PLEXUS: A Pattern-Oriented Runtime System Architecture for Resilient Extreme-Scale High-Performance Computing Systems

Saurabh Hukerikar
Christian Engelmann
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA

***Abstract*—**
For high-performance computing (HPC) system designers and users, meeting the myriad challenges of next-generation exascale supercomputing systems requires rethinking their approach to application and system software design. Among these challenges, providing resiliency and stability to the scientific applications in the presence of high fault rates requires new approaches to software architecture and design. As HPC systems become increasingly complex, they require intricate solutions for detection and mitigation for various modes of faults and errors that occur in these large-scale systems, as well as solutions for failure recovery. These resiliency solutions often interact with and affect other system properties, including application scalability, power and energy efficiency. Therefore, resilience solutions for HPC systems must be thoughtfully engineered and deployed.

In previous work, we developed the concept of resilience design patterns, which consist of templated solutions based on well-established techniques for detection, mitigation and recovery. In this paper, we use these patterns as the foundation to propose new approaches to designing runtime systems for HPC systems. The instantiation of these patterns within a runtime system enables flexible and adaptable end-to-end resiliency solutions for HPC environments. The paper describes the architecture of the runtime system, named Plexus, and the strategies for dynamically composing and adapting pattern instances under runtime control. This runtime-based approach enables actively balancing the cost-benefit trade-off between performance overhead and protection coverage of the resilience solutions. Based on a prototype implementation of PLEXUS, we demonstrate the resiliency and performance gains achieved by the pattern-based runtime system for a parallel linear solver application.

I. Introduction

High-performance computing (HPC) systems provide the computational capabilities for driving the simulation, modeling and analysis in various areas of scientific research. Driven by the quest for greater computational performance and by rapid technological changes, the complexity of node and system architectures of modern supercomputing systems, which are massively parallel systems, has rapidly increased over the past decade. For the next generation of exascale HPC systems, ever increasing levels of parallelism and emerging heterogeneity of computational and memory resources are forcing a reevaluation of the current approaches to designing the application and system software for HPC systems. In addition to harnessing the parallelism for scalable application performance, the software stack must also contend with the challenges of resiliency, power and energy efficiency. In particular, managing the resilience of future extreme-scale systems is a complex, multidimensional challenge. Various types of faults are expected to increase significantly in high-performance computing (HPC) systems as the number of system components increases and use technologies that include smaller feature sizes, near-threshold voltage, which make the components inherently less reliable. As HPC systems approach exaflops scale, the sheer frequency of occurrence of faults and errors in these systems in addition to the scale of the systems makes the detection and mitigation of faults and recovery from failures a difficult challenge [1] [2]. The need for fault management solutions to be cognizant of the often-divergent objectives of robust computation, scalable performance and power efficiency requires careful thought when selecting and deploying resilience solutions in HPC systems.

In previous work, we developed the concept of resilience design patterns [3] to foster a structured approach to address the resilience challenge in HPC systems. The patterns are descriptions of well-established solutions that are used to address the various types of faults, errors and failure events in HPC systems. Inspired by design patterns used in object oriented programming [4] and architecture [5], the patterns are templated solutions that formalize the set of techniques used to deal with specific types of faults, errors or failure. The patterns enable the design of flexible resilience solutions through integration of multiple patterns into composite solutions. However, the resilience design patterns are described in [3] at a sufficiently high level of abstraction such that

This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

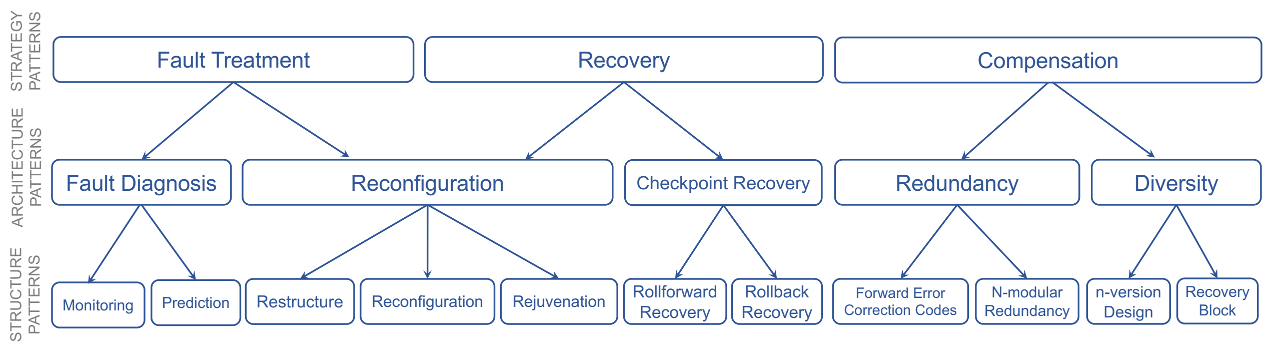


Fig. 1. Classification of Resilience Design Patterns

they are free of any implementation details. In this work, we leverage the resilience patterns to develop a new approach to designing runtime systems that emphasize resilient operation in HPC systems, called Plexus. At the heart of our approach is the instantiation of an intricate network of resilience design patterns in the runtime system to provide flexible end-to-end resilience for HPC applications. Given the key role of runtime systems play in supporting the OS in optimizing various application and system parameters, they are well-positioned to systematically manage the instances of the resilience design patterns. Plexus defines an architecture, a set of runtime components and associated mechanisms that constitute a framework for building resilient HPC software environments. The main contributions of this work are:

