Incorporating Fault-Tolerance Awareness into System-Level Modeling and Simulation

Trokon Johnson
Electrical and Computer Engineering Department
The University of Florida
Gainesville, Florida, USA
johnson2319@ufl.edu

Herman Lam
Electrical and Computer Engineering Department
The University of Florida
Gainesville, Florida, USA
hlam@ufl.edu

2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS) | 978-1-6654-2059-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/FTXS54580.2021.00008

***Abstract*—As the design space for high-performance computer (HPC) systems grows larger and more complex, modeling and simulation (MODSIM) techniques become more important to better optimize systems. Furthermore, recent extreme-scale systems and newer technologies can lead to higher system fault rates, which negatively affect system performance and other metrics. Therefore, it is important for system designers to consider the effects of faults and fault-tolerance (FT) techniques on system design through MODSIM.**

BE-SST is an existing MODSIM methodology and workflow that facilitates preliminary exploration & reduction of large design spaces, particularly by highlighting areas of the space for detailed study and pruning less optimal areas. This paper presents the overall methodology for adding fault-tolerance awareness (FT-awareness) into BE-SST. We present the process used to extend BE-SST, enabling the creation of models that predict the time needed to perform a checkpoint instance for the given system configuration. Additionally, this paper presents a case study where a full HPC system is simulated using BE-SST, including application, hardware, and checkpointing. We validate the models and simulation against actual system measurements, finding an average percent error of less than 17% for the instance models and about 20% for system simulation, a level of accuracy acceptable for initial exploration and pruning of the design space. Finally, we show how FT-aware simulation results are used for comparing FT levels in the design space.

***Index Terms*—fault-tolerance aware system design and evaluation, System-level modeling and simulation, high-performance computing, design space exploration**

I. Introduction

Designing high-performance computing (HPC) systems involves evaluating the various options (e.g., hardware components, applications, libraries) within a design space to meet system objectives, such as performance, cost, and power. Recently, the growth of this design space has accelerated, due to trends such as larger systems (e.g., greater node count, greater bandwidth, larger application problem size) and more system diversity (e.g., co-processors, newer programming models). This growth places greater importance on the field of modeling and simulation (MODSIM), which can facilitate system design by predicting system metrics and behavior before and throughout system design and construction, leading to a more informed design process. Different MODSIM methods address this large design space with various techniques, each with different strengths and weaknesses. At one extreme, cycle-accurate simulators can provide extremely precise and detailed results for individual components and sub-systems, but the complexity and simulation time increases very quickly with system scale. At the other end of the spectrum, a purely analytical modeling approach provides quick results turnaround, but can fail to capture more nuanced system behavior and interaction. There a spectrum of MODSIM techniques with advantages and trade-offs.

The trends driving design space growth are predicted to lead to an increased number of system faults as well, further complicating the design space [1]–[3]. System faults can cause errors that negatively affect system operation, such as application crashes or data corruption. Many systems employ a wide range of fault-tolerance (FT) techniques to mitigate these negative effects [4]–[6]. Common techniques include checkpoint-restart (C/R), error correcting codes (ECC), algorithm-based fault-tolerance (ABFT), and process replication [7], [8]. However, fault-tolerance techniques incur some degree of overhead by consuming additional system resources. This overhead can directly or indirectly affect system metrics and behavior, such as an FT technique that adds compute time or memory requirements to an application, reducing the effective performance or scaling behavior. In some cases, increasing the level of parallelism can reduce performance, as the negative effect of additional fault sources overshadows additional parallelism. [9], [10]. As a result, it is important to better understand the balance between the benefit and cost of various FT techniques, in order to predict their effects on performance and other objectives. Critically, this balance depends on how the system components, i.e., the architecture, application, and fault-tolerance techniques, interact and affect each other. For example, GPUs have different fault vulnerabilities, such as a higher potential for data corruption to spread [11], [12]. This, combined with GPUs different performance profiles, affect the cost-benefit relationship evaluated during DSE. Therefore, when performing MODSIM for emerging and future systems, it is increasingly important to investigate how the relationship between architecture, application, and fault-tolerance affect the

This work is supported by the U.S. Department of Energy, National Nuclear Security Administration, Advanced Simulation and Computing Program, as a Cooperative Agreement under the Predictive Science Academic Alliance Program, under Contract No. DE-NA0002378. The work was also supported by the National Science Foundation under grant CNS-1718033.

978-1-6654-2059-4/21/$31.00 ©2021 IEEE
DOI 10.1109/FTXS54580.2021.00008

system.

This paper presents an FT-aware extension to our previous work, BE-SST, a MODSIM tool, methodology, and workflow developed at a DOE PSAPP-II Center and used for multiple validated full system experiments [13]. BE-SST was developed as a coarse-grained, multi-level MODSIM methodology and workflow, and is used to accelerate the exploration and reduction of a large design space. BE-SST achieves rapid DSE through Behavioral Emulation (BE), an approach to MODSIM that uses coarse-grained models of applications and architectural components to construct and simulate candidate HPC system designs [13]. This abstract modeling approach allows for faster DSE by accelerating both the process of creating models and simulating systems, while maintaining an acceptable level of accuracy for design space exploration and reduction.

Specifically, this paper presents an expansion to the Model Development phase of the BE-SST workflow, allowing the creation of models for different levels of fault-tolerance via checkpointing. Additionally, we demonstrate the methodology by presenting an analysis and validation of the models at the function level. These validated models are then used in a system-level case study using the BE-SST simulator. The simulation results using the FT-aware system models are compared to the original system models, which are not FT-aware, in order to demonstrate the impact of fault-tolerance awareness predictions of system performance and scalability, and, thus, DSE.

