WHAT BROKE WHERE FOR DISTRIBUTED AND PARALLEL APPLICATIONS — A WHODUNIT STORY

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Subrata Mitra

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

PhD is a long and difficult journey and many people have helped me to make the path smoother and kept me motivated. I sincerely thank my advisor Prof. Saurabh Bagchi for teaching me the basics to become a good researcher. He always guided me towards the right direction and motivated me to solve interesting research problems. I got the opportunity to collaborate with many established researchers and their guidance and suggestions were immensely helpful. Specially, I would like to thank Greg Bronevetsky (Google), Rajesh Panta (AT&T Research), Moo-Ryong Ra (AT&T Research), Todd Gamblin (LLNL), Ignacio Laguna (LLNL), Martin Schulz (LLNL), Dong Ahn (LLNL), and Sasa Misailovic (UIUC). A big thanks also goes to the members of my research group who helped me in my PhD journey by giving many fruitful suggestions and encouragements. I also thank my PhD committee members Prof. Sam Midkiff, Prof. Milind Kulkarni and Prof. Jennifer Neville for providing many constructive feedbacks. This PhD would not have been possible without the support and encouragement I received from my parents and my wife.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Mitra, Subrata PhD, Purdue University, December 2016. What Broke Where for Distributed and Parallel Applications — a Whodunit Story. Major Professor: Saurabh Bagchi.

Detection, diagnosis and mitigation of performance problems in today's large scale distributed and parallel systems is a difficult task. These large distributed and parallel systems are composed of various complex software and hardware components. When the system experiences some performance or correctness problem, developers struggle to understand the root cause of the problem and fix in a timely manner. In my thesis, I address these three components of the performance problem related issues. First, we focus on diagnosing performance problems in large scale parallel applications running on supercomputers. We developed techniques to localize the performance problem for root-cause analysis. Parallel applications, most of which are complex scientific simulations running in supercomputers, can create up to millions of processes or task and communicate using the message passing paradigm. We developed a highly scalable and accurate automated debugging tool (PRODOMETER), which uses sophisticated algorithms to first, identifies the task where the problem originated. Second, it creates a logical progress dependency graph of the tasks to highlight how the problem spread through the system manifesting as a system-wide performance issue. Finally, PRODOMETER pinpoints the code region corresponding to the origin of the bug. Second, we developed a tool chain to detect performance anomaly using machine-learning techniques. This input-aware performance anomaly detection system consists of a scalable data collection framework to collect performance related metrics from different granularity of code regions, an offline model creation and prediction error characterization technique and a threshold based anomaly

detection engine for production runs. Our system can learn from few training runs and can handle unknown inputs and parameter combinations by dynamically calibrating the threshold according to the characteristics of the input data. Third, we developed a new performance problem mitigation scheme for erasure-coded distributed storage systems. Specifically, repair operation of failed blocks in erasure-coded distributed storage system takes really long time in a networked constrained datacenter. The reason being, a lot of data from multiple nodes are gathered into a single node before a mathematical operation is performed to reconstruct the missing part and this process severely congests the links toward the destination. We proposed a novel distributed repair technique, called Partial-Parallel-Repair (PPR) that performs this reconstruction in parallel on multiple nodes and reduces the chance of network congestion and greatly speeds up the repair process. Fourth, we study how for a class of applications, performance can be improved (or performance problems can be mitigated) by selectively approximating some of the computations. For many applications, the main computation happens inside a loop that can be logically divided into few temporal segments, we call phases. We found that while approximating initial phases might have a serious effect on the quality of the result, approximating the computation for the last few phases have very small impact on the final accuracy of the result. Based on this observation, we developed an optimization framework that for a given budget of accuracy-loss would find the best approximation settings for each phases in the execution.

1. INTRODUCTION

Detection, diagnosis and mitigation of performance problems continues to be one of the hardest challenges in large scale distributed or parallel systems. Distributed applications are often complex – they involve complex interactions among many software modules, use many third party libraries, and need to run on various kinds of system architectures. And yet, they are tasked with running 24X7, with strict requirements for low latency and high throughput. Consider the abundance of such distributed systems all around us – from e-commerce web services, social network and email services, (landline or cellular) telecommunications systems; aircraft control systems, super computers, video streaming services, to the massively multiplayer online games in which we spend our countless hours. It is well-nigh impossible for a single developer or even a group of developers to understand the internals well enough to tame the complexity of distributed applications. Naturally when the system experiences some performance or correctness problem, developers struggle to understand the root cause of the problem in a timely manner. They generally start with some intuitions, but due to the complexity of the distributed application, often their mental model is wrong and they get into a tedious "trial and error" cycle to debug the problem. Since many of these applications are revenue critical and data-centers and supercomputers have large operating cost, failures at large scale results in substantial loss of money and reputation. Thus localizing and fixing the problem as soon as possible is a necessity and this is possible only if automated tools can direct the developer's attention to the possible root causes. In my research, so far I have worked on performance problem detection, diagnosis and mitigation for mainstream production quality software applications and benchmarks. These applications are a mix of massively parallel scientific applications, distributed applications and complex serial applications. My main research contributions can be summarized as as follows:

1. **Identifying the root-cause of performance problems at massive scale:** Today's largest supercomputers consist of hundreds of thousands of machines connected via different kinds of network architectures. Parallel applications, most of which are complex scientific simulations, can create up to millions of processes or task and communicate using the message passing paradigm. Many problems arise during large scale runs which can originate in several libraries being used, congestion in the network, bugs in the underlying system software, hardware problems in individual machines or due to a bug in the actual application itself. Very often such problems cannot be reproduced at small scale or cannot be detected using static analysis of the program. Traditional break point-based manual debugging simply does not work at these large scales. I developed a highly scalable tool named PRODOMETER [1], which monitors the application run, detects a performance problem, and performs a dynamic analysis of the application. This analysis, first, identifies the task where the problem originated. Second, it creates a logical progress dependency graph of the tasks to highlight how the problem spread through the system manifesting as a system-wide performance issue. Finally, PRODOMETER pinpoints the code region corresponding to the origin of the bug. With this information, the software developer might just attach a simple serial debugger such as GDB to the identified process, get to the identified code region and fix the bug. Currently this technique was tested to scale up to 16K processes, can handle real applications involving complex loops and code structures. It causes acceptable slowdown and can perform the analysis in seconds. In some case studies, we found this tool can reduce the debugging time from weeks to few hours and our open source tool [2] has been used within Lawrence Livermore National Lab (who co-developed it) and by researchers at other supercomputing facilities. In Chapter 2, I describe the analysis technique of PRODOMETER in detail and present the results.

2. **Input-aware performance anomaly detection:**

Complex software such as scientific simulators, image processing applications, financial applications etc. often come with many configuration parameters which can be used to calibrate internal algorithms or achieve certain quality of results. Often, these parameters also influence the performance characteristics (i.e. run-time, memory footprint etc.) of the applications. Performance also depends on properties of input data. For example, for a graph search application, run-time and resource requirement for a dense-graph will be much more than a sparse-graph. Often, performance prediction models are created for these applications through multiple training runs. These models are then used to estimate the runtimes of these applications and also to detect performance anomalies. But these models have inherent prediction errors which become prominent when input parameters at production do not resemble the ones used for training. In this work [3], I show how such prediction errors can introduce inaccuracies during anomaly detection. Further, we present a technique to characterize such prediction errors and show how such information can be used to build an input-aware anomaly detection engine. In Chapter 3 I will illustrate these techniques in detail and present the results.

3. **Improving repair performance in erasure-coded distributed storage:**

   With an increased need to store massive amount of data, erasure-coded storage has emerged as an attractive alternative to replication because it simultaneously provides significantly lower storage overhead and better reliability. Reed-Solomon (RS) code is the most widely used erasure-coding scheme because of its capability to provide maximum reliability for a given storage overhead as well as flexibility in choosing suitable coding parameters. However, after a chunk of data is lost or gets corrupted, the reconstruction time to recreate this unavailable data becomes prohibitively long in the traditional repair technique. The main reason being the bottlenecks created at the network due to large amount of data transfer required by the traditional repair technique. Some proposed solutions

either use additional storage or severely limit the coding techniques that can be used. In this work, we propose a novel distributed repair technique, called Partial Parallel Repair (PPR) [4], which divides the reconstruction operation to small partial operations and schedules them on multiple nodes already involved in the data reconstruction. A distributed protocol then runs to progressively reconstruct the unavailable data blocks and this drastically reduces the network transfer time. Theoretically, our technique can complete the network transfer in $ceil(log_2(k+1))$ time, compared to k time needed for a (k, m) Reed-Solomon code. Our experiments show, using our technique the whole reconstruction time gets significantly reduced and degraded read throughput significantly increases. Moreover, our technique is compatible with almost all existing erasure codes and does not require any additional storage overhead. We demonstrate this by overlaying PPR on top of two prior schemes, Local Reconstruction Codes and Rotated Reed-Solomon code, to gain additional savings in reconstruction time. In Chapter 4 I will present the design, implementation and evaluations of PPR in details.

4. **Improving application performance through phase-aware approximation:**

   A class of applications from machine-learning, image-processing, financial-analysis etc. can have inherent ability to sustain moderate amount of error introduced through inexact or approximate computation. We discovered that a majority of these applications also exhibits a temporal or *phase-specific behavior* because of a common computation pattern where the main computation is iteratively performed inside an outer-loop. Computation inside this outer-loop can be logically divided into certain number of phases. This gives a novel control over the error generated due to approximation and the resulting performance improvement by controlling not only how much to approximate but also controlling *when* the approximation should be performed. We found, in most of the cases,

approximation in the later phases introduce very small approximation error.. In this work, we presents OPPROX [5], a novel system for application's execution-phase-aware approximation. For a user provided error budget and target input parameters, OPPROX first identifies different program phases and then searches for the optimum phase-specific approximation settings that would maximize the performance improvement of the application while keeping the resulting error within the user provided budget. Our evaluations with five popular benchmarks and applications show that OPPROX become extremely useful while operating under small error budget. In Chapter 5, I will present the phase-specific characterization of the applications and OPPROX's technique in detail.

2. IDENTIFYING THE ROOT-CAUSE OF PERFORMANCE PROBLEMS AT MASSIVE SCALE

2.1 Introduction

Debugging large-scale parallel applications is a daunting task. The traditional debugging paradigm [6, 7] of interactively following the execution of individual lines of source code can easily break down on the sheer volume of information that must be captured, aggregated, and analyzed at large scale. Perhaps, more importantly, even if such approaches were feasible, programmers would be simply overwhelmed by massively large numbers of threads of control and program states, which are involved in large-scale parallel applications. Instead, (semi-)automated data aggregation and reduction techniques offer much more attractive alternatives. For serial programs, several projects including Cooperative Bug Isolation (CBI) [8, 9] and DIDUCE [10] already target such techniques for bug detection and identification. However, these techniques cannot be used to debug large-scale parallel programs since they do not capture and model communication-related dependencies.

Driven by these challenges, a few recent efforts provide semi-automated techniques to debug large-scale parallel applications [11, 12]. Their key insight is that, although there is a large number of tasks in a large-scale application, the number of behavioral equivalence classes is much smaller and does not grow with the scale of the application (i.e., task counts and input data). These classes are mostly defined in terms of the control-flow graph of all involved tasks. These approaches identify the behavioral equivalence classes and isolate any task, or a small group of tasks, that deviates from their assigned behavior class.

Hangs and performance slowdowns are common, yet hard-to-diagnose problems in parallel high-performance computing (HPC) applications. Due to the tight coupling

of tasks, a fault in one task can quickly spread to other tasks, causing a large number of tasks, or even the entire application, to hang or to slow down. Most large-scale HPC applications use the message-passing interface (MPI) [13] to communicate among tasks. If a faulty task hangs, tasks that communicate with the faulty task will also hang during point-to-point or collective communication that involves the faulty task. These tasks will also cause other non-faulty tasks to hang, leading the application to an entire hang.

Previous work [14] proposed the notion of *progress* of tasks as a useful model to diagnose hangs and slowdowns. Intuitively, progress is a partial order for tasks, based on how much execution a task has made in relation to other tasks. The notion of progress is useful in parallel debugging because the *least-progressed* (LP) tasks[1] often contains rich information of the root-cause (i.e., the task where the error originated). Thus, traditional debuggers can be used to inspect these LP tasks in more detail.

Several static and dynamic techniques to identify LP tasks at large scales exist. However, they largely suffer fundamental shortcomings when they are applied to HPC applications. The most relevant dynamic technique is AutomaDeD introduced by Laguna *et al.* [15]. It draws probabilistic inference about progress based on a coarse control-flow graph, captured as a Markov model, that is generated through dynamic traces. However, if two tasks are in the same loop, but in different iterations when a fault occurs, AutomaDeD may not accurately determine which task is progress-dependent on which other task. This is a fundamental drawback, as most HPC applications spend most of their execution time in loops [16]. For example, in scientific applications, a common use of HPC, there is typically a large outer loop, which controls the time step of the simulation, and within it, there are many inner loops, often with deep nesting levels. Thus, AutomaDeD may fail to infer progress dependencies for a large number of fault cases in HPC applications, as we show empirically in Section 2.5.2.

[1]Since progress is a partial order more than one task can be considered least progressed. In the following we refer to this set of tasks as LP tasks.

A similar approach is the Stack Trace Analysis Tool (STAT), which was introduced by Ahn *et al.* [14]. STAT relies on static analysis and uses the concept of *temporal ordering*, which creates a partial order among tasks representing their relative progress in the logical execution space, even within a loop. To identify the LP tasks, users simply select the first task in the temporal-ordering list. The temporal-ordering feature of STAT, called STAT-TO, builds def-use chains to identify *Loop Order Variables* (LOV) and uses them to determine relative progress among tasks in different iterations of the same loop. However, there are constraints on when an LOV can be identified by STAT-TO. For example, they must be explicitly assigned in each iteration and must increase or decrease monotonically; thus, a simple `while` loop that iterates over a pointer-based linked list may not have an LOV. We show this effect empirically for a variety of applications, in Section 2.5.2. Even in the cases where an LOV can be identified, the overhead of static analysis needed for their identification is prohibitive for complex loops as an interactive tool.

In this section, we present PRODOMETER [2], a novel loop-aware progress dependency analysis tool that can accurately determine progress dependence and, through this, identify the LP tasks, even in the presence of loops. It is implemented purely as a run-time tool and uses the main building blocks of AUTOMADED: It keeps track of per-task control-flow information using a Markov model. States in the model are created by intercepting MPI calls, and represent code executed within and between MPI calls. At any arbitrary point in the execution, PRODOMETER can recreate the partially ordered set of progress that each task in the application has made. It sidesteps the problem of STAT by avoiding static analysis, allowing us to keep track of the progress of a task in a highly efficient way. To achieve scalability, as in AUTOMADED, we trade off the granularity at which progress is measured—it is not done at the line-of-code granularity, but for a block with multiple lines of code.

In particular, we make the following contributions:

[2] PRODOS is Greek for progress and METER is measure.

- A highly accurate novel run-time technique that compares the progress of parallel tasks in large-scale parallel applications in the presence of loops
- An evaluation of accuracy, precision, and performance of our technique against the two state-of-the-art approaches (i.e., STAT-TO and AutomaDeD) via fault injection in six benchmarks
- A case study that shows our proposed technique can significantly aid in localizing a bug that only manifested in a large-scale production environment

Our fault-injection experiments on six representative HPC benchmark programs shows that Prodometer achieves 93% accuracy and 98% precision on average, and that this is 45% more accurate, 56% more precise, and more time-efficient than existing approaches. Our overhead evaluation suggests that the instrumentation overhead of Prodometer slows down the target programs between a factor of 1.29 and 2.4, and its per-task memory overhead is less than 9.4MB. Further, our scalability evaluation shows that its analysis itself is also highly efficient and scales logarithmically with respect to the number of tasks, up to 16,384 MPI tasks. Finally, our case study demonstrates that Prodometer significantly helped diagnosing an MPI bug that affected a large-scale dislocation dynamic simulation.

2.2 Need For Accurate Loop-aware Analysis

We detail the significance of progress dependencies as a scalable and powerful debugging idiom, as well as critical gaps in the state-of-the-art techniques that can infer them.

2.2.1 Progress Dependencies as a Scalable Debugging Marker

In the message-passing paradigm, parallel tasks progress in a highly coordinated fashion. They explicitly exchange messages to send or receive data relevant to their computation, and the need for matching sends and receives in point-to-point commu-

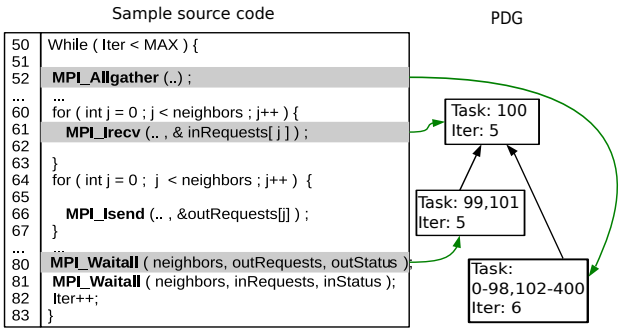


Fig. 2.1.: Iterative solver with 400 MPI tasks. Tasks are inside an outer `while` loop. Task 100 has progressed the least. Other tasks form two groups and waiting for task 100.

nication and collective communication calls point to the requirement of tight coordination needed for progress. For example, receivers cannot make progress until senders complete the matching send operation. This causes the progress of some tasks (e.g., receivers) to become *dependent* on other tasks (e.g., senders).

The ability to analyze such progress dependencies provides significant insight into the origin of many types of errors. Any error that disrupts the orderly progress plan of an application can reveal important clues, and thus resolving dependencies can point to the task containing the root cause.

We use Figure 2.1 to elaborate on this point. The code first exchanges a set of local data (e.g., ghost cells) with neighbor tasks using `MPI_Isend` and `MPI_Irecv`, non-blocking point-to-point communication calls, and then gathers computed data from *all* tasks and distributes the combined data back to *all* tasks through `MPI_Allgather`, a *collective* call. As many scientific codes model scientific phenomena over time (e.g., modeling the evolution of dislocation lines in a structure), this code iterates this computational step using the `while` loop to advance physical time steps.

Figure 2.1 highlights the source lines at which tasks are blocked when a single task encounters a fault (e.g., an infinite loop), showing its global impact. Specifically, task 100 triggers a fault right before executing `MPI_Irecv` and this causes its neighbor tasks 99 and 101 to form a group and to wait in `MPI_Waitall` for task 100 to complete.

Meanwhile, the majority of tasks complete this loop iteration, and wait in `MPI_Allgather`, which cannot complete until the other delayed tasks can join.

In general, a fault often causes tasks to form a *wait-chain* among them due to their natural progress dependencies, which eventually manifests itself as a global hang or deadlock. It is desirable to detect such conditions and to infer the dependencies automatically. Indeed, the graph on the right in Figure 2.1 shows that at the root of the corresponding progress dependencies lies the faulty task.

While the number of tasks in an application increases exponentially, the number of MPI calls, where tasks are stuck, is often limited. This is mainly because MPI programs are written often in a single program, multiple data (SPMD) style, which causes them to progress in an almost lock-step fashion through the same code segments, limiting the number of possible state combinations across tasks. The same holds for most multiple program, multiple data (MPMD) codes, since they are typically composed from only a small number of different programs, which are themselves SPMD. As a consequence, tasks often wait at a limited number of states, and those at the same state form a *progress-equivalence* group, which can be used as a scalable debugging marker.[3]

2.2.2 Markov Models as a Scalable Summary of Execution

Identifying and exploiting progress equivalence groups requires a representation of parallel program-control flow. Traditional control-flow graphs (CFGs) capture the execution flow of instructions either statically or dynamically. For MPI applications, however, control paths that capture multiple tasks and the dependencies among them, generally (except for simple programs) cannot be generated through static analysis, since the analysis of matching messages is infeasible in the general case. In contrast, CFGs created through dynamic analysis are based on the history of executed instructions and therefore implicitly capture all matched messages accurately. Never-

[3]A more formal definition of progress dependency can be found in [15].

theless, large-scale and long-running applications can produce very large CFGs [17], presenting scalability challenges.

To overcome these challenges, Laguna *et al.* [15] used *Markov models* (MMs) as a compact, scalable summary of the dynamic execution history. They create *states* in the MM by intercepting each MPI function call, and by capturing the call stack before and after the actual call to the underlying MPI runtime (through an PMPI function call). Edges between the states (i.e., nodes) represent control-flow transitions through two types of code: (1) communication code (executed inside MPI calls), and (2) computation code (executed between two adjacent MPI calls), as depicted in Figure 2.2.

In addition, a *transition probability* is tracked on each edge. This probability represents the fraction of times a particular edge is traversed (out of the total number of transitions seen from that state). For example, nodes with only one outgoing edge will always transition to the next node pointed to by the outgoing edge: the transition probability is 1.0; nodes with multiple outgoing edges can have different probabilities in choosing a next node for a transition, and this depends on the previous observations. This approach provides a compact abstraction of the CFG on each nodes and can be captured even for long-running applications. Further, it can be used in subsequent steps for a scalable cross-node aggregation by forming equivalence classes of MMs.