- We present the architecture of the Plexus runtime system, which implements pattern instances to provide a resilient environment for HPC applications while supporting existing numerical and domain specific libraries, other runtime systems and frameworks (Section III)
- We develop strategies for the resilience patterns to be instantiated, modified and destroyed by the runtime based on static and dynamic policies to meet the resiliency needs of HPC applications (Section IV)
- We present a prototype implementation and evaluate the cost and benefit of these runtime techniques with the instancing of failure detection and recovery patterns for a large-scale parallel application (Sections V and VI)

II. Resilience Design Patterns

In this section, we provide a description of the concept of resilience design patterns, the motivation behind the development of the patterns, their classification scheme and brief synopses of the patterns to enable to reader to understand the remainder of the paper. The complete specification of the resilience design patterns is detailed in [6]. These resilience patterns capture solutions that have been developed for HPC systems and evolved over time. Every HPC resilience solution consists of the following core capabilities: **(1) Detection:** Identifying the presence of an anomaly in the data or control value, or the discovery of error or failure events in a HPC system; **(2) Containment:** When an error or failure is discovered in a system, containment strategies assist in limiting the impact of the event on other components in the system; **(3) Recovery:** The elimination of the error or failure condition, or isolation and bypassing of a subcomponent with the error or failure. Solutions that provide these capabilities have been developed and optimized for large-scale HPC systems and the essential techniques they employ will continue to be relevant for future generations of supercomputing systems. A resilience design pattern formalizes these proven techniques and the trade-offs involved when using them.

Through extensive surveys and studies of HPC resilience techniques, we captured the descriptions of the HPC resilience design patterns, which are written down in a standardized template form. Each design pattern describes a solution to a recurring HPC resilience problem under a set of clearly defined assumptions about the type of the fault, error or failure it deals with and the constraints about the system behavior it guarantees. Yet the resilience design patterns are specified at a high level of abstraction and describe solutions that are free of implementation details. Expressing resilience solutions as abstract design patterns makes them more accessible to hardware and software designers of new HPC systems. From the collection of patterns we compiled a comprehensive pattern catalog [3] [6] in which, we codify the resilience design patterns in a layered hierarchy, which classifies the patterns in the catalog, and clearly conveys the relationships among them. The classification is illustrated in 1. In this hierarchical organization, the high-level patterns describe the outline of the solution in abstract, and lower level patterns provide additional guidelines and constraints that are encountered during the implementation of the pattern in an HPC environment. This organization suggests a number of ways in which these patterns can be combined to develop complete resilience solutions and also makes it easy to choose between design alternatives [7].

The resilience patterns in the catalog are broadly classified into *State* patterns and *Behavioral* patterns. The *State* patterns (not shown in Figure 1) encapsulate the protection domain of a resilience solution, i.e., they define the aspects of the application and/or system state for which the detection and

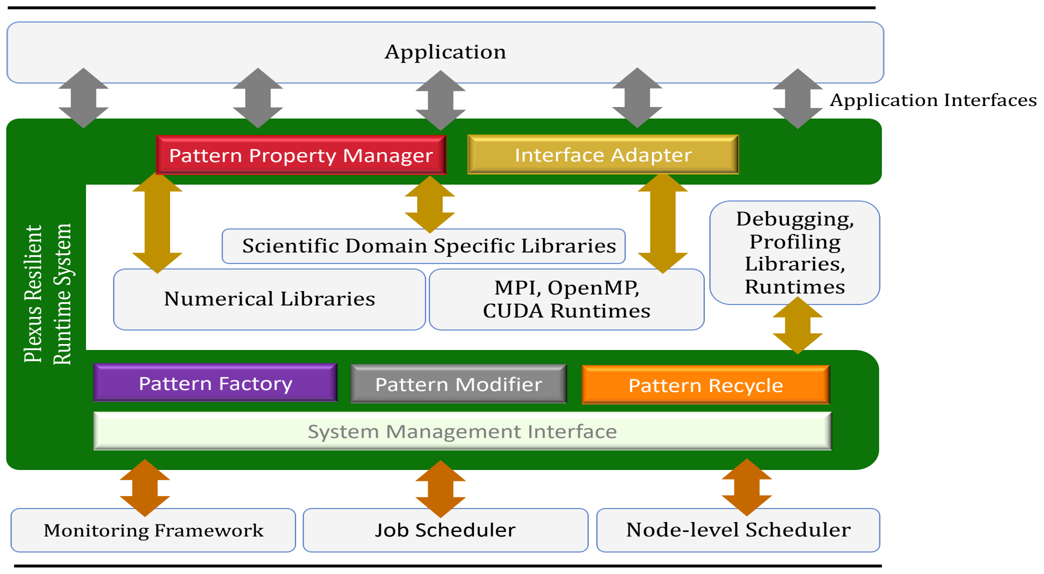


Fig. 2. Plexus: Pattern-Oriented Runtime System Architecture

mitigation applies. They also define the containment scope, i.e., the scope of how far a fault or error event propagates. The *Behavioral* patterns contain solutions that enables the system that instantiates them to cope with the presence of fault, error, or failure events by providing detection, containment and/or recovery capabilities. These patterns are hierarchically classified into: (i) **Strategy** patterns, which define high-level policies of a resilience solution and whose descriptions are deliberately abstract to contemplate the overall organization of a resilience solution; (ii) **Architecture** patterns, which convey specific methods necessary for the construction of a resilience solution and provide the outline of the actions necessary to handle a fault event and, (iii) **Structural** patterns, which provide concrete descriptions of the solution that are intended to guide the implementation of a resilience solution. This hierarchy is illustrated in 1.

