Concepts for OpenMP Target Offload Resilience*

Christian Engelmann[1], Geoffroy R. Vallée[2], and Swaroop Pophale[1]

[1] Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN 37831, USA
{engelmannc,pophaless}@ornl.gov
[2] Sylabs, Inc, 1191 Solano Ave, Unit 6634, Albany, CA 94706
geoffroy@sylabs.io

Abstract. Recent reliability issues with one of the fastest supercomputers in the world, Titan at Oak Ridge National Laboratory (ORNL), demonstrated the need for resilience in large-scale heterogeneous computing. OpenMP currently does not address error and failure behavior. This paper takes a first step toward resilience for heterogeneous systems by providing the concepts for resilient OpenMP offload to devices. Using real-world error and failure observations, the paper describes the concepts and terminology for resilient OpenMP target offload, including error and failure classes and resilience strategies. It details the experienced general-purpose computing graphics processing unit (GPGPU) errors and failures in Titan. It further proposes improvements in OpenMP, including a preliminary prototype design, to support resilient offload to devices for efficient handling of errors and failures in heterogeneous high-performance computing (HPC) systems.

Keywords: Supercomputing · Resilience · OpenMP.

1 Introduction

Resilience, i.e., obtaining a correct solution in a timely and efficient manner, is a key challenge in extreme-scale HPC. Heterogeneity, i.e., using multiple, and potentially configurable, types of processors, accelerators and memory/storage in a single platform, adds significant complexity to the HPC hardware/software ecosystem. The diverse set of compute and memory components in today's and future HPC systems require novel resilience solutions.

* Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy. 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).

There is only preliminary work in resilience for heterogeneous HPC systems, such as checkpoint/restart for GPGPUs using OpenCL with VOCL-FT [16]. There is also fine-grain transaction-based application-level checkpoint/restart with the Fault Tolerance Interface (FTI) [2]. Rolex [12] is an initial set of C/C++ language extensions for fine-grain resilience, which specify how data variables and code block execution may be repaired during program execution.

In contrast, the Titan supercomputer at ORNL experienced severe GPGPU reliability issues over its life time (2012–2019). In late 2016, 12 out of Titan's 18,688 GPGPUs failed per day [21]. Approximately 11,000 GPGPUs were replaced in the 2017–2019 time frame due to failures or high failure probability. The only mitigation available was application-level checkpoint/restart, which was never designed to efficiently handle such high failure rates. Titan's successor, the Summit supercomputer at ORNL [20], has 27,648 GPGPUs. While it is the expectation that Titan's severe reliability issues were a rather unique experience, hope is not a strategy. There is an urgent need for fine-grain and low-overhead resilience capabilities at the parallel programming model that permit specifying what types of errors and failures should be handled and how.

Efficient software-based solutions to fill gaps in detection, masking, recovery, and avoidance of errors and failures require coordination. Based on the underlying execution model and intrinsic resilience features of the hardware, the various components in a heterogeneous system can be organized into protection domains. Employed resilience solutions can handle errors and failures in specific components and granularities where it is most appropriate to do so and in coordination with the rest of the system, which prevents errors from propagating and failures from cascading beyond these protection domains.

This paper describes concepts for resilience in OpenMP based on real-world observations from the largest heterogeneous HPC system in the world. It focuses on offload to devices as a first step toward resilience in OpenMP. The paper describes the used concepts and terminology, including general fault, error and failure classes. It derives error and failure scopes and classes for OpenMP target offload and maps them to the experienced GPGPU errors and failures in Titan. Using these concepts, this paper proposes improvements to enable resilience for OpenMP offload to devices and details a preliminary prototype design based on the concept of quality of service (QoS).

2 System Model

This section describes the involved concepts and terminology for OpenMP target offload. It continues with a short overview of general fault, error and failure classes and common terms that will be used in this context. It further defines the error and failure scopes and classes for OpenMP target offload.

2.1 OpenMP Target Offload

An *OpenMP thread* offloads the code and data of a *target region* in the form of a *target task* from the *host device* (*parent device*) to a *target device* using a *target construct*. The *target device* can be specified by a *device number*, otherwise the *default device number* is used. The *target task* may be *undeferred*, *i.e.*, the *OpenMP thread* waits for the completion of the *target task*, or *deferred*, *i.e.*, the *OpenMP thread* does not wait for the completion of the *target task*. *Target task* input and output data is *mapped* to and from the *host device* to the *target device*. Space for *target task* runtime data may be allocated on the *target device*.

The work presented in this paper primarily focuses on an *OpenMP thread* running on a conventional processor core and offloading a *target region* as a *target task* to a GPGPU. It does not focus on an *OpenMP thread* executing an *OpenMP task* on the *host device*, as the shared memory aspects are significantly more complex and require different error and failure models. This work is, however, applicable to a great extend to offloading a *target region* as a *target task* to other types of *target devices* that OpenMP may support.

