from N-version programming are that only one version is executed at a time and the acceptability of results is decided by an adjudicator test rather than by majority voting.

Known Uses:

- Containment Domains provide recovery blocks in order to recover from errors in the computation included within the domain in order to generate correct output values.
- The SwiFT library provides language based implementation of the recovery block for use in C language programs.

7.5 State Patterns

7.5.1 Persistent State Pattern

Pattern Name: Persistent State Pattern

Problem: The `Persistent State Pattern` solves the problem of the separating the part of the system state that remains unchanged for the entire duration of system operation.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state.
- The system state may be described in terms of state that remains invariant for the duration of the system operation and state that changes during system operation.

Forces:

- The state patterns present an application-centric view of systems. The precise definition of aspects of the system state are invariant and those that change depends on the layer of system abstraction. At the application-level, the separation of this state is straightforward. However, in the hardware and system software layers, distinguishing between these types of state is non-trivial.

Solution: The persistent state refers to all aspects of a system's state that is computed when the system is initialized, but is not modified during the system operation. From the perspective of an HPC application, the persistent state includes program instructions and variable state that is computed upon application initialization.

Capability: The correctness of the persistent state is essential to correct execution of a program. The presence of any errors in the persistent state may not necessarily lead to immediate catastrophic failure of an application program's execution, but might lead the program on divergent paths that cause a failure at a future point in the system's operation.

Protection Domain: The `Persistent State` pattern defines the scope of the application program state that is computed during initialization.

Resulting Context: The persistent state pattern defines the scope of the static program state. Such scope forms the protection domain for a resilience behavioral pattern.

Examples: Various algorithm-based fault tolerance methods leverage the property of invariance in the persistent state. These methods maintain redundant information about the application variables in the static state that enables recovery to their default data values at any time during application execution.

Rationale: The isolation of the state that is persistent throughout an application program execution is supported by this pattern. The state invariance feature provided by this pattern enables the use of resilience behavioral patterns that leverage this property to detect and recovery errors/failure of such state.

Related Patterns: Together with the `Dynamic State` pattern and `Environment State` pattern, the `Persistent State` pattern defines the overall state of a system.

7.5.2 Dynamic State Pattern

Pattern Name: Dynamic State Pattern

Problem: The `Dynamic State Pattern` solves the problem of the encapsulating the part of the system state that changes during system operation.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state.
- The system state may be described in terms of state that remains invariant for the duration of the system operation and state that changes during system operation.

Forces:

- The state patterns present an application-centric view of systems. The precise separation of aspects of the system state that will be invariant and those that change depends on the layer of system abstraction. At the application-level, the separation of this state is straightforward. However, in the hardware and system software layers, distinguishing between these types of state is non-trivial.

Solution: The `Dynamic State Pattern` encapsulates the system state that changes as the system makes forward progress.

Capability: The state refers to all aspects of the program state that continuously changes as an application program executes. This includes the data values that are computed during system operation, or those that enable forward progress of the system (control-flow variables).

Resulting Context: For the perspective of an HPC program, the encapsulation of the dynamic state for the purpose of defining its resilience behavior enables correct operation and forward progress of the system.

Examples: Algorithm-based fault tolerance strategies that guarantee resilience of the dynamic state actively track changes to state. Redundancy methods maintain copies of the change to the dynamic state in order to recover the version that is impacted by an error or failure.

Rationale/Capability: The isolation of the dynamic state that is updated throughout an application program execution is supported by this pattern. The *dynamic* feature of this state pattern implies that any errors/failure in such state amounts to lost work. However, this isolation of dynamic state enables the use of resilience behavioral patterns that leverage this property to recover the error/failure without the need to abort and restart an application program.

Related Patterns: Together with the `Dynamic State` pattern and `Environment State` pattern, the `Persistent State` pattern defines the overall state of a system.

7.5.3 Environment State Pattern

Pattern Name: Environment State Pattern

Problem: The `Environment State Pattern` solves the problem of encapsulating the system state that supports the operation of the system.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state.
- The system state may be described in terms of state relevant to the core function of the system, called the *primary state* and the system state that supports its function, called the *ecosystem*.

Forces:

- The state patterns present an application-centric view of systems. The aspects of the system state that serves the primary function of the system and those that form the ecosystem is easier due to the convention of defining system layers of abstractions and the availability of well-defined interfaces between the layers. These interfaces enable designers to distinguish between the system's primary state and the environment state.

Solution: The `Environment State` pattern defines state that a system relies on to access the system resources and services that enable the system to fulfill its function. The environment also facilitates and coordinates the operation of various sub-systems.

Capability: An error/failure in the environment state is often immediately catastrophic to the operation of the primary system. The encapsulation of the environment state provided by this pattern enables the development of separate resilience strategies for the ecosystem.

Protection Domain: The `Environment State` pattern defines the scope of the components in the ecosystem that support the operation of the primary system. For an HPC program, this scope includes productivity tools and libraries, a runtime system, the operating system, file systems, communication channels, etc.