The `Fault Diagnosis` pattern is a behavioral pattern that identifies the presence of the fault and determines its root cause. The solution consists of deploying an auxiliary monitoring system. The `Reconfiguration` pattern entails maintaining logical groupings of the sub-systems and modifying their interconnections to isolate the affected (sub-)system in response to failures. The `Checkpoint-Recovery` pattern, which is a formalization of well-known solution for handling system failures, is based on the creation of snapshots of the system state and maintenance of these checkpoints on a persistent storage system during the error- or failure-free operation of the system; these snapshots are used to recreate last known correct system state. The `N-modular Redundancy` and `Forward Error Correction Code` patterns offset the effects of an error/failure by provisioning excess resources or information in the system design. The `N-modular Redundancy` pattern employs additional resources to recover system operation by replacing subsystems that have encountered errors or have failed; the `Forward Error Correction Code` pattern uses some form of redundant information about the system to detect and correct corruptions of the system state. The `Design Diversity` pattern creates distinct but functionally equivalent versions of the same design specification. The versions are created by different individuals or teams, or developed using different tools with intent of eliminating design bugs in the system. The detailed descriptions of resilience design pattern solutions, including structure of the solution, scenarios where the pattern is applicable, pitfalls and consequences, implementation hints as well as the relatedness to the other patterns in the layered hierarchy are included in the full pattern catalog [6]. Complete resilience solutions that offer detection, containment and recovery capabilities for a specific component or the complete HPC system, are often composed by combining one or more patterns in the catalog.

III. PLEXUS

A. *Design Principles*

In modern large-scale HPC systems, runtime systems are a critical part of the software stack that complement the operating system (OS), the application programming interface (API) and the compiler frameworks. Runtime systems provide adaptive means to guide resource management and optimize application execution through the use of static and dynamic policies. They exploit knowledge about the status of the application and gather information about the system hardware operation to make decisions about task scheduling, synchronization, message-driven communication, power optimization, etc. The role of the runtime system in the HPC software stack is to bridge the gap between the system capabilities and the actual achievable performance for realistic HPC applications.

Designing a runtime system that supports resiliency for large-scale HPC systems requires carefully balancing the efficiency, scalability, productivity, and portability properties provided by existing libraries and runtime systems in the software stack with the capabilities that provide detection, containment and recovery. For example, the Message Passing Interface (MPI) has been the dominant standard for writing parallel applications that run on HPC platforms; productivity libraries such as BLAS and LAPACK are widely used for highly-optimized numerical linear algebra routines; and, the OpenMP API and runtime system are used for shared memory multithreaded processing. One of the primary goals of this work is to ensure that the runtime system maintains interoperatibility with these runtime systems and library frameworks while providing a resilient environment for HPC applications by managing the detection, containment and recovery capabilities based on the requirements of the applications and the state of the HPC system. Additionally, for addressing the different types of fault, error and failure events, separate resiliency solutions may need to be deployed by the runtime system. The selection of these solutions may be driven by static and dynamic policies that are driven by the needs of the application, the system architecture and the state of the available system resources. The implementations of the solutions may require varying levels of engagement from the system user and application programmer. Therefore, the high-level design goals of the runtime system architecture are:

- Flexibility in adapting resiliency capabilities based on the needs of the HPC application
- Integration with existing HPC programming models and leveraging existing libraries and runtimes
- Modularity in defining the scope and capabilities of the solutions
- Opportunistic selection of resiliency solutions among alternatives
- Interoperatibility with existing runtime systems and programming interfaces

B. Runtime System Architecture

To design a resilience-oriented runtime system that is cognizant of these objectives, we define the architecture of the PLEXUS runtime system that is based on leveraging the design patterns to create resilient environments for applications running on HPC systems. For a specific type of fault event, the solution for its detection, containment and recovery by the runtime system will typically require instantiation and combination of several resilience patterns. While the implementation of individual patterns in the catalog may be independently customized to the needs of the solution being designed, the behavior of the composite of patterns may need to be dynamically adapted and modified to meet the needs of the HPC application and adapted to constraints of the system architecture and software environment. Therefore, the runtime instantiation and adaptation of an intricate network of various patterns from the catalog to handle the multitude of fault, error and failure types that occur in HPC systems is central to the design philosophy of the PLEXUS runtime system. We believe that the instantiation and management of the resilience design patterns under the control of the runtime system offers opportunities to create adaptable and flexible, end-to-end resilience solutions for HPC systems.