2.2.3 Loops Hamper Accuracy of Dependency Analysis

Laguna *et al.* [15] used a *path probability*-based approach to resolve progress dependencies. If tasks are stuck in control-flow states S_i and S_j, they calculate probability of going from $S_i \rightarrow S_j$ as forward-path probability (P_f) and probability of going from $S_j \rightarrow S_i$ as backward-path probability(P_b). If $P_f > P_b$, then they conclude that it is highly likely that tasks at S_i eventually reach S_j (based on the execution history seen so far). Therefore tasks at S_j are more progressed than tasks at S_i. In other words, tasks at S_j are progress-dependent on tasks at S_i.

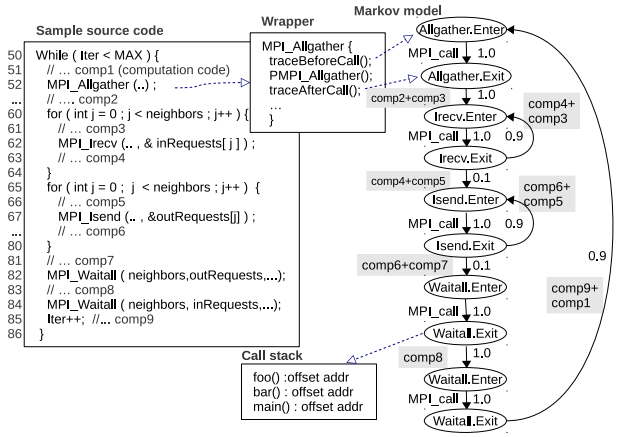


Fig. 2.2.: Markov model creation: MPI calls are intercepted and `.Enter`, `.Exit` nodes are added to the MM using the call stack *before* and *after* the actual `PMPI` calls. *Computation* and *communication*(corresponding to MPI library calls) edges alternate showing transition probabilities along them.

The major drawback of this approach is, however, that such inference does not work well in the presence of loops. For example, in Figure 2.2 the forward-path probability from the state corresponding to `MPI_Allgather` to the state corresponding to `MPI_Irecv` is 1.0 (because every task in the former eventually reaches the latter). The backward-path probability from state B to state A is either less than 1.0 (i.e., some tasks eventually exit the loop) or equal to 1.0 (a hang arises before any task exits).

In the first case, the probability-based approach will infer that tasks at `MPI_Irecv` are waiting for tasks at `MPI_Allgather`. Clearly, this is incorrect because tasks at `MPI_Allgather` have already passed `MPI_Irecv` and are waiting at the collective in the *next* iteration. In the second case, probability-based approach will not be able to infer any dependency.

Loops are very common in real-world applications, ranging from the main time-step loop to many internal computation loops. Thus, we need a highly accurate and scalable analysis technique that can resolve dependencies even in the presence of loops.

2.3 Approach

To address the challenges laid out above and to go beyond the shortfalls of current tools, we designed and implemented PRODOMETER, a highly accurate and scalable analysis tool that can resolve dependencies even in the presence of loops. It detects the least-progressed (LP) tasks and uses them to pinpoint the tasks where a fault is first manifested. For its analysis, it builds a dynamic progress-dependency graph (PDG), which gives insight into the progress relationships of all MPI tasks, by first creating space-efficient, per-node Markov models (MMs) capable of abstracting long-running executions, and then grouping them into progress-equivalence groups for scalability.

Figure 5.6 gives an overview of the PRODOMETER's workflow. Programmers link PRODOMETER, e.g., by preloading its shared library, to their MPI application (Step 1 in the figure). In Steps 2 and 3, PRODOMETER monitors the application at run-time and creates an MM for each task. The MM creation is fully distributed (i.e., no communication is involved). During the execution, PRODOMETER uses a helper thread in each task to detect hangs: when this thread detects inactivity in the main application thread (i.e., it does not see any state transition in a configurable amount of time), it signals a fault, which triggers its analysis including the creation of the PDG (Step 4). Finally (Step 5), PRODOMETER allows users to visualize the PDG, the LP tasks, call stack trees, and annotations on the source code where different tasks are waiting.

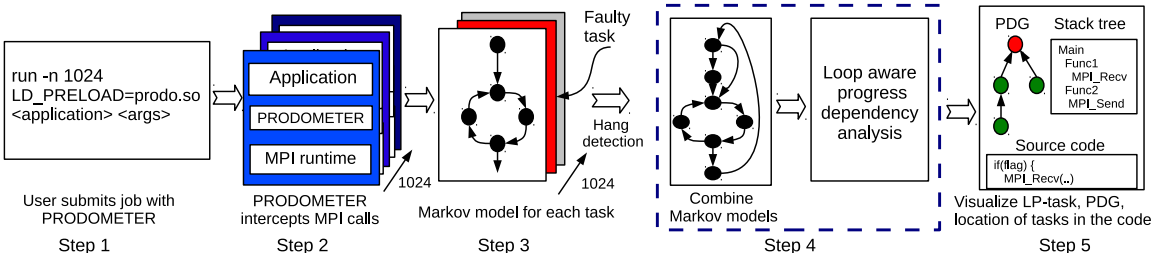


Fig. 2.3.: Workflow of PRODOMETER

2.3.1 Markov Model Creation

PRODOMETER uses MPI wrappers to intercept calls to MPI functions. Within each wrapper, it identifies the MPI call as well as the computation since the previous MPI call using the call stack observed at that point and adds each of two states to the MM, if they are not yet present. In this case, it also assigns an integer identifier to the newly created state in increasing order. Thus, this identifier represents the order in which different states are created.

While a traditional MM only keeps transition probabilities on edges [15], this model cannot capture loop iteration information. Therefore, PRODOMETER augments the MMs to keep track of inter-state *transition counts*. A transition count captures the number of times a transition between two states in MM has been observed.

Since the density of calls to MPI functions in the code is reasonably high in most applications, our technique provides appropriate granularity for localizing a problem in the code region. For the applications where MPI call density is low, a binary instrumentation technique could be used to insert additional markers. At each marker, the corresponding wrapper will insert a state in the MM increasing the granularity of diagnosis. This technique does come with some run-time overhead, but users can control it by choosing an appropriate sampling rate.

2.3.2 Concept of Progress

To infer progress dependencies, our algorithms treat each MM as a coarse representation of a dynamically generated control-flow graph. Thus, we assume that loop properties, such as entry and exit nodes, backedges, and domination [18] also apply to our MM analysis. In particular we assume: (1) a loop has an *entry* and an *exit* node; (2) a node x *dominates* node y, if every path of directed edges from the start node to y must go through x [19]; (3) a loop has a *backedge* (identified as an edge whose head dominates its tail), and (4) a loop with a single entry node is called *reducible* [20].

Next, we define the concepts of loop-nesting order, and relative progress, which we use later for our algorithms.

Loop-nesting order. Let L_x and L_y be two loops in an MM. Let N_x and N_y be the sets of nodes that belong to L_x and L_y, respectively. A loop-nesting order exists between L_x and L_y if $N_y \subset N_x$, i.e., all nodes in L_y also belong to L_x. Then, we call L_x has higher loop-nesting order than L_y, $L_x > L_y$. Intuitively, L_x is the outer loop in a loop nesting.

Relative progress. Let two tasks T_1 and T_2 be at node i and j in an MM. If i and j are not inside a loop then T_2 has made more progressed than T_1 if there is a *path* from i to j (i.e., it is possible that T_1 can reach T_2 by following a sequence of forward edges) but not vice versa. If i and j are inside a nesting of loops then T_2 has made more progress if loop-nesting order exists between these loops and T_2 has made more transitions along a path in the outer loops. Let $L_1, .., L_i, L_j, .., L_n$ be n nested loops, where $L_i > L_j$. Let t_{1i} denote the number of transitions made by task T_1 along loop L_i. Then, T_2 is more progressed than T_1, $T_1 \preceq T_2$, iff $t_{1i} < t_{2i}$ and $t_{1k} = t_{2k} \forall k < i$ or $t_{1k} = t_{2k} \forall k \leq n$, i.e., the lexicographical order between t_{1k} and $t_{2k} \forall k \leq n$.

Intuitively, we compute relative progress in a nesting of loops by first comparing the number of transitions made in the outermost loop, if equal, comparing the next inner loop, and so on.

Ahn et al. [14] showed that relative progress is a partial order because it is reflexive ($T_i \preceq T_i \forall i$), antisymmetric ($T_i \preceq T_j$ and $T_j \preceq T_i \Rightarrow T_i = T_j \forall i, j$) and transitive ($T_i \preceq T_j$ and $T_j \preceq T_k \Rightarrow T_i \preceq T_k \forall i, j, k$). If relative progress cannot be resolved between two tasks, we call the tasks *incomparable*. Such tasks would be executing in two separate branches in the MM. For example, relative progress order between two tasks stuck in distinct branches (e.g., *if* and *else* branches) of a conditional statement cannot be resolved, unless they are inside a loop and have completed different iterations in that loop.

2.3.3 Iteration Counts in Markov Model

We define the *iteration count* of a loop as the number of transitions a task has *completed*—i.e., it has traversed the *backedge* to the loop-entry point—along only that loop. Due to loop nesting, an edge belonging to a loop in an MM can also be shared by other loops. Our tool keeps track of the number of transitions along MM edges and from this, it derives the number of iterations of a loop that have been executed thus far.

Let t_i denote the number of transitions made by the program along edge e_i in the MM. If there are n loops $l_1, .., l_n$ which share this edge, and if we denote IC_k as the iteration count for loop l_k, then for the backedges, $t_i = \sum_n IC_k$, i.e. the transition count along a backedge is the summation of all the iteration counts of the loop nesting surrounding that edge. But, for an iteration in progress (not completed yet), the edges on the *forward path* of the loop will have one extra transition making the transition count greater than the summation of ICs. Thus in general we can write $t_i \geq \sum_n IC_k$.

Characteristic edge. We define the characteristic edge of a loop as the edge that is not part of any other loop. Therefore, the transition count on that edge accurately represents the iteration count of that loop. Let E_l be a set of all edges e_i which constitute loop l. Each edge might belong to more than one loop. Thus, a characteristic edge e_k of a loop x will be such that $e_k \in E_x$ and $e_k \notin E_x \cap E_y : \forall y \neq x$. In 2.3.4 we discuss how can we identify a characteristic edge for all practical loops.

2.3.4 Analysis Step: PDG Creation

Definition. A progress-dependency graph (PDG) represents relative dependencies that prevent tasks from making further execution progress in case of a fault [15]. The graph shown in Figure 2.1 is an example of PDG which shows two task groups (99 and 101, and all the others) being dependent on task 100. Thus in a PDG, the nodes

corresponding to LP tasks will be the nodes that have no further dependencies on other tasks and hence no outgoing edge.

To create such a PDG, we first need to resolve relative progress between different tasks through the following steps.

Aggregation of models. After a fault is detected, PRODOMETER begins the analysis step. PRODOMETER gathers all MMs from each task into the *root* task (rank 0) and creates an *aggregated* MM. This is done by a custom reduction operation, using individual models as input. We use an aggregated model, instead of using distributed models, because it gives us a global picture of all the states in which the MPI tasks are in and the history of control-flow paths of each task. Figure 2.4 shows how the aggregation algorithm creates a single MM at the root-task by combining all the states and edges of individual MMs from each task. In this figure first Task-0 and Task-2 combines MMs from Task-1 and Task-3 respectively by using the union of all the states and edges of 2 participating tasks. In the next step, Task-0 or the root task follows similar procedure to combine MM from Task-2 with its own MM to create the final aggregated MM. MM aggregation allows PRODOMETER to identify loops that could not be identified by looking at individual MMs. For example, two tasks might

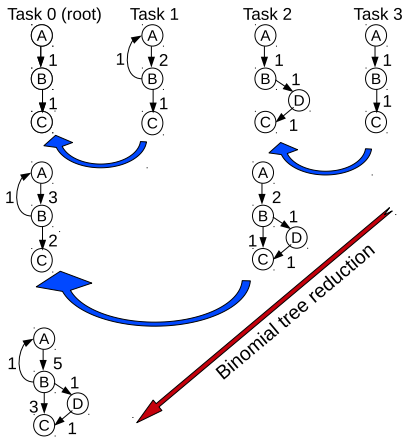


Fig. 2.4.: Aggregation of models: We aggregate MMs of individual tasks into a single (global) model by a reduction that uses a binomial-tree algorithm. Transition counts on the edges are also combined.

observe partial paths between two states in their MMs, while a global picture might reveal the existence of a loop when their per-task paths are combined.

All edges in the aggregated MM are annotated with transition probabilities and counts along with the unique identifier of the corresponding task. After the aggregation, if a state s has k outgoing edges and T_i represents the transition count for i^{th} edge, which connects to next state r, we calculate the transition probability from state s to state r as $T_i/\sum_k T_i$.

We keep raw transition counts corresponding to each task for subsequent analysis. To achieve scalability, instead of using a linear buffer to store transition counts for each task, we group the tasks based on transition counts—on each edge of the aggregated MM, we store unique transition counts as the *key* in a compact task list. To represent consecutive MPI ranks in a group, we use ranges of values. Thus, each entry in this representation (i.e., a table) are the tasks that have seen the same number of transitions along that edge, as shown in Figure 2.5. This approach makes our tool scalable; it greatly reduces the memory footprint because the number of task groups is far fewer than the number of tasks. This is because the tasks, large in number as they are, are found to be waiting in only a few places in the code. In practice, we have found the size of this table to be in the order of tens for an application with hundreds of thousands tasks.

Equivalence states. When a fault occurs, multiple tasks may be in the same state in the MM (i.e., they are executing the same code region). This behavior is due to the SPMD nature of MPI applications, and it simplifies our problem—it naturally creates *equivalence states*. Our analysis deals with equivalence classes of tasks rather than each task individually. An equivalence MM state and a set of iteration counts over all of the containing loops uniquely define a progress-equivalence group of tasks.

Compression

We eliminate unnecessary states from the MM before the analysis. An MM can have a large number of states because the same MPI call can be made from multiple different calling contexts. However, not all states and edges are interesting from the point of view of progress-dependency analysis. We are only interested in identifying progress dependencies between the equivalence MM states. Even though this step is not necessary for loop aware analysis, using it makes the algorithm scalable.

The compression algorithm works as follows. First, we replace all *small* loops with a single state. Small loops contain only two states, with a cycle between them, and they are created mainly by send/receive operations. Second, we merge consecutive linear chains of edges (i.e., sequence of edges with transition probability of 1.0) into a single edge. As shown in Figure 2.5, states between 1 and 4 were compressed after eliminating small loop between states 2 and 3. In all cases, the algorithm keeps consistent the transition probabilities of the entire MM—the sum of probabilities along all outgoing edges of a node is 1.0. But while doing the compression, PRODOMETER preserves all of the equivalence MM states and all loop structures containing them. For example, in Figure 2.5 it does not compress away states 6 and 7 because there are different groups of tasks which are waiting in these states.

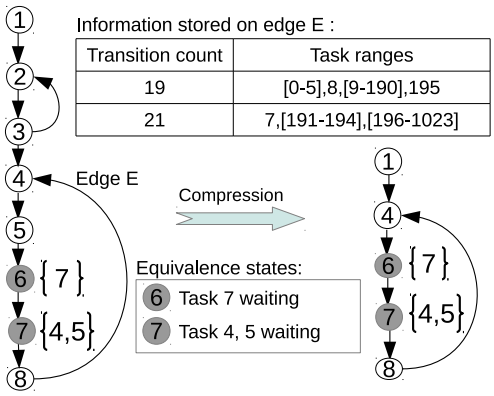


Fig. 2.5.: Tasks are grouped based on transition counts and stored in a scalable manner on each edge. Markov model is compressed to keep only useful information about control flow structure and equivalence states.

Progress Dependency Analysis

The algorithm for resolving progress dependency can be divided into two cases: (1) when two equivalence states are inside some loop(s), or (2) when they do not share any common loop. For case (2), PRODOMETER simply uses the algorithm in Laguna *et al.* [15]. For two equivalence states S_i and S_j, it first calculates *transitive closure*. Then for each *path* in closure it calculates forward and backward probabilities between those two states using transition probabilities present in the MM. It resolves the final dependency based on which one of these is higher. In the rest of this section, we describe how PRODOMETER resolves dependencies in case (1), our primary contribution. Algorithm 1 shows the overall procedure.

Algorithm 1 Progress dependency analysis

Input: *mm*: Markov model

statesSet: set of equivalence states where tasks waiting

Output: *matrix*: adjacency-matrix representation of PDG

```
1: procedure PDGCreation
2:     mm ← compressGraph(mm, stateSet)
3:     allLoops ← identifyAllLoops(mm)
4:     mergedLoops ← mergeLoops(allLoops)
5:     for all pair (s1, s2) in statesSet do
6:         loopSet ← getCommonLoops(s1, s2)
7:         if loopSet ≠ empty then
8:             d ← loopBasedDependency(loopSet, s1, s2)
9:         else
10:            d ← probabilityBasedDependency(s1, s2)
11:        end if
12:        matrix[s1, s2] ← d
13:    end for
14: end procedure
15: procedure getCommonLoops(s1,s2)
16:    loopSet1 ← getLoopsWithNode(s1) //loops containing s1
17:    loopSet2 ← getLoopsWithNode(s2) //loops containing s2
18:    return loopSet1 ∩ loopSet2 //return intersection:loops shared by s1 and s2
19: end procedure
```

Loop identification. PRODOMETER uses the Johnson's algorithm [21] to identify all loops in the compressed MM. This algorithm finds all the elementary circuits in a directed graph and runs in time bounded by *O((n + e)(c+1))*, where MM has n states, e edges and c loops. Internally, PRODOMETER uses a hash function to create an integer representation of the loop. The input to the hash function is an ordered

Algorithm 2 Loop aware progress dependency analysis

Input: $s1$, $s2$: Two equivalence states being compared

$loopSet$: Set of loops containing those two states

Output: Dependency relation between $s1$, $s2$ [$x \xrightarrow{d} y$ implies x depends on y]

```
1: procedure LOOPBASEDDEPENDENCY(loopSet, s1, s2)
2:     orderedLoops ← GETLOOPNESTINGORDER(loopSet) //sort loops
3:     for all loop in orderedLoops do
4:         ic1 ← GETITERATIONCOUNT(s1, loop)
5:         ic2 ← GETITERATIONCOUNT(s2, loop)
6:         if ic1 > ic2 then
7:             return s1 -d-> s2
8:         else if ic1 < ic2 then
9:             return s1 <-d- s2
10:        end if
11:    end for
12:    /* Here s1,s2 are in the same iteration for all the loops /*
13:    outerLoop ← orderedLoops.first //use only outer loop to break tie
14:    return DISTANCEBASEDDEPENDENCY(outerLoop, s1, s2)
15: end procedure
16: procedure DISTANCEBASEDDEPENDENCY(outerLoop,s1,s2)
17:    entry ← GETLOOPENTRY(outerLoop)
18:    dis1 ← GETDISTANCEFROMLOOPENTRY(entry, s1)
19:    dis2 ← GETDISTANCEFROMLOOPENTRY(entry, s2)
20:    if dis1 > dis2 then
21:        return s1 -d-> s2
22:    else if ic1 < ic2 then
23:        return s1 <-d- s2
24:    end if
25: end procedure
26: procedure GETITERATIONCOUNT(s,loop)
27:    taskSet ← TASKSWAITINGAT(s) //tasks waiting at this equivalence state
28:    ic ← 0
29:    backEdgeSet ← GETBACKEDGES(loop)
30:    for all backEdge in backEdgeSet do
31:        ic ← ic + GETTRANSITIONCOUNT(backEdge, taskSet)
32:    end for
33:    return ic
34: end procedure
```

list of the states that constitutes the loop. This integer-based representation helps PRODOMETER use faster comparisons and lookups for subsequent analysis, than if we were to use a string representation of the states.

Finding common loops. To compare two states, PRODOMETER first finds the set of loops that *contain* those states. Then, it uses our loop-aware algorithm to resolve progress dependency. If there are no common loops that contain those states, it applies case (2), as stated above. Note that HPC applications typically have multiple nested loops, which could also create a nesting of loops in the MM.

Loop merging. Different loops in the aggregated MM appear depending on when MPI calls are made in the source code, as Figure 2.6 illustrates. Our approach assumes that loops are reducible [20] (which implies that the code does not use "goto" statements, for example). Figure 2.6 shows the basic loop categories that PRODOMETER can handle.

In our survey of multiple HPC benchmarks and from our experience with scientific applications, we observed that most (non-goto) loop structures found in HPC applications are composed of these basic loop categories. For example, if PRODOMETER encounters a complex loop-nesting structure in the MM, it breaks it down to simpler structures, and tries to map each structure to one of these basic categories.