The contributions of this paper are as follows:

- A presentation of the fault-tolerance aware extensions to our existing coarse-grained MODSIM methodology, workflow, and simulation platform, as well as plans for additional extensions.
- An analysis and validation of the performance models for the application and checkpointing, based on real benchmarking data. The models are also used for prediction of the performance of the functions running on larger, notional systems. The level of accuracy allows for design space pruning & further study with more detailed simulators.
- A full system case study using the BE-SST simulator, contrasting the FT-aware and non-FT-aware system predictions with different levels of checkpointing for a full application run on Quartz, an HPC system housed at Lawrence Livermore National Lab.

The rest of the paper is organized as follows. Section 2 discusses related works, particularly other MODSIM research efforts that are also fault-tolerance aware. Section 3 describes the background on the MODSIM workflow that we extend, BE-SST, as well as methodology for incorporating fault-tolerance awareness into that workflow. Section 4 presents a DSE case study using the extended workflow, including the experimental setup, comparison of non-FT-aware and FT-aware performance models, validation and analysis of the models, and the use of these models for scaling predictions via BE-SST simulation. Section 5 summarizes our contributions and details our future directions.

II. Related Works

There are a large variety of MODSIM techniques used for HPC system DSE, often to maximize system performance. Because of the recognition of faults as a performance concern, recent works have begun to integrate fault-tolerance awareness into their performance models to better optimize runtime [9], [10], [14]–[16]. In this section, we will discuss these fault-tolerance aware MODSIM approaches.

Cavelan, et al. [15] modifies Amdahl's law to find the optimum number of processes to minimize execution time while considering the effects of faults and the overhead of checkpoint-restart. Zheng, et al. [9], [10] perform similar studies, but extends Gustafson's law in addition to Amdahl's, to model and predict strong and weak scaling. Both works verify these models with fault injection simulations. These works highlight the significant effect of faults on performance by showing the difference between model predictions for a system assumed to be fault free, a system with faults, and a system with faults and fault-tolerance. They find that faults can greatly decrease the predicted system speedup, but checkpoint-restart can mitigate these losses. An important finding of these works is that while the original Amdahl's and Gustafson's laws have monotonically increasing speedup as the number of nodes increases, additional nodes can also increase the system-wide failure rate. As a result, an increase in number of nodes can actually result in a decrease in performance. While these works offer great insight into how faults affect performance on HPC systems, their basis in Amdahl's and Gustafson's law make the works very abstract, though applicable to many systems. By leveraging BE-SST's more concrete MODSIM approach, our methodology workflow for predicting performance under faults is more specific and can capture more nuanced system behavior.

Hussain, et al. [14] builds upon the idea of FT-aware performance models by modifying Amdahl's law to include dual replication as a fault-tolerance technique, along with checkpoint-restart. Through analytical models and simulations, they find that replication allows for a greater maximum speedup than checkpoint-restart alone, due to the higher fault rate of more nodes being mitigated. This work differs from others in that while replication is a well-known fault-tolerance technique, it is much less studied than C/R. Similar to the works mentioned earlier, our work is more concrete, capturing more of the system behavior. However, we focus on more studied C/R techniques.

Jin et al. [16] presents an analytical queuing model to maximize application performance. They seek to optimize performance by finding optimal values for processes, checkpoint frequency, and spare nodes. They also use this model for exploring the design space and giving directives for scalable HPC systems. This work is similar to ours, but, like others, takes an abstract view of applications, whereas we gather application benchmark data from real systems to build

our models. Additionally, the analysis of spare nodes as a fault-tolerance method is helpful in understanding how less prevalent fault-tolerance methods affect performance, as C/R is already widely studied.

III. Methodology

As previously stated, this work involves extending our current BE-SST methodology to be fault-tolerance aware. BE-SST is a simulator framework and coarse-grained model development methodology and workflow developed at the DOE PSAAP-II Center for Compressible Multiphase Turbulence [13]. The simulator leverages the Structural Simulation Toolkit (SST) from Sandia National Lab, which provides a framework for component-based, parallel discrete-event simulation [17]. The behavioral emulation (BE) MODSIM philosophy focuses on abstracting away fine-grained system details, thereby accelerating the model generation process and reducing simulation time. However, BE-SST can use models at various levels of granularity to more finely balance speed and accuracy. This in turn facilitates faster evaluation of large numbers of candidate system designs in the design space while still maintaining acceptable accuracy. After the reduction of the design space, slower, more detailed fine-grained simulators can be used to further evaluate the promising candidate designs, if necessary.

BE-SST provides a distributed parallel simulation library for Behavioral Emulation, as well as the framework for developing coarse-grained BE models. The workflow supports plug-and-play DSE with different architecture and application subsystems (e.g., node architecture, interconnect topology, kernel libraries, etc.) which facilitates studying abstract, emerging, and notional systems. Fig. 1 shows previously published benchmarking and validation results collected using BE-SST workflow and simulator as an example DSE case [13]. Here, BE-SST is being used to simulate the Vulcan HPC system, previously operational at Lawrence Livermore National Laboratory. The application under test is CMT-bone, a proxy app version of CMT-nek, which is based on the Nek5000 computational fluid dynamics solver [18]. The scatter plot on the left side of Fig. 1 shows both actual benchmarked performance results (in orange) and simulation results (in blue) of the system. Execution time is measured and simulated for different parameter combinations of MPI ranks and application problem size. Furthermore, because actual machine performance is non-deterministic due to noise and other factors, BE-SST implements Monte Carlo simulations to capture the variance that exists in the calibration samples, to better emulate real machine behavior. As such, each of the points on the graph represent a distribution of results for performance runtime of one timestep. This is illustrated by the pop-out on the right side of Fig. 1.