The Plexus runtime manages the creation of the resilience patterns. This entails providing detection or recovery capabilities within the runtime system or leveraging the capabilities of existing library frameworks that provide resilience capabilities. The seamless integration of the interfaces of such library frameworks and the APIs used by the HPC application are managed by the Plexus runtime. Based on the rate of fault events in the system, or other static or dynamic policies, the pattern behavior may need to be adapted dynamically to modulate its resiliency properties. The Plexus runtime also manages the properties of instantiated patterns depending on the extent to which the application programmer or system user is involved in creating and modifying the policies. Certain group of patterns are instantiated, modified and destroyed under complete runtime control without any engagement with the HPC application program; other patterns are created and modified at the application level using Plexus runtime interfaces.

To cater to these various modes of managing the pattern-based resilience solutions, the architecture of the Plexus runtime is based on an extensible library that: (i) implements a core set of detection, containment and recovery functions; these functions are exposed to the HPC applications as a set of Plexus APIs (ii) provides a set of basic interfaces for extensibility, which allow for the integration of other libraries and frameworks that provide these capabilities; and (iii) contains interfaces that modify pattern behavior based on static or dynamic policies. These interfaces enable applications to request the invocation and termination of patterns and modification of policies for the patterns' behavior. The instantiation of a pattern under runtime control entails invoking one of the native Plexus routines for providing the resiliency functions for an application object, or setting extension objects to use other library functionality. Metadata about the pattern is also maintained by the Plexus runtime to manage the pattern behavior and to create logical groups of patterns. This architecture of Plexus facilitates the creation of resilience solutions that address specific fault models by invoking and managing the patterns as logical groups. It also permits the integration of other libraries and runtimes to leverage their respective productivity and optimization features. The Plexus interfaces to define the static and dynamic policies for the pattern management enables the resilience solutions to be applied based on well-defined rules, or opportunistically in response to error or failure events in the system.

C. Components

The PLEXUS runtime system is based on a component-based architecture. This design approach allows the runtime system to achieve the flexibility to introduce newer capabilities for new fault types and develop new policies tailored to application requirements and system architecture. The architecture of PLEXUS also enables rapid integration of other productivity libraries and runtime systems, which continuously evolve to leverage the advancements in the hardware and software systems. Associated with each component is a set of functions related to the role of the component in managing the pattern instances. Figure 2 shows the conceptual design of Plexus and its components. The key components of the PLEXUS runtime system are:

Pattern Factory

The *Pattern Factory* is the component that is responsible for instantiating the patterns. It exposes a set of routines for intializing native resiliency functionality in Plexus and provides the extensibility interfaces that allow the integration of other library frameworks. This component activates the metadata for each pattern instance and links related patterns during instantiation to enable creation of composite pattern solutions.

Pattern Modifier

The *Pattern Modifier* adapts the behavior of the already instantiated patterns based on policy decisions. The properties of the pattern and rules for its behavior are derived from the pattern metadata. The component's routines permit HPC applications to specify static or dynamic policies about pattern behavior.

Pattern Recycle

The *Pattern Recycle* destroys the patterns when the resilience capabilities are no longer required. The recycle of a pattern instance by the runtime allows it to only maintain pattern object instances and related state information for patterns relevant to the HPC application and those that minimize the overhead to system resources. The recycle entails the termination of the pattern object, unlinking any external library objects, and the clean up of any resources allocated and metadata related to the pattern instance. The metadata and policy information related to the pattern are also removed.

Interface Adapter

The *Interface Adapter* component serves as the gateway for function calls made by the HPC application program. This entails invocation of the native Plexus runtime-based resilience modules. This component also provides routines for extensibility, which enables other library objects to be integrated and handles forwarding of requests to these libraries with the appropriate policy parameters to tune the required resiliency features provided by the libraries.

Pattern Property Manager

This component is responsible for maintaining the metadata for the active resilience patterns in the runtime system. It also maintains the policies related to groups of linked patterns. The component's routines allow the pattern policies to be initialized and dynamically modified during application execution.

IV. Paradigms for Runtime System-driven Pattern Management

The composition of a set of patterns provide a complete resilience solution for specific fault model. As the Plexus runtime instantiates several such groups of patterns, the network of patterns becomes fairly complex. In an effort to systematically organize the patterns, we explore organizational paradigms for the patterns to effectively managed by the runtime. The paradigms outline guidelines for how patterns are interconnected and how they interact and influence the strategies for instantiating, modifying and destroying the patterns. We develop three paradigms: the *Integrated*, *Interception*, *Service-Oriented*. A collection of patterns linked together in accordance with one of these paradigms forms a logical grouping of patterns. The organizational paradigms also determines how the HPC application interacts with the Plexus runtime system. These paradigms are inspired by similar fault tolerance strategies used for objects in the Fault Tolerant CORBA (FT-CORBA) standard [8].

A. *Integrated Resilience Pattern Paradigm*

The patterns in this paradigm are organized to provide the HPC application with explicit control over the pattern instantiation and behavior. This requires the application code to include the Plexus routines that interact directly with the runtime components. The workings of the integrated resilience strategy is illustrated in Figure 3. With this pattern organization, the application defines the state pattern, i.e. the scope of the protection domain, and is aware of its resiliency properties since it explicitly selects the behavioral patterns for detection and recovery.