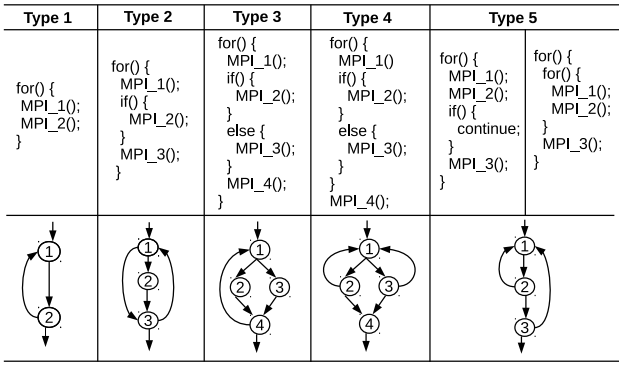


Fig. 2.6.: Categories of loops. MPI_x denotes any MPI call.

PRODOMETER uses purely dynamic analysis. As a result, it initially detects multiple loops in the MM corresponding to a single source-code loop. For example, in Figure 2.6, for Type-3, PRODOMETER initially determines one loop as 1 - 2 - 4 - 1 and a separate loop as 1 - 3 - 4 - 1. Iteration counts in these initially separated loops do not provide a complete picture and cannot be used to resolve relative progress. For example, an *if-else* statement inside a loop might appear as two separate loops in MM. To resolve relative progress between two tasks, one of which took the *if* path and the other one *else* path, we compare their iteration count in the actual source-code loop, which encloses the *if* and the *else* path.

PRODOMETER identifies different loops created from a single source-code-level loop and merges those to create a single loop which represents the original source-level loop. Assuming a reducible MM, each loop has only one *entry point* [19]. Therefore, we consider all loops with the same entry point as a single loop. The only category of loops that creates ambiguity is Type-5. As shown in Figure 2.6, such MMs might be created either from a single loop through *if-continue* statements, or from two nested loops. But due to the SPMD nature of MPI applications, we do not need to distinguish between these two cases for our analysis.

Identifying characteristic edges. Due to nesting, most of the edges in an MM belong to multiple loops. Thus, a transition count on those edges corresponds to a combined total count of many different loops. This problem can be solved by solving a system of linear equations and inequalities. The unknown quantities of these equations would be the iteration counts of various loops, and the known quantities would be the transition counts of various edges. However, for practical applications, solving the system of linear inequalities is a computationally expensive procedure for a dynamic tool. PRODOMETER avoids this expensive solution by a simple observation: A loop makes a transition along the backedge(s) when it completes one iteration. Also the backedge of a loop is not shared with any other loop, after loop entry-point based merging has been done as described above. For loops with a single backedge (Types

1,2, and 3), the transition count along the backedge correctly represents the iteration count of the loop. Thus, a backedge satisfies all the properties of a characteristic edge discussed before.

An exception is the case when loops have multiple backedges (Types 4, 5). In these cases, instead of considering a single backedge, we consider the *combination* of backedges as the characteristic edge and use it to find the iteration count of the loop. Then the iteration count of the loop becomes $\sum Tb_i$ where Tb_i is the transition count on i^{th} backedge.

Lexicographic comparison. Our tool resolves relative progress between two tasks inside a complex nesting of loops by comparing iteration counts in the *lexicographic order* (i.e., in the order from outer to inner loop). This is important because there might be cases where, between two tasks inside a 2-level nesting, one task has completed more iterations on the inner loop while the other made more progress in the outer loop. In this case, we assume that the task with more iterations on the outer loop has made more overall progress. To identify outer and inner loop in an MM, PRODOMETER considers the loop whose entry state *dominates* the other. This can be simply checked using state identifiers assigned to each state. Entry-point of the outer-loop will always be created before the inner loop and therefore will have a smaller identifier.

Distance-based comparison. In some situations, two equivalence MM states may have the *same* iteration count for all nesting levels of loops. Then, PRODOMETER uses a *hop-count distance* from the loop entry-point as the metric for progress, i.e., the number of edges traversed between the entry-point of the loop and the current state. A state that has a higher value of the hop-count distance is more progressed than one with a lower value. Algorithm 2 formally describes the loop-aware analysis procedure.

Finally, a PDG is created from these pairwise dependencies between equivalence MM states. In a PDG, a directed edge goes from a *more-progressed* state to a *less-*

progressed state showing their relative dependencies. Note that a state can contain multiple tasks, all of which are currently waiting in that state. A PDG is a graphical representation of the partial order.

2.4 Implementation

PRODOMETER is implemented in C++ as an extension to AUTOMADED's framework [15]. We implemented and tested it on x86/Linux and IBM Blue Gene/Q architectures, although the design is portable to any MPI-based parallel platform. The source code for PRODOMETER is available at [22] as part of the AutomaDeD project. In this section we discuss implementation-related aspects, in particular how we aggregate MMs in a scalable manner, how we detect faults, and how users can easily visualize the LP tasks.

2.4.1 Scalable Reduction of MMs

We implement a scalable binomial-tree-based algorithm, which merges MMs from individual tasks in $O(log(p))$ time, where p is the number of tasks. We cannot use `MPI_Reduce` to combine MMs because tasks can contribute states of different sizes. Since we keep an integer representation of each state, we can easily map states from different tasks into a state in the aggregated MM with efficient integer-based comparison. The merged MM contains a union of all states and edges, and thus can handle the case where individual MMs differ from one another. Figure 2.4 depicts this logarithmic reduction technique using an example of 4 tasks.

2.4.2 Fault Detection

Fault detection is orthogonal to our root cause detection in PRODOMETER. It can be combined with any technique available by the target platform or can even be done in cooperation with the application. By default, we include a platform and

application-independent heuristic based on a timeout mechanism. For this we use a *helper thread* per task that monitors the application to determine if a hang has occurred. The thread *caches* a sequence of the last N states that the application has seen. Each time it sees a state, it checks if the state is present in the cache. If it is not, it resets a timer and inserts that state into the cache. As a result, if it does not encounter a new state for a long time and only repeatedly cycles through the states in the cache, then the timer expires and it signals a fault.

Our default technique can detect hangs arising from deadlocks, livelocks, and slow code regions. PRODOMETER can infer a reasonable timeout threshold from the mean and standard deviation of previous state transitions in the MM. Users can also provide a timeout period to account for special application characteristics. We have found in practice that a period of 60 seconds is good enough to detect a fault in most of the applications. Cache size N is a configurable parameter and depends on application characteristics. A low value of N decreases the coverage of the fault detection whereas a large value might trigger false alarm for large loops. We found a value of 10 works reasonably well for real applications.

2.4.3 Determination of LP Tasks

PRODOMETER computes the LP tasks from the PDG, by identifying the nodes that do not have any progress dependency, i.e., nodes with no outgoing edges in the PDG. If it finds more than one such node, PRODOMETER uses point-to-point message send information to reduce this list. For example, if it currently has both nodes i and j in the set of LP tasks and the MPI trace contains a point-to-point message from i to j, but not vice-versa, it discards node j from the LP task set due to the observation that node j expects a message from node i.

2.4.4 Visualization

After the analysis, PRODOMETER opens a graphical interface to visualize the PDG as a graph. The LP tasks are highlighted with different colors. It also marks if the bug was identified in a communication node or computation node with different shapes of nodes. Users can interact with the graph by selecting one or multiple nodes, which will show a *parallel stack tree* of call-graphs and highlight corresponding lines in a source code viewer.

2.5 Evaluation

In this section, we show accuracy and precision of PRODOMETER using controlled experiments, followed by a real world case study.

2.5.1 Setup of Controlled Experiments

To evaluate the effectiveness of PRODOMETER, we set up controlled experiments in which we dynamically inject faults into applications, and measure its precision and accuracy in identifying the task that was injected. We compare our results to two existing state-of-the-art techniques, STAT-TO and AUTOMADED.

We implement the fault injection using the binary instrumentation library PIN [23] and use it to randomly inject an infinite loop as the fault at runtime. To cover a wide range of HPC application patterns, we choose three applications (AMG, LAMMPS and IRS) from the Sequoia procurement benchmark suite [24], a widely studied proxy application (LULESH [25]), and two programs from the NAS parallel benchmark (BT and SP), totaling six programs.

As commonly found in real-world HPC applications, most of these benchmark programs have two distinct simulation phases: a setup and a solver phase. During the setup phase, they generate their basic data structures, e.g., a mesh, and distribute the input data across MPI tasks. Once done, they move to the solver phase where the

tasks start to iterate through a time-step or solver loop and solve the given problem. While production applications spend most of their simulation time in their solver phase [26], these benchmark programs can spend a relatively large portion of time in the setup phase, due to relatively small input data set sizes as well as artificially reduced iteration counts, which makes them more suitable for experimentation and procurement testing while not changing the computational characteristics in each phase. To compensate for this bias, we inject faults only into the solver phase.

We first run each of these programs under PIN and profile all functions invoked in its solver phase. We filter out function calls from within well-known libraries, like `libc`, MPI and math libraries, to capture the fact that faults are more likely to be in the application than in these well-known and widely tested libraries.

We then randomly select various parameters to make our fault injection campaign statistically fair. Of all unique functions found in the profile, we randomly select 50 functions, and then pick one invocation of one of these functions for injection—this ensures we inject a fault into a random iteration of a loop. Similarly, we select one task out of all of the MPI tasks as the target for this injection.

Finally, we run these programs at different scales to observe any scale-dependent behavior of our technique. We use 128, 256 and 512 tasks for the cases, where the programs do not have restrictions on the task count to use; for some other benchmarks such as LULESH, IRS and BT, which have specific restrictions, we use the closest integers to these counts.

2.5.2 Accuracy and Precision

We use two metrics to summarize the findings of our controlled experiments and to quantify the quality of root cause analysis: *Accuracy* and *Precision*. *Accuracy* is the fraction of cases that a tool correctly identifies the Least-Progressed (LP) tasks. *Precision* is the fraction of the identified LP tasks that are actually where the fault

was injected. Since we inject a fault only into a single task, ideally ProDometer should detect only one task as the LP task.

In the first study we compare ProDometer to AutomaDeD. As mentioned above, AutomaDeD uses a similar approach in gathering runtime statistics using MMs, but is not capable of dependencies across loop boundaries.

Table 2.1 summarizes the accuracy results for ProDometer and AutomaDeD. ProDometer achieves over 93% accuracy on average, across all tested programs and scales, and its accuracy is not affected by scale. Further, the data shows that ProDometer's accuracy is significantly higher than that of AutomaDeD (64%). This is mainly because faults are injected into the solver phases which typically contain many complex loop-based control flows. Nevertheless, the accuracy of AutomaDeD, which does not have a special logic to infer progress inside a loop, is not close to zero, even on those programs with time-step loops governing the entire solver phase. This can be caused by faults that prohibit the completion of even a single iteration of the time-step loop. Thus, from the perspective of the Markov model, the loop was never entered, and AutomaDeD could infer progress of this region as if there was no loop. For BT, accuracy of ProDometer is relatively low, which is caused by the use of *goto* statements inside loops. Our current loop-detection algorithm is based on finding "natural loops", i.e., loops with a single head node and a backedge in the CFG. The *goto* statement violates this assumption, and we leave support for such cases to our future work.

Table 2.2 shows the summary of the precision results. ProDometer detects LP tasks with very high precision (above 98% on average), which means that in most cases, ProDometer will point the developer to a single task, which she can focus on for purposes of debugging, using standard single process debuggers.

However, we believe that there are fundamental limits to the precision of any tool that determines progress dependence. This is because the concept of progress dependency is itself a partial order, and thus there exist cases where states simply cannot be ordered. Notably, one cannot resolve the ordering of two tasks that are

Table 2.1.: Accuracy: PRODOMETER (PR) vs. AUTOMADED (AU)

	128 tasks		256 tasks		512 tasks	
Benchmarks	PR	AU	PR	AU	PR	AU
LAMMPS	1.00	0.54	1.00	0.48	1.00	0.58
AMG	0.92	0.56	0.94	0.46	0.88	0.67
IRS	1.00	0.50	1.00	0.76	1.00	0.78
LULESH	0.90	0.60	0.90	0.60	0.92	0.56
BT	0.82	0.52	0.84	0.66	0.84	0.68
SP	0.94	0.80	0.92	0.82	0.92	0.82

Table 2.2.: Precision: PRODOMETER (PR) vs. AUTOMADED (AU)

	128 tasks		256 tasks		512 tasks	
Benchmarks	PR	AU	PR	AU	PR	AU
LAMMPS	0.98	0.75	0.99	0.68	0.98	0.47
AMG	1.00	0.89	1.00	0.73	0.99	0.71
IRS	0.96	0.54	0.98	0.67	0.97	0.75
LULESH	0.97	0.46	0.98	0.25	0.94	0.28
BT	0.98	0.67	1.00	0.63	1.00	0.42
SP	1.00	0.87	0.98	0.84	1.00	0.74

executing in distinct branches of a conditional statement, in the same iteration count. In this case, PRODOMETER may identify both tasks as LP, which affects precision. PRODOMETER's mechanism for determining forward- and backward-paths is probabilistic, and if the prior observations are not representative enough or large enough, these introduce errors in the analysis.

Second, we compare the accuracy of PRODOMETER with STAT-TO, which is, to our knowledge, the only existing tool that is capable of finding loop dependencies. This is done in STAT-TO by detecting *Loop Order Variables* (LOVs) (which govern loops) via static analysis. Since STAT-TO requires a manual intervention and guidance, we compare the two tools by applying STAT-TO manually to some of the experiments for which PRODOMETER has succeeded. We first randomly select five cases from each of three benchmark programs (AMG, LAMMPS and LULESH), which PRODOMETER analyzed correctly using the new technique (i.e., cases with loops). Manual inspection reveals that the selected cases involve 1–3 structured loops

(e.g., `while`) for each benchmark and 1–3 program points (i.e., a line in a source file) for each loop. In addition, we find that only a single program point is within a *single* loop, while all others are inside triple-nested loops.

Then, we manually apply STAT-TO's Loop Order Variables (LOV) analysis to those program points that are contained in a loop. This represents the static analysis step in STAT-TO, which is most essential to resolve temporal order of the program points within loops. Further, STAT-TO requires a set of program points to be analyzed together for ordering, and thus we apply this analysis to sets of program points involved in each case. This amounts to nine distinct sets of LOV analysis runs summarized in Table 2.3.

In terms of accuracy, this static analysis fully retrieves a LOV in six out of the nine cases—66% retrieval rate. It completely fails to identify LOVs for two cases: one in LAMMPS and one in in LULESH. But for one case—i.e., `Main-cycle Loop`—where the program points are included in triple-nested loops, it partially fails to identify LOVs: it fails to retrieve a LOV for the outermost loop while successfully identifying

Table 2.3.: STAT-TO accuracy and performance

Codes	Loops	Points	LOV	Secs
AMG	PCG solver	498,	`i`	9.0
	Coarsening	595, 609, 1183	`level`	294.1
	Coarsening	1221, 1292	`level`	295.6
	V-Cycle	237	`cycle_count`	10.5
	Main cycle	263, 335	Not found	14.6
LAMMPS	Input	187	Not found	4.6
	Verlet	206, 264	`i`	3.6
	Verlet	206, 253	`i`	3.6
LULESH	Time step	2775, 2776	Not found	13.17

LOVs for both of the inner loops. To complete temporal ordering, however, STAT-TO must fully resolve all of the loops so we log this case as *Not found.*

In terms of performance, for all but one case, LOV analysis finishes its analysis in under 15 seconds, which would be acceptable to support even an interactive tool like a parallel debugger. However, for AMG's coarsening loop, it jumps to 295.6 seconds, a factor 20 larger overhead than other loops. We find that this is due in large part to the high complexity of this loop, which triggers a longer static analysis. The def-use table used in STAT-TO exhibits over one hundred variables defined outside the loop while being used inside the loop, and over thousand references to these variables from within the loop body. Given the complexity of a def-use chain analysis algorithm, $O(N^2 * V)$ where (N) is the number of definitions and uses and (V) is the number of variables, this case has the computation complexity of $O(10^8)$. This suggests that a static analysis technique can become unwieldy, as the complexity of target loops becomes higher.

2.5.3 Performance and Scalability

Our second set of experiments targets the run-time overhead of PRODOMETER, in terms of execution time and memory use with the target programs. We define *slowdown* as the ratio of times it takes for the program to complete with and without PRODOMETER. Memory overhead is the memory consumed by the tool. Since different tasks can have different memory usage, we use the average number across all the tasks. Table 2.4 summarizes the results measured with 512 tasks for the four largest codes: the three Sequoia benchmarks and LULESH.

Table 2.4.: Slowdown and memory overhead for model creation

Benchmarks	Slowdown	Memory overhead (in MB)
AMG	2.4	9.4
LAMMPS	1.3	3.67
IRS	1.29	4.7
LULESH	1.44	2.2

PRODOMETER is a dynamic analysis tool, and its interception of each MPI call followed by a system call to capture a call path is the primary reason for the increased run-time overhead. Nevertheless, the overhead is still reasonable, in particular for a debugging tool, and—most importantly—small enough to still enable the execution of full scale applications with realistic input sets. Memory overhead is a function of the number of unique states and edges in the Markov model. PRODOMETER stores call-path information in each state, and keeps track of the number of transitions on each edge.

In this experiment we statically linked the library and used return addresses from GNU *backtrace* utility to represent a call-path. Statically linking ensures that object code is loaded at the same addresses on all tasks. With dynamic linking, and with static linking on operating systems that use security features like address space randomization (typically not used on HPC systems, but default for many desktop OSs), libraries' load addresses can vary from task to task. To properly identify equivalence states across all tasks, we normalize the addresses in call-paths by representing them as a tuple (M, O) where M is the name of the module or library containing the address and O is the offset within that library. With the use of this normalization feature, we have observed slowdowns of up to 4.5x, when libraries are dynamically linked (and on systems for which this normalization feature is needed). The highest slowdown occurs for AMG. We plan to address this problem in our future work by implementing more efficient normalization and by using a faster stack tracing tool (such as libunwind).

Scalability The final set of controlled experiments evaluates the scalability of PRODOMETER's progress analysis itself by measuring model-aggregation and dependency-analysis performance. We perform this test with AMG and LULESH with up to 16,384 tasks on an IBM Blue Gene/Q (BGQ) machine. Each BGQ compute node consists of 16 PowerPC cores with 16GB of RAM, connected through a custom 5-D torus network. For our scalability test, we inject a fault close to the final execution phase of the

programs so that an analysis must handle the largest Markov model. Our objective is to evaluate how efficiently PRODOMETER aggregates large Markov models from a large number of tasks and analyzes this aggregated model to determine progress dependence.

Figure 2.7 summarizes the scalability results. *Aggregation_time* denotes the time it takes for PRODOMETER to aggregate Markov models gathered from all tasks using a binomial tree-based reduction technique. *Analysis_time* denotes the time taken to identify relative progress, to build a progress dependence graph and to identify LP task(s). *Aggregation_time* increases with scale for both benchmark programs, and the trend is logarithmic with the R^2 value of a logarithmic fit (with $alog_2x + b$) is 0.98 for AMG and 0.96 for LULESH. As for *Analysis_time*, that of AMG increases with scale while LULESH stays almost constant. In the case of LULESH, the complexity of the application does not change with scale and thus the number of states remains constant, while in the case of AMG the algorithmic complexity grows with scale (e.g., the number of levels in the multi-grid method increases with scale) and thus PRODOMETER must handle larger numbers of states at larger scales. Nevertheless, the worse-case overall time is less than 16 seconds, which is quite tolerable as an automated tool for debugging.

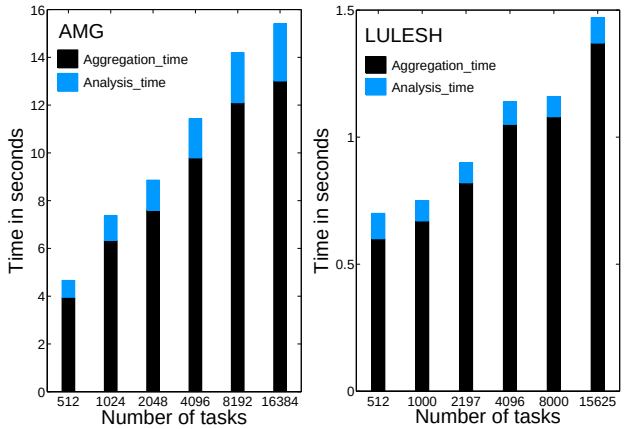


Fig. 2.7.: Scalability of PRODOMETER progress-analysis time

2.5.4 Case Study: Using PRODOMETER on a Real MPI Bug

A dislocation simulation code recently encountered intermittent hangs during production runs on our IBM BG/Q machine soon after our computing facility had rolled out a new driver, which included a new version of the MPI library. The cause of the problem was unknown. We observed this issue more frequently at larger scales. For instance, it almost always showed up for runs with 32,768 tasks. The scientist who was developing this code reported the issue to a system analyst. He then extracted its control flow and communication patterns, and put together a highly deterministic reproducer at a reduced scale.