The system model assumes that *target task* input data is transferred or made accessible to the *target device* before the *target task* starts, *target task* runtime data is allocated before it starts, and *target task* output data is transferred to or made accessible to the *host device* after it ends. Only the *target task* modifies its input, output, and runtime data during its execution, *i.e.*, the data is not shared with the *host device*. The *target task* is typically a parallel execution on the GPGPU and the data may be shared between threads on the GPGPU, i.e., *target task* data may be shared within the *target device* during its execution.

2.2 Faults, Errors and Failures

Error and failure behavior in OpenMP is currently undefined. Consequently, implementations are left to handle them (or not) in a non-uniform way. In general, a *fault* is an underlying flaw/defect in a *system* that has potential to cause problems. A fault can be *dormant* and can have no effect. When *activated* during system operation, a fault leads to an *error* and an illegal *system state*. A *failure* occurs if an *error* reaches the service interface of a *system*, resulting in behavior that is inconsistent with the system's specification. Prior work [11,19] identified the following general fault, error and failure classes and common terms:

- {*benign,dormant,active*} {*permanent,transient,intermittent*} {*hard,soft*} *fault*
  - *Benign:* An inactive fault that does not activate.
  - *Dormant:* An inactive fault that potentially becomes active at some point.
  - *Active:* A fault that causes an error at the moment it becomes active.
  - *Permanent:* The presence of the fault is continuous in time.
  - *Transient:* The presence of the fault is temporary.
  - *Intermittent:* The presence of the fault 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*.

- {*undetected,detected*} {*unmasked,masked*} {*hard,soft*} *error*
  - *Undetected:* An error whose presence is not indicated.
  - *Detected:* An error whose presence is indicated by a message or a signal.
  - *Masked:* An error whose impact is compensated so that the system specification is satisfied despite the incorrect state; its propagation is limited.
  - *Unmasked:* An error that has not been compensated and has the potential to propagate.
  - *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:
    - *Latent error* or *silent error:* Any type of *undetected error*.
    - *Silent data corruption (SDC):* An *undetected unmasked hard* or *soft error*.

- {*undetected,detected*} {*permanent,transient,intermittent*} {*complete,partial, Byzantine*} *failure*
  - *Undetected:* A failure whose occurrence is not indicated.
  - *Detected:* A failure whose occurrence is indicated by a message or a signal.
  - *Permanent:* The presence of the failure is continuous in time.
  - *Transient:* The presence of the failure is temporary.
  - *Intermittent:* The failure is temporary but recurring in time.
  - *Complete:* A failure that causes service outage of the system.
  - *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 failure classes:
    - *Fail-stop:* An *undetected* or *detected failure* that completely halts system operation, which often causes an irretrievable loss of state.
    - *Fail-safe:* A mode of system operation that mitigates the consequences of a system failure.

While a *fault* is the cause of an *error*, its manifestation as a *state change* is considered an *error*, and the transition to an incorrect service is observed as a *failure* (see Fig. 1). A *fault-error-failure chain* is a directed acyclic graph (DAG) with *faults*, *errors* and *failures* represented by its vertices. In a system composed of multiple components, *errors* may be transformed into other *errors* and *propagate* through the system generating further *errors*, which may eventually result in a *failure*. A *failure cascade* occurs when the failure of a component A causes an error and subsequently a failure in component B interfaced with A.

2.3 OpenMP Target Offload Error and Failure Scopes and Classes

In terms of hardware errors and failures, OpenMP offloading has a *host device* and *target device* scope. In terms of software errors and failures, *OpenMP thread* and *target task* scopes exist. The *host device* and *OpenMP thread* scopes are

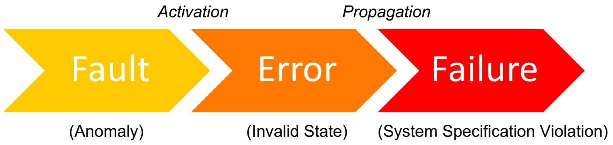

Fig. 1. Relationship between fault, error and failure

not considered in this work due to the complex shared memory aspects it involves. Only *target device* and *target task* errors and failures are considered. The following error and failure classes are defined:

- {*undetected, detected*} {*unmasked, masked*} {*hard, soft*} *target device error*
- {*undetected, detected*} {*unmasked, masked*} {*hard, soft*} *target task error*
- {*undetected, detected*} {*permanent, transient, intermittent*} {*complete, partial, Byzantine*} *target device failure*
- {*undetected, detected*} {*permanent, transient, intermittent*} {*complete, partial, Byzantine*} *target task failure*

A total of 16 error classes for *target devices* and *target tasks* are defined based on the general error classes. Undetected masked errors are rather irrelevant, as the masking makes them undetectable by any error detector. Detected masked errors are less relevant, as the masking already limits error propagation. A resilience strategy may still do something about a detected masked error though, such as to avoid it in the future. Undetected errors may become detectable through a resilience strategy. Undetected errors that do not become detectable are problematic, as no resilience strategy is able to deal with them.