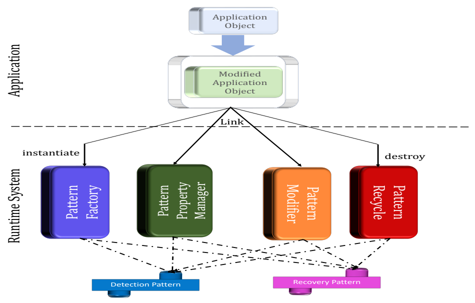


Fig. 3. Integrated Resilience Pattern Paradigm

B. *Interception Resilience Pattern Paradigm*

In this organization of the patterns, the HPC application is oblivious to the presence of the pattern-based capabilities provided by the Plexus runtime. The distinctive advantage of this paradigm is the transparency of the resiliency features; the application code requires no modification or recompilation. The management of the patterns is completely under the control of the runtime system. However, the *Iterface Adapter* component contains a shim library that transparently intercepts API calls made by the application. The pattern instantiation entails tracking the scope of the protection domain based on the arguments of the API calls and setting up fault detection and recovery functionality by invoking native Plexus routines or another library integrated through the extensible interface. Figure 4 shows interception-based paradigm in the runtime system.

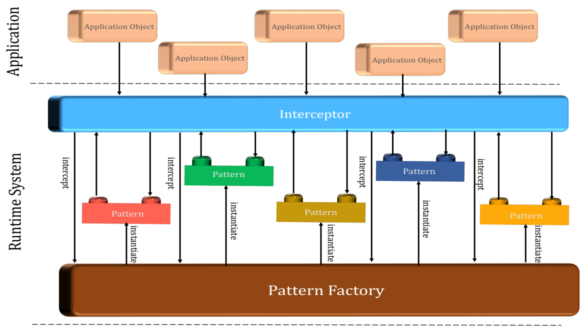


Fig. 4. Interception Resilience Pattern Paradigm

C. Service-Oriented Resilience Paradigm

In this paradigm the pattern instances that are created by the runtime are decoupled from the HPC application and the other components in the software environment. The key benefit of this pattern organization is ability to activate and deactivate the pattern-based detection, containment and recovery capabilities dynamically and to modify their properties as required. The patterns are activated under set of runtime parameters that define policies for the behavior of the patterns.The instantiation and management of the patterns is completely under the control of the runtime system. The pattern-based capabilities can also be disabled during the application execution. The workings of this paradigm of pattern management is shown in Figure 5.

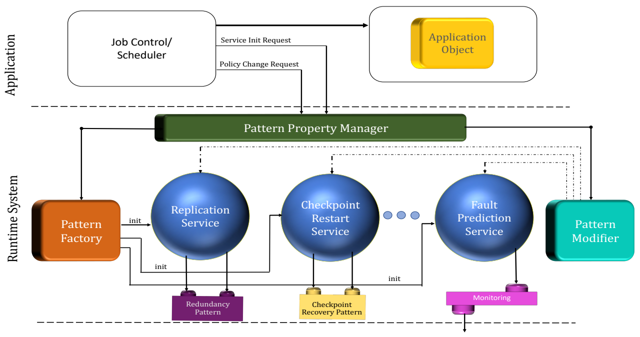


Fig. 5. Service-Oriented Resilience Paradigm

V. Prototype Implementation

We developed a prototype of the Plexus runtime that addresses two of the most important classes of faults that occur in large-scale HPC systems: (i) process failures on account of one or compute nodes malfunctioning, and (ii) silent data corruption in the application program data on account of transient faults in the system. For each type of fault, we instantiate a set of patterns that provide detection, containment and recovery functions; however, the patterns for the process failures and transient faults are organized using different paradigms.

For addressing the process failures, we aim to develop a pattern-driven solution that provides transparent failure recovery, which calls for the use of the *Interception* paradigm. The dominant model of parallel computation in HPC applications is based on message passing based on the MPI standard. However, most implementations of MPI such as OpenMPI and MPICH are not capable of gracefully handling process failures. The default response in case of even a single process failure is to crash the application since the library's default error handler is MPI_ERRORS_ARE_FATAL. In our Plexus implementation, the *Interface Adapter* component implements a shim library that transparently intercepts MPI library calls. It also instantiates failure detection and recovery patterns, which are realized using the User Level Failure Mitigation (ULFM) [9] library. ULFM includes a set of functions and primitives that are proposed extensions to be included in a future MPI standard, which enable tolerating process failures. While the ULFM extensions are concise, they require applications to make significant modifications to their application codes to leverage the fault tolerance features. By invocation of the patterns under the control of the *Pattern Factory*, much of the complexity of detection and recovery is handled by the patterns allowing an HPC application to run on a large-scale system unmodified, providing the abstraction of a failure-free system.