Resulting Context: The system does not normally have complete control over its environment, but may have partial control to affect the environment through well-defined interfaces. Any changes to the environment typically affect the system operating within the environment directly. The encapsulation of the environment enables the resilience behavior of the environment state to be reasoned about separately from the resilience behavior of the primary system state.

Examples: Operating-system based resilience mechanisms focus on the correctness of the data structure state within the kernel without concern for the resilience features of the application program.

Rationale: The encapsulation of the supporting sub-systems from the primary system state enables the application of specific behavioral resilience patterns that are general-purpose, which do not rely on any feature of the primary system.

Related Patterns: Together with the `Persistent State` pattern and `Dynamic State` pattern, the `Environment State` pattern defines the overall state of a system.

7.5.4 Stateless Pattern

Pattern Name: Stateless Pattern

Problem: The `Stateless Pattern` solves the problem of defining resilience strategies that are independent of state.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state.
- The behavior and the progress of the system is does not depend on specific parts of the system state.

Forces:

- The state patterns present an application-centric view of systems. The precise separation of aspects of the system state that will be invariant and those that change depends on the layer of system abstraction. At the application-level, the separation of this state is straightforward. However, in the hardware and system software layers, distinguishing between these types of state is non-trivial.

Solution: The `Stateless` pattern provides the notion of the *null* state in order to define resilience solutions that are independent of system state.

Resulting Context:

- The stateless pattern is utilized together with behavioral resilience patterns whose actions do not necessitate operating on any aspect of the application program state.
- The behavioral pattern used to build a resilience solution using the stateless pattern must be capable to deal with any additional side-effects in the system.

Examples: The use of the *transaction* model to provide resilient behavior is an example of the `Stateless` pattern. Transactions support execution of a sequence of operations that may complete as a unit, or fail; the notion of partial execution is not supported. While the transaction may entail performing computation on data variables, the resilience of the data is independently managed; the resilience solution may be defined with a `Stateless` pattern.

Rationale/Capability: The pattern is the equivalent of a *null* pattern that enables resilience solutions to be constructed without the requirement for the behavioral patterns to operate on the program state.

Related Patterns: While the `Persistent State` pattern `Dynamic State` pattern, and the `Environment State` pattern defines the complement of the overall state of a system, the `Stateless` pattern offers the notion of `null` state.

8 Building Resilience Solutions using Resilience Design Patterns

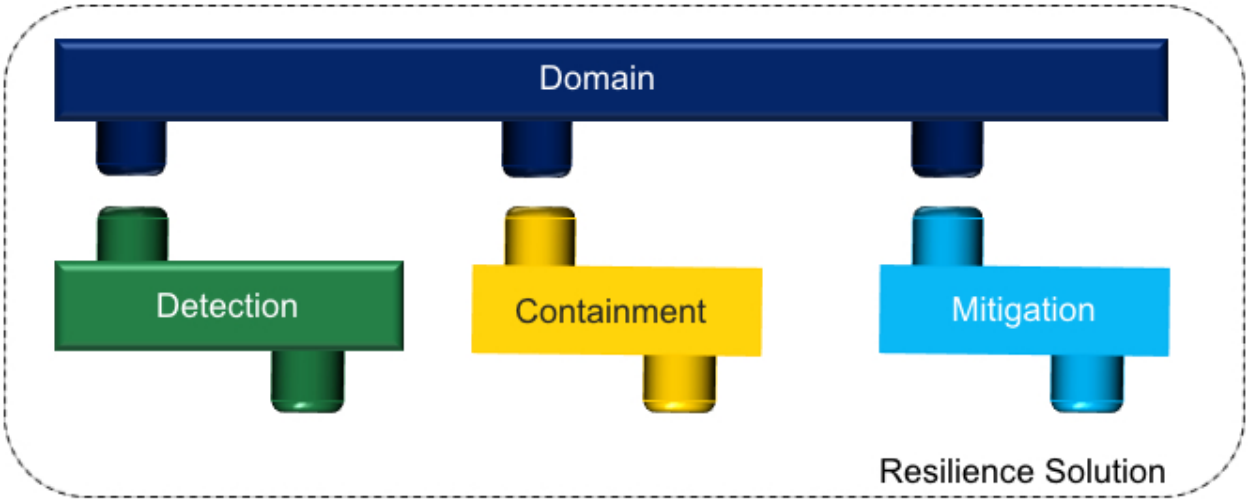

Figure 4: Elements of a resilience solution for HPC systems and applications

8.1 Features of Resilience Solutions

The resilience design patterns presented in the catalog offer solutions to problems that repeatedly occur in the design of resilience capabilities for HPC. Each pattern in the catalog presents a solution to a specific problem in detecting, recovering from, masking an error, or the scope of system state that is of interest to the resilience solution. These key constituents of a complete solutions are shown in Figure 4.

The artifacts of a design process that uses the resilience design patterns are complete resilience solutions that provide fault/error/failure detection, containment and mitigation capabilities for a specific fault model. These solutions may be instantiated at multiple layers of system abstraction, and are relevant to various application and system scales. However, many of the patterns in the catalog individually provide partial solutions by supporting only one or two out of the detection, containment and mitigation solutions. For system and application designers to use these patterns in the construction of resilient versions of their designs, these patterns must be organized into a well-defined system of patterns.