Additionally, once simulations have been validated for existing machines, BE-SST can be used to perform predictions of notional machines. An example of a simple notional extension to an existing machine is to increase the number of ranks beyond the number of cores in the actual machine. This can be seen in Fig. 1, where validation was performed up to

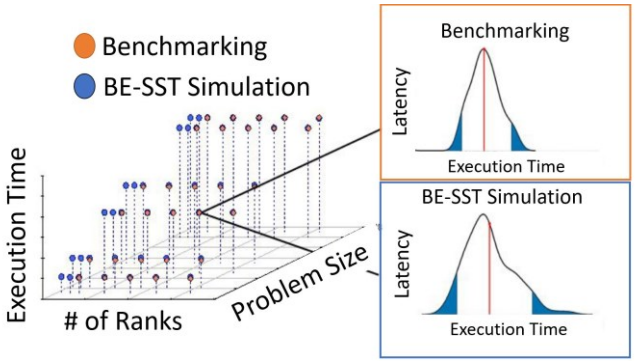

Fig. 1. BE-SST Simulation Results used for DSE of CMT-bone-BE Proxy Application on Vulcan Supercomputer [13]

our allocation of 128,000 cores (blue and orange points) and predicted up to 1 million cores (blue only). Physically, Vulcan had a total of 400,000 CPU cores. However, we were able to perform simulations of Vulcan's architecture up to the max size of the machine, and beyond, by using models validated at smaller sizes. This capability of BE-SST allows us to explore more hypothetical areas of the design space.

BE-SST also facilitates DSE through the plug-and-play nature of SST to perform notional system simulation. With BE-SST, models from different machine subsystems (e.g., node architecture, network topology, memory system, etc.) can be used together to construct and simulate full notional system designs. The Behavioral Emulation (BE) workflow, illustrated in Fig. 2, consists of two major phases: (1) Model Development on the left, which includes the design, validation, and calibration of performance models, and (2) Hardware/Software Co-Design on the right, which uses the validated models from the previous phase for full system simulation and performance prediction. The rest of this section will give in-depth explanations of the two workflow phases, as well as the fault-tolerance awareness extensions to each phase.

A. Model Development Methodology

The individual steps of the Model Development phase can be seen in the left side of Fig. 2. The output models of this phase are *Behavioral Emulation Objects* (BEOs) for both the application and the hardware architecture, which are called AppBEOs and ArchBEOs, respectively. An *AppBEO* is a list of abstract instructions that represents the major functions and control flow of the application under study. An *ArchBEO* describes the system hardware architecture that is simulated, defines system operations, and connects the performance models to the instructions listed in the AppBEO. To create an AppBEO, we identify the significant computation blocks and communication patterns in the source code at the desired granularity level. These are then translated into the abstract instructions for the AppBEO. The instructions are created to only accept parameters that affect the performance, which can be decided from previous benchmarking or knowledge of the application. When an abstract instruction is executed

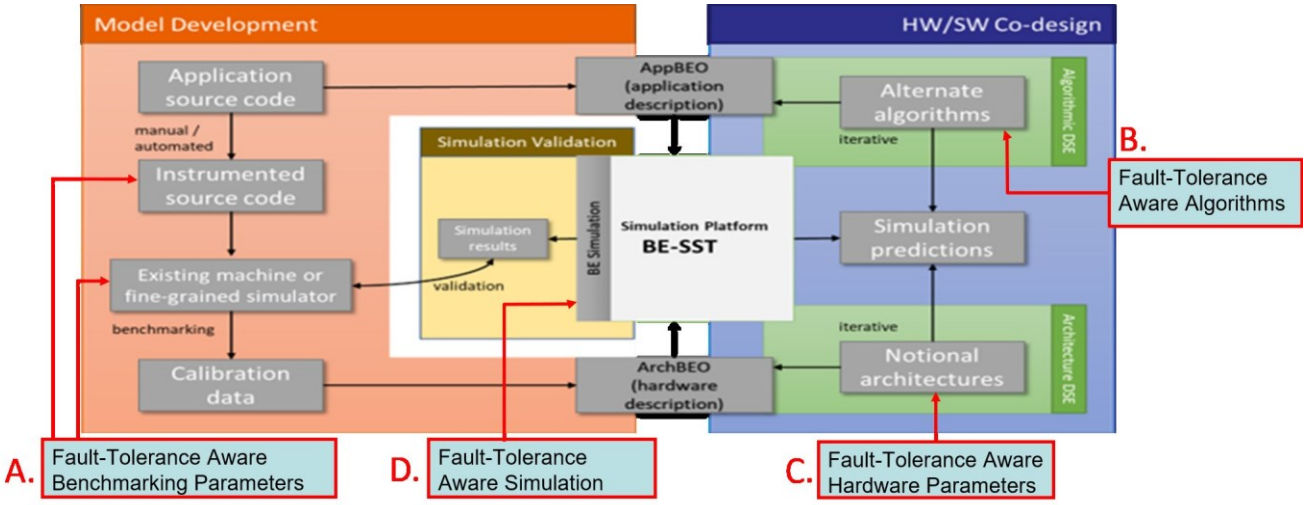

Fig. 2. Existing BE-SST Workflow, with Fault-Tolerance Aware Extensions in Red

by the BE-SST simulator, the simulator calls the associated performance model to return the predicted run time, rather than actually carrying out what could be a costly computation or communication call during real execution. The simulation clock is then updated to reflect the time.

To create an ArchBEO, we begin by instrumenting the application code under study with timer calls corresponding to the same blocks and patterns used for the AppBEO and running the code on existing machines or fine-grained simulators to collect benchmarking data as shown in the left side of Fig. 2. We collect multiple timing samples for each system parameter combination in the design space to account for system noise and other sources of variation. These samples are used to create models using one of two currently implemented methods: interpolation and symbolic regression. For our interpolation method of modeling, the training data is organized into lookup tables based on the corresponding system parameters. When a function from the AppBEO is called during simulation, the corresponding lookup table is searched for the function arguments, and one of many samples is selected for a runtime prediction. If the parameters in the current function call do not have an existing sample for this combination, the simulator estimates a value by using one of several implemented methods to interpolate a data point between two existing data values. We also use a second method of modeling, which is used in the case study experiments presented in this paper, based on symbolic regression [19]. In the symbolic regression method, the benchmarking data is split into training data and testing data. The training data is used as input to our symbolic regression tool to create models through an iterative process. The testing data is used to evaluate model accuracy at each iteration.