A total of 36 failure classes for *target devices* and *target tasks* are defined based on the general error failure classes. Undetected failures may become detectable through a resilience strategy. Undetected failures that do not become detectable are problematic, as no resilience strategy is able to deal with them.

3 Observed Errors and Failures

This section provides an overview of the observed and inferred errors and failures in the Titan supercomputer at ORNL that are relevant for OpenMP target offload with GPGPUs. It maps these errors and failures the previously defined OpenMP offloading error and failure classes.

3.1 GPGPU Errors and Failures in Titan

The Titan supercomputer deployed at ORNL in November 2012 as the fastest in the world will be decommissioned in 2019, still being the 9th fastest. It is a hybrid-architecture Cray XK7 with a theoretical peak performance of 27 PFlops and a LINPACK performance of 17.95 PFlops. Each of Titan's 18,688 compute nodes consists of an NVIDIA K20X Kepler GPGPU and a 16-core AMD Opteron processor. A significant amount of work has been published about the observed and inferred errors and failures in Titan [13,9,15,14,21]. The following Titan GPGPU (*target device/task*) errors can be mapped to the previously defined OpenMP offloading error classes (see Table 1 for a summary):

- *Target device error correcting code (ECC) double-bit error:* A *detected unmasked soft error* in *target device* memory. This error is detected and signaled by the *target device*. It typically transitions to a *target task abort*.
- *Target device SDC:* An *undetected unmasked soft error* in *target device* memory or logic. It is not signaled and can propagate to a *target task SDC*, a *target task abort*, or a *target task delay*, including an indefinite delay (hang).
- *Target task SDC:* An *undetected unmasked soft error* in *target task* data. It is not signaled and can transition to a *target task abort* or a *target task delay*, including an indefinite delay. It may propagate to incorrect *target task* output.

These Titan GPGPU (*target device/task*) failures can be mapped to the previously defined OpenMP offloading failure classes (see Table 2 for a summary):

- *Target device Peripheral Component Interconnect (PCI) bus width degrade:* A *detected transient, intermittent or permanent partial failure* of the PCI connection between the *host device* and the *target device*. It is typically caused by a PCI hardware failure. This failure results in degraded transfer performance for *target task* input and output data. It can cascade to a *target task delay*.
- *Target device PCI bus disconnect:* A *detected permanent complete failure* of the PCI connection between the *host device* and the *target device* or a *detected permanent complete failure* of the *target device*. It is typically caused by a PCI hardware or GPGPU failure. This failure can cascade to a *target task abort*.
- *Target device dynamic page retirement (DPR):* A *detected transient complete failure* of the *target device* memory. It is typically caused by the GPGPU when preventing or repairing a *detected permanent partial failure* of the *target device* memory. This failure can cascade to a *target task abort*.
- *Target device SXM power off:* A *detected permanent complete failure* of the *target device*. It is typically caused by a voltage fault. This failure can cascade to a *target task abort*.
- *Target task abort:* A *detected permanent complete failure* of a *target task*. It is typically caused by a *target task error* or a *target device error* or *failure*.
- *Target task delay:* A *detected permanent partial failure* of a *target task*. It is typically caused by *target task SDC* or a *target device PCI width degrade*.

4 Resilience for OpenMP Target Offload

Errors may propagate or transition to failures and failures may cascade in other parts of the system, such as the *host device* and *OpenMP threads*, depending

Table 1. Mapping of Titan GPGPU errors to the OpenMP offloading error classes

Error	**Error Class**
Target device ECC double-bit error	Detected unmasked soft target device error
Target device SDC	Undetected unmasked soft target device error
Target task SDC	Undetected unmasked soft target task error

Table 2. Mapping of Titan GPGPU failures to the OpenMP offloading failure classes

Failure	**Failure Class**
Target device PCI width degrade	Detected transient partial target device failure Detected intermittent partial target device failure Detected permanent partial target device failure
Target device PCI disconnect	Detected permanent complete target device failure
Target device DPR	Detected transient complete target device failure
Target device SXM power off	Detected permanent complete target device failure
Target task abort	Detected permanent complete target task failure
Target task delay	Detected permanent partial target task failure

on employed resilience strategies. Since OpenMP currently does not employ resilience strategies, a *target task abort failure* will cascade to an *OpenMP thread abort failure* and a *target task delay failure* will cascade to an *OpenMP thread delay failure*. Additionally, any *complete target device failure* will cascade to an *OpenMP thread abort failure*. *Target task SDC* may propagate to a *OpenMP thread SDC*, which then may transition to an *OpenMP thread delay* or *abort failure* or propagate to incorrect *OpenMP thread* output. This section discusses the individual needs for changes in the OpenMP standard and implementations to employ a reasonable set of resilience strategies for OpenMP offload to devices.