A pattern framework enables the creation of the outline of the resilience solution that captures the dimensions and capabilities of the patterns, reveals and clarifies the relationships between the patterns. The combination of these patterns based on the guidelines offered by the hierarchical classification scheme enables the complete solutions for resilience to specific fault models in HPC systems. However, there is sufficient flexibility to adapt the solution to specific situations.

8.2 Design Spaces

We define a framework that enables the composition of the resilience design patterns into practical solutions.

In order to articulate a systematic method for customized designs, the framework is based on *design spaces* (Figure 5). These design spaces provide guidelines for the decision making in the design process, which

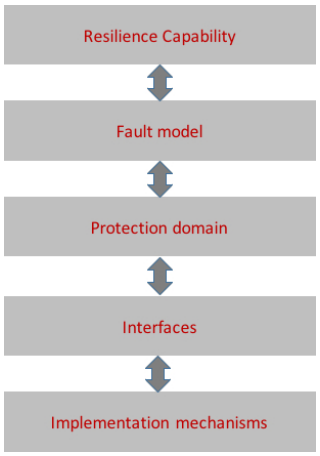

Figure 5: Design Spaces for construction of resilience solutions using patterns

consists of selection of the appropriate patterns based on the requirements of protection and the cost of using specific patterns.

- **Capability**: The patterns must support capabilities that enable the detection, containment, mitigation of faults/errors/failure events.
- **Fault model**: The identification of the root causes of fault events and their impact and propagation through the system must be well-understood to provide effective solutions.
- **Protection domain**: The definition of the protection domain enables clear encapsulation of the system scope over which the resilience patterns operate.
- **Interfaces**: The identification and implementation of the activation and response interfaces for behavioral patterns affect the propagation of fault/error/failure event information.
- **Implementation mechanisms**: The implementation design space is concerned with constraints imposed by specific features of hardware, execution or programming models, software ecosystems.

The structured design process enabled by these design spaces supports various approaches to create resilience solutions, including (i) a *top-down approach*; (ii) a *bottom-up approach*, as well as (iii) various *hybrid* approaches that enable designers to create solutions in the presence of practical constraints imposed by any hardware or software system features. Design spaces also provide a framework to guide the creation of cross-layered resilience solutions that leverage capabilities from multiple layers of the system abstraction. With the use of resilience patterns in the context of the framework provided by the design spaces, HPC system designers, users and application developers may evaluate the feasibility and effectiveness of novel resilience techniques, as well as analyze and evaluate existing solutions.