Fatal errors that cause process failures in MPI programs are discovered by *Detection* pattern. The default response in case of MPI process failure is to crash the application since the default error handler is MPI_ERRORS_ARE_FATAL. The detection pattern implementation leverages the primitives offered by the ULFM implementation of MPI: the MPI_COM_AGREE, MPI_COMM_REVOKE, MPI_COMM_FAILURE_ACK. ULFM guarantees that all MPI communications should return an ERR_PROC_FAILED error code if the runtime detects a process failure in the communicator. The notification of failures to other processes is propagated through constructs such as MPI_ERR_REVOKED and MPI_ERR_PROC_FAILED. The *Recovery* pattern isolates a failed process from the MPI communicator, and rebuilds the communicator with the exclusion of the failed process. The implementation uses ULFM extension MPI_COMM_SHRINK primitive for this purpose. The scope of the protection domain, i.e., the state pattern encompasses all processes in the MPI communicator

For the detection and correction of silent errors in the program state caused by transient faults, the application programmer is well-positioned to understand the resiliency needs for the application variables. Therefore, we employ the *Integration* strategy by providing Plexus routines for memory allocation and deallocation. These routines enable the runtime to define the scope of the protection domain. During pattern instantiation by the *Factory*, checksum vectors for matrix and vector data structures are established (an instance of the *Forward Error Code* Pattern). For pointer variables, the runtime instantiates *N-modular Redundancy* patterns, creating triple modular redundant copies of the pointers. The detection pattern implementation is embedded in the Plexus routines for read and write operations that enable the patterns in the runtime to check validity of the checksums or compare the redundant copies of the pointer variables. The recovery pattern repairs a corrupted data value from the checksum or the redundant copies of pointer variables.

VI. Experimental Evaluation

A. Test Application

The linear system solver, which solves a problem of form A · x = b, makes up an important kernel of many scientific computing applications. The Conjugate Gradient (CG) method, which is useful when the operand is a sparse matrix, expresses x as a linear function of n vectors p_1, p_2, ... p_n, where each pair of vectors in the set are conjugate in A. With an initial approximation x_0; the residual $r_0 = b - A \cdot x_0$, which is the direction of the error in x_0, serves as the first conjugate vector, p_0. Subsequent iterations compute the residual r_k and use it to compute the next conjugate vector p_k. The process is repeated until r_k falls below some threshold. We use an MPI-implementation version of the application, which uses detection and recovery patterns under the control of the Plexus runtime for process failure and transient faults.

B. Experiment Setup

We use an HPC system consisting of a 40 compute node Linux cluster, where each compute node has 2 AMD Opteron processors with 12 cores per chip (total of 24 cores per node) and 64 GB memory, for a total of 960 processor cores. The nodes are connected with a dual-bonded 1 Gbps Ethernet interconnect. The patterns instantiated by PLEXUS runtime that support detection and recovery of process failures rely on ULFM release 1.1, which is derived from Open MPI-1.7.1, which provides the primitives for failure detection, notification, and rebuilding failed communicators. For the evaluation of pattern-based detection and recovery, we perform extensive fault injection experiments. Hard error events that cause MPI processes to fail are simulated by terminating an MPI process during application execution. This is done by randomly sending a kill signal to one of the MPI processes in the communicator. Transient fault injection is performed by separate fault injection process forked on every node; this process sends an interrupt signal to one of the compute processes and perturbing a random data variable in the process state.

C. Results

Figure 6 shows the results from the fault injection experiments. We perform our fault injection experiments with 32, 64, 128, 256, 512 and 1024 MPI ranks. The results are averaged across random injections from 10000 experiment runs. In these

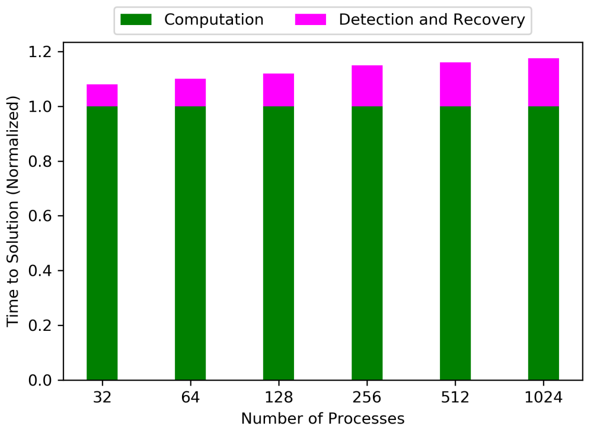


Fig. 6. Results: Process Failure Recovery Time For Single Failure Recovery

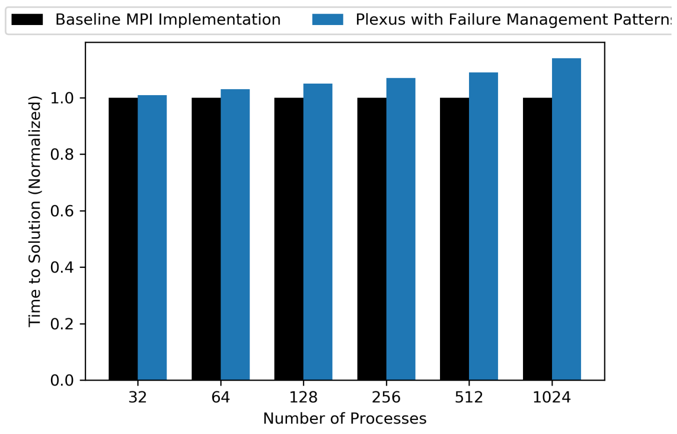


Fig. 7. Results: Plexus Managed Failure Detection and Recovery Pattern Overhead for a Failure-Free Scenario

experiments, a single failure event is generated during the application execution. The recovery pattern isolates the failed process, shrinks and rebuilds the MPI communicator. From the application's perspective, the recovery after the failure of a process causes the application to execute with N-1 processes. This affects the load balance of the work across processes causing a larger overhead to total time to solution for small number of processes. However, the time for the ULFM library to communicate the failure to the remaining processes and rebuild the communicator scales with the number of processes leading to a proportional higher increase in overhead for large number of processes. The overhead is in the range of 12 to 18%, which is related to the time required for the repair of the MPI communicator. We also compare the overhead of the runtime system in a failure-free scenario. Figure 7 shows the overhead of deploying the patterns at runtime under the control of the runtime. The comparison is between a baseline implementation using the standard MPI library and a modified version that uses Plexus.