4.1 Error and Failure Detection and Notification

Errors and failures need to be detected and employed resilience strategies need to be notified in order to be able to deal with them.

Errors and failures detected by the *target device* are reported to the OpenMP runtime after attempted *target task* execution. Employed resilience strategies may transparently handle them. However, some resilience strategies need application feedback to decide on the course of action, such as to asses if an error or failure is acceptable. A reporting and feedback capability for device-detected errors and failures is needed in OpenMP. This could be implemented using function callbacks and an OpenMP language feature for defining resilience policies using the previously defined OpenMP offloading error and failure classes. Since detailed error and failure information could be helpful to make decisions, such as to assess the severity of a *target device ECC double-bit error*, OpenMP support for *target device* error reporting to the application is needed.

Errors and failures may also be detected by the application, such as by checking the correctness of *target task* output. A notification capability for application-detected errors and failures is needed in OpenMP to enable the use of resilience strategies by the application. This could be implemented using an OpenMP language feature for raising error notifications to the OpenMP runtime.

4.2 Fail-Fast and Graceful Shutdown

The *fail-fast* resilience strategy is designed to detect and report errors and failures as soon as possible. It also stops normal operation if there is no other resilience strategy in place to handle a specific error or failure. At the very least, the default error and failure behavior of OpenMP in general should be defined as *fail-fast*. This permits resilience strategies that are in place outside of OpenMP to efficiently handle errors and failures. A primary example is application-level checkpoint/restart, where any computation an application continues after an unrecoverable error or failure is wasted time.

For OpenMP target offload, *fail-fast* means that the *host device* detects and reports errors and failures as soon as possible. It also means that the OpenMP runtime aborts *target tasks* impacted by the error or failure as soon as possible. For performance failures, such as the *target device PCI width degrade* that can cascade to a *target task delay*, this means aborting a *target task*. The resilience strategy of graceful degradation, which would risk/accept a *target task delay* is described in the following subsection. The *fail-fast* strategy can also be employed in conjunction with application-level error or failure detection, such as through an application-level correctness check of the *target task* output and a corresponding abort upon error detection.

Graceful shutdown avoids error propagation and failure cascades beyond the component that is being shutdown. An uncontrolled stop of normal operation, such as a crash, can result in errors or failures in other system components. Operating system (OS) features usually prevent such effects by triggering cleanup procedures, such as after a crash. However, the OS may not have control over everything an OpenMP application is involved in, such as when an OS bypass is employed for networking/storage or a workflow software framework is used. Another example is the clean execution of an Message Passing Interface (MPI) abort after an OpenMP abort due to a *target task failure*. *Error handlers* can perform application-level cleanup during a *graceful shutdown*, but they would need to be triggered by the OpenMP runtime upon a *fail-fast* abort.

4.3 Graceful Degradation

Graceful degradation continues operation after an error or failure at the cost of performance or correctness that is deemed acceptable. In case of a performance failure, such as the *target device PCI width degrade* that can cascade to a *target task delay*, this means not aborting a *target task* and accepting the possible performance degradation, but reporting the failure to the application/user.

In case of a resource outage, such as the *target device PCI disconnect* that can cascade to a *target task abort*, this means continuing operation with less resources while employing a resilience strategy for aborted tasks. For example, an aborted task may be re-executed on a different *target device* using a rollback recovery strategy (described in the following subsection) while the failed *target device* is removed from OpenMP's pool of *target devices*. This requires OpenMP support for shrinking the number of *target devices* after a failure.

In case of a detected error, *graceful degradation* means to continue operation despite the error and to accept a possible error propagation. The application may need to make a decision if an error is acceptable.

4.4 Rollback Recovery

The *rollback recovery* resilience strategy transparently re-executes an erroneous or failed *target task* using the original *target task* input. The re-execution may be performed on the same *target device*, assuming that it is available and has not been removed from OpenMP's pool of *target devices* due to *graceful degradation*. If it has been removed, the re-execution is performed on a different *target device*. Successive *target task errors* or *failures* may result in corresponding successive re-executions. The number of successive rollbacks should be restricted to avoid endless rollbacks. On systems where the *target task* input is not copied to the *target device* but used in-place, the input may be backed up before offloading to assure its integrity, i.e., to protect it from being corrupted.