With either method, (interpolation or symbolic regression) the simulator returns a predicted execution time for each function call. The modeling process is designed to incorporate and mimic the interactions of the whole system (e.g., hardware architecture, libraries, interconnect, etc.) into the performance models. The simulator then advances the simulation time by the amount predicted. This holistic modeling approach allows for more accurate results for system-level DSE.

In order to develop FT-aware performance models, the existing BE-SST Model Development process was extended to add fault-tolerance aware computation and communication blocks to the BEOs, as well as additional FT-aware benchmarking parameters. These extensions to the model development process are visualized in the left side of Fig. 2 (labelled ”A”). The goal is to maintain the creation of models that capture interactions between hardware and software, but now include interactions with a new dimension of the design space, fault-tolerance. The specific alterations vary depending on the fault-tolerance techniques being benchmarked or simulated. The fault-tolerant version of an application may include changes to the source code, such as alternate algorithms that are more resilient, or checkpointing subroutines. Changes may extend to the control flow structure as well. As a result, the AppBEO must be updated to include the abstract instructions to reflect these source code changes. Also, the changes must be included in the instrumentation process to collect benchmarking data used to develop the corresponding ArchBEOs and their performance models.

Fig. 3 provides a visual example of a simple fault-tolerance extension to an application, as well as visualizing how the Model Development process is affected. A common fault-tolerance strategy for applications is checkpoint-restart, where critical application data is periodically backed up, to be restored in the event of an error or crash. The performance of this added checkpointing function must be modeled for this application. Moreover, the parameters that will affect this checkpoint function must be identified. These parameters may not have been factors for any functions in the non-fault aware versions of the application. For example, a parallel application that consists primarily of local, compute bound

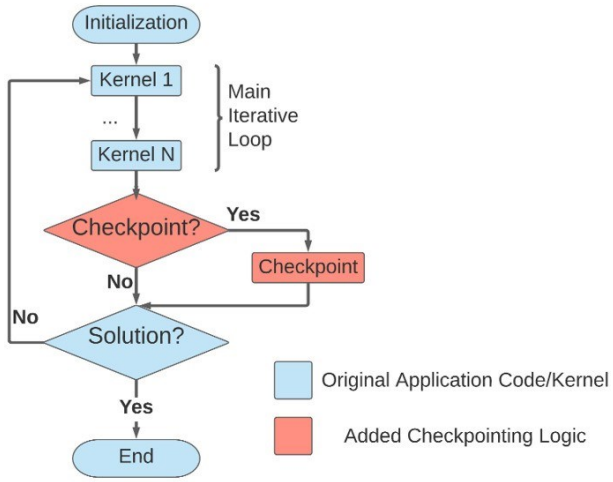

Fig. 3. Fault-Tolerance Aware Iterative Solver

functions may demonstrate good weak scaling performance; as such, the number of nodes has a minor effect on overall application time to solution if no FT method is used. However, a coordinated checkpoint-restart function can introduce communication overhead, which is much more dependent on the number of ranks, and network limitations [20]. Given the size of the FT design space, consisting of different strategies, which themselves consist of different libraries and implementations, this can lead to multiple performance models for the same application, all of which can be explored and compared in the Co-Design phase.

B. Hardware/Software Co-Design Methodology

The BEOs and models developed and validated from the Model Development phase can be used in the Co-Design phase for design space exploration (DSE) of applications on existing architectures (algorithmic DSE), as shown in the right side of Fig. 2 (labelled ”B”). Furthermore, by modifying and extending the ArchBEO simulation parameters (e.g., network bandwidths, latencies, or topology) or replacing BEOs models with other validated BEO models, it becomes possible to perform architectural DSE, including DSE of notional systems.

Algorithmic DSE involves interchanging models to determine how different algorithms affect the performance of the overall application. Different algorithms can serve similar functions but have different scaling behaviors, depending on their design, the system, and other factors. For example, an application that includes computation of a Fast Fourier Transform can have several algorithms from which to choose [21]. This could range from the commonly used Cooley-Tukey algorithm to a bespoke implementation provided by the FFTw libraries. These two example algorithms may have performance profiles that differ depending on the size and type of the input dataset (e.g., real vs. complex, single-precision vs. double-precision) or the details of the node architecture (e.g., cache size, accelerator). Once the models for multiple algorithms have been created, we can use simulation to determine their performance under different conditions and recommend one for the system without having to run on the system. This can be helpful if runs are prohibitively long, or if the system does not yet exist.

To include fault-tolerance awareness for algorithmic DSE, specific fault and FT parameters should be considered. As an example, C/R has its own set of parameters separate from the application, including implementation, scale, memory/storage level, etc. These parameters can interact with the application, hardware and software design spaces, which ultimately affects the performance of the entire system. In the case study of this paper, we show an example of how an FT-aware application can have its checkpointing cost modeled and evaluated, and how the application parameters affect checkpoint behavior.

Additionally, including fault-tolerance awareness in the architecture under study requires incorporating FT-aware hardware parameters, such as hardware fault rates and recovery times, into the ArchBEOs, as shown in the bottom right of Fig. 2 (labelled “C”). Different hardware components (e.g. processors, memory technologies, etc.) have failure rates that can be found through various means, such as documentation or failure logs [3]. Co-design involves balancing hardware and software trade-offs to maximize performance, but changing system scale, hardware architecture and algorithms are all decisions that can affect the fault rate and fault-tolerance of a system and can therefore affect system-level performance.

Other fault-tolerance techniques can be added for more intentional fault-tolerance aware algorithmic design space exploration, such as algorithm-based fault-tolerance (ABFT). ABFT takes the form of alternate algorithms that perform the same operations but with more resilience and overhead, such as using a checksum in a matrix-based code to guard against silent data corruption. This can lead to direct performance overhead due to time needed to compute the checksum, or indirect overhead due to memory usage to store the checksum information, which harms performance. These factors can vary by application and parameters, which requires more trade-offs for study.