9 Case Study: Checkpoint and Rollback

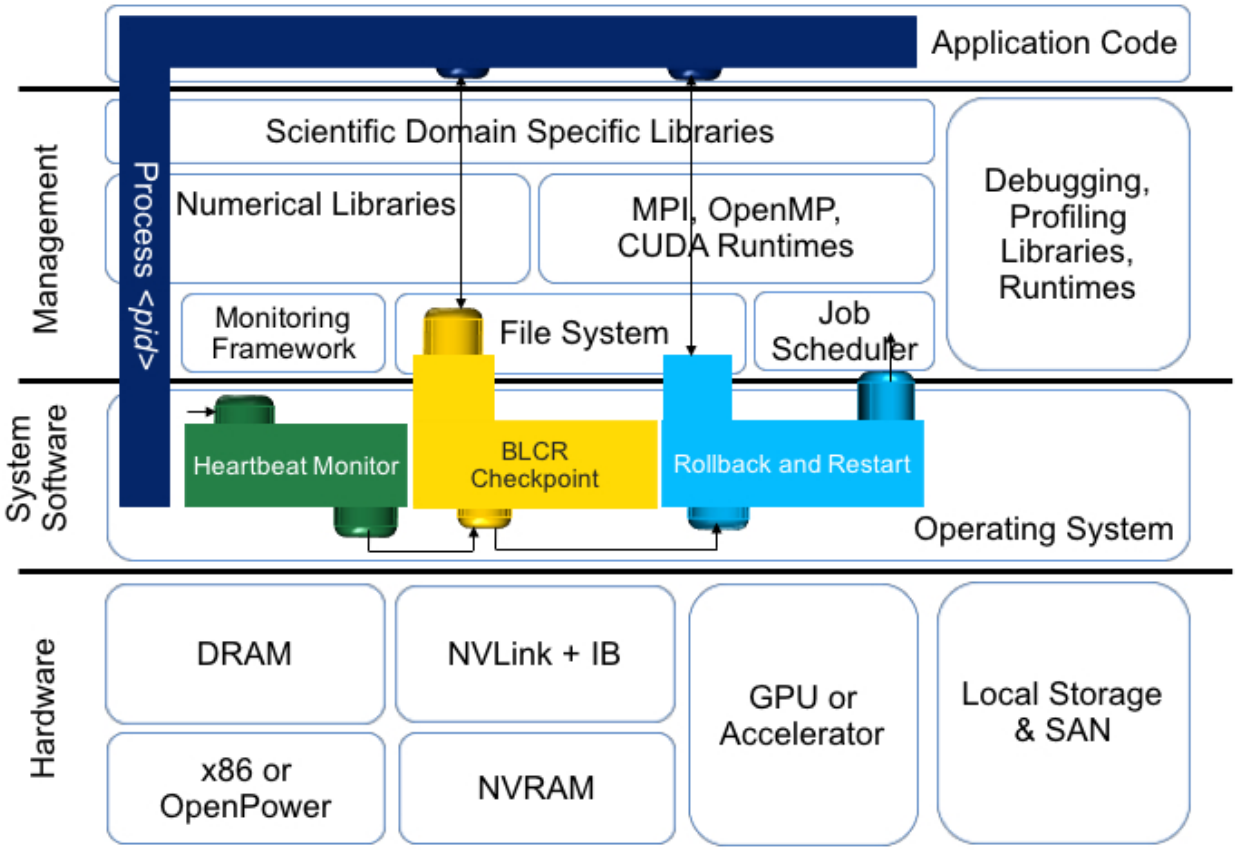

Figure 6: Resilience Solution Case Study: Checkpoint & Restart using BLCR

Checkpoint and restart (C/R) solutions are the most widely-used resilience solution in HPC systems. These solutions capture the image, or snapshot, of a running process and preserve it for later recovery. For parallel applications, the C/R framework's coordination protocols produce a global snapshot of the application by combining the state of all the processes in the parallel application. The checkpoint is typically committed to parallel file system on disk. Since C/R is a well-understood resilience strategy used in production HPC systems, the goal of this case study is to breakdown this solution and cast it within the resilience design patterns-based framework. The reexamination of this well-known resilience solution demonstrates the utility of the pattern-based framework in understanding the protection domain, capabilities, as well as the limitations of the C/R solution.

For this case study, the resilience solution, which is illustrated in Figure 6, is built using the BLCR (Berkley Lab's Checkpoint/Restart) [28] framework for a single process. In order to deconstruct this solution based on the structured pattern-based approach, we navigate the design spaces to methodically construct the solution. we focus on developing a complete resilience solution that enables systems to contend with single process failure. Since the fault model that our solution addresses is process failure, it is not necessary to identify the root cause of the fault and error that cause this failure.

We identify patterns for:

- ***Detection:*** For the detection of a process failure, we require an instantiation of the `Fault Treatment` pattern. Specifically, the solution requires `Fault Diagnosis` pattern to discover the location of the failure and the type of event, which is enabled by a `Monitoring` pattern. The instantiation of the `Monitoring` pattern is a kernel-level heartbeat monitor, which is deployed in the system to detects whether the process is alive.

- ***Containment:*** The BLCR framework provides containment by recovering a failed process from the last known stable process state from disk, which prevents the propagation of the failure.

- ***Recovery:*** BLCR also manages the recovery of the failure by instantiating the `Recovery` pattern, specifically the `Roll-back` structure pattern, whose architecture is framed using the `Checkpoint-Recovery` pattern.

- ***Domain:*** Since the solution aims to provide resilience capabilities for the complete process, the solution fuses the `Persistent` and `Dynamic` state patterns. Therefore, the protection domain associated with the system-level checkpointing solution extends to the entire memory associated with a process.

BLCR provides a completely transparent checkpoint of the process, which saves the current state of a process. The framework uses a coarse-grain locking mechanism to interrupt momentarily the execution of all the threads of the process, giving them a global view of its current state, and reducing the problem of saving the process state to a sequential problem. Since the entire state is saved (from CPU registers to the virtual memory map), the function call stack is saved. From the perspective of an application programmer, the checkpoint routine returns with a different error code, to let the caller know if this calls returns from a successful checkpoint or from a successful restart. The recovery after the detection of a process failure by the instantiation of the `Monitoring` pattern (the heartbeart monitor), entails restoring the checkpoint from the parallel file system on the same hardware, with the same software environment.