An OpenMP language extension is needed to specify the *rollback recovery* resilience strategy and its parameters, such as the maximum number of rollbacks, for each *target task*. The OpenMP runtime relies on *target device error* and *failure* detection and on application error detection notification to initiate rollbacks.

4.5 Redundancy

Redundancy in space executes *target tasks* at the same time on different *target devices*, while redundancy in time executes them sequentially on the same *target device*. A mix between both executes them on multiple *target devices*, where at least one *target device* is being reused. Common levels of redundancy are two and three, where two redundant *target tasks* detect a *target task error* and detect and mask a *target task failure*. Three redundant *target tasks* detect and mask a *target task error* and two *target task failures*. Error detection uses *target task* output comparison, while error masking uses the output of the majority. Failure detection and masking uses the output of the fastest surviving *target task*.

An OpenMP language extension is needed to specify the *redundancy* resilience strategy and its parameters, such as redundancy level (2 or 3) and resource usage (space, time or both). The OpenMP runtime relies on *target device error* and *failure* detection and on application error detection notification. It also relies on *target task* output comparison (e.g., bit-wise comparison or error bounds).

5 Preliminary Prototype

We detail in this section some aspects of the design and implementation of our solution for OpenMP target offload resilience. Both are driven by software engineering concerns, best-practices in extreme scale computing and available standards and libraries.

5.1 Design Details

Because our work is in the context of complex software components (a compiler), a standard (OpenMP) and a set of new concepts (QoS), one of our main challenges from a design and software engineering point-of-view is the separation of concerns. It is for example beneficial to have a clear separate implementation of the QoS and OpenMP support, and enable a fine-grain interaction of the resulting libraries. By doing so, it becomes easier to define, implement, modify and maintain each component, as well as explicitly and precisely define how these components interact. We believe this is especially critical when using complex production-level software such as a main-stream compiler (Low Level Virtual Machine (LLVM)). Another level of complexity comes from the asynchronous aspect of the problem we are trying to solve: the QoS runtime needs to asynchronously interact with the OpenMP runtime to enable system monitoring, fault detection and potentially recovery.

Our design centers on a novel concept for QoS and corresponding OpenMP language and runtime extensions. The QoS language extensions allow application developers to specify their resilience strategy without focusing on the implementation details. The QoS runtime extensions create a corresponding contract that maps application resilience requirements to the underlying hardware and software capabilities.

A QoS contract is defined as a set of QoS parameters that reflect the users' resilience requirements by identifying the requested resilience strategies. We propose a QoS language that provides all the required semantics to manipulate QoS parameters which can be applied to both application's data and tasks. These parameters are handled via generic "get/set" interfaces, and can be expressed as: (1) key/value pairs; (2) bounded values; and (3) ranges of values. The interface uses the block concept, similarly to OpenMP, to define the scope in which parameters are valid and the QoS contract with the runtime system. To simplify the definition of new QoS contracts, predefined QoS classes offer coherent sets of parameter that achieve popular resilience strategies. By using these classes, users only need to specify a few, if any, parameters, and let the system manage QoS policies that are already available. The following example uses QoS key/value pairs for specifying triple redundancy for a *target task*:

```
#pragma omp qoskv resilience (TASK_REDUNDANCY, BOOL, TRUE)
#pragma omp qoskv resilience (TASK_REDUNDANCY_FACTOR, INT, 3)
{
  #pragma omp target ...
  ...
}
```

An implementation of OpenMP is extended to offer an event-based QoS-aware runtime for resilience with a QoS Scheduler and QoS Negotiation Schemes at its core (see Fig. 2). The QoS Negotiation Schemes drive the method to enforce the QoS requirements, specifying the type of contract that the application and the system establish. Two types of schemes are proposed: (1) *best effort*, for which the system will match the requirements without strong guarantees, i.e., breaches of QoS contracts are possible, reported, but do not stop the execution of the application; (2) *guaranteed*, for which the system will match the requirements with strong guarantees, i.e., the application will stop in the event of a breach.