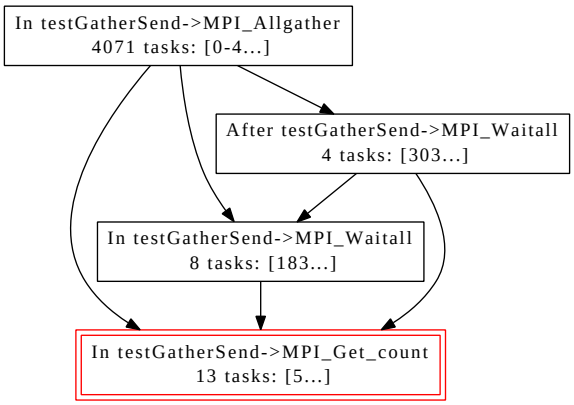


Fig. 2.8.: PRODOMETER on Dislocation Dynamics Reproducer

To help diagnose this issue further, we applied PRODOMETER to this reproducer. Figure 2.8 shows the global state PRODOMETER captured when the reproducer code was hung at 4,096 MPI tasks. The tool immediately helped us understand the global hang state without overwhelming us, as it expresses the state in a form of progress-equivalence classes (i.e., nodes). While this program was run at 4,096 tasks, our tool showed the state with only four progress-equivalence classes with dependencies (i.e., edges) among them.

Clearly, this diagnosis shows the reason for the global hang: the majority of the tasks (4,071 tasks) were not making progress because of their dependencies on a small number of tasks (25 tasks). Further, PRODOMETER identified the group that are still

in an MPI communication routine called `MPI_Get_count` as the least-progressed group. With this information, it was likely that the root cause of this hang would be in the vicinity of the code that these less-progressed groups were executing.

Given that this reproducer was not hung under the older MPI drivers, and that it was written simply and in a way to avoid elusive non-deterministic concurrency or memory bugs, we immediately suspected a bug in the underlying communication layer itself.

Indeed, using the same reproducer, the IBM software team quickly discovered a software bug in the communication layer of their new driver whereby a new collective communication optimization was too aggressive and was causing other concurrent communications to starve. As shown in Figure 2.8, large numbers of tasks reached and started `MPI_Allgather` first, and this large-scale collective communication significantly starved the communication subsystem of those tasks that were still performing logically earlier point-to-point communications. In fact, the reproducer actually injects a random delay prior to certain point-to-point calls into a small number of tasks to induce this condition more frequently.

Manual analysis would have been far more confusing, since `MPI_Allgather` appears earlier in the source listing. While it is obvious that this collective call is included in the main time-step loop in the reproducer code, it is far less obvious in the real case with the full dislocation dynamics code where this collective call is buried in a function being called by an upper-routine loop.

2.6 Related work

Debugging and root cause analysis Debugging is one of the most crucial and time consuming processes in software development cycle. Traditional breakpoint based debugging with GDB or "print debugging" is particularly not suitable for large scale parallel applications. Parallel debuggers such as Totalview [6] and DDT [7] control multiple processes and aggregate distributed states. However, identifying the

faulty process or finding the matching code location still requires interactive manual effort. Recent research on semi-automated statistical debugging has produced tools for sequential codes [8, 27–29] that, in the presence of sufficient historical data, can diagnose the root cause of a bug. Other techniques include use of boolean SAT and MAX-SAT [30, 31] for detecting program errors. Even though these techniques are quite promising, it is difficult to immediately apply those to debug parallel applications at large scale. Some formal verification based tools [32] and assertion based techniques [33] can overcome scalability challenges and adapt to parallel applications. However, these tools are mainly suited for debugging accuracy problems and are complementary to our approach. Laguna *et al.* [12] and Mirgorodskiy *et al.* [34] both monitor applications timing behavior and identify processes that exhibit unusual behaviors. DMTracker [35] uses statistical technique to find bugs in MPI applications by identifying anomalous data movements. There are other techniques [36, 37] that target general MPI coding errors and deadlock detection. These tools are also complimentary to our approach and can be used to detect a problem and trigger ProDometer for further analysis. The closest prior work that follows a similar aspect of relative progress as ProDometer are AutomaDeD [15] and the temporal ordering extension of STAT [14]. While AutomaDeD suffers from significant drawback of not being able to handle the common case — analysis in the presence of loops, STAT's static analysis based algorithm suffers from extensive static analysis times while building def-use chain and fails in the absence of loop-order-variables. Our loop-aware dynamic technique addresses both of these issues.

Loop analysis Loop analysis is an established field in compiler technology. There are many well accepted algorithms for identifying natural loops in the program and used in compilers for loop-unrolling [38], tiling [39], resolving dependency between different variables [19]. These techniques are mainly based on static analysis of the program and the goal is to improve parallelism and cache behavior. Other studies [40] use loop characterization at the hardware level to improve branch prediction

and parallelism. Our goal for dynamic analysis of loops is fundamentally different. We perform our loop-analysis on Markov models in-order to extract information about iteration count and loop nesting. We then perform lexicographical order based comparison to resolve progress between different groups of tasks.

2.7 Conclusion

Our novel loop aware progress dependency analysis technique can diagnose faults in large scale HPC applications with high accuracy. These are faults, like hangs and performance slowdowns, that are a dominant class of software problems encountered in HPC applications. This fully dynamic technique is easy to use and does not require modifications to the application. Its ability to handle complex loops and its approach based on runtime analysis makes it more accurate and precise in debugging complex applications, compared to existing state-of-the-art techniques [14, 15]. Further, we achieve high scalability by using Markov models to summarize the application's dynamic control-flow as well as deploying a binomial reduction of the models across tasks. Our fault injection study on 4 major applications and 2 NAS parallel benchmarks show that the least-progressed task identified through this technique can be effectively used to identify the root-cause, i.e., the faulty task and corresponding code region. On average PRODOMETER achieved over 93% accuracy and 98% precision. The case study presented in this section shows how this technique was able to diagnose an unknown non-deterministic bug, reproducible only at large scale, in a full scale dislocation dynamics simulation code.

3. INPUT AWARE PERFORMANCE ANOMALY DETECTION

3.1 Introduction

Techniques to predict the dynamic properties of application executions are a critical part of various tools. For example, schedulers need to predict an application's execution time and resource use (*e.g.,*, network bandwidth requirements), while anomaly detection tools need to differentiate an application's normal behavior from anomalous behavior that indicate software or hardware problems. Similarly, performance profiling tools need to describe how various parts of an application utilize system resources to enable developers to focus their code optimization efforts. The key capability required by these tools is to predict, before an application executes, various metrics of its execution (*e.g.,*, total execution time, energy use, cache miss counts and code execution paths), both at the granularity of the whole application as well as for individual code segments (*e.g.,*, function calls and loops).

Prior work on statistical techniques to make such predictions has demonstrated their utility in the design of real tools [41,42]. These techniques observe multiple runs of the target application, or of individual code regions (a given code region may be executed multiple times in a single application run) to build a statistical model of the various metrics that are observed during these runs. These models are then used to either predict the values of these metrics for future application runs or to determine whether a given metric value is consistent with a normal application execution or is somehow anomalous. *A key observation of prior research is that to predict these metrics accurately, modeling techniques must take into account properties of application input, configuration and state* (denoted collectively here as "parameters"). For exam-

ple, to predict the execution time of a shortest-paths graph algorithm it is necessary to know properties of the graph, such as its diameter.

However, the key challenge that limits the use of these techniques in practice is that the number of application parameters that control application behavior is large and their values may span a large range. Hence, for a reasonable-sized training set for the statistical techniques, it can only cover a very small fraction of the overall parameter space. This means that most predictions that will be made in reality will be on application parameters that are outside the predictor's training set. Expectedly, the accuracy with which a model predicts the behavior of a given application decreases as the run grows more different from the runs the model was trained on. For example, when an application runs in production environment, it may use a larger scale and a larger and different characteristic dataset. The application behavior then shows large deviations from that predicted by the model. Hence, we have to either ignore the predictions of the model altogether or run the risk of high false alarms or poor resource utilization, depending on the use case for the prediction. As a simplified example of a real bug, consider the situation in Figure 3.1. Here, an unseen value of a command-line parameter (q) causes a performance anomaly. During software testing, the values of q used along with calculated values of r from input-data leads to moderate values of `iterCount` which control how many times a compute intensive routine `doMoreCalculations` will be invoked.

What we would rather like to have is a way to characterize the accuracy of the prediction as the parameters deviate from those seen during the training runs. Armed with such a characterization, we can do the equivalent of "uncertainty quantification" for our prediction. We could, for example, put statistically rigorous error bounds on the prediction. This translates to usable characterization of uncertainty in the various use cases. For the anomaly detection use case, this could allow the user to set application-specific bounds on the rates of false alerts and missed alerts. For performance analysis tools, this could let the user set bounds on the resource utilization of a region of the code.

Code for a hypothetical program:

```
1.  void main (argc, argv){

2.    int p = readInput()
3.    int q = argv[1];
4.    int r = calculationsOnInput(p);
5.    float s = otherCalculations()
6.    float someVal = (r % q) + 0.001;
7.    int iterCount = s / someVal;
8.    for (int i=0; i < iterCount; i++){
9.          doMoreCalculations();
      }
    }
```

Testing values:

q	Calculated r	Calculated s	SomeVal	iterCount
2	21	10	1.001	9
3	14	20	2.001	9
5	28	30	3.001	9
40	85	40	5.001	7

Production values:

>$ ./prog –arg1=20 #performance anomaly

q	Calculated r	Calculated s	someVal	iterCount
20	40	10	0.001	**10000**

Fig. 3.1.: A program takes a command-line parameter (stored in q) and reads an input data file. For certain combination of q and input data, the calculated iteration bound of a loop containing a heavy calculation becomes very large, leading to a performance anomaly.

In this section, we provide a method to precisely address this requirement, *i.e.*,, quantify the accuracy of the prediction. It describes a technique to use a limited training set of application runs, that cover a small fraction of a target application's parameter space, to train an arbitrary regression model that predicts the application's behavior across this entire space. Further, it quantifies the errors of these predictions as a function of how "different" they are from the training runs. Our technique takes as input a set of observations of the parameters and behavioral metrics of the executions, of entire applications or individual code regions[1]. Examples for behavioral metrics are the execution time, the number of instructions/floating point instructions, the percentage of conditional branches taken, etc. It then trains regression models on this dataset that predicts the behavioral metrics of code regions given their parameters. Our first contribution is a technique to create a distribution of the model's prediction error for parameter values *within* the region of the parameter space that the model was trained on. Our second contribution is a method to extrapolate the

[1]Whether we will use a single model for the entire application or have one for arbitrarily-defined code regions is a choice that depends on how homogeneous the behavior of the application is, across its different code regions.

prediction error outside this region. This is accomplished by considering one parameter at a time and creating the error characterization for that parameter. We then combine the error characterizations when all the parameter values deviate from what had been seen during the training runs.

We demonstrate the utility of our approach by creating a concrete performance anomaly detection tool, called GUARDIAN. We show off the use case of anomaly detection in seven popular applications, drawn from a diversity of domains — scientific simulations, matrix-vector algebra, graph search, and financial options trading.

Figure 5.6 shows the complete workflow of the anomaly detection system composed of data-collection framework (SIGHT), SCULPTOR (modeling tool), E-ANALYZER (error analysis tool) and GUARDIAN (anomaly detection engine). The toolchain is composed of

- SIGHT, which tracks application code regions, their parameters and behavioral metrics,
- SCULPTOR, which a builds statistical model for each code region that predicts the region's behavioral metrics from its parameters,
- E-ANALYZER, which quantifies the prediction error of SCULPTOR, and
- GUARDIAN, which detects anomalies in application behavior by comparing actual observations of code region behavioral metrics to their values predicted by SCULPTOR, accounting for the prediction error computed by E-ANALYZER.

Experimental evaluations demonstrate that by using error predictions computed by E-ANALYZER, GUARDIAN is able to achieve false positive rates below 2% which is a 94% improvement over over anomaly detection techniques that ignore model prediction error. Further, our studies that inject synthetic faults into application execution show that this improvement is achieved without reducing the detection accuracy relative to these alternative techniques.

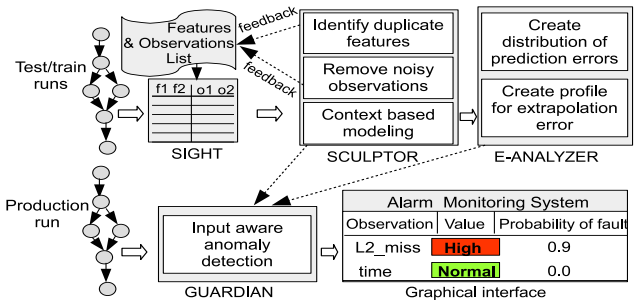


Fig. 3.2.: Workflow of anomaly detection system

3.2 Data collection framework

Example of SIGHT APIs:

```
float func (float x, float y) {
    SightModule funcMod("func", inputs("x",x,"y",y));  // creating a module for the scope of this function
    DoCalculations();                // some computation done by the function
    r = calculateResidual ();   // function computes a residual value to verify convergence
    funcMod.outputs("r", r);    // SIGHT tracks the residual as an output observation
}
```

Fig. 3.3.: SIGHT instrumentation, context aware modules.

We collect parameters and behavioral metrics at the granularity of arbitrary application code regions (denoted "modules"), which may include code blocks, functions or the entire application. This data is collected using a data collection tool we developed called SIGHT, which provides simple APIs for developers to annotate the modules to be modeled and specify the information to be collected for each one. Section 3.3.1 details the parameters and behavior metrics we collect as part of our analysis. Figure 3.3 shows an example of SIGHT annotation APIs that user may use to mark a module (the duration of function `func`'s module is the lifetime of variable `funcMod`), report its parameters (`x`, `y`) and behavior metrics (output `r` and performance metrics). Modules are tracked in a context-aware manner, collecting observations separately for a single module depending on the calling context it is executed in. Currently module annotation and specification of module inputs and outputs is manual, while call context identification and collection of performance metrics is automatic. While automatic identification of the proper granularity for the modules and the choice of module pa-

rameters is orthogonal to our work, SCULPTOR provides feedback to the developer (detailed in Section 3.3.2) if it determines that the data provided is not sufficiently detailed to create an accurate model.

3.3 Modeling application behavior

In this section we describe our model building tool, SCULPTOR and how it statistically models application behavior.

3.3.1 Application properties

Accurate modeling and prediction of application behavior requires knowledge of the parameters that control it and metrics that quantify it.We execute a limited number of application training runs (e.g. during regression testing) and use our data collection framework (discussed in Section 3.2) to collect the parameters and behavior metrics of each application module. We introduce two terms that will hold center stage through the rest of the paper — **input features** and **observation features**. *Input features* are comprised of all collected parameters — the configuration properties, the command-line arguments, the input properties of a data structure in the application, such as matrix sparsity, graph diameter, or the loop iteration index. *Observation features* are comprised of all those features that quantify the behavior of the application. Application behavior is quantified in terms of various performance metrics (*e.g.,*, execution time or hardware performance counter values), application-level quality metrics (*e.g.,*, video frame rate, residual values in a linear algebra solver) and output values from a module (*e.g.,*, number of search results).

Table 3.1 lists the applications we studied and a few of the input and observation features that SIGHT collects for each of them. CoMD (Classical Molecular Dynamics) [43] and LULESH(Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [25] are scientific simulations. FFmpeg [44] is widely used video processing software. SpMV [45] is a benchmark for sparse matrix vector multiplication which

appears in scientific applications. LINPACK [46] solves a dense system of linear equations while PageRank [47] is graph search algorithm and used to compute a ranking of all the vertices. Black-Scholes [48] benchmark is a popular stock option pricing benchmark. Note that since our goal is statistical predictability rather than logical correctness, the features we selected are aggregate summaries of application state, whose values characterize application behavior. We do *not* try to model or predict the values of individual scalar values in application state, *e.g.*,, values of individual elements of the output matrix in case of matrix multiplication).

3.3.2 Feature selection

The accuracy of a model depends on the granularity with which an application's behavior is modeled and the utility of the input features for predicting the observation features. At the time of instrumentation, it is not always clear to developers what granularity and choice of features is most useful. This makes it necessary to preprocess this information to make it maximally useful for subsequent analysis and to identify ways in which the feature set may be improved. Concretely, our preprocessing procedure (i) identifies the input features that are unlikely to be useful

Table 3.1.: Examples of input and observation features. Performance measures (perf measure) include, time, total number of instructions executed (TOT_INS), number of floating point instructions(FP_INS), number of load instructions (LD_INS), number of L2 data cache miss (L2_DCM), cache miss-rate (L2_DC_MR), MFLOPS etc. PSNR is peak signal-to-noise ratio.

Applications	Input features				Observation features
	Configuration parameters	Properties of input data	Runtime parameters	Internal variables	
LULESH	num of cycles to run, length of cube mesh, number of distinct regions		num processes	timestep loop iteration number	absolute diff, relative diff, perf measure
SpMV	memory-limit, time-limit	size, sparsity	block-size		mflop rates, perf measure
FFmpeg	crf,target resolution	size, bitrate,resolution			PSNR, output bitrate, perf measure
Black-Scholes	num runs	num options	num threads		avg error in predicted price, perf measure
LINPACK	dimension		num processes	num operations	perf measure
CoMD	num of unit cells, lattice parameter, time step		num processes	timestep loop iteration number	result quality, perf measure
PageRank	number of vertex	graph density			convergence diff, perf measure

for predicting *any* observation feature so that they can be ignored by subsequent analysis and (ii) identifies the observation features that cannot be predicted so that the developer can be prompted to provide additional information as new input features. SCULPTOR accomplishes these by using a theoretical technique *Maximal Information Coefficient* (MIC). We first give a brief background on MIC and then explain how it is used by SCULPTOR.

Background: Maximal information coefficient analysis

The MIC [49] algorithm attempts to determine the strength of the relationship between a given pair of input and observation features, regardless of the actual shape of this relationship and works with both functional and non-functional relationships. MIC is based on the key intuition that if a relationship exists between two variables, then a grid can be drawn on the scatterplot of the two variables that partitions the data to encapsulate that relationship. Thus, to calculate the MIC of a set of two-variable data, we explore all grids up to a maximal grid resolution, dependent on the sample size, computing for every pair of integers (x, y) the largest possible mutual information achievable by any $x \times y$ grid applied to the data. We then normalize these mutual information values to ensure a fair comparison between grids of different dimensions and to obtain modified values between 0 and 1. We define the characteristic matrix $M = (m_{x,y})$, where $m_{x,y}$ is the highest normalized mutual information achieved by any $x \times y$ grid, and the statistic MIC to be the maximum value in M. MIC has been shown to find relationships that cannot be identified by other methods such as Spearman correlation coefficient or linear regression.

Using MIC as a filter

A key goal of our data collection mechanism is to minimize the developer's instrumentation effort. A key aspect of this is that developers are free to provide as many input features as they like, without worrying about whether they are actually

correlated with each other or with the observation features. However, this means a filtering step is needed before the input features are considered in our model.

Identify duplicated features: One issue that may occur is that multiple features actually represent a single piece of information. For example, in `LINPACK`, problem size `N` and internal variable `ops` (number of floating point calculations), capture the same information and are related by the equation: $ops = (2/3)N^3 + 2N^2$. SCULPTOR performs a MIC analysis between the input features to identify such duplicates (MIC-score (≥ 0.95)) and removes them.

Remove noise: If an input feature is not related to *any* observation feature, including it as an input to a statistical model can add noise and reduce the model's accuracy. These are detected by checking if there exist any observation features for which their MIC is larger than a threshold and are removed if there are none. Conversely, there may be observation features that are not related to *any* input features. These are detected as above and are then communicated to the developer to encourage them to collect more expressive input features that might be useful for predicting these observation features. It might happen, even after few iterations of input feature refinement that some observation features are still not related to any input features. For example in FFmpeg, given an input bitrate, resolution, and a quality metric *Constant Rate Factor (CRF)* [50], we could not properly predict bitrate of the output video. This is because the output bitrate also depends on the actual content of input video (such as motion) and while maintaining same quality (CRF), FFmpeg applies more compression to a high motion video than to a slow video, leading to low output bitrate for the former. Since here we are not considering the actual content of the video, we can not properly predict output bitrate. This kind of observations features with low prediction accuracy are recorded and discarded from further analysis.

3.3.3 Choosing the best regression function

While MIC technique can identify useful input features, it cannot provide a model that predicts the observation features from these input features. SCULPTOR creates a predictive model for each observation feature by applying a range of linear regression methods. In the experiments reported in Section 3.6 we used Least Absolute Shrinkage and Selection Operator (`LASSO`) [51] with a polynomial fit (upto degree 3), which work well for the applications we studied. Thus, a regression model is built for each output feature separately in terms of potentially all the input features. LASSO has the property that it tries to reduce the sum of coefficients of the terms, ultimately leading to a smaller number of terms. Our model, being of order three, has terms of the type $x_1, x_1^2, x_1^3, x_1x_2, x_1^2x_2, x_1x_2^2$ and being a linear regression model, it is a linear combination of such terms. SCULPTOR also verifies the quality of fit and avoids over-fitting using `k-fold` cross-validation (for our experiments we used $k = 3$). We record the observation features for which the prediction model could not achieve a sufficiently high R^2 fit score and do not use those as *detectors* for anomaly detection use-case, because such observation features might raise false alarms.