10 Case Study: Proactive Process Migration

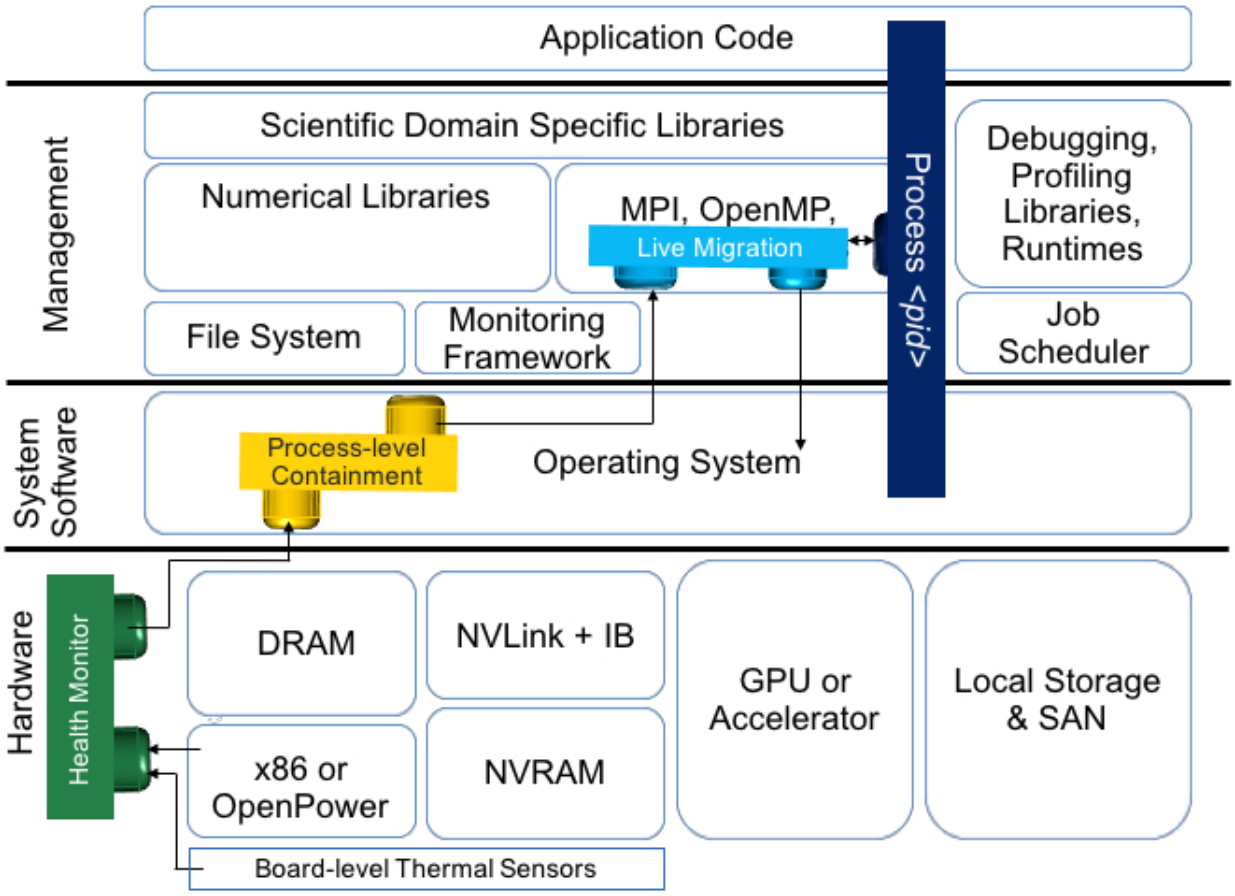

Figure 7: Resilience Solution Case Study: Process Migration

Various resilience strategies are inherently reactive, i.e., they respond to the occurrence of a fault, error or failure event and seek to prevent the event from affecting the correct execution of an HPC application. In this case study, we evaluate a proactive resilience solution using the resilience design pattern framework that enables a system to anticipate failures and provides failure avoidance capability through a process-level live migration mechanism [85]. The solution uses a combination of hardware and system software to handle resiliency in a manner that is transparent to the application developer. Also, the solution requires no changes in the application codes.

For the systematic study of the construction of the complete resilience solution based on the structured pattern-based approach, we address the specific design constraints that are imposed within each of the design spaces. For this case study, our emphasis is on the development of a complete resilience solution that avoids single process failure (Figure 7). We do not concern ourselves with the root cause of the fault and error that may cause the failure.

We identify patterns for:

- ***Detection:*** In order to proactively anticipate the occurrence of a failure, the solution must observe critical indicators that will predict the likelihood of a failure. We require the `Fault Treatment` strategy

pattern, which must be instantiated as a `Fault Diagnosis` pattern within the system architecture. Since the detection entails anticipation of a future failure event, the structure pattern selected is the `Prediction` pattern, whose implementation requires reading board-level thermal sensors for health monitoring for each of the processors on the compute node.

- ***Containment:*** A kernel level module provides containment for the fault by identifying the process that is executing on the CPU for which the `Prediction` pattern has assessed as vulnerable to a failure.
- ***Recovery:*** The live migration is a kernel level module that is integrated with the MPI execution environment to support parallel applications. The `Recovery` strategy pattern is used by this solution. The architecture of the system must instantiate the `Reconfiguration` pattern and specifically the `Restructure` structural pattern in order to isolate the processor on which failure is predicted by the `Prediction` pattern and move the process to an alternative processor.
- ***Domain:*** Since the solution aims to provide resilience capabilities for the complete process, the solution fuses the `Persistent` and `Dynamic` state patterns. Therefore, the protection domain associated with the system-level checkpointing solution extends to the entire memory associated with a process.

The `Prediction` pattern instantiation uses the standardized Intelligent Platform Management Interface (IPMI) which provides message-based interface to collect sensors readings for health monitoring, including the data on temperature, fan speed, and voltage. The instantiation of the `Restructure` pattern gathers the sensor data, and when the sensor reading exceeds a threshold value, the scheduler determines the availability of new destination nodes to complete the restructure of the job. If no spare nodes are available, the scheduler selects a compute node with lowest utilization. With the use of these pattern instantiations, this solution supports migration of a process as a precaution to potentially imminent failure by monitoring the health of each node.