C. BE-SST Simulator

The BE-SST simulator (shown in the middle of Fig. 2) must be extended to integrate fault-tolerance awareness for the system-level simulation, shown at the bottom of Fig. 2 (labelled ”D”). This consists of two steps: *FTA model integration*, and *fault injection/simulation capability*. Currently, BE-SST performs simulations of systems without fault-tolerance awareness or fault injections; this is visualized as Case 1 in Fig. 4. The simulator “executes” the abstract instructions in the AppBEO. Each instruction in the AppBEO causes the simulator to poll the ArchBEO to determine the runtime for that event and advance the simulator clock for that rank, communicating with other ranks if necessary. Our work integrating FT-aware models into BE-SST allows for simulating Case 3, systems with FT-aware performance models. By doing so, we can

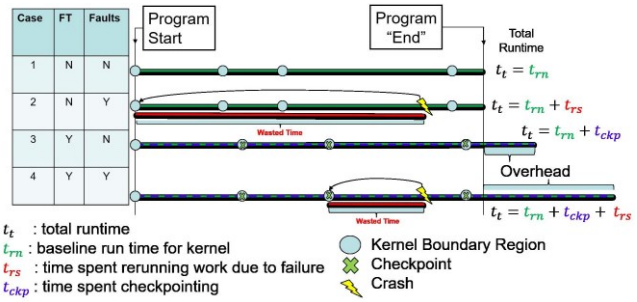

Fig. 4. Different Fault Assumption Cases for BE-SST DSE

determine expected system scaling behavior based on the different overhead of fault-tolerance techniques and parameters. As previously noted, with fault-tolerance becoming a greater concern in emerging systems, this feature becomes a more important addition to BE-SST. Moving forward, adding the capability to inject faults into the BE-SST simulator will allow the simulation of Case 2, systems with different fault profiles, and Case 4, systems with both faults and fault-tolerance. Adding both fault awareness and fault-tolerance awareness allows for added versatility for exploring a fault aware design space.

The case study in the following section presents results of the FT-aware extensions to the Model Development phase of the BE-SST workflow, along with the extensions to algorithmic DSE in the Co-Design phase. Together with our ongoing work on the other proposed extensions to the co-design phase, (i.e., architecture DSE and BE-SST simulator capabilities to support fault injection and fault aware simulation) these extensions will allow greater design space exploration by modeling and simulating how fault-tolerance design choices influence and interact with hardware and software design choices, how FT is affected in return, and how these choices and interactions affect performance and other metrics.

IV. Fault-Tolerance Aware DSE Case Study

This section presents a case study illustrating FT-aware DSE enable via the aforementioned fault-tolerance extensions BE-SST.

A. Experimental Setup

The target architecture for our case study is Quartz, an Intel Xeon machine housed at Lawrence Livermore National Laboratory (LLNL). The machine consists of 2,988 nodes, each with 2 Intel Xeon E5-2695v4 CPUs, for 36 total cores, and 128 GB of memory. The nodes are connected in a two-stage bidirectional fat-tree topology using Omni-Path interconnect technology.

The application used for the case study is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) application, which was developed as part of the DARPA Ubiquitous High Performance Computing (UHPC) program [22], [23]. LULESH was designed as a compute

TABLE I
Checkpointing Levels of the Fault Tolerance Interface (FTI)

Checkpoint Level	Checkpoint Method
Level 1	Checkpoint file saved on local node
Level 2	Checkpoint file saved on local node AND sent to neighbor node in group
Level 3	Checkpoint files encoded via Reed-Solomon (RS) erasure code
Level 4	All checkpoint files flushed . to parallel file system

focused, weak scaling hydrodynamic proxy application, which lends itself to high-performance computing design space exploration. It has been used to explore traditional and emerging programming models, and has been implemented in over 10 languages and programming models, including C++ with MPI+OMP, the version used in this case study [24]. LULESH has one main parameter: the problem size, or elements per rank (epr). The problem size determines how many individual elements, or spatial regions, will be assigned to each rank for computation. The other parameter used for our case study is the number of MPI ranks, which allows solving hydrodynamic problems with a larger spatial domain or higher resolution. The overall cubic domain for the entire application run is divided into a single cubic subdomain per rank, and the subdomains are in turn divided into elements, corresponding to the problem size parameter. Because of the decomposition algorithm divides the cubic subdomain into smaller cubes, LULESH is limited to running only on a number of ranks that are perfect cubes (e.g., 8, 27, 64, ...).

The Fault Tolerance Interface (FTI) is a checkpointing library which provides access to multiple levels of checkpointing, fault-tolerance regions, and other parameters that can be tailored towards a system [25]. FTI includes 4 different checkpointing levels, which determine how the checkpoint is stored, including the level of fault-tolerance (i.e., the amount and kinds of failures that can be tolerated by a system). These checkpoint files allow an interrupted application to be resumed at a later time, either after the failure has been corrected, or using new hardware. Generally, as the levels increase from 1 to 4, the resilience of the system increases as well. However, the performance overhead and demand on system components increase as well. Each level allows for parameters such as the checkpointing frequency to be set independently, allowing for flexibility in balancing fault-tolerance and performance overhead.

Table I acts as a quick reference of each checkpoint level. Level 1 saves the checkpoint file to the node locally. If the node experiences a failure that halts its progression, the application can restart from the most recent successful checkpoint on all nodes. Level 2 and level 3 both make use of FTI groups, which are collections of nodes that keep their own checkpoints, as well as checkpoint files from other nodes in the group. This creates semi-independent fault-tolerant regions that can tolerated multiple failures between them. The number of nodes per group is set using FTI's groupsize parameter. For

Level 2 checkpointing, each node saves its checkpoint locally, as in Level 1, but also sends it to two neighboring nodes within the group. Therefore, if a node fails and loses its checkpoint file, recovery is possible as long as one of the two neighboring nodes retain the copy of the lost checkpoint. Level 3 checkpointing uses Reed-Solomon codes to encode and partition a single node's checkpoint file among all members of a group. If any node fails and loses its checkpoint file, the file can be recreated through the encoded partition on the group's other nodes. Through this process, any FTI group, and thus the application, can tolerate and recover from up to ½ of the nodes concurrent failures and loss of checkpoint in one group and still recover. Level 4 checkpointing involves flushing the checkpoint to the parallel file system (PFS), where checkpoints are the least likely to be lost.