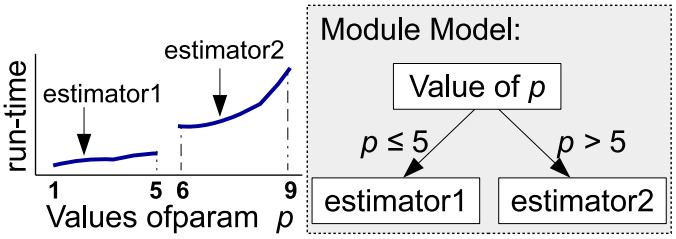


Fig. 3.4.: Handling discontinuous behavior with incremental model update. Here run-time shows discontinuous behavior for values of input parameter p larger than 5. We add a new estimator to the model to predict that behavior.

3.3.4 Incremental model update

In some scenarios, the application's performance behavior might be discontinuous. For example, there might be a big jump in run-time when the size of working set no-longer fits in L1-cache. Such discontinuous behaviors typically happen when the application hits some resource bottleneck, say, available cache, memory, network, I/O or memory bandwidth. Another cause of discontinuity might appear in the presence of input features that act as control variables for the program flow, leading to multiple behaviors of a module. When SCULPTOR finds a discontinuous behavior in the observations, it adds a new estimator to model the next continuous section after the discontinuity. Thus, for each continuous segment of an observation behavior, with few discontinuities, SCULPTOR maintains separate estimators in a *decision-tree*-like data structure, as illustrated in Figure 3.4. Since we handle such discontinuities by incrementally updating our model to capture the behavior, users do not have to retrain the entire model.

3.4 Quantification of prediction errors

Our modeling is based on a set of measurements of a set of application runs, denoted *"available data"*, that cover some representative portion of the input feature space of its modules. When the application is executed in production, the input feature of its modules are denoted as *"production points"*. In an ideal scenario, available data could be gathered by sweeping through all possible points in the input feature space and collecting the corresponding observation features. While this would be ideal since every production point's observation features would be known, it is not practical since the input feature space is far too large for most real-world applications.

If the model created based on available data is not perfect (which is the case for most real applications), this situation leads to predictors that behave reasonably well when the production point is close to available data, but its prediction error grows as the production point moves further away from it. We call this the *"extrapolation*

error". Moreover, since prediction models generally cannot capture all the details of a dataset, there is some prediction error even for the production points within the available data set. We call this the *"interpolation error"*.

Quantification of such interpolation and extrapolation error is necessary for many use cases. For example, a job scheduler needs to consider the uncertainty in a job's predicted execution time when deciding whether to schedule it in a given time slice, since the job will be aborted if it runs over its allocation. In this paper, we focus on a use case of anomaly detection, where interpolation and extrapolation errors are used to calibrate the thresholds that determine whether a value of the given observation feature is sufficiently different from its predicted value to signal an alarm. Sections 3.4.1 and 3.4.2 present error estimation for interpolation and extrapolation error, respectively.

3.4.1 Distribution of prediction errors within the available data: Interpolation error

We characterize the interpolation error of our model by systematically evaluating the distribution of its prediction error within the available data set. Our error analysis engine, called E-ANALYZER, uses *Bootstrapping* to create multiple small training and test subsets of the available data set. Members of each subset are sampled at random, while ensuring each training set is disjoint from the corresponding test set. Then SCULPTOR trains a model on the training set and uses it to predict the observa-

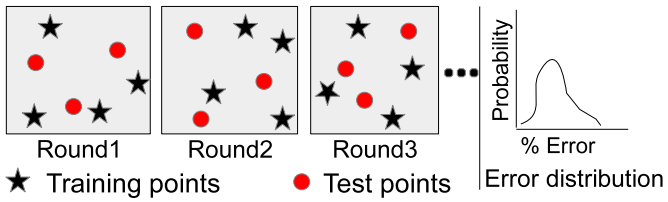


Fig. 3.5.: Resampling train and test data points from training data space. Multiple such rounds are performed to obtain an error distribution for each predictor.

tion features of the points in the test set, and records the distribution of these errors. Finally, the points are placed back into the set of available data and more samples are taken until the distribution of prediction error reaches statistical significance. The result is a histogram of model prediction errors. Since the raw counts of prediction errors are noisy, E-Analyzer uses "Kernel Density Estimation" (KDE) to smooth out irrelevant variation. Finally, these smoothed histograms are normalized to add up to 1, so that they can be interpreted as the probability distributions of error magnitudes (note that individual values may be larger than 1). These distributions, provide an insight into the model's accuracy in the case where test data resembles training data. For example, a highly accurate model will have a narrow distribution around 0, while an inaccurate one will have high probability mass away from 0. Further, these distributions make it possible to calculate `Prob(observedValue | predictedValue)`, the probability that a particular observation feature value `observedValue` will be seen in a normal execution, given that the model predicted a value `predictedValue` for that observation feature at that production point. In Section 3.5, we show how we use this information to achieve an anomaly detection scheme with low false positives. In Fig. 3.9, we show the empirically measured distribution of interpolation errors for a few interesting observation features from the applications that we studied.

3.4.2 Error as a function of distance: Extrapolation error

Models generated by Sculptor can predict the expected performance characteristics of the application with high accuracy, when production data points are close to the available data used for modeling. For production points which are far from the available data, the error in prediction typically increases. We want to quantify such extrapolation error as a function of the *distance* between the input feature points in the available data and those encountered during production runs, as illustrated by the graph in Figure 3.6 for a 2-dimensional input-space.

Distance measurement

Our approach to quantifying the dependence of model's error in predicting the output features of a given production point leverages the points in our available data set. First we look at the space of input feature values of the points in the available data set and train the model on just the points in a corner of this space. We then observe how prediction errors grow as this model attempts to predict remaining points within the available data set as their distance to the model's training points grows.

This approach faces two major challenges. First, we must compute the distance from one input feature vector to another. The Euclidean or Mahalanobis distance formulas cannot be used here because the different dimensions of these vectors may span vastly different ranges of values. Further, for some input features, such as the *size* feature in PageRank [47] or the matrix size in LINPACK [46] there is no a priori bound on this range and therefore we can not normalize to make all features comparable in terms of numeric values.

The second issue is that we must find the region of the input feature space on which to train our model. The ideal location would be a corner in the space, as shown in Figure 3.6 for the example of a 2D space, so that we can select test points that move away from the training sub-region in *equal steps* along *all dimensions*. This approach does not work in a high-dimensional space, because the training data is too sparse in practice for there to be a large number of points in any corner.

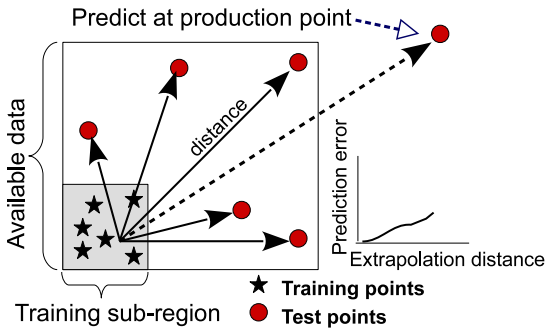


Fig. 3.6.: Quantifying prediction error with extrapolation distance.

We propose an approach that addresses both of the above issues. The key idea is that although it is difficult to find a sufficient number of points in a corner of a high-dimensional space, this is made much easier if we cover the space by considering one dimension at a time. This way we can train a model on points in the input feature space that have small or large values in a given dimension and consider how prediction errors increase as the point we predict moves away along the same dimension. We can then combine these per-dimension error measurements to create the cumulative error for distance along all dimensions and compute prediction error at a particular production point (detailed in Section 3.4.3).

Algorithm 3 details our approach. For each input feature dimension E-Analyzer first *sorts* the data-points with respect to this feature. It then creates set S to contain the data-points that have similar values for the remaining features and selects the k points from S with the smallest values for this feature to create set S_{train}. It then trains a regression model on S_{train} and predicts the value of each observation feature for the remaining points in S. Finally, it records how the error in predicting each observation feature for each point p relates to its distance from S_{train} (errors for each combination of observation and input features are tracked separately). The distance is computed as $|\frac{f_p - \mu_{S_{train}}}{\sigma_{S_{train}}}|$, where f_p is the value of the current input feature dimension at point p and $\mu_{S_{train}}$ and $\sigma_{S_{train}}$ are the mean and standard deviation, respectively, of the value in this dimension of all points in S_{train}. The choice of k trades off between providing enough points to train an accurate model and leaving enough points to capture the scaling of extrapolation error. We found that `k`=50% worked well in our test cases.

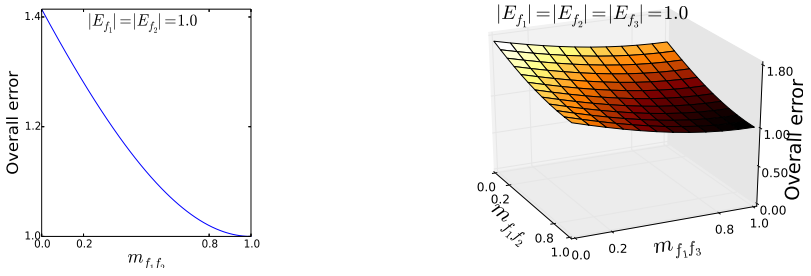
After the pairs ⟨distance, prediction error⟩ are collected for each input feature and observation feature dimension, E-Analyzer builds a simple polynomial regression model on this data. This makes it possible to predict how prediction scales as distances grow beyond our set of available data, and we maintain one such estimator for each combination of input and observation feature dimension.

Algorithm 3 Creating extrapolation error profile estimators

```
1: F ← Set of features
2: O ← Set of Observation features
3: D ← All available data-points and corresponding observations
4: procedure CreateErrorProfile
5:     for all f in F do
6:         Sort D w.r.t f
7:         S ← Get data-points from D with similar values for {F − f}
8:         S_train ← First k data-points in S
9:         model ← CreateModel(S_train)
10:        for all pt in {S − S_train} do
11:            for all o in O do
12:                d ← Distance(pt,S_train)
13:                e ← PredictionError(pt,o,model)
14:                errorProfile[o][f].addToList(d,e)
15:            end for
16:        end for
17:    end for
18:    for all f in F do
19:        for all o in O do
20:            errEst ← CreateEstimator(errorProfile[o][f])
21:        end for
22:    end for
23: end procedure
```

3.4.3 Model error estimation

Given the input feature values for a production application run, we want to estimate the error in the model's prediction of each of its observation features. For a given observation feature, we want to do this by applying the error estimation model of each input feature dimension and combining the errors that all the single-dimension models predict. While it is possible to combine these error estimates via a simple aggregate such as an average or the Root Mean Square (RMS) norm, this estimate of overall error would be biased because some of the input features are correlated with each other. Giving equal weight to each of these correlated features would have the effect of weighting the data source behind this correlation with disproportionate importance. We thus designed a new technique to account for such correlations that gives the same weight to all sources of information that feed into the error estimates along all the individual input features. It works by removing weight from error predictions along individual input feature dimensions in proportion to how correlated they are to other dimensions that have already been accounted for. We start with the the RMS norm of the error predictions along all input features. Then we select one feature as a base and compute the correlation between it and all other features using the *Maximal Information Coefficient* (MIC), which ranges between 0 and 1. Finally, we reduce the weight of the other features in proportion to the correlation and increase



(a) Variations in $m_{f_1f_2}$ (b) Variations in $m_{f_1f_2}$ & $m_{f_1f_3}$

Fig. 3.7.: Variation of overall error with variation in MIC between (a) 2 error components, (b)3 error components

the importance of the base feature by accounting for the correlated contributions. Thus the *overall error* $\Psi(E_{f_1}, E_{f_2}, E_{f_3}, ...) =$

$$\sqrt{[E_{f_1}]^2 + [(1 - m_{f_1 f_2})E_{f_2}]^2 + [(1 - m_{f_1 f_3})E_{f_3}]^2 + ...} \tag{3.1}$$

Where f_1, f_2, f_3 etc. are the features while E_{f_1}, E_{f_2}, E_{f_3} etc. are the individual error components with respect to those features. We chose E_{f_1} as the *base error metric*. $m_{f_1 f_2}$ is the MIC of E_{f_2} w.r.t E_{f_1}. Similarly, $m_{f_1 f_3}$ is the MIC of E_{f_3} w.r.t E_{f_1} and so on.

Intuitively, in a correlated error scenario, while calculating contribution from E_{f_1} we have already accounted for some of the contributions (captured through MICs: $m_{f_1 f_2}$, $m_{f_1 f_3}$ etc.) from other components such as E_{f_2}, E_{f_3} etc. Thus, while calculating the overall error, we reduce the raw contributions of E_{f_2}, E_{f_3} by a factor of $1 - m_{f_1 f_2}$ and $1 - m_{f_1 f_3}$ respectively, to maintain the balance.

As can be seen, this formula gracefully satisfies two boundary conditions. When all the components are *independent* (i.e. $m_{f_1 f_k} = 0$), it reduces to the RMS norm. Further, when all the error components are *fully correlated* (i.e. $m_{f_1 f_k} = 1$ and $E_{f_1} = E_{f_2} = E_{f_3}$), we only consider error coming from the base feature *i.e.,* E_{f_1}.

To provide a stronger intuition for how Equation 3.1 works, Figure 3.7 shows the overall error it computes across input features f_1, f_2 and f_3 (error is predicted to be 1.0 for each dimension), as the correlations (MIC) among f_1 and f_2, and f_2 and f_3 are varied. It shows that overall error computed by Equation 3.1 lies in the range $[1, \sqrt{2}]$ when considering just two features and in the range $[1, \sqrt{3}]$ when considering all three. As correlation increases from 0 to 1 the total error approaches 1.0 and when no features are correlated, the predicted error rises to $\sqrt{k}$ for k features (identical to RMS norm). It might be interesting to understand why overall error predicted by Equation 3.1 would grow with number of features contributing to the prediction error. The reason is, as we extrapolate along more input feature dimensions, we move further away from the training region of the prediction model. Hence, it is only natural that our prediction error due to extrapolation would grow.

3.5 Anomaly detection as a use case

This section evaluates the utility of our error prediction technique by using it as a part of a probabilistic anomaly detector engine, GUARDIAN. By anomaly we mean, deviations of values seen for the observation features of an application module (whole application or a code region) relative to values that would be expected given the module's input features. Currently anomaly detection (*e.g.*,, [41, 52, 53]) is done by creating a statistical model, predicting the behavior at the production point using the statistical model, and then using a predefined thresholds (*e.g.*,, 50% deviation) for the difference between the predicted and the observed value. The current approach does not take into account the distance of input features from the training space. In contrast to this, GUARDIAN uses interpolation and extrapolation error analysis performed by E-ANALYZER (as discussed in Section 3.4) to calibrate the threshold based on the *distance* of production point from the training space. GUARDIAN only uses the input and observation features selected by SCULPTOR during modeling as discussed in Section 3.3.2.

Algorithm 4 Anomaly detection during production runs

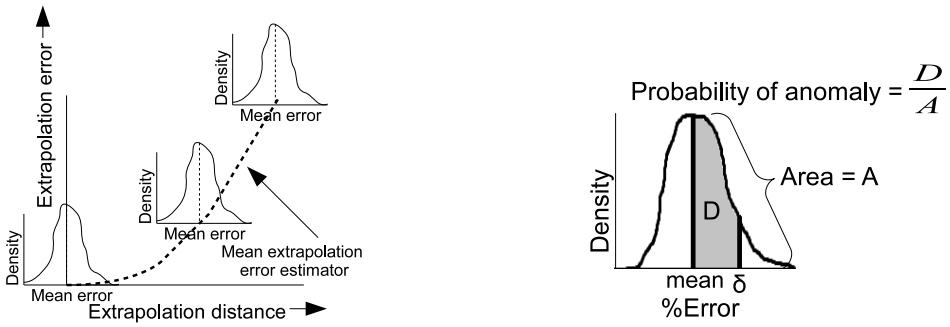
```
1: P ← Production data-point (input features in production run)
2: Oset ← Set of selected Observation features
3: predictorObjects ← Dictionary of predictor objects for each observation
4: procedure AnomalyDetectionPerModule
5:     for all obs in Oset do
6:         doAnomalyDetection(obs)
7:     end for
8: end procedure
9: procedure doAnomalyDetection(obs)
10:     actualVal = getObservedValue(obs)
11:     predVal = predictorObjects[obs].pred(P) //Calculate predicted value at P
12:     E_overall = calculateOverallExtrapolationError(P) //Expected error
13:     ObservedError = |actualVal − predVal| / predVal
14:     if ObservedError ≤ E_overall then
15:         δ = 0
16:     else
17:         δ = ObservedError − E_overall //Deviation from expected error
18:     end if
19:     prob = calculateProbability(δ)
20:     if prob > threshold then
21:         Flag anomaly error for obs
22:     end if
23: end procedure
```

The key idea of our approach, detailed in Algorithm 4 is as follows. During a production application run, our SIGHT tool observes the input and observation features of multiple modules. We use the models generated by SCULPTOR to predict for each observation feature the value $predVal$ that is expected to be observed given the module's input features, and compare it to the value $actualVal$ that is observed during the production run. If the percentage difference of $actualVal$ from $predVal$ is within the range of prediction error due to extrapolation ($E_{overall}$), we consider that value as *normal* because we know that this difference is within our predictor's error bounds. If $actualVal$ is outside this range of mean extrapolation error, we use the probability distribution of interpolation error to calculate the probability that $actualVal$ deviates from the $predVal$ because of the prediction error or because this production run was anomalous.

As before, let $f_1, f_2, ...$ be the set of pre-selected input features of a given module. Let $P(f_1 = x_1, f_2 = x_2, ...)$ denote the input features observed for a single execution of a module during a production run, where features hold values x_1, x_2, etc. respectively. Let o be an observation feature we wish to validate for anomalies. First, using our predictor models, GUARDIAN calculates the predicted value of o as: $predVal = pred(P)$, as discussed in Section 3.3.3. Then we calculate the *distance* of P from the training sub-regions along the feature dimensions as discussed in Section 3.4.2. Let d_{f_k} be the extrapolation distance of P along feature f_k. We calculate the error components E_{f_k} at P using extrapolation error estimators of each feature: $E_{f_k} = ErrEst_{f_k}(d_{f_k})$.
These components are then combined to calculate the overall extrapolation error $E_{overall}$ according to Equation 3.1. Let $actualVal$ be the value observed for observation o during the production run. To compute whether $actualVal$ is anomalous we first compute δ, the *deviation* of prediction error from the expected extrapolation error. If, $actualVal$ is within the range $predVal \pm (predVal \times E_{overall})$, then we consider $\delta = 0$. Otherwise, we calculate the deviation from expected extrapolation error as:
$\delta = |\frac{actualVal - predVal}{predVal}| - E_{overall}$ (Lines 10-17 in Algorithm 4)
Now, it is to be noted that, $E_{overall}$ is essentially the *mean* of extrapolation errors

at that particular data point P. As discussed in Section 3.4.1, prediction errors due to interpolation error generally form a distribution around the mean value, as illustrated in Figure 3.8a. We assume that properties of this distribution remain the same irrespective of the distance of the production points from the training set, while extrapolation error represents how the mean value of the distribution changes with distance from the training set. Therefore, intuitively, we check whether it is unlikely that during normal execution we would see a larger deviation (of the observed error from the expected extrapolation error) than δ, and if so, signal an anomaly. Concretely, we calculate the *probability of an anomaly* given a δ deviation of observed error from its predicted error by calculating what is the probability that deviation of error under normal execution will be lower. If this calculated probability is lower than a threshold, then we consider the execution as normal, otherwise it is flagged as anomalous.

As illustrated in Figure 3.8b, GUARDIAN calculates the probability that we can experience a *higher* deviation(δ) from the expected extrapolation error under normal circumstances. If the area under the distribution curve, between the mean (*i.e.*,, $E_{overall}$) and ∞, is A and the area under the curve between the mean and δ is D, then we calculate the probability of anomaly as: D/A.When this probability is high,



(a) Prediction error estimation

(b) probability of anomaly

Fig. 3.8.: **a)** Estimation of prediction error: Interpolation error distribution is put on top of estimated mean extrapolation error. **b)** Calculating *probability of anomaly* from distribution of errors, for a deviation of δ from expected extrapolation error.

we flag that observation as anomalous. Algorithm 4 summarizes these main steps which are followed during production run, for each code module in the application that has a prediction model.