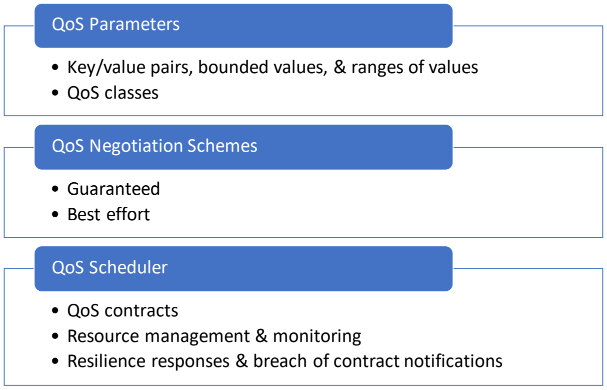
QoS Parameters
• Key/value pairs, bounded values, & ranges of values
• QoS classes
QoS Negotiation Schemes
• Guaranteed
• Best effort
QoS Scheduler
• QoS contracts
• Resource management & monitoring
• Resilience responses & breach of contract notifications

Fig. 2. Core components of a QoS-aware parallel programming model runtime

The QoS Scheduler instantiates QoS parameters and resilience strategies, deploying a QoS contract that relies on system services (e.g., for monitoring of task offloading and error/failure detection), as well as resource allocators (e.g., for deploying a task on a specific GPGPU). The QoS scheduler ensures that everything complies with the QoS contract. If a discrepancy is observed, a breach of contract will be raised (software exception). This generates an event that activates the configured responses, such as resilience actions. Application developers are able to specify a function (handler) that would be automatically called by the QoS scheduler upon a breach of a QoS contract. This enables a programmatic way to handle breaches of QoS contracts when custom actions are required, without imposing complex modifications of the application's code. Fig. 3 presents an overview of the core components that are involved for the specification, implementation and control of QoS contract.

Our design requires coordination between the QoS and OpenMP runtimes. Such an inter-runtime coordination requires the following capabilities: (i) *notifications*, e.g., in order to guarantee progress, a runtime should be able to raise an event to generically notify another runtime/library for coordination purposes (e.g., resource management); and (ii) a *key/value store* shared by runtimes/libraries, for example to store and load QoS parameters. Fortunately, the

PMIx [4] standard supports these features and existing libraries can easily be extended to be PMIx compliant by using the PMIx reference implementation.

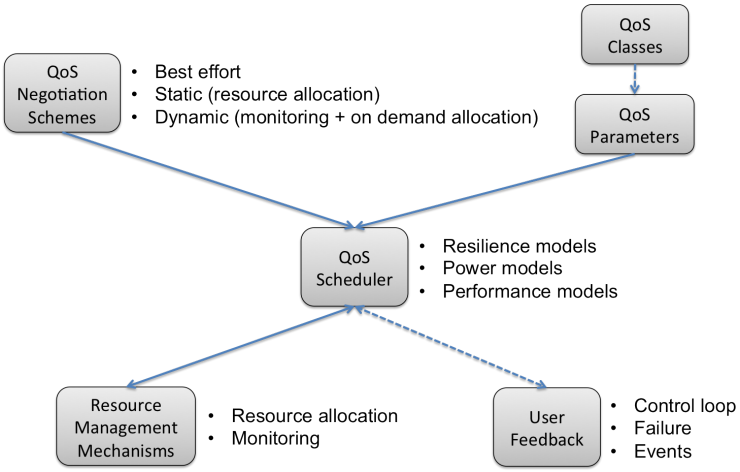

Fig. 3. Schematic overview of the QoS solution

5.2 Implementation details

As previously stated, our QoS library, ORQOS, developed to provide the QoS runtime capabilities, is based on PMIx. We also extended the OpenMP runtime to be PMIx-compliant, which ultimately enables inter-library communication and coordination. Practically, our prototype is therefore composed of our QoS library, ORQOS, and an extension of OpenMP based on the LLVM 7.0.0 release. Specifically, the QoS directives and clauses for OpenMP were added to `clang` and LLVM. As a result, the QoS library is fairly easy to maintain because of its limited size and can potentially be reused in a different context. For example, we are considering reusing it with other programming languages, such as MPI. Similarly, the LLVM extensions remain fairly limited and easy to maintain.

Fig. 4 shows the workflow for compiling OpenMP code with QoS extensions. When the OpenMP code is compiled, it is transformed into an intermediate code with the QoS directives converted into calls to ORQOS. These calls perform two tasks: (i) initialize PMIx to permit data exchange between libraries through its key/value store; and (ii) store QoS key/value pairs to make them accessible to other runtimes. After generating the intermediate code, LLVM creates the binary with all the required library dependencies, including PMIx and ORQOS. At runtime, a PMIx server that is hosting the key/value store is implicitly created when the ORQOS and OpenMP runtimes connect to it. This enables inter-runtime coordination through PMIx key/value pairs and PMIx events. Monitoring and enforcement of QoS contracts is implemented only in the ORQOS runtime, limiting the need for further modifying other components.