11 Case Study: Cross-Layer Hardware/Software Hybrid Solution

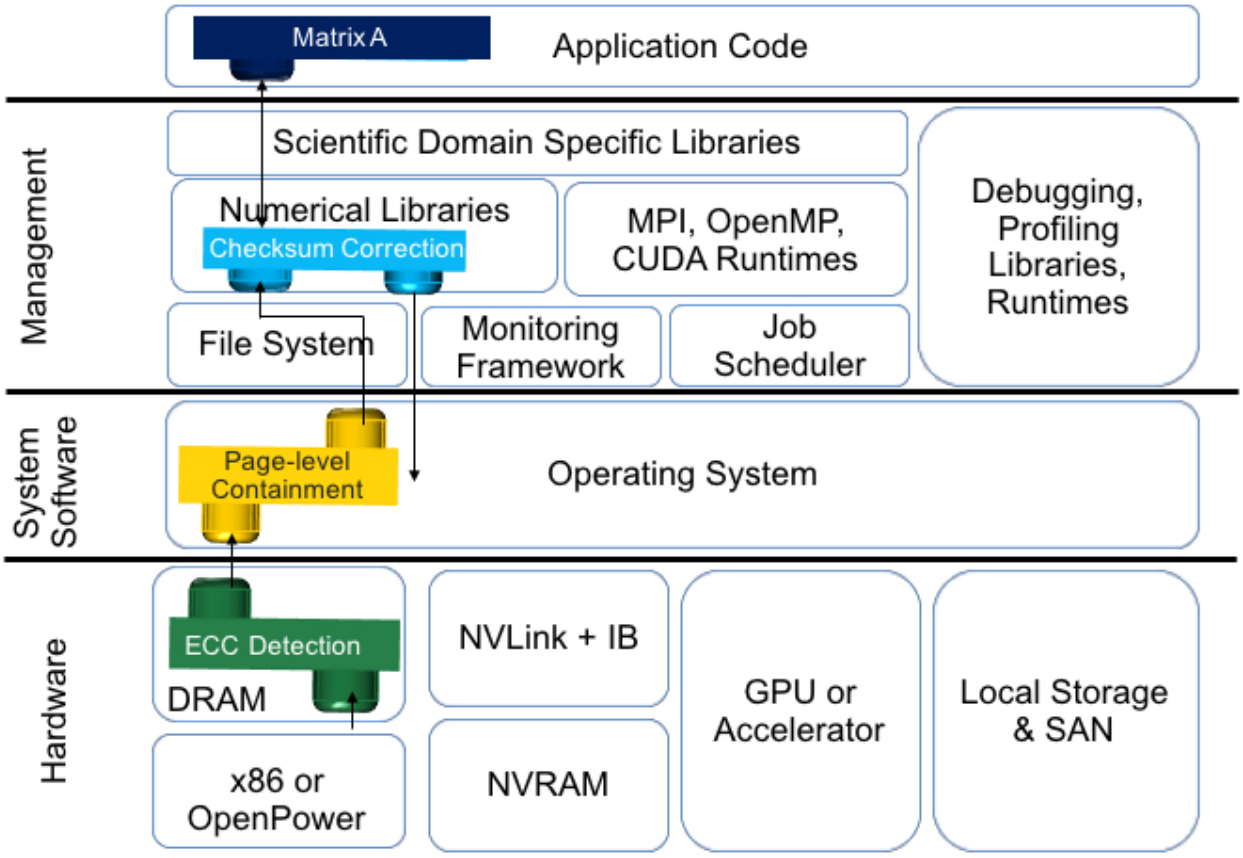

Figure 8: Resilience Solution Case Study: Cross-Layer Design using ECC with ABFT

The pattern-based structured approach enables the design of resilience solutions that combine techniques across various layers of the system stack, which is referred to as cross-layer resilience. The aim of this case study is to use the framework of resilience design patterns to systematically explore various techniques available at multiple layers of the system stack and design a cross-layer combination that supports targeted, cost-effective resilience capabilities for a specific data structure within an application. The case study demonstrates practicality and effectiveness of our framework in developing novel resilience solutions. The proposed solution (Figure 8) is intended to provide resilience capabilities for a matrix data structure in an application that uses a numerical method. The solution is designed to protect the matrix structure A from the impact of multi-bit corruptions due to errors in the DRAM memory.

We identify patterns for:

- ***Detection:*** Since the memory is protected by error correcting codes (ECC), the solution leverages the hardware-based ECC. This is an instantiation of the `Compensation` pattern, and specifically the `State Diversity` pattern that supports the single error correction and double error detection capability.
- ***Containment:*** The presence of double-bit error corruptions is detected by the `State Diversity` pattern and is communicated to the operating system via an interrupt mechanism. The containment

is support by a module in the OS kernel that maps the physical address to the application address space, and notifies the library that provides the recovery pattern instantiation.

- ***Recovery:*** The matrix A is protected by an algorithm-based fault tolerance (ABFT) method, specifically a checksum routine. This routine is an instantiation `Compensation` pattern, and specifically the `State Diversity` that is implemented within the numerical library.