3.6 Evaluation

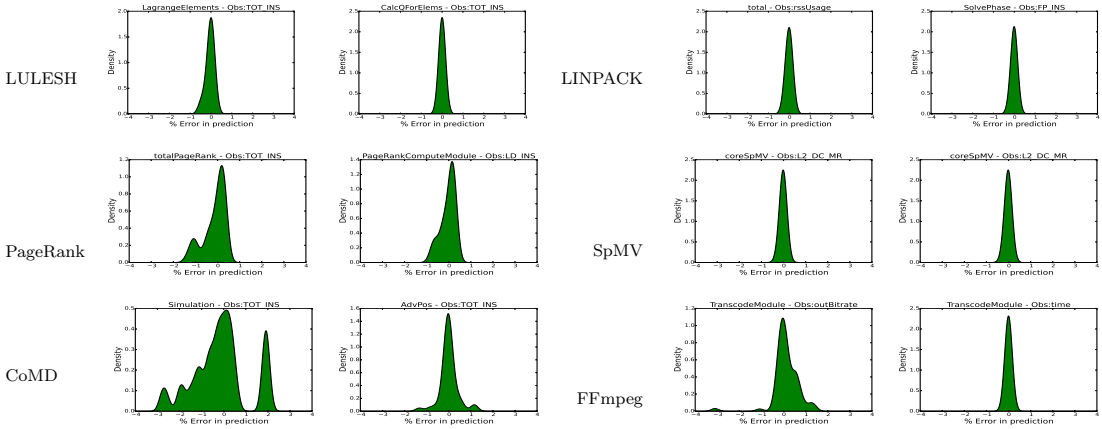


Fig. 3.9.: Distribution of interpolation errors for applications

3.6.1 Error analysis case studies

In this section we experimentally evaluate how effective our error prediction technique is at improving the effectiveness of the GUARDIAN tool for detecting anomalies in application behavior. Our evaluation focuses on the seven representative applications and benchmarks listed in Section 3.3, selected from a wide variety of domains: CoMD, LULESH (scientific simulations), FFmpeg (video processing), SpMV, LINPACK (numerical processing), Black-Scholles (financial modeling) and PageRank (graph processing).

Figure 3.9 focuses on the interpolation errors (predictions for points within the bounds of the training set), as described in Section 3.4.1. It includes two plots for each application each showing the probability distribution of interpolation errors for two representative combinations of module (code region that is measured) and observation

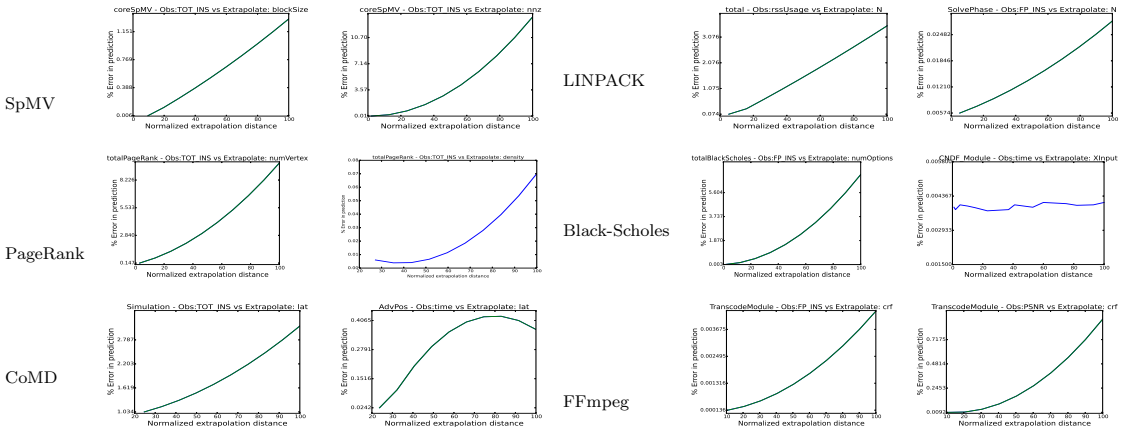


Fig. 3.10.: Extrapolation errors for applications. "Extrapolate: X" in the titles mean, extrapolation is along input feature dimension X.
SpMV: nnz is the number of non-zero rows. LINPACK: N is matrix size. PageRank: $numVertex$ is the number of vertex in the input graph, $density$ is the density of the graph. Black-Scholes: $numOptions$ is the number of stock options in input data, $XInput$ is the input to CNDF function. CoMD: lat is lattice parameter. nx is the number of unit cells in x. FFmpeg: crf is constant rate factor, $inputSize$ is size of the input video.

feature that is predicted. For example, the top-left plot presents error in predicting the total number of instructions executed within calls to the `LagrangeElements` function in LULESH. The Y-axis of each plot is the density, as obtained from the Kernel Density Estimation[2], and the X-axis is the relative error in prediction. Most of the observations have a narrow distribution around the mean of zero error, which demonstrates that the modeling accuracy of SCULPTOR is high when data points fall within the bounds of training set. This accuracy is primarily controlled by the granularity at which application is tracked and the information content of the input features. For example, consider the CoMD plots in Figire 3.9, The left plot represents prediction error distribution for the entire `simulation`, which consists of setup-phase, timestep simulation loop and some epilogue. In contrast, the right plot represent the prediction errors for `AdvPos` function which is inside the timestep loop. The prediction error is notably lower for the latter, more focused, module because (i) its behavior is

[2]Y-axis is *not* the probability; area under the curve is the probability.

more homogeneous and (ii) additional parameters that control its behavior, such as the *loop iteration index*, are available when it executes, which enables more accurate prediction. The same pattern repeats for LULESH, PageRank and LINPACK, where the left plot corresponds to a larger application region than the right (in SpMV and FFmpeg we only instrumented the core function). Further, FFmpeg shows a related phenomenon where the bitrate of the output video has higher error than other predictions (left plot). This is because this bitrate depends on the actual content of the video, which we do not track as an input feature (recall our discussion in Section 3.3.2.

Figure 3.10 focuses on extrapolation error and shows representative examples E-Analyzer's predictions of how the errors in predictions of various individual output features vary with increasing distance from the model's training set along various individual input features. Each plot focuses on a specific application, code region (module), the input feature, and the observation feature. We show two examples for each application. In each plot the X-axis represents a normalized distance from the training region according to the measure we introduced in Section 3.4.2, while the Y-axis is the prediction error. As expected, extrapolation error increases with distance from training set. Interestingly, it grows with different patterns and slopes in different cases. For example, for SpMV, the error in total instruction count relative to the block size is predicted to grow linearly, while the same metric grows quadratically relative to nnz (number of non-zeroes in the matrix). Further, even when patterns are linear, a variety of slopes are observed, for example, in the LINPACK, slopes of total floating point instructions and rssUsage (resident set size) relative to the size of input matrix (N) are different. Further, in cases where we instrument code at a very fine granularity, such as a few lines of code that perform a very deterministic action, extrapolation error is independent of distance. This is the case for the *CNDF* module in Black-Scholes, which simply calculates a formula value for the cumulative normal distribution function. Another interesting example is the right-most plot of FFmpeg where E-Analyzer achieves reasonable accuracy in predicting the peak signal-to-noise ratio, a video quality metric, based on the input feature CRF which

controls the perceptible image quality, even though they are not directly related by any equations. Finally, an interesting phenomenon is observed in the right-most plot of CoMD, where the prediction error for execution time relative *lat*, decreases with distance for large distances. This is caused by the fact that a 2^{nd} polynomial was used to model this data, as discussed in Sec. 3.4.2, and a concave shape is an artifact of using this function. In the future work, we will explore the use of a wider range of function to model extrapolation errors, focusing on *isotonic* regression estimators.

3.6.2 Evaluation of Guardian

This section presents a detailed experimental evaluation of Guardian, when integrated with an input-aware modeling scheme and using threshold calibration based on our model error prediction technique. Our evaluation compares this variant of Guardian to traditional options that use fixed thresholds to decide whether to signal an alarm. The fixed threshold based scheme goes through same modeling steps as done by Sculptor but for anomaly detection uses a predefined threshold `T`, where deviations of the actual value of an observation from the predicted one that are larger than `T` (actually T% of the predicted value) are flagged as anomalous. We vary values of `T` as 10%, 50%, 100% to control the detector's sensitivity level. Further, we evaluate two variants of Guardian. In variant `A` only the mean of the error is extrapolated and any deviations larger than this mean are flagged as anomalous. Variant `B` includes the full functionality of Guardian, illustrated in Figure 3.8a. Deviations of actual errors are compared to the extrapolated mean errors and for those that are larger, the algorithm signals an anomaly if the probability of observing such a large deviation is $< 10\%$.

False alarms

In this experiment we evaluate the false positive rate (FP) of the different tool variants presented above. Specifically, we evaluate how FP of Guardian varies

as the *distance* between available training data and the production point increases. We ran each application 15 times, with production points having feature values at *low distance* (distances between 5-10 from the training space), and *high distance* (distances between 30-35 from the training space). These distances are measured using the formula defined in Section 3.4.2 and are multiples of the standard deviation of points within the training set, along each input feature. Since the applications were run under normal conditions, albeit with different parameter values, and the applications had no bugs, there should not be any false alarm.

Figure 3.11, summarizes the results. On average (the rightmost solid red bar), GUARDIAN variant A has 11% FP for production points at low distance and 4% FP for production points at high distance. GUARDIAN variant B, has only 2% FP which is significantly better than 74% and 38% average FP rates achieved by fixed threshold based detector with thresholds set at 10% and 50% of the predicted value, respectively. Threshold-based scheme with T=100% achieves comparable FP rates at low distance, but suffers from poor detection accuracy, as will be seen from the following

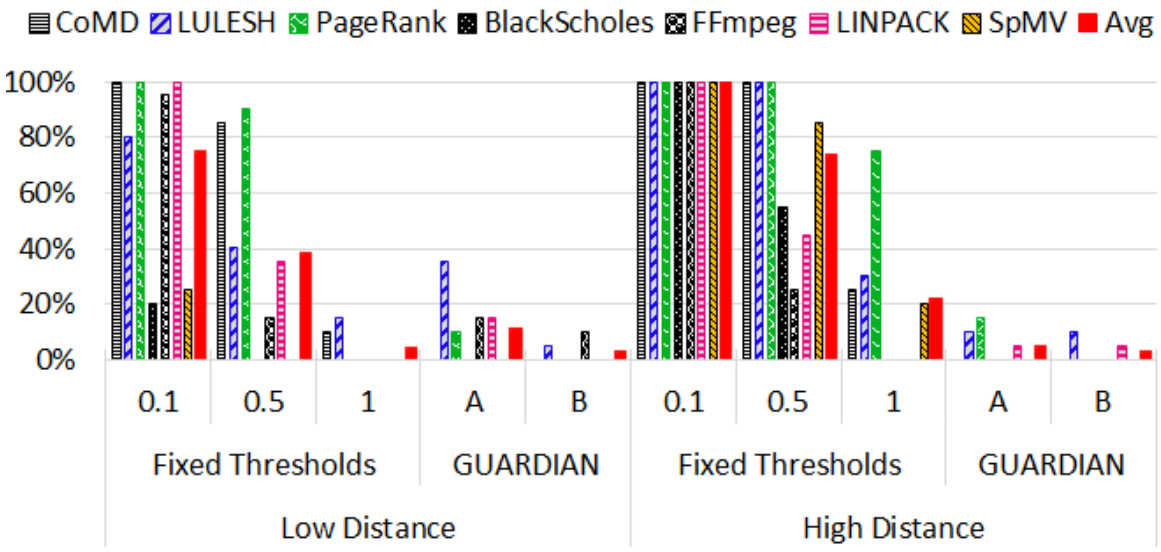


Fig. 3.11.: False positive rates: GUARDIAN (GD) vs. Fixed threshold based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Lower is better)

experiment (Section 3.6.2). For the production points at high distance, GUARDIAN shows a larger improvement relative to the threshold-based variants because it can widen the acceptable range of values with the help of extrapolation error prediction. Another important observation is that at higher distances, variants A and B perform almost equally well. The reason is, since in most of the cases, interpolation based distribution is very narrow, it becomes insignificant at higher distances, when the mean extrapolation error is large.

Accuracy

This experiment evaluates the detection accuracy of GUARDIAN in the presence of synthetically-injected performance faults. We implemented a fault injector using binary instrumentation tool Pin [23]. For each application, we inject 20 *bugs* of two categories:

- Comp: extra computing loops, and
- Mem: allocates and reads randomly and multiple times from a dummy memory array, which also creates cache contention.

We also vary the intensity of such bugs between *low* and *high*. We record how many times these injected bugs were flagged as anomaly and present the results for Comp in Figure 3.12 and for Mem in Figure 3.13. Binary instrumentation through Pin has significant monitoring overhead. To make sure, that our training runs and bug-injected runs experience similar overheads, even for training runs we injected dummy *no ops* using Pin. Since we saw in our earlier experiment that at high distances between the production and the training runs, the fixed threshold scheme is overwhelmed with false alerts, our accuracy experiments focus on low distances where the fixed threshold scheme may still be competitive with GUARDIAN. For GUARDIAN, we again use variants A and B, as discussed above and for fixed threshold-based technique, we vary the detection threshold between 10%, 50% and 100%.

We summarize the results of Comp injected faults in Figure 3.12. For the fixed threshold scheme, a threshold of T=10% has best detection accuracy. However, it is not useful in practice due to its high FP rates. On the other hand, T=100% is too insensitive and missed most of the low intensity faults. T=50% provides the best balance between the FP and detection rates and is thus a reasonable choice for comparison with GUARDIAN. Both the variants of GUARDIAN perform almost equally well and most of the time perform better than the fixed threshold scheme with T=50%, with one exception, PageRank. Our hypothesis is that in this case the intensity of the injected bug was too low compared to the normal variation of PageRank's behavior, which resulted in low accuracy for GUARDIAN. At the same time accuracy of the fixed threshold detector looks good due to its inherent high FP rate (Figure 3.11). Another observation is that variant A of GUARDIAN is slightly better than variant B because, it considers mean extrapolation error as a hard threshold and therefore is more sensitive to anomalies. For the same reason, this tends to have higher FP rates.

The accuracy results for Mem injected faults is shown in Figure 3.13 and follows the same trend. GUARDIAN variants A and B perform significantly better than fixed threshold based detection with T=50%. We noticed that even though for Mem faults we injected repeated memory read operations, the majority of the time it was detected either as increased number in total instruction count, increased L2 cache miss or increased load instructions and not by increased resident set size.

3.7 Related Work

Extrapolation: In the machine learning community, it is well-known, that quality of modeling is only as good as the training data. An interesting work [54] related to k-fold cross validation classifier provides a theoretical foundation of estimation errors but they do not cover extrapolation. In the context of pattern discovery, a recent work by Wilson *et al.* [55] used expressive closed-form kernel-based approach for Gaussian process extrapolation but did not handle multi-dimensional inputs. An

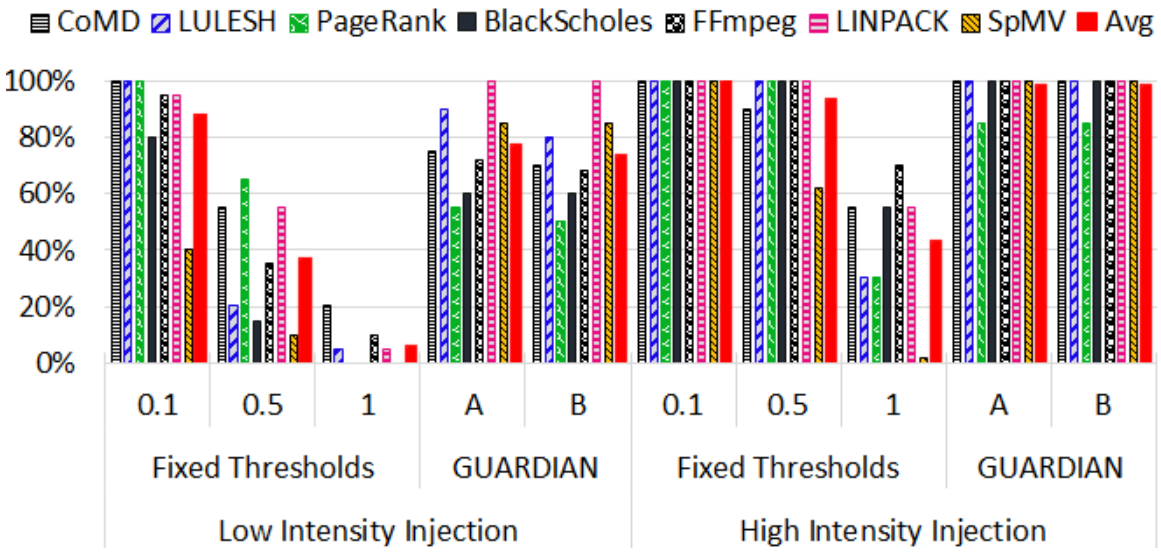


Fig. 3.12.: Detection accuracy for extra computation bug (Comp): GUARDIAN vs. Fixed threshold-based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Higher is better)

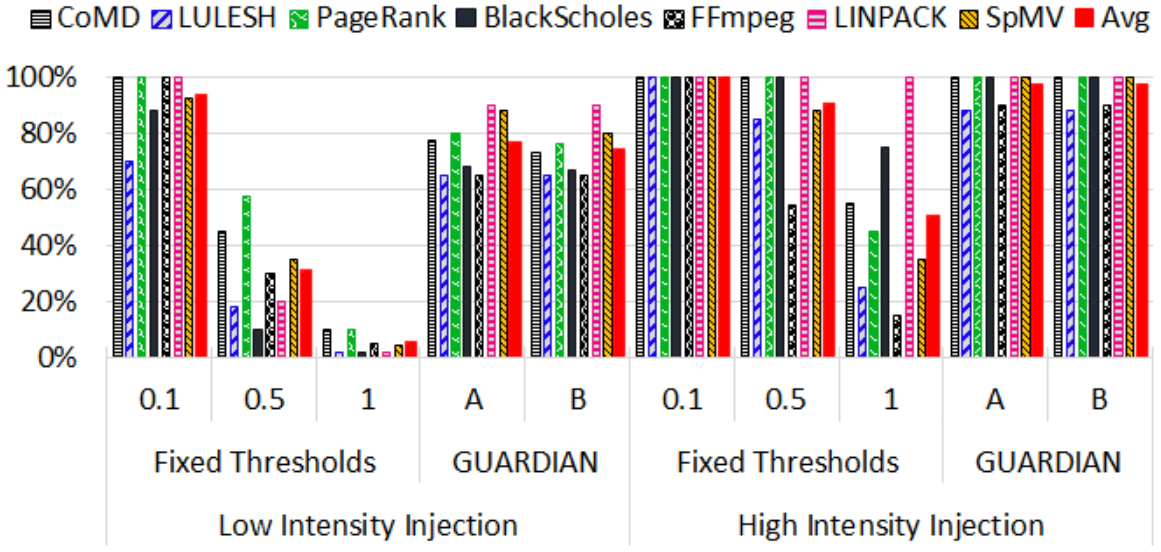


Fig. 3.13.: Detection accuracy for extra memory read bug (Mem): GUARDIAN vs. Fixed threshold-based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Higher is better)

improved approach for kernel learning for multi-dimensional pattern extrapolation was presented in [56].

Performance prediction: Another body of work focuses on performance forecasting [57,58] which addresses the issue of predicting the performance of the application as a whole. They do not try to model the application at the granularity of code regions. More recent advanced variations of these techniques [59,60] also consider the size of application inputs and try to predict how the resource usage would scale for bigger input sizes. These works do not consider how prediction error would grow in the presence of extrapolation. Moreover, for our modeling we use not only the size of inputs but also more interesting features such as matrix sparsity and graph density. An interesting work by Jiang *et al.* [53] shows that some variables are an early indicator of how other variables will vary later on in the program and can therefore be used for early prediction of later program behavior. Specifically, in the context of dynamic program optimization, they argue, correlation between trip counts of loops as one such key indicator.

Anomaly detection and diagnosis: Performance problems have always been a source of frustration for system administrators and software engineers. There is much research diagnosing such problems [1, 52, 61–63], some of which specifically looks at correlation between performance metrics. These tools do not track program properties such as values of input and internal variables and cannot operate in a production environment far away from the training environment. Probably two works which are closest two our approach are Daikon [64] and DIDUCE [65]. Both focus on automatic invariant detection. DIDUCE is more advanced as it can also have an anomaly detection engine based on those inferred invariants. Both Daikon and DIDUCE involve some training runs of the correct application to identify invariants, an approach similar to ours. Since they do not consider input-aware model creation, strict invariants identified by these tools can raise many false alarms in production when they encounter completely new set of values. We consider inaccuracy in predictor models and use input data to quantify such errors and dynamically calibrate our thresholds

for detection. Thus GUARDIAN is more resilient to prediction errors with unseen datasets.

3.8 Conclusion

Techniques to predict the performance and functional behavior of applications are important in the design of many tools. When these tools take into account application input and configuration parameters they are able to achieve high levels of predictive accuracy but face the challenge that they can only be trained on a small fraction of the overall space of inputs and configurations. Since this means that in most production runs the models will need to make predictions about configurations that are well outside the space on which they were trained, it is necessary for such models to quantify their prediction errors across the entirety of this space. In this section, we present a systematic approach to quantify such prediction errors and demonstrate the utility of our approach via a practical use case of an anomaly detector. This detector calibrates its alarm thresholds based on the error estimates provided by our technique. Our experimental evaluations confirm that this tool achieves a low false positive rate and while maintaining a high detection accuracy compared to fixed threshold based anomaly detectors that do not use our technique.

4. IMPROVING REPAIR PERFORMANCE IN ERASURE-CODED DISTRIBUTED STORAGE

4.1 Introduction

Tremendous amount of data has been created in the past few years. Some studies show that 90% of world's data was created in the last two years [66]. Not only are we generating huge amounts of data, but the pace at which the data is being created is also increasing rapidly. Along with this increase, there is also the user expectation of high availability of the data, in the face of occurrence of failures of disks or disk blocks. Replication is a commonly used technique to provide reliability of the stored data. However, replication makes data storage even more expensive because it increases the cost of raw storage by a factor equal to the replication count. For example, many practical storage systems (e.g., HDFS [67], Ceph [68], Swift [69], etc.) maintain three copies of the data, which increases the raw storage cost by a factor of three.