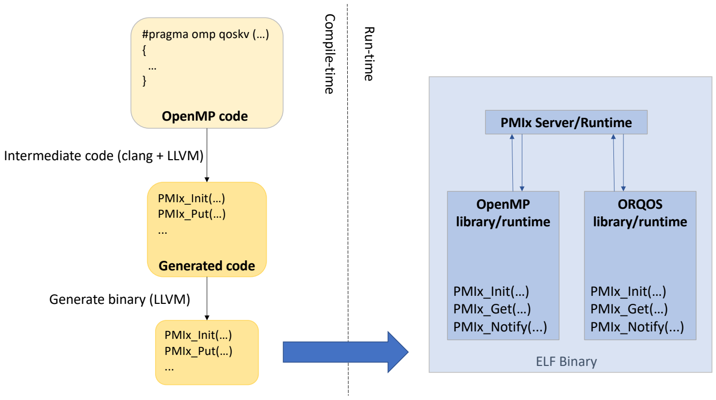

Fig. 4. Compile-time workflow and run-time interactions of the prototype using LLVM.

6 Related Work

The current state of practice for HPC resilience is global application-level checkpoint/restart. It is a single-layer approach that burdens the user with employing a strategy at extreme coarse granularity, i.e., the job level. Part of the current state of practice for HPC resilience are also hardware solutions at extreme fine granularity, such as ECC for memories, redundant power supplies, and management systems for monitoring and control.

The current state of research is more advanced and includes fault-tolerant MPI, fault-aware MPI, redundant MPI, proactive fault tolerance, containment domains, and resilient algorithms. MPI solutions provide resilience at process granularity. Fault-tolerant MPI [3] and fault-aware MPI [10] require global reconfiguration and either local or global recovery. Redundant MPI [8] has significant overheads. Containment domains [5], or sometimes referred to as recovery blocks, use finer-grain checkpoint/restart strategies, such as at the sequential execution block level (e.g., task) or the parallel execution block level (e.g., parallel loop, iteration or application phase). There is also fine-grain transaction-based application-level checkpoint/restart with the FTI [2] essentially implements containment domains at the parallel execution block level. Resilient algorithms [7,6,18,1] utilize data redundancy, computational redundancy, or self stabilization. Individual solutions tend to be algorithm specific.

Resilience Oriented Language Extensions (Rolex) [12] offers C/C++ data type qualifiers for resilience and C/C++ pragma directives for fault tolerant execution blocks. While developed independently from OpenMP, Rolex does offer OpenMP-like resilient programming. It does not offer support for heterogeneous systems. Rolex is also not transparent, as it requires the application programmer to specify resilience strategies in detail. Another OpenMP pragma-based resilience scheme explored in DIvergent NOde cloning (DINO) [17] focuses on data protection by immediately performing correctness check after the last use of a variable based on a vulnerability factor metric. This scheme is limited to soft errors in memory.

There is only preliminary work in resilience for heterogeneous systems. VOCL-FT [16] offers checkpoint/restart for computation offloaded to GPGPUs using OpenCL. VOCL-FT transparently intercepts the communication between the originating process and the local or remote GPGPU to automatically recover from ECC errors experienced on the GPGPU during computation.

7 Conclusion

This paper is motivated by experiences with GPGPU errors and failures from the largest heterogeneous HPC system in the world. It offers concepts for resilience using target offload as a first step toward resilience in OpenMP. It describes the underlying concepts and terminology and the observed errors and failures. It derives error and failure classes for OpenMP target offload from the observations using the underlying concepts and terminology. This paper proposes a number of improvements to enable OpenMP target offload resilience, including a preliminary prototype design and some implementation aspects using a novel concept for QoS.

Future work includes improving the prototype to demonstrate the proposed improvements on a large-scale heterogeneous HPC system with a scientific application. Its evaluation will use appropriate metrics, such as, ease of use, performance, and resilience. The ease of use evaluation identifies how much effort in terms of additional lines of code and implementation time is required to use the QoS capabilities. The performance evaluation compares an unmodified OpenMP with the developed prototype under error- and failure-free conditions. The resilience evaluation performs error and failure injection experiments and measures the time to correct solution under various error and failure conditions with different QoS contracts. Additional future work will focus on the OpenMP language extensions for QoS, specifically on clearly defining QoS parameters and classes. Other future work could also focus on the reuse of our QoS library in the context of other HPC programming languages. For instance, it would be interesting to investigate whether the concept of QoS could be used in the context of MPI to specify resilience, performance and energy consumption requirements in a portable manner. In this context, users could use QoS contracts to specify requirements at both the application and job level but let the runtime find the best compromise to satisfy all or most of the expressed requirements. If possible this would enable a new set of capabilities without drastically increase the size and complexity of standards such as OpenMP and MPI.