As previously stated, each level of checkpointing has a different amount of overhead, and can recover different kinds and numbers of failures. These different checkpointing levels and parameters change the resilience and performance profiles of the application, and the system as a whole, growing the design space. The performance overhead depends not only on FTI parameters, such as the group size and checkpoint frequency, but also, indirectly, on other system parameters. This can include the level of parallelism (e.g., the number of ranks and nodes used to run the application) and application parameters (e.g., problem size increasing the amount of data saved in a checkpoint file). Additionally, the speed of system components, such as local storage (Level 1), communication and network congestion (Level 2), computational performance (Level 3) and write speed to the parallel file system (Level 4) also affect overhead, depending on which levels are implemented. System performance parameters and fault rates can determine what level of fault-tolerance is necessary to optimize performance. As a result, these new parameters and operations further increase the scope of the design space. The exploration of this expanded, fault-tolerance design space using BE-SST's accelerated modeling and simulation workflow is presented in the remainder of section IV.

The version of LULESH with FTI integration that was used for this paper was found in a publicly available GitHub repository owned by Maxime Kermarquer [26]. For this work, we focused on the checkpointing levels 1 and 2, the levels with the least amount of communication, which do not require extensive communication modeling. While we have constructed and validated communication models for other HPC systems, Quartz requires additional modeling of the Fat Tree network. We intend to model and validate Quartz communication in the future, at which point we can more fully explore the higher levels of fault-tolerance.

As the case study involved measuring runtime for the application and the fault-tolerance routine, this case study falls under Case 3 of Fig. 4 (i.e., fault-tolerance without fault injection), which illustrates the overhead performance cost of using a FT method. The parameters used for the case study are listed in Table II. For this study, we elected to keep both the group and node sized fixed at low values, 4 and 2 respectively.

TABLE II
CASE STUDY PARAMETERS

Parameters	**Values**				
Problem Size (epr)	5	10	15	20	25
Ranks	8	64	216	512	1000
Group Size	4				
Node Size	2				

Changing either FTI parameter could potentially affect communication patterns, and while communication at Level 1 and 2 is minimal, we did not want to risk introducing additional variance without modeling the communication. Additionally, it should be noted that FTI requires the number of ranks to be a multiple of group_size*node_size. Coupled with LULESH's perfect cube number of ranks requirement, we ran on every perfect cube number of ranks that is evenly divisible by 8. All possible perfect cube number of ranks that would fit on our allocation of the Quartz partition were run, maxing out at 1000 ranks. The experiments were run for every combination of problem size and number of ranks, leading to 25 unique parameter combinations.

B. Validation of Performance Models

This subsection presents the validation of the performance models BE-SST workflow. We perform this validation by comparing the ***modeled***, or predicted, runtime for different problem sizes and numbers of ranks against the ***measured*** runtime for the same parameters, using Mean Average Percentage Error (MAPE) as an error metric As previously mentioned, we want to ensure that the models can predict both the performance and the trends of the machine with acceptable accuracy, as the models are used for low-cost simulations. This subsection also contains an example using the models for prediction of larger parameter values than were benchmarked, where both the problem size and the number of ranks are beyond our ability to run on Quartz, acting as a demonstrative step towards notional system prediction.

Figs. 5-6 present the scaling behavior for LULESH and the 2 levels of checkpointing. Both figures present the same data, with Fig. 5 showing scalability primarily by problem size, and Fig. 6 by number of ranks. In both graphs, the vertical black dashed line demarcates the boundary between validation on the left (both benchmarked and modeled data), and prediction on the right (only modeled data). The predicted region of Fig. 5 predicts runtime of a larger problem size, simulating a notional system with more memory per node. The prediction region of Fig. 6 simulates a system with 1331 ranks, above the 1000 rank limit we encountered on Quartz.

In Figs. 5-6, it is clear that the relative costs of the functions stay mostly ordered. The LULESH timestep takes the least amount of time and scales the most slightly with either parameter, problem size or number of ranks. This is consistent with LULESH being a computationally focused, weak-scaling mini-app. As expected, both checkpointing levels have a higher time cost, and scale much more quickly with either parameter, most likely due to FTI being a coordinated checkpointing solution

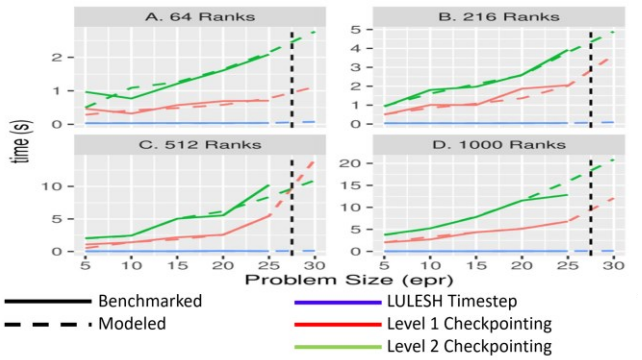

Fig. 5. Model Validation for LULESH_FTI Checkpointing and Timestep Functions vs Problem Size (epr)

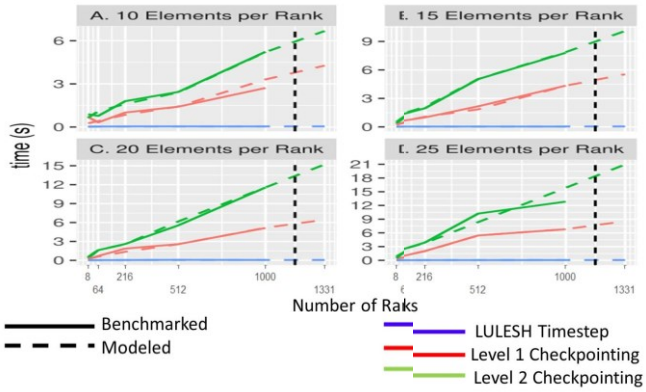

Fig. 6. Model Validation for LULESH_FTI Checkpointing and Timestep Functions vs Number of Ranks

that touches storage and communication, thus scaling with level of parallelism and amount of data to store respectively.