In recent years, erasure codes (EC) have gained favor and increasing adoption as an alternative to data replication because they incur significantly less storage overhead, while maintaining equal (or better) reliability. In a (k, m) Reed-Solomon (RS) code, the most widely used EC scheme, a given set of k data blocks, called *chunks*, are encoded into $(k + m)$ chunks. The total set of chunks comprises a *stripe*. The coding is done such that any k out of $(k + m)$ chunks are sufficient to recreate the original data. For example, in RS (4, 2) code, 4MB of user data is divided into four 1MB blocks. Then, two additional 1MB parity blocks are created to provide redundancy. In case of a triple replicated system, all four 1MB blocks are replicated three times. Thus, an RS (4, 2) coded system requires $1.5x$ bytes of raw storage to store x bytes of data and it can tolerate up to two data block failures. On the other hand, a triple

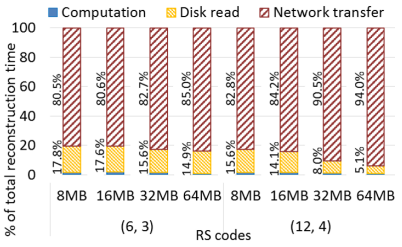


Fig. 4.1.: Percentage of time taken by different phases during a degraded read using traditional RS reconstruction technique.

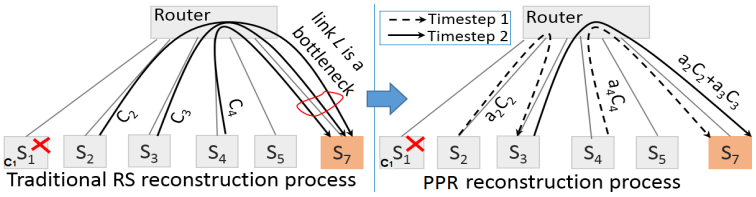


Fig. 4.2.: Comparison of data transfer pattern between traditional and PPR reconstruction for RS (3, 2) code. C_2, C_3, etc. are the chunks hosted by the servers. When Server S_1 fails, Server S_7 becomes the repair destination. Network link L to S_7 is congested during traditional repair.

replication system needs $3x$ bytes of raw storage and can tolerate the same number of simultaneous failures.

Although attractive in terms of reliability and storage overhead, a major drawback of erasure codes is the expensive repair or reconstruction process — when an encoded chunk (say c bytes) is lost because of a disk or server[1] failure, in a (k, m) code system, $k \times c$ bytes of data need to be retrieved from k servers to recover the lost data. In the triple replicated system, on the other hand, since each chunk of c bytes is replicated three times, the loss of a chunk can be recovered by copying only c bytes of data from any one of the remaining replicas. This k-factor increase in network traffic causes reconstruction to be very slow, which is a critical concern for any production data center of reasonable size, where disk, server or network failures happen quite regularly, thereby necessitating frequent data reconstructions. In addition, long reconstruction time degrades performance for normal read operations that attempts to read the erasured[2] data. During the long reconstruction time window, the probability of further data loss increases, thereby increasing the susceptibility to a permanent data loss.

It should be noted that, while it is important to reduce repair traffic, practical storage systems also need to maintain a given level of data reliability and storage overhead. Using erasure codes that incur low repair traffic at the expense of increased

[1]We use the term "server" to refer to the machine that stores the replicated or erasure-encoded data or parity chunks.
[2]An erasure refers to loss, corruption, unavailability of data or parity chunks.

storage overhead and inferior data reliability is therefore a non-starter. However, reducing repair traffic without negatively impacting storage overhead and data reliability is a challenging task. It has been shown theoretically that there exists a fundamental tradeoff among data reliability, storage overhead, volume of repair traffic, and repair degree. Dimakis *et al.* [70] provide a mathematical formulation of an optimal tradeoff curve that answers the following question—for a given level of data reliability (i.e. a given (k, m) erasure coding scheme), what is the minimum repair traffic that is feasible while maintaining a given level of storage overhead? At one end of this optimal curve lies a family of erasure codes called *Minimum Storage Codes* that require minimum storage overhead, but incur high repair bandwidth. At another end of the spectrum lies a set of erasure codes called *Minimum Bandwidth Codes* that require optimal repair traffic, but incur high storage overhead and repair degree. Existing works fall at different points of this optimal tradeoff curve. For example, RS codes, popular in many practical storage systems [71, 72], require minimum storage space, but create large repair traffic. Locally repairable codes [73–75] require less repair traffic, but add extra parity chunks, thereby increasing the storage overhead.

In this section, we design a practical EC repair technique called PPR, which reduces repair time without negatively affecting data reliability, storage overhead, and repair degree. Note that our technique reduces repair time, but not the total repair traffic aggregated over the links. Further, our approach is complementary to existing repair-friendly codes since PPR can be trivially overlaid on top of *any* existing EC scheme.

Key insight A key reason why reconstruction is slow in EC systems is the poor utilization of network resources during reconstruction. A reconstruction of the failed chunk requires a *repair server* to fetch k chunks (belonging to the same stripe as the failed chunk) from k different servers. This causes the network link into the repair server to become congested, increasing the network transfer time. The measurements in our clusters show that network transfer time constitutes up to 94% of the entire reconstruction time, as illustrated in Fig. 4.1. Other researchers have also reported

Table 4.1.: Advantages of PPR: Potential improvements in network transfer time and maximum bandwidth requirement per server

Code params	Users	Possible reduction in network transfer	Possible reduction in maximum BW usage/server
(6,3)	QFS [76], Google ColossusFS [77]	50%	50%
(8,3)	Yahoo Object Store [78]	50%	62.5%
(10,4)	Facebook HDFS [79]	60%	60%
(12,4)	Microsoft Azure [73]	66.6%	66.6%

similar results [72,75,80,81]. Fig. 4.2 shows an example of a reconstruction of a failed chunk in a (3, 2) EC system. The network link into the repair server (server S_7) is three times more congested than network links to other servers. PPR attempts to solve this problem by redistributing the reconstruction traffic more uniformly across the existing network links, thereby improving the utilization of network resources and decreasing reconstruction time. In order to redistribute the reconstruction traffic, PPR takes a novel approach for performing reconstruction — instead of centralizing reconstruction in a single reconstruction server, PPR divides reconstruction into several *partial parallel repair* operations that are performed simultaneously at multiple servers, as shown in Fig. 4.2. Then these results from partial computation are collected using a tree-like overlay network. By splitting the repair operation among multiple servers, PPR removes the congestion in the network link of the repair server and redistributes the reconstruction traffic more evenly across the existing network links. Theoretically, PPR can complete the network transfer for a single chunk reconstruction in $O(log_2(k))$ time, compared to $O(k)$ time needed for a (k, m) RS code. Table 4.1 shows expected reduction in network transfer time during reconstruction for typical erasure coding parameters used in practical systems. Although PPR does not reduce the total amount of data transferred during reconstruction, it reduces reconstruction time significantly by distributing data transfers more evenly across the network links.

One of the benefits of PPR is that it can be overlaid on top of almost all published erasure coding schemes. The list includes, but is not limited to, the most widely used RS code, LRC code (Locally Repairable Code or Local Reconstruction Code [73–75]), PM-MSR code [72], RS-Hitchhiker code [82], Rotated RS [83] code. This is because the distribution of PPR is orthogonal to the coding and placement techniques that distinguish these prior works.

In considering the effect of any scheme on reconstruction of missing chunks in an EC system, we need to consider two different kinds of reconstruction. The first is called *regular repair* or *proactive repair*, in which a monitoring daemon proactively detects that a chunk is missing or erroneous and triggers reconstruction. The second is called *degraded read*, in which a client tries to read a lost data chunk that has not been repaired yet and then has to perform reconstruction in the critical path. PPR achieves a significant reduction in times for both these kinds of reconstruction. Degraded reads are an important concern for practical storage systems because degraded read operations happen quite often, more frequently than regular repairs. Transient errors amount to 90% of data center failures [84], because of issues like rolling software updates, OS issues, and non-disk system failures [73,83]. In these cases, actual repairs are not necessary, but degraded reads are inevitable since client requests can happen during the transient failure period. Furthermore, many practical systems delay the repair operation to avoid initiating costly repair of transient errors [80].

PPR introduces a load-balancing approach to further reduce the reconstruction time when multiple concurrent requests are in progress. We call this variant m-PPR. When selecting k servers out of $(k + m)$ available servers for reconstruction, PPR chooses those servers that have already cached the data in memory, thereby avoiding the time-consuming disk IO on such servers. The m-PPR protocol tries to schedule the simultaneous reconstruction of multiple stripes in such a way that the network traffic is evenly distributed among existing servers. We present further details of m-PPR in Sec. 4.6.

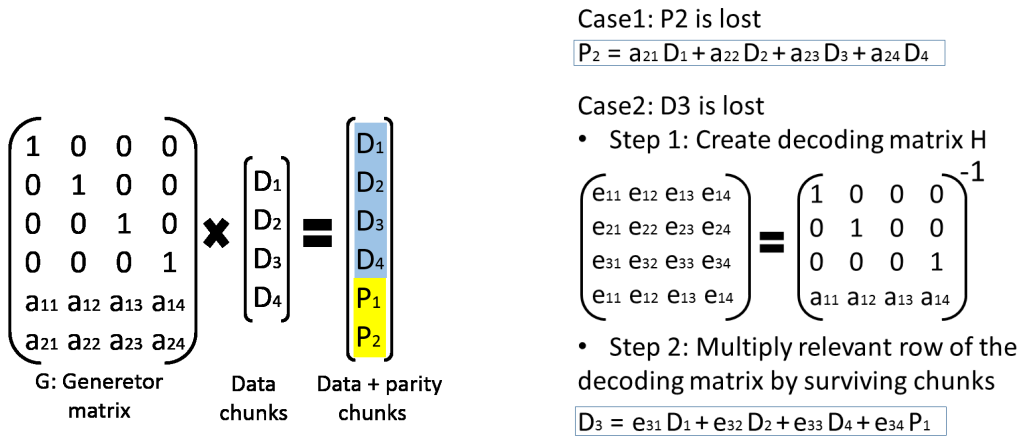
We implemented PPR on top of the Quantcast File System (QFS) [76], which supports RS-based erasure coded storage. For typical erasure coding parameters depicted in Table 4.1, our prototype achieves up to a 59% reduction in repair time out of which 57% is from reduction in network transfer time alone. Such significant reduction in reconstruction time is achieved without degrading data reliability or increasing storage overhead.

This section makes the following contributions:

- We introduce PPR, a novel distributed reconstruction technique that significantly reduces network transfer time and thus reduces overall reconstruction time for erasure coded storage systems by up to 59%.
- We present additional optimization methods to further reduce reconstruction time: a) a caching scheme for reducing IO read time and b) a scheduling scheme targeted for multiple simultaneous reconstruction operations.
- We demonstrate our technique can be easily overlaid on previous sophisticated codes beyond Reed-Solomon, such as LRC and Rotated RS, which were targeted to reduce repair time. PPR provides additional 19% and 35% reduction in reconstruction time, respectively, over and above these codes.

4.2 Primer on Reed-Solomon Coding

Erasure coded storage is attractive mainly because it requires less storage overhead for a given level of reliability. Out of many available erasure coding techniques, ***Reed-Solomon (RS) coding*** [85] is the most widely used. RS code belongs to the class of Maximum Distance Separable (MDS) codes [86], which offers the maximum reliability for a given storage overhead. For a (k, m) RS code, the available data item of size N is divided into k equal ***data chunks*** each of size N/k. Then m additional ***parity chunks*** are calculated from the original k data chunks. The term ***stripe*** refers to this set of $(k + m)$ chunks that is created from the original data. The mathematical



(a) RS (4, 2) encoding (b) RS (4, 2) reconstruction

Fig. 4.3.: Encoding and Reconstruction in Reed-Solomon coding

property, based on which the parity chunks are created, ensures that *any* missing chunk (data or parity) can be reconstructed using any k of the remaining chunks. After the reconstruction process, the server where the reconstructed data is hosted is referred to as the ***repair site***. Thus, the repair site is a server for a regular repair while for degraded read, it is the client component which has issued the read request.

RS Encoding: An RS encoding operation can be represented as a matrix-vector multiplication where the vector of k data chunks is multiplied by a particular matrix G of size $(k+m) \times k$, as illustrated in Fig. 4.3a for a (4, 2) RS code. This matrix G is called the *generator matrix* and is constructed from the *Vandermonde* matrix [87] and the elements a_{ij} etc. are calculated according to Galois Field (GF) arithmetic [85]. In GF arithmetic, addition is equivalent to XOR; thus, adding chunk A with chunk B would involve bit-wise XOR operations. Multiplying chunks by a scalar constant (such as the elements of G) is equivalent to multiplying each GF word component by the constant.

RS Reconstruction: In Fig. 4.3a, when a chunk is lost, it can be reconstructed using some linear algebraic operations with G and a remaining chunk set from the

stripe. For example, in Case-1 in Fig. 4.3b, if a parity chunk (*e.g.*,, P_2) is lost, it can be recalculated by multiplying the corresponding *row* (*i.e.,* the last row in the example) of G by the data chunk vector. On the other hand, if a data chunk (*e.g.*,, D_3) is lost, the reconstruction involves two steps: the first step calculates a *decoding matrix* H, by taking the inverse of a matrix created using any k (*i.e.*,, four in our example) surviving rows of G. We refer to the elements of H as decoding coefficients. The second step multiplies the previously selected k surviving chunks (a combination of data and parity) by the row of the decoding matrix corresponding to the lost chunk (*i.e.,* the 3^{rd} row in the figure). Thus the decoding process is to solve a set of independent linear equations.

4.3 The Achilles' Heel of EC Storage: Reconstruction Time

Both for regular repair and degraded read, the reconstruction path consists of three major steps: multiple servers read the relevant chunks from their own disks (usually done in parallel at each server), each server sends the read chunk to the repair site over the network and finally some computation is performed at the repair site to reconstruct the erasured chunk. For regular repairs, the reconstructed chunk is finally written back to the disk while for degraded reads, the data is directly used by the user request. Thus, the reconstruction time for (k, m) RS coding can be approximated as follows

$$T_{reconst} = \frac{C}{B_I} + \frac{kC}{B_N} + T_{comp}(kC) \tag{4.1}$$

Where C is chunk size, B_I and B_N denote the IO and network bandwidth, respectively. T_{comp} is the computation time, which is a function of a total data size (kC).

As we see from Fig. 4.1, network transfer and IO read are the two most time consuming steps, while the computation time is relatively insignificant. Among these, network transfer time is the most dominant factor because k chunk transfers are required per reconstruction. Often such huge data transfer creates a network bottle-

neck near the repair site. For example, Facebook [75] uses RS(10, 4) code with a data chunk size of 256MB. In this case, for repairing a single chunk, more than 20Gbits need to be funneled into one server. This volume of data has been found to overwhelm network resources in many practical cases leading to extremely long reconstruction time. In spite of recent advances in network technology, with the rapid growth of network heavy applications, the network still remains the most scarce resource in data centers and we anticipate network transfer time will continue to remain a bottleneck for reconstruction operations in EC storage.

Such long reconstruction time would still have been a non-issue if reconstructions were infrequent enough. However, traces of failures from large data centers [75, 79] indicate, that is not the case. Analyzing failures in Facebook data centers, Rashmi *et al.* [79] report on average 50 machine unavailability events (where the machine fails for more than 15 minutes) per day, in a data center with a few thousand machines, each of which has a storage capacity of 24-36TB. To maintain data reliability, these events ultimately lead to reconstruction operations. Moreover, Sathiamoorthy *et al.* [75] report that transient errors with no permanent data loss correspond to 90% of data center failure events. These cases often lead to degraded reads where the reconstruction operation happens in the critical path of the user read request.

Thus, long reconstruction time is the main hindrance toward wide scale adoption of erasure coded storage for distributed storage and network transfer time is expected to remain the primary cause for this for the foreseeable future.

This observation has also been made by many prior researchers [82, 88]. Their solutions have taken two forms. In the first form, several solutions design new coding schemes that reduce reconstruction traffic, but incur a higher storage overhead [73,88]. In the second form, the proposed solutions place erasure encoded data in such a way that the amount of data that needs to be read for the common failure cases is kept small [82, 83].

In this work, we observe that there is a third way of reducing the network bottleneck during recovery in erasure coded storage: determining intelligently *where* the

repair takes place. In all existing repair schemes, the repair operation happens in a centralized location — the repair site — which is either the server where the recovered chunk will be placed, or the client that initiates the read request for the lost data. We propose a distributed repair technique where partial results are computed locally at the server hosting the chunks. Then these results are aggregated to reconstruct the missing chunk. This distributed technique may not appear to be significant because the computational burden of repair in erasure codes is minimal. However, the process of conveying *all* the chunks to a single point in itself creates a bottleneck and load imbalance on some network links. The process of distributing the repair burden among multiple servers has the benefit of removing such a bottleneck and load imbalance. This forms the key innovation in our proposed system PPR. It distributes the task of decoding among multiple servers, in a fashion reminiscent of binomial reduction trees from the High Performance Computing (HPC) world [89].

Because of a mathematical property of the repair operation, this distribution means that the amount of traffic coming out of any aggregator server is exactly half of the sum of the traffics coming in from the two inputs, into the aggregator server. The final destination of the repair traffic, where the complete reconstructed data is finally available, is not overloaded with network traffic in its incoming link. Rather, with PPR, even the incoming link to that destination server gets approximately as much traffic as the first aggregator server. This mathematical property has the desired effect of reducing the network transfer time during repair from erasure coded storage.

4.4 Design: Partial Parallel Repair (PPR)

We present an efficient reconstruction technique that focuses on reducing network transfer time during reconstruction. PPR divides the entire repair operation into a set of partial operations that are then scheduled to execute in parallel on multiple servers. PPR reduces the pressure on the two primary constrained resources, network capacity and disk reads.

We address the *reconstruction latency problem* in two steps; first, using the main PPR algorithm (Sec. 4.4.1), we make *single chunk* reconstruction highly efficient. Second, we speed up simultaneous reconstructions resulting from multiple chunk failures[3] by evenly apportioning the load of these multiple reconstructions. The multiple reconstruction scenario arises most commonly because of a hard drive failure. We discuss this aspect of the solution, which we call multiple-PPR (m-PPR), in Sec. 4.5.

4.4.1 Efficient single chunk reconstruction: Main PPR

As discussed before, to reconstruct an erasured chunk, the EC storage system needs to gather k other chunks and perform the required computation. This step often incurs high latency because of the large volume of data transfer over a particular link, namely, the one leading to the final destination, which becomes the bottleneck.

Based on the repair operation of RS code, we make the following two observations that fundamentally drive the design of PPR:

- The actual reconstruction equation used for computing the missing chunks (either data or parity), as shown in Fig. 4.3b, is linear and the XOR operations (*i.e.,*, the additions) over the terms are *associative*.

- The multiplication by the scalar decoding coefficients or a XOR between two terms do not increase the size of the data. Thus, the size of all the terms that would be XORed, as well as the size of the final reconstructed chunk, is the same as the size of the original chunks that were retrieved from different servers. For instance, let $R = a_1C_1 + a_2C_2$ be the equation for reconstruction. Here a_1, a_2 are the decoding coefficients and R denotes a missing chunk that will be reconstructed from the existing chunks C_1 and C_2. All individual terms in the above equation, *e.g.,*, C_1, C_2, a_1C_1, and a_2C_2, will have the same volume of data which is equal to the chunk size (*e.g.,* 64MB).

[3]Each individual chunk failure is still the only failure in its corresponding stripe. Such single chunk failure in a stripe captures almost 99% of the failure cases (Sec. 4.1).

These two observations lead to the fundamental design principle of PPR: *distribute* the repair operation over a number of servers that only computes a *partial* result locally and in parallel, and then forward the intermediate result to the next designated server en route to the final destination. The servers involved in the distributed operations are the ones that host the surviving chunks of that stripe. This design ensures that the part of the data needed for reconstruction is already available locally.

PPR takes a few *logical timesteps* to complete the reconstruction operation, where in each timestep a set of servers perform some partial repair operations to generate intermediate results in parallel. These partial operations constitute either a scalar multiplication of the local chunk data by the corresponding decoding coefficient [4] (this operation happens only during the first logical timestep) or an aggregate XOR operation between the received intermediate results from the earlier servers. For example, in Fig. 4.2, chunk C_1 is lost because of a failure in server S_1. Server S_7 is chosen as a new host to repair and store C_1. Now C_1 can be reconstructed using the equation: $C_1 = a_2C_2 + a_3C_3 + a_4C_4$, where a_2, a_3, and a_4 are the decoding coefficients corresponding to chunks C_2, C_3, and C_4. In `timestep 1`, S_2 sends its partial result a_2C_2 to S_3. In *parallel*, S_4 sends its partial result a_4C_4 to S_7, while at the same time S_3 also computes its own partial result a_3S_3. In `timestep 2`, S_3 sends its aggregated (*i.e.,* XORed) results to S_7 reducing the overall network transfer time by a factor of 1/3 or 33%. This behavior can be explained as follows. Let the chunk size be C MB and the available network bandwidth be B_N MB/s. In traditional reconstruction, $3C$ MB of data goes through a particular link, resulting in a network transfer time of approximately $3C/B_N$ sec. In PPR, in each timestep, only one chunk is transferred over *any particular* link (since parallel transfers have different source and destination servers). Thus, the network transfer time in each timestep is C/B_N sec, and since there are *two* timesteps involved in this example, the total network transfer time is $2C/B_N$. The number of timesteps required in PPR can be generalized as $\lceil log_2(k+1) \rceil$, as we will elaborate next.