- ***Domain:*** The solution is designed in order to guarantee the resilience of the data structure A, which as an operand matrix is an instantiation of the `Persistent` state pattern, since it is initialized during the initialization of the numerical method and does not change until the application converges.

12 Summary

The key to addressing the resilience challenge for future extreme-scale HPC systems make the design of resilience techniques an essential part of the system architecture and software development efforts. Towards the development of a systematic methodology for designing resilience solutions, design patterns provide reusable templates that may be used to build and refine resilience solutions. In this document we presented a set of design patterns that provide solutions to problems specific to the management of resilience in HPC systems. We identified and presented solutions that support detection, containment, recovery and masking in a structured design pattern format. We developed a classification scheme to enable designers to understand the capabilities of each pattern and the relationship between the various patterns in the pattern catalog. We developed a design framework to enable the composition and refinement of resilience solutions using the design patterns. The resilience design patterns and the design framework offer a systematic way to investigate the effectiveness and efficiency of a resilience solution. They also provide a structured approach for optimizing the trade-off, at design time or runtime, between the key system design factors: performance, resilience, and power consumption.

References

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual International Conference on Supercomputing*, ICS '04, pages 277–286, 2004.

[2] A.M. Agbaria and R. Friedman. Starfish: Fault-tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of The Eighth International Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

[3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, August 1977.

[4] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable systems on unreliable fabrics. *IEEE Design Test of Computers*, 25(4):322–332, 2008.

[5] Algirdas Avižienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, April 1997.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, pages 11–33, January 2004.

[7] D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, 2005.

[8] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[9] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[10] 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 *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, 2002.

[11] Patrick G. Bridges, Mark Hoemmen, Kurt B. Ferreira, Michael A. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/os dram fault recovery. In *4th Workshop on Resiliency in High Performance Computing in Clusters, Clouds, and Grids*, Bordeaux, France, September 2011.

[12] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, pages 155–164, 2008.

[13] Jeremy Casas, Dan Clark, Phil Galbiati, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MIST: PVM with Transparent Migration and Checkpointing. In *In 3rd Annual PVM Users' Group Meeting*, 1995.

[14] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 73–84, 2011.

[15] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan 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*, pages 58:1–58:11, 2012.

[16] John Daly, Bill Harrod, Thuc Hoang, Lucy Nowell, Bob Adolf, Shekhar Borkar, Nathan DeBardeleben, Mootaz Elnozahy, Mike Heroux, David Rogers, Rob Ross, Vivek Sarkar, Martin Schulz, Mark Snir, Paul Woodward, Rob Aulwes, Marti Bancroft, Greg Bronevetsky, Bill Carlson, Al Geist, Mary Hall, Jeff Hollingsworth, Bob Lucas, Andrew Lumsdaine, Tina Macaluso, Dan Quinlan, Sonia Sachs, John Shalf, Tom Smith, Jon Stearley, Bert Still, and John Wu. Inter-agency workshop on hpc resilience at extreme scale. February 2012.

[17] John T. Daly, Lori A. Pritchett-Sheats, and Sarah E. Michalak. Application mttfe vs. platform mttf: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2008: Workshop on Resiliency in High Performance Computing (Resilience)*. IEEE Computer Society, May 2008.

[18] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 162–171, 2011.

[19] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 497–508, 2010.

[20] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 475–486, 2012.

[21] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the hec community and path-forward for research and development. *Whitepaper*, December 2009.

[22] Timothy J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Technical report, IBM Microelectronics Division, November 1997.

[23] T.J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Technical report, IBM Microelectronics Division Whitepaper, November 1997.

[24] J. Dinan, A. Singri, P. Sadayappan, and S. Krishnamoorthy. Selective recovery from failures in a task parallel programming model. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 709–714, 2010.

[25] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara

Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The International Exascale Software Project Roadmap. *International Journal on High Performance Computing Applications*, pages 3–60, February 2011.

[26] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, February 2010.

[27] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 225–234, 2012.

[28] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Lawrence Berkeley National Lab (LBNL), December 2002.

[29] E.N.Elnozahy, Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godfrey, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[30] Christian Engelmann. *Symmetric Active/Active High Availability for High-Performance Computing System Services*. PhD thesis, Department of Computer Science, University of Reading, UK, 2008. Thesis research performed at Oak Ridge National Laboratory. Advisor: Prof. Vassil N. Alexandrov (University of Reading).

[31] Christian Engelmann and Swen 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)*, pages 15–17, 2011.

[32] Irene Eusgeld, Felix C. Freiling, and Ralf Reussner, editors. *Dependability Metrics: Advanced Lectures [Result from a Dagstuhl seminar, October 30 - November 1, 2005]*, volume 4909 of *Lecture Notes in Computer Science*. Springer, 2008.

[33] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, 2000.

[34] Kurt Ferreira, Jon Stearley, James H. Laros, III, and et al. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[35] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12. IEEE Computer Society Press, 2012.