Furthermore, we observe that the model predictions follow the trends for the real data quite well. This can be seen through the MAPE of 6.64% for the LULESH timestep models, and 17% and 15% for L1 and L2 checkpointing, respectively. The checkpoint error may be higher due to complications from communication and secondary storage, or the parameters used for symbolic regression generation, all of which can be refined via finer grained modeling. However, as checkpointing occurs much less frequently than the timestep function in iterative solvers (e.g., 1 checkpoint per 10-100 timesteps) this error is less problematic for overall runtime. This is further verified in the full system simulation runs of the next section.

From Figs. 5-6, it is clear that the models follow the trends of the real data for the majority of the data points, primarily in the center of the graphs. The benchmarks and models diverge at 2 key outlier areas: Figs. 5A, 5D and 6D. These are all extreme areas of design space, which are the most difficult to prune. For Fig. 5A, 8 epr, this area is low priority, as it is a low cost run. It has an extremely small runtime, which is more easily affected by noise. However, actual runs can be used quite easily for data. For the high cost runs, Figs. 5D and 6D, 25 epr and 1000 ranks respectively, these areas are highlighted by BE-SST as areas of interest for more detailed study with fine-grained simulators. However, the models cover the majority of the other points on the design space in this low-cost manner.

TABLE III
MODEL VALIDATION VIA MEAN AVERAGE PERCENT ERROR

Kernel	MAPE
LULESH Timestep	6.64%
Level 1 Checkpointing	16.68%
Level 2 Checkpointing	14.50%

C. Full System Case Study using BE-SST with Fault-Tolerance Awareness

As discussed earlier, BE-SST has been used with performance models to predict full system performance of applications and systems. By using the validated application and checkpointing performance models discussed in the previous section as input to the BE-SST simulator, BE-SST can be used to simulate performance overheads for different levels of fault-tolerance.

Figs. 7-8 show the total application runtime for 200 timesteps under three different fault-tolerance scenarios: 1) no fault-tolerance, 2) level 1 checkpointing 1, and 3) levels 1 & 2 checkpointing. Scenario 1 (shown in blue) is used as the baseline, representing simulation of the system without any fault-tolerance awareness (i.e., the traditional BE-SST workflow). Scenarios 2 & 3 (shown in red and green, respectively) are possible due to the FT-aware extensions to the BE-SST methodology and workflow, and serve to expand the design space further. Both Level 1 and Level 2 have a checkpointing period of 40 timesteps, marked by the black dots on Figs. 7-8.

Similar to the individual functions, the accuracy for the system-level simulations can be seen in how well the simulations follow the benchmarked data, or in the MAPE levels of in Table IV, of 20%, 17% and 14% for no fault-tolerance, L1, and L1 & L2 respectively. This full system simulation case study leads to 2 important insights. **1). Predicting full application performance does not significantly increase error over individual timestep prediction.** The error values of the full system simulations are comparable to the single function runs, as seen in tables III and IV. This is most likely due to aggregate error: since the full system simulation involves BE-SST predicting each individual model call, error can add up for larger predictions. However, as long as the variance is centered around the mean of the prediction, positive and negative error should cancel this out long-term, as Fig. 8 shows more divergence between prediction and measured values as more timesteps pass. **2). The percent error from the FT-aware scenarios are lower than the non-FT-aware scenario.** This is most likely due to the benchmarked and measured values of scenario 1 being much smaller than scenarios 2 and 3, and, therefore more susceptible to machine variation and other sources of noise.

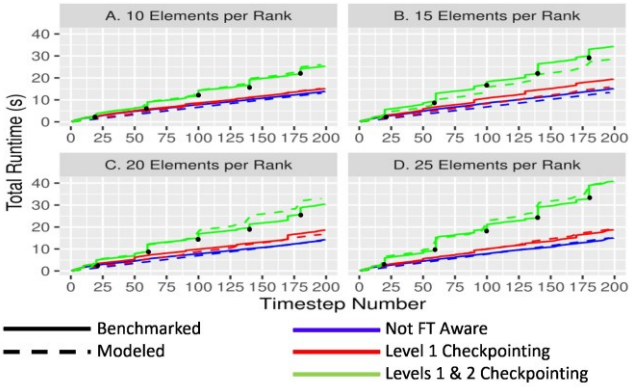

Fig. 7. Full Application Runtime Prediction for 64 Ranks

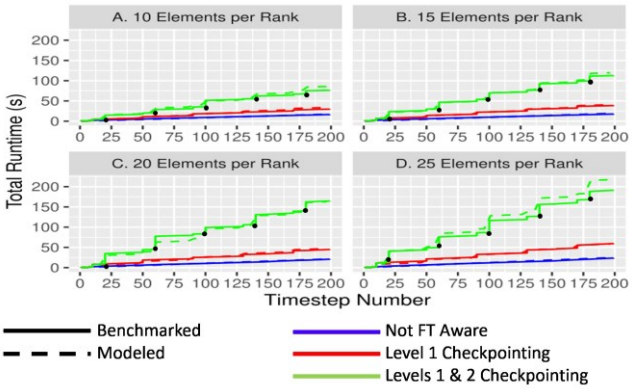

Fig. 8. Full Application Runtime Prediction for 1000 Ranks

As mentioned, a major facet of BE-SST workflow is facilitating DSE. With the full system simulations, the results can be used to predict how much overhead will result from each configuration without having to run the configuration. Fig. 9 demonstrates this by displaying the amount of overhead for different points in the design space based on the problem size, number of ranks, and fault-tolerance level, allowing a quick overview of how these factors affect system behavior.