In the transient fault injection experiments, we explore the pattern-based protection for variables separately. This requires the specific data structures that need to be protected by the Plexus-based runtime to be allocated and referenced through the Plexus library routines. The detection and recovery patterns are only instantiated for these structures. The remaining structures are allocated using the standard memory allocation interfaces. The application allocates the matrix A, the vector b and the solution vector x. Additionally, the conjugate vectors p and the residual vector r are also referenced during each iteration of the algorithm. The CG algorithm being iterative by nature is inherently tolerant to certain transient error without loss of correctness in the final outcome of the solver. Therefore, the protection of operand matrices A, B and solution vector x have successful outcome in $>70\%$ transient fault injections without any detection/recovery capabilities. With the protection of these structures using patterns that implement checksums for detection and recovery, the percentage of success increases by 20% as shown in 8(a). The baseline case is the completion rate in the presence of transient faults, but without any detection or recovery capability for the structure. When the intermediate vectors p and r are protected, we achieve a 40% increase in application resiliency. The most significant increase in rate of successful outcomes is achieved when the pattern-based detection and recovery are applied to the pointers and address structures. These are highly sensitive to transient faults since these cause they cause misaligned or illegal memory accesses, which are fatal to the applications. When all variables are protected using the patterns, the rate of successful outcomes becomes as high as 97%.

The burden on performance overhead by protecting different application structures is shown by the results in 8(b). Protecting the large, sparse matrices using checksums incurs 30% increase in time to solution, which yields an increase of 20% in ratio of successful outcomes. The highest return on investment is achieved by protecting the critical address and pointer variables, which despite the use of a pattern that implements triple modular redundancy, incurs only 7% increase in time to solution, but provides an increase of over 80% in rate of successful outcomes. The important benefit of the pattern-based management is that it permits this trade-off between overhead and benefit to the application's resiliency to be exploited by the Plexus runtime system.

VII. Related Work

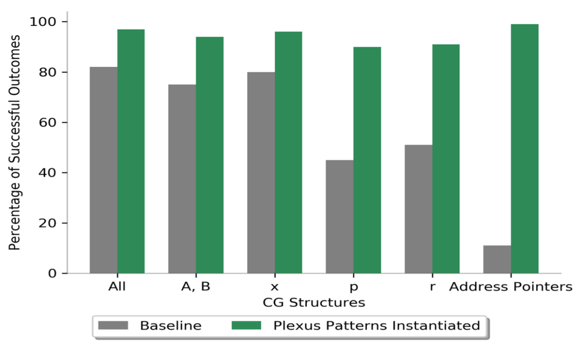
Historically, HPC systems have limited the use of runtime systems for avoiding overheads imposed by the inclusion of such software in the system stack. In order to optimize the parallelism in applications various lightweight runtime systems have been employed. This includes runtime control employed by the most widely used programming interfaces of OpenMP [10] and MPI [11]. The Charm++ runtime [12] optimizes the scheduling of tasks by supporting dynamic load balancing. The HPX-5 runtime [13] provides support for distributed asynchronous multi tasking (AMT) in HPC systems. At the node-level, runtime systems that support Cilk [14], Intel TBB [15], OpenMP 4.0 [16] and Habanero [17] implement task scheduling algorithms that optimize the scheduling of threads in many-core systems. The Argo project [18] proposes a distributed runtime system for extreme-scale HPC systems that consists of various services that communicate through a global information bus for resource monitoring and configuration of system services and job scheduling. The GEOPM runtime framework [19] enables optimization of performance and energy efficiency by discovering patterns in application execution to control certain hardware-level settings.

Supporting resilience capabilities at the runtime level has been previously explored. For example, the Scalable runTime Component Infrastructure (STCI) [20] runtime contains entities called agents; fault tolerance is achieved by assigning the roles of task execution and task control to different agents. To handle process failures transparently for parallel MPI applications, process recovery has been implemented for the PMIx runtime [21]. At the node level, the Rolex [22] runtime system provides detection and thread-level recovery capabilities in support of language extensions that capture the application programmer's knowledge about the resiliency of program variables and code regions. However, the PLEXUS approach of composing solutions through combinations of patterns offers distinct advantages of modularity of solutions for specific fault types, ability to handle multiple fault models, the flexibility to invoke and modify the solutions dynamically.