References

1. Ashraf, R., Hukerikar, S., Engelmann, C.: Pattern-based modeling of multiresilience solutions for high-performance computing. In: Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE) 2018. pp. 80–87 (Apr 2018). https://doi.org/10.1145/3184407.3184421
2. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: High performance fault tolerance interface for hybrid systems. In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC11). pp. 1–12 (Nov 2011). https://doi.org/10.1145/2063384.2063427
3. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of MPI communication capability: Design and rationale. The International Journal of High Performance Computing Applications **27**(3), 244–254 (2013). https://doi.org/10.1177/1094342013488238
4. Castain, R.H., Solt, D., Hursey, J., Bouteiller, A.: Pmix: Process management for exascale environments. In: European MPI Users' Group Meeting (EuroMPI'17). pp. 14:1–14:10 (Sep 2017). https://doi.org/10.1145/3127024.3127027
5. Chung, J., Lee, I., Sullivan, M., Ryoo, J.H., Kim, D.W., Yoon, D.H., Kaplan, L., Erez, M.: Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12). pp. 58:1–58:11. IEEE Computer Society Press (Nov 2012). https://doi.org/10.1109/SC.2012.36
6. Davies, T., Chen, Z.: Correcting soft errors online in LU factorization. In: Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC'13). pp. 167–178 (2013). https://doi.org/10.1145/2493123.2462920
7. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: 28th International Parallel and Distributed Processing Symposium (IPDPS'14). pp. 1193–1202 (May 2014). https://doi.org/10.1109/IPDPS.2014.123
8. Fiala, D., Mueller, F., Engelmann, C., Ferreira, K., Brightwell, R., Riesen, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the 25th IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12). pp. 78:1–78:12 (Nov 2012). https://doi.org/10.1109/SC.2012.49
9. Gupta, S., Patel, T., Engelmann, C., Tiwari, D.: Failures in large scale systems: Long-term measurement, analysis, and implications. In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC17). pp. 44:1–44:12 (Nov 2017). https://doi.org/10.1145/3126908.3126937
10. Hassani, A., Skjellum, A., Brightwell, R.: Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 750–755 (June 2014). https://doi.org/10.1109/DSN.2014.78
11. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale (version 1.2). Tech. Rep. ORNL/TM-2017/745, Oak Ridge National Laboratory (Aug 2017). https://doi.org/10.2172/1436045
12. Hukerikar, S., Lucas, R.F.: Rolex: Resilience-oriented language extensions for extreme-scale systems. The Journal of Supercomputing pp. 1–33 (2016). https://doi.org/10.1007/s11227-016-1752-5

13. Meneses, E., Ni, X., Jones, T., Maxwell, D.: Analyzing the interplay of failures and workload on a leadership-class supercomputer. In: Cray User Group Meeting (CUG'14) (Mar 2014), `https://cug.org/proceedings/cug2015_proceedings/includes/files/pap169.pdf`
14. Nie, B., Xue, J., Gupta, S., Engelmann, C., Smirni, E., Tiwari, D.: Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In: International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'17). pp. 22–31 (Sep 2017). https://doi.org/10.1109/MASCOTS.2017.12
15. Nie, B., Xue, J., Gupta, S., Patel, T., Engelmann, C., Smirni, E., Tiwari, D.: Machine learning models for GPU error prediction in a large scale HPC system. In: International Conference on Dependable Systems and Networks (DSN'18). pp. 95–106 (Jun 2018). https://doi.org/10.1109/DSN.2018.00022
16. Pena, A.J., Bland, W., Balaji, P.: VOCL-FT: Introducing techniques for efficient soft error coprocessor recovery. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'15). pp. 1–12 (Nov 2015). https://doi.org/10.1145/2807591.2807640
17. Rezaei, A., Mueller, F., Hargrove, P., Roman, E.: DINO: Divergent node cloning for sustained redundancy in hpc. Journal of Parallel and Distributed Computing **109**, 350–362 (November 2017). https://doi.org/10.1016/j.jpdc.2017.06.010
18. Sao, P., Vuduc, R.: Self-stabilizing iterative solvers. In: Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA'13). pp. 4:1–4:8 (Nov 2013). https://doi.org/10.1145/2530268.2530272
19. Snir, M., et al.: Addressing failures in exascale computing. International Journal of High Performance Computing Applications (IJHPCA) **28**(2), 127–171 (May 2014). https://doi.org/10.1177/1094342014522573
20. Vazhkudai, S., et al.: The design, deployment, and evaluation of the CORAL pre-exascale systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'18). pp. 52:1–52:12 (Nov 2018). https://doi.org/10.1109/SC.2018.00055
21. Zimmer, C., Maxwell, D., McNally, S., Atchley, S., Vazhkudai, S.S.: GPU age-aware scheduling to improve the reliability of leadership jobs on Titan. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'18). pp. 7:1–7:11 (Nov 2018). https://doi.org/10.1109/SC.2018.00010