TABLE IV
VALIDATION FOR FULL SYSTEM SIMULATION

Fault-Tolerance Level	MAPE
LULESH + No Fault-Tolerance	20.13%
LULESH + Level 1 Checkpointing	17.64%
LULESH + Levels 1 & 2 Checkpointing	14.54%

64 Ranks	10	15	20	25	1000 Ranks	10	15	20	25
No FT	100%	109%	103%	108%	No FT	119%	127%	151%	170%
L1	109%	140%	135%	135%	L1	215%	278%	324%	428%
L1 & L2	183%	247%	220%	294%	L1 & L2	550%	810%	1185%	1374%

Fig. 9. Overhead Prediction for Full System Simulation

Using this new extended workflow, BE-SST is now capable of simulating full systems with fault-tolerance. While this case study only looked at one fault-tolerance method and implementation, this methodology opens the door to simulation and evaluation of fault-tolerance aware systems multiple checkpointing implementations, as well as other FT methods such as algorithm based fault-tolerance. This, in turn grants a greater design space to explore. In future work, we plan to implement fault injection, which will allow us to optimize for different fault rates and scenarios as well.

V. CONCLUSION & FUTURE WORK

In this paper, we have presented a methodology to extend our current BE-SST workflow and platform to incorporate fault awareness. A case study was used to demonstrate how the extended methodology can create and validate FT-aware performance models, which predict checkpoint overhead for different system parameters, and use these models for system-level simulation. Specifically, we used LULESH, an HPC proxy application, to show how fault-tolerance aware performance models can be used to perform DSE within the BE-SST workflow. We presented and validated both fault-tolerance aware performance models and checkpoint performance models, with average errors of less than 17% for individual functions, and 21% for full system runs. We analyzed the trends of these models, and showed predictions, demonstrating how they could be used for predictive DSE of notional systems, specifically for use predicting the effect of fault-tolerance on performance.

Currently, we are integrating the HW/SW Co-Design phase of our workflow with fault-tolerance awareness. We will also further our ability to explore the fault-tolerance aware design space by investigating other fault-tolerance techniques. The latter is interesting because BE-SST is already being used to study multiple applications and architectures, but we will incorporate a more formal methodology for including and comparing differing checkpointing libraries and algorithm based fault-tolerant methods, as well as building up those libraries for BE-SST. Finally, by including the capabilities of fault injection and checkpoint-restart into the BE-SST simulator, we can perform full system simulations to determine how both faults and fault-tolerance affect performance predictions and the overall design space.

REFERENCES

[1] N. DeBardeleben, "Extreme scale and bleeding edge technology lead to a need for resilient high performance computing systems," in *2016 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2016, pp. 3B–1.

[2] N. DeBardeleben, S. Blanchard, D. Kaeli, and P. Rech, "Field, experimental, and analytical data on large-scale hpc systems and evaluation of the implications for exascale system design," in *2015 IEEE 33rd VLSI Test Symposium (VTS)*. IEEE, 2015, pp. 1–2.

[3] D. Jauk, D. Yang, and M. Schulz, "Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[4] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of resilience techniques for exascale computing platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 914–923.

[5] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu, "Sdc is in the eye of the beholder: A survey and preliminary study," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 72–76.

[6] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[7] N. B. S. H. N. H. J. D. M. Zounon, "Uniman) algorithm-based fault tolerance techniques."

[8] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 117–130.

[9] Z. Zheng, L. Yu, and Z. Lan, "Reliability-aware speedup models for parallel applications with coordinated checkpointing/restart," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1402–1415, 2014.

[10] Z. Zheng and Z. Lan, "Reliability-aware scalability models for high performance computing," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–9.

[11] A. R. Anwer, G. Li, K. Pattabiraman, M. Sullivan, T. Tsai, and S. K. S. Hari, "Gpu-trident: efficient modeling of error propagation in gpu programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[12] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "Gpgpus: How to combine high computational power with high reliability," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–9.

[13] A. Ramaswamy, N. Kumar, A. Neelakantan, H. Lam, and G. Stitt, "Scalable behavioral emulation of extreme-scale systems using structural simulation toolkit," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–11.

[14] Z. Hussain, T. Znati, and R. Melhem, "Enhancing reliability-aware speedup modelling via replication," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 528–539.

[15] A. Cavelan, J. Li, Y. Robert, and H. Sun, "When amdahl meets young/daly," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 203–212.

[16] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun, "Optimizing hpc fault-tolerant environment: An analytical approach," in *2010 39th International Conference on Parallel Processing*. IEEE, 2010, pp. 525–534.

[17] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[18] T. Banerjee, J. Hackl, M. Shringarpure, T. Islam, S. Balachandar, T. Jackson, and S. Ranka, "Cmt-bone—a proxy application for compressible multiphase turbulent flows," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 173–182.

[19] S. P. Chenna, G. Stitt, and H. Lam, "Multi-parameter performance modeling using symbolic regression," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 312–321.

[20] O. Subasi, F. Zyulkyarov, O. Unsal, and J. Labarta, "Marriage between coordinated and uncoordinated checkpointing for the exascale era," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 2015, pp. 470–478.

[21] Y. Li, L. Zhao, H. Lin, A. C. Chow, and J. R. Diamond, "A performance model for fast fourier transform," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–11.

[22] I. Karlin, "Lulesh programming model and performance ports overview," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[23] I. Karlin, J. McGraw, E. Gallardo, J. Keasler, E. A. Leon, and B. Still, "Memory and parallelism exploration using the lulesh proxy application," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 1427–1428.

[24] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang *et al.*, "Exploring traditional and emerging parallel programming models using a proxy application," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 919–932.

[25] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–32.

[26] M. Kermarquer, "Fault tolerance interface." [Online]. Available: https://github.com/Maxime91860/LULESH$_{FTI}$.