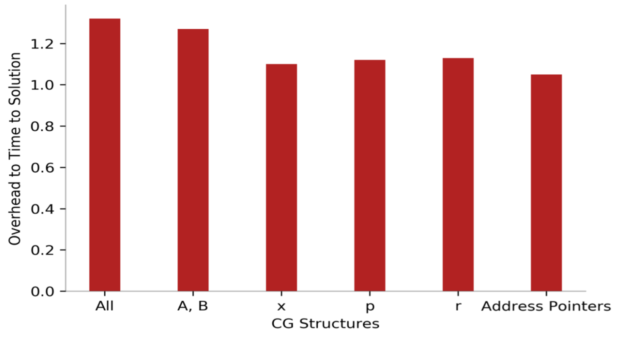
VIII. Conclusion

In this paper, we presented a new approach to architecting runtime systems for extreme-scale HPC systems based on resilience design patterns. The emergence of resilience as one of the key challenges for the next generation of supercomputing systems has motivated the need to explore new approaches to the design of HPC software. Runtime system software is an important component of the HPC stack that offers opportunities to augment the resiliency of applications. The PLEXUS runtime system is based on the runtime instantiation of the resilience patterns and their management through static and dynamic policies. We developed three new strategies for runtime pattern management. We defined the architecture of the PLEXUS runtime system and detailed the roles of the key components of the runtime that that lend the organizational structure for the strategies to be realized in a HPC software stack. The paper also defined the PLEXUS interfaces and describes a prototype implementation that implements these components and services in support of parallel MPI-based linear solver-based application.

We demonstrated that the PLEXUS runtime instantiated pattern-based solutions for MPI rank failure detection and recovery delivers effective resilience in the presence of actual failure events and incurs very low implementation overhead. The key benefit of the pattern-based approach is highlighted by the seamless integration and runtime management of several pattern instances that provided resilience capabilities for different fault types; yet the subset of patterns for each type is modularly controlled by the runtime. The prototype implementation also demonstrates that pattern-based architecture of the runtime allows an application to leverage the capabilities of an



(a)



(b)

Fig. 8. Results for Transient Fault Injection Experiments: (a) Successful Outcomes with Pattern-based Protection for differnt application variables; (b) Overhead of Transient Fault Detection and Correction on Performance

existing MPI library as well as utilize the PLEXUS runtime's native resilience capabilities. Therfore, the pattern-based approach to designing runtime-driven resilience solutions and the extensible nature of the PLEXUS runtime provides a roadmap for the runtime to stitch together solutions for a range of faults, errors, and failures, as well as to dynamically consider alternatives between patterns towards performance, power and energy efficiency.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Early Career Program, with program managers Robinson Pino and Lucy Nowell.

References

[1] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod, "High-end computing resilience: Analysis of issues facing the hec community and path-forward for research and development," December 2009.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham *et al.*, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014.

[3] S. Hukerikar and C. Engelmann, "Resilience design patterns: A structured approach to resilience at extreme scale," *Supercomputing Frontiers and Innovations*, vol. 4, no. 3, 2017.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, August 1977.

[6] Saurabh Hukerikar and Christian Engelmann, "Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale (Version 1.2)," 2017. [Online]. Available: ORNL/TM-2017/745

[7] S. Hukerikar and C. Engelmann, "A pattern language for high-performance computing resilience," in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, ser. EuroPLoP '17, 2017.

[8] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt, "Doors: towards high-performance fault tolerant corba," in *Proceedings DOA'00. International Symposium on Distributed Objects and Applications*, September 2000, pp. 39–48.

[9] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure Recovery of MPI Communication Capability: Design and Rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[10] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[11] M. P. Forum, "Mpi: A message-passing interface standard," University of Tennessee, Knoxville, TN, USA, Tech. Rep., 1994.

[12] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore, "The CHARM Parallel Programming Language and System: Part II – The Runtime system," *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[13] Center for Research in Extreme Scale Technologies (CREST), "Hpx-5 runtime system." [Online]. Available: https://wiki.modelado.org/HPX-5

[14] R. D. Blumofe and C. E. Leiserson, "The Cilk Runtime System." [Online]. Available: http://cilk.mit.edu/runtime/

[15] Intel Corporation, "Intel Threading Building Blocks (Intel TBB)." [Online]. Available: https://software.intel.com/en-us/tbb

[16] OpenMP Architecture Review Board, "OpenMP application program interface version 5.0," November 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[17] Habanero Research Group, "A C/C++ task-based programming model for shared and distributed memory parallel computing," 2017.

[18] S. Perarnau, J. A. Zounmevo, M. Dreher, B. C. V. Essen, R. Gioiosa, K. Iskra, M. B. Gokhale, K. Yoshii, and P. Beckman, "Argo nodeos: Toward unified resource management for exascale," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 153–162.

[19] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, and S. Jana, "Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions," in *High Performance Computing*. Springer International Publishing, 2017, pp. 394–412.

[20] G. Valle, T. Naughton, S. Bohm, and C. Engelmann, "A runtime environment for supporting research in resilient hpc system software tools," in *2013 First International Symposium on Computing and Networking*, 2013, pp. 213–219.

[21] D. Zhong, A. Bouteiller, X. Luo, and G. Bosilca, "Runtime level failure detection and propagation in hpc systems," in *Proceedings of the 26th European MPI Users Group Meeting*, ser. EuroMPI 19, 2019.

[22] S. Hukerikar and R. F. Lucas, "Rolex: Resilience-oriented language extensions for extreme-scale systems," in *Journal of Supercomputing*, 2013, pp. 213–219.