[36] Hajime Fujita, Robert Schreiber, and Andrew A. Chien. It's time for new programming models for unreliable hardware, provocative ideas session. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[38] D. Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2010.

[39] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518 –528, june 1984.

[40] Saurabh Hukerikar and Robert F. Lucas. Rolex: Resilience-oriented language extensions for extreme-scale systems. *The Journal of Supercomputing*, pages 1–33, 2016.

[41] Saurabh Hukerikar, Keita Teranishi, Pedro C. Diniz, and Robert F. Lucas. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming*, pages 1–27, 2016.

[42] 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 *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2007.

[43] Cray Inc. Cray xc40 computing platform, 2014.

[44] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf 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, 2012.

[45] J.-Y. Jou and J.A. Abraham. Fault-tolerant fft networks. *IEEE Transactions on Computers*, 37(5):548–561, 1988.

[46] Jing-Yang Jou and Jacob A. Abraham. Fault-tolerant matrix operations on multiple processor systems using weighted checksums. pages 94–101, 1984.

[47] K. Keutzer and T. Mattson. Our pattern language (opl): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, 2009.

[48] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dallya, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale systems. Technical report, DARPA, September 2008.

[49] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, Burlington, MA, USA, July 2007.

[50] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM Journal Scientific Computing*, 30:102–116, November 2007.

[51] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical report, 1993.

[52] Wei-Jih Li and Jyh-Jong Tsay. Checkpointing Message-Passing Interface (MPI) Parallel Programs. In *Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 147–152, 1997.

[53] J. Lidman, D.J. Quinlan, C. Liao, and S.A McKee. ROSE::FTTransform - a Source-to-Source Translation Framework for Exascale Fault-tolerance Research. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.

[54] Michael Litzkow and Miron Livny. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, CA, January 1992.

[55] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[56] Dennis McEvoy. The architecture of tandem's nonstop system. In *Proceedings of the ACM '81 conference*, New York, NY, USA, 1981. ACM.

[57] A. Mishra and P. Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Transactions on Computers*, 52(9):1089–1099, 2003.

[58] S. Mitra, K. Brelsford, and P. N. Sanda. Cross-layer resilience challenges: Metrics and optimization. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1029–1034, March 2010.

[59] Kathryn Mohror, Adam Moody, Greg Bronevetsky, and Bronis R. de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, 99:1, 2013.

[60] Todd K Moon. Error correction coding: Mathematical methods and algorithms. 2005.

[61] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 23–32. ACM, 2007.

[62] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in srams. In *34th European Solid-State Circuits Conference*, pages 222–225, 2008.

[63] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-scalar Processors. *IEEE Transactions on Reliability*, pages 63–75, 2002.

[64] Adam J. Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 14–23, 2006.

[65] Hoang Pham. *System Software Reliability*. Springer Series in Reliability Engineering. Springer, 2007.

[66] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[67] J.S. Plank, Youngbae Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 351–360, 1995.

[68] J.S. Plank, K. Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[69] A.L.N. Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10):1304–1308, 1990.

[70] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization, 2005*, pages 243–254, 2005.

[71] J. Rexford and N.K. Jha. Algorithm-based fault tolerance for floating-point operations in massively parallel systems. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 2, pages 649–652 vol.2, 1992.

[72] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE Transactions on Computers*, 45(11):1239–1247, 1996.

[73] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *In Proceedings of LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[74] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, November 2011.

[75] 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 Transactions on Dependable and Secure Computing*, pages 135–148, 2009.

[76] T.J. Slegel, III Averill, R.M., M.A. Check, and et. al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, pages 12–23, 1999.

[77] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, 2012.

[78] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[79] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[80] G. Stellner. CoCheck: Checkpointing and Process migration for MPI. In *Proceedings of The Tenth International Parallel Processing Symposium*, pages 526–531, 1996.

[81] Nitin H. Vaidya. A case for two-level distributed recovery schemes. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.

[82] Hubertus J. J. van Dam, Abhinav Vishnu, and Wibe A. de Jong. A case for soft error detection and correction in computational chemistry. *Journal of Chemical Theory and Computation*, 9(9):3995–4005, 2013.

[83] Enrique Vargas. High availability fundamentals. *Sun Blueprints*, November 2000.

[84] Marco Vassura, Luciano Margara, Pietro Di Lena, Filippo Medri, Piero Fariselli, and Rita Casadio. Ft-comar: Fault tolerant three-dimensional structure reconstruction from protein contact maps. *Bioinformatics*, 24(10):1313–1315, 2008.

[85] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 43:1–43:12, 2008.

[86] Cheng Wang, H. Kim, Y. Wu, and V. Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization, 2007*, pages 244–258, 2007.

[87] Sying-Jyan Wang and N.K. Jha. Algorithm-based fault tolerance for fft networks. In *IEEE International Symposium on Circuits and Systems*, volume 1, pages 141–144 vol.1, 1992.

[88] S. Yajnik and N.K. Jha. Synthesis of fault tolerant architectures for molecular dynamics. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 4, pages 247–250 vol.4, 1994.

[89] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, and Mateo Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Saint Malo, France, 2010.

[90] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 87–98, 2010.