[4] We use the term *decoding coefficient* in a generic way. During reconstruction of a parity chunk for RS codes, an encoding operation may be performed. In that case, such coefficients will be 1.

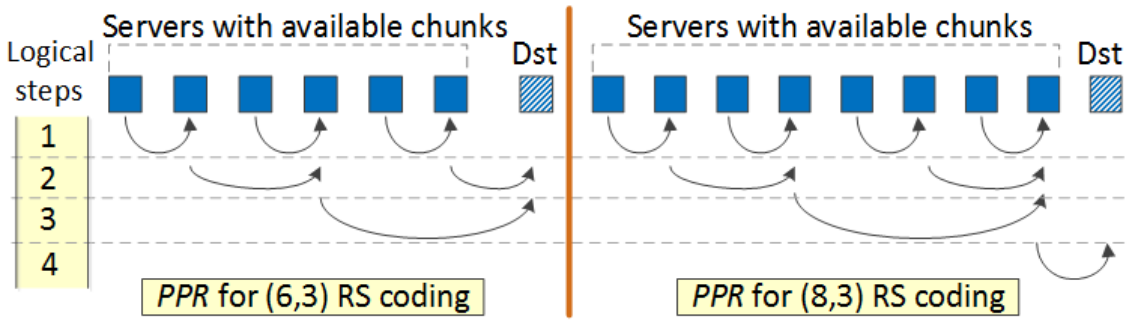


Fig. 4.4.: Data transfer pattern during traditional reconstruction for (6, 3) and (8, 3) RS coding

4.4.2 Reduction in network transfer time

Even though PPR takes a few logical *timesteps* to complete the reconstruction process, in reality, it significantly reduces the total reconstruction time. Essentially, PPR overlays a *tree-like* reduction structure (also referred to as a Binomial Reduction Tree in HPC [89,90]) over the servers that hold the relevant chunks for reconstruction. Fig. 4.4 shows more examples of PPR-based reconstruction techniques for RS codes (6,3) and (8,3) where network transfers are completed in only three and four logical timesteps, respectively. Each timestep takes C/B_N amount of time where, C is the chunk size and B_N is the available bandwidth, which results in a total network transfer time of $3C/B_N$ and $4C/B_N$, respectively. In comparison, traditional RS reconstruction for RS (6,3) and (8,3) would bring six and eight chunks to a particular server with a network transfer time of $6C/B_N$ and $8C/B_N$ respectively. Thus PPR can reduce network transfer time by 50% in both cases. We introduce the following theorem to generalize the observation.

Theorem 4.4.1 *For (k, m) RS coding, network transfer time for* PPR*-based reconstruction is $\lceil(log_2(k+1))\rceil \times C/B_N$ as compared to $k \times C/B_N$ for the original reconstruction technique. Thus* PPR *reduces the network transfer time by a factor of* $\frac{k}{\lceil(log_2(k+1))\rceil}$.

Proof: PPR *reconstruction*: During reconstruction, in total $(k+1)$ servers are involved, out of which k servers host the relevant chunks and the remaining one is the repair site. PPR performs a binary tree-like reduction where $(k+1)$ servers are the leaf nodes of the tree. Completion of each logical timestep in PPR is equivalent to moving one level up towards the root in a binary tree, while reaching the root marks the completion of PPR. Since the height of a binary tree with $(k+1)$ leaves is $log_2(k+1)$, PPR requires exactly $log_2(k+1)$ logical steps to complete when $(k+1)$ is a power of two; the `ceil` function is used if that is not the case. During each step, the network transfer time is C/B_N since the same amount C is being transferred on each link and each link has bandwidth B_N. Thus, the total network transfer time is $\lceil(log_2(k+1))\rceil \times C/B_N$.

Baseline EC reconstruction: A total of k chunks, each of size C, will be simultaneously retrieved from k servers. Thus the ingress link to the repair server becomes the bottleneck. If B_N is the bandwidth of that ingress link, the total network transfer time becomes $k \times C/B_N$.

Thus, PPR reduces the network transfer time by a factor of $\frac{k}{\lceil(log_2(k+1))\rceil}$.

If $k = 2^n - 1$, where $n \in \mathbb{Z}^+$, then the network transfer time is reduced by a factor of $\Omega(\frac{2^n}{n})$. This reduction in network transfer time becomes larger for increasing values of n, *i.e.,*, for larger values of k. Since larger values of k (for a fixed m) can reduce the storage overhead of erasure coded storage even further, coding with high values of k is independently beneficial for storing large amounts of data. However, it has not been adopted in practice mainly because of the *lengthy reconstruction time problem.*

■

Moreover, as an additional benefit, the *maximum* data transfer over *any* link during reconstruction is also reduced by a factor of approximately $\lceil(log_2(k+1))\rceil/k$. In PPR, the cumulative data transfer across all logical timesteps and including both ingress and egress links is $C \times \lceil log_2(k+1)\rceil$. This behavior can be observed in Fig. 4.4 which illustrates PPR-based reconstruction technique for RS codes (6,3) and (8,3).

Such a reduction facilitates a more uniform utilization of the links in the data center, which is a desirable property.

Network architecture: We assume the network to have either a *fat-tree* [91] like topology, where each level gets approximately full bisection bandwidth with similar network capacity between any two servers in the data center, or a VL2-like [92] architecture, which gives the illusion of all servers connected to a monolithic giant virtual switch. These architectures are the most popular choices in practice [93]. If servers have non-homogeneous network capacity, PPR can be extended to use servers with higher network capacity as aggregators, since these servers often handle multiple flows during reconstruction, as depicted in Fig. 4.4.

When is PPR **most useful?** The benefits of PPR become prominent when network transfer is the bottleneck. Moreover, the effectiveness of PPR increases with higher values of k as discussed before. Interestingly, we found that PPR also becomes more attractive for larger chunk sizes. For a given k, larger chunks tend to create higher contention in the network. Nevertheless, for other circumstances, PPR should be at least as good as traditional reconstruction since it introduces negligible overhead.

PPR **vs staggered data transfer:** Since the reconstruction process causes network congestion at the server acting as the repair site, a straightforward approach to avoid congestion could be to stagger data transfer, with the repair server issuing

Table 4.2.: Faster reconstruction: Less computation per server because of parallelism in PPR technique

PPR reconstruction computation	Traditional RS reconstruction computation
Creation of the decoding matrix +	Creation of the decoding matrix +
One Galois-field multiplication with coefficients (since parallel at multiple servers) +	k Galois-field multiplications with coefficients +
$ceil(log_2(k+1))$ number of XOR operations (done by aggregating servers)	k number of XOR operations

requests for chunks one-by-one from other servers. However, staggering data transfer adds unnecessary serialization to the reconstruction process and increases the network transfer time. The main problem with this approach is that it avoids congestion in the network link to the repair server by under-utilizing the available bandwidth of network links. Thus, although simple and easy to implement, staggered data transfer may not be suitable for scenarios where reconstructions need to be fast, e.g., in case of degraded reads. PPR decreases network congestion and simultaneously increases parallelism in the repair operation.

Compatibility with other ECs: Since the majority of the practical erasure codes are linear and associative, PPR-based reconstruction can be readily applied on top of them. PPR can also be easily extended to handle even non-linear codes, as long as the overall reconstruction equation can be decomposed into a few independent and partial associative operations.

4.4.3 Computation speed-up and reduced memory footprint

PPR provides two additional benefits.

Parallel computations: PPR distributes the reconstruction job among multiple servers that perform partial reconstruction functions in parallel. For example, scalar multiplication with decoding coefficients[5] and some aggregating XOR operations are done in parallel, as opposed to traditional serial computation at the repair site. For RS(k, m) code Table 4.2 highlights the difference between PPR and traditional RS reconstruction, in terms of the computation on the critical path.

Reduced memory footprint: In traditional RS reconstruction, the repair site collects all the k necessary chunks and performs the repair operation on those chunks. Since the processor actively performs multiplication or bitwise XOR operations on

[5]In Cauchy-Reed Solomon coding, multiplications are replaced by XOR operations [94]

these k chunks residing in memory, the memory footprint of such reconstruction operation is on the order of kC, where C is the chunk size. In PPR, the maximum bound on memory footprint in any of the involved servers is $C \times \lceil log_2(k+1) \rceil$, because a server deals with only $\lceil log_2(k+1) \rceil$ chunks at most.

4.4.4 Reducing disk IO with in-memory chunk caching

To reduce the reconstruction time as much as possible, in addition to optimizing network transfer, we also try to reduce disk IO time. Although read operations from multiple servers can be done in parallel, disk read still contributes a non-trivial amount of time to reconstruction, up to 17.8% in the experiment, as shown in Fig. 4.1. We design an in-memory least recently used (LRU) cache that keeps most frequently used chunks in each server. As a result, the chunk required for reconstruction can be obtained from memory, without incurring the cost of reading it from disk. In addition, PPR maintains a usage profile for chunks that are present in the cache using the associated timestamp. The usage profile can influence the decision regarding which chunk failures should be handled urgently. A chunk that is frequently used, and hence in the cache, should be repaired urgently. Even though caching helps reducing the total reconstruction time, the technique itself is orthogonal to the main PPR technique. Caching can also be used with traditional repair techniques to reduce IO time.

4.5 Multiple Concurrent Repairs: m-PPR

In any reasonably sized data center, there can be multiple chunk failures at any given time because of either scattered transient failures, machine maintenance, software upgrade, or hard disk failures. Although proactive repairs for such failures are often delayed (*e.g.*,, by 15 minutes by Google [84]) in anticipation that the failure was transient, multiple simultaneous reconstructions can still happen at any point in time. A naive attempt to perform multiple overlapping reconstructions may put

pressure on shared resources, such as network and disk IO, leading to poor reconstruction performance. We design m-PPR, an algorithm that schedules multiple reconstruction-jobs in parallel while trying to minimize the competition for shared resources between multiple reconstruction operations. At the core, each repair job uses the PPR-based reconstruction technique described earlier. Scheduling of multiple reconstructions through m-PPR is handled by a *Repair-Manager* (RM), which runs within a centralized entity (*e.g.,* the *Meta-Server* in our *Quantcast File System* based implementation).

Algorithm 5 m-PPR: Scheduling algorithm for multiple reconstructions

```
1: for all missingChunk ∈ missingChunkList do
2:     hosts ← GETAVAILABLEHOSTS(missingChunk);
3:     reconstSrc ← SELECTSOURCES(hosts); //Choose best sources
4:     reconstDst ← SELECTDESTINATION(hosts, allServers); //Choose the best destination
5:     // Schedule a PPR-based single reconstruction
6:     SCHEDULERECONSTRUCTION(reconstSrc, reconstDst);
7:     // Update state to capture the impact of scheduled reconstruction
8:     UPDATESERVERWEIGHTS( );
9: end for
10: // Choose k out of k + m − 1 available sources
11: procedure SELECTSOURCES(hosts)
12:     sortedHosts ← SORTSOURCES(hosts);
13:     selectedSources ← [];
14:     while selectedSources.size ≤ k do
15:         anotherSourceServer ← sortedHosts.pop();
16:         selectedSources.add(anotherSourceServer);
17:     end while
18:     return selectedSources;
19: end procedure
20: //Find a destination server as repair site
21: procedure SELECTDESTINATION(hosts, allServers)
22:     ifdegraded read return Client; //Degraded read:client is destination
23:     // For reliability, exclude existing hosts
24:     possibleDsts ← FINDPOSSIBLEDESTINATIONS(hosts,allServers);
25:     sortedDsts ← SORTDESTINATIONS(possibleDsts);
26:     chosenDst ← sortedDsts.pop(); //Choose the best repair site
27:     return chosenDst
28: end procedure
```

The RM keeps track of various information for all the servers, such as whether a chunk is available in its in-memory cache, the number of ongoing repair operations scheduled on the server, and the load that users impose on the servers. Based on these information, the RM uses *greedy heuristics* to choose the best source and destination servers for each reconstruction job. For the source servers, m-PPR selects the k best servers out of the remaining $k + m - 1$ servers. For the destination server, it

chooses one out of the available $N - (k + m)$ servers, where N is the total number of available servers. In practice, the number of possible destination servers is further constrained by various factors. For example, some applications might require the chunks corresponding to one data stripe to be in close network proximity. Others might want affinity of some data to specific storage types, such as SSD. Some applications might want to avoid servers with identical failure and upgrade domains [71]. The RM calculates, for each potential server, a *source weight* and a *destination weight* as follows:

$$\begin{aligned} w_{src} &= a_1(hasCache) - a_2(\#reconstructions) - \\ &\quad a_3(userLoad) \qquad (4.2) \\ w_{dst} &= -[b_1(\#repairDsts) + b_2(userLoad)] \qquad (4.3) \end{aligned}$$

Here a_is, b_is in Eq.(4.2) and Eq.(4.3) are the coefficients denoting the importance of various parameters in the source and destination weight equations. The `hasCache` is a binary variable denoting whether the relevant chunk is already present in the in-memory cache of that particular server. The number of reconstructions (`#reconstructions`) in Eq.(4.2) represents how many reconstruction jobs are currently being handled by the server. `userLoad` measures the network load handled by that server as part of regular user requests. The value of `#reconstructions` gives an indication of the maximum possible network bandwidth utilization by reconstruction operation at that server. In Eq.(4.3), the number of repair destinations (`#repairDsts`) represents how many repair jobs are using this server as their final destination. Intuitively, these source and destination weights represent the goodness of a server as a source or destination candidate for scheduling the next repair job.

Choosing the coefficients: We calculate the ratio of a_1 and a_2 as $\alpha(ceil(log_2(k + 1)))/\beta$. Here α represents the percentage reduction in the total reconstruction time, if a chunk is found in the in-memory cache of a source server. β denotes the ratio of net-

work transfer time to the total reconstruction time in PPR. Intuitively, we compare how many simultaneous reconstructions would be onerous enough to offset benefit of having a chunk in the cache. We calculate the ratio a_2 and a_3 as $C \times \lceil log_2(k) \rceil$. Essentially, from `userLoad` we calculate an equivalent number of PPR-based reconstruction operations that would generate similar traffic. The ratio of b_1 and b_2 is identical to this. For simplicity, we set a_2 and b_1 to one and calculate other coefficients. For example, for RS(6, 3) code, 64MB chunk size, and cluster with 1Gbps network we calculate values of a_1, a_2, and a_3 to be `0.36`, `1.0`, and `0.005` when `userLoad` is measured in MB.

Scheduling: The RM maintains a queue with missing chunk identifiers. To schedule reconstruction of multiple chunks in a batch using m-PPR algorithm, it pops up items one by one from the head of the queue and greedily schedules reconstruction jobs for each of those missing chunks. The RM uses Eq.(4.2) to calculate goodness score for servers holding relevant chunks of the missing data item and iteratively selects the best k source servers to schedule a PPR job. If fewer than k source servers are available, the RM skips that reconstruction and puts it back at the end of the queue for re-trial. The RM also needs to find a suitable destination server to schedule a repair job. However, not all available servers in the data center are good candidates for the destination server because of reliability reasons. The servers already holding the corresponding data or parity chunks from the same stripe and the ones in the same *failure domain*[6] or *upgrade domain*[7] should be avoided for reliability reasons. For the remaining destination candidates, the RM calculates a weight to capture the current load on that server using Eq.(4.3). Finally, the most lightly loaded server is selected as the final destination for that repair job. After scheduling a job, all the weights are updated to reconsider the impact on the shared resources. This entire process is illustrated in Algo. 5.

[6]Servers that can fail together *e.g.*,, within a rack.
[7]Servers that are likely to be down at the same time because of the software or hardware upgrades.

The overall complexity of m-PPR for scheduling a reconstruction is $O\{N \log(N)\}$. Again, N is the number of possible destination servers and also $N \gg k, m$.

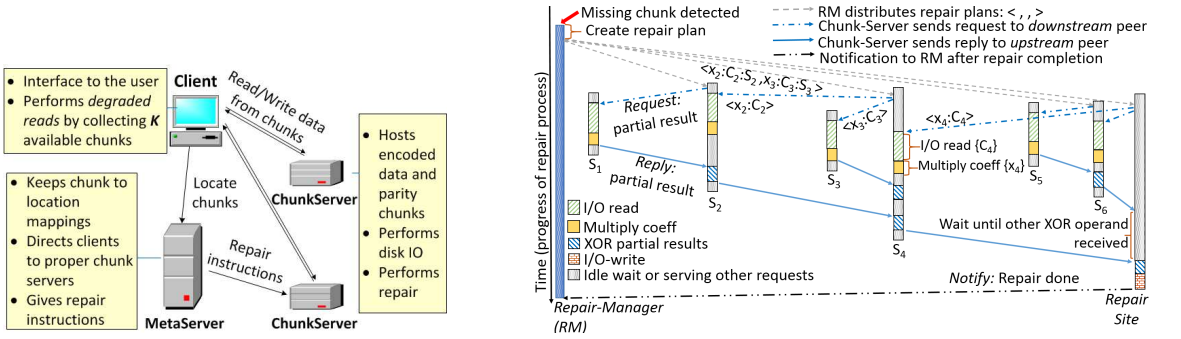
Staleness of information: Some of the parameters used in Eq.(4.2) and Eq.(4.3), such as `hasCache` and `userLoad` can be slightly stale as RM collects these metrics through *heartbeats* from the servers. Such staleness is limited by the frequency of heartbeats (5 sec in our setup). Thus, such minor staleness does not affect the usability of m-PPR. Moreover, RM monitors all the scheduled reconstructions and, if a job does not finish within a threshold time, RM reschedules it for choosing a new set of servers. Essentially, m-PPR is a greedy algorithm because for each reconstruction it chooses the best server combination possible at that point.

Beyond a centrally managed server: The scheduling load in m-PPR can be easily distributed over multiple RMs. Each one of these RMs would be responsible for coordinating repairs of a subset of chunk failures. In a more distributed architecture, one of the source servers can also take the responsibility of choosing a new destination server and distribute a repair plan to coordinate the repair with other peers.

4.6 Design and Implementation

4.6.1 Background: QFS architecture

Quantcast File System (QFS) is a popular high-performance distributed file system that provides stable RS-based erasure coding for lower storage overhead and higher reliability [76]. QFS evolved from the Kosmos File System originally developed at Microsoft. The QFS architecture has three major components, as illustrated in Fig. 4.5a. A centralized ***Meta-Server*** manages the file system's directory structure and how RS chunks are mapped to physical storage locations. A ***Chunk Server*** runs on each machine where the data is hosted and manages disk IO. A ***Client*** refers to an entity that interfaces with the user requests. During read (or write) operation,



(a) Three major components in the QFS architecture

(b) Message exchanges among Chunk Servers and the RM for RS(6, 3) reconstruction using PPR

Fig. 4.5.: (a) QFS architecture and (b) PPR protocol timeline

a Client communicates with the Meta-Server to identify which Chunk Server holds (or will hold, in the case of write) the data, then directly interacts with a Chunk Server to transfer the data. Chunk Servers periodically send *heartbeat* messages to the Meta-Server and the Meta-Server periodically checks the availability of the chunks (*monitoring*). If the Meta-Server detects a disk or server failure (through heartbeat), or a corrupted or a missing chunk (through monitoring), it starts the repair process, first designating one Chunk Server as a repair site and then performing the traditional repair process. In case of degraded read, where the client identifies a missing chunk while trying to read, the reconstruction happens in the critical path initiated by the client, which again first gathers k chunks before executing a decoding operation.

4.6.2 PPR protocol

In this section, we elaborate on the relevant implementation details to enable PPR in QFS.

Reconstruction during regular repairs: For a regular repair, the Meta-Server invokes a ***Repair-Manager (***RM***)***. The RM selects k out of the remaining $k+m-1$ chunks for the reconstruction of the missing chunk. This selection is done by the m-

PPR algorithm (Algo. 5). The RM first analyzes which chunks are available for repair and computes the decoding matrix accordingly. From the decoding matrix, the RM calculates decoding coefficients corresponding to each participating chunk. The RM distributes these coefficients along with a ***repair plan*** to only $k/2$ Chunk Servers (*e.g.*,, S_2, S_4, S_6 in Fig. 4.5b) and also to the repair site.

In Fig. 4.5b, a Chunk Server S_4 receives a plan command $< x_2{:}C_2{:}S_2,\ x_3{:}C_3{:}S_3 >$ from the RM, where x_i's are the decoding coefficients, C_i's are the chunk identifiers (chunkId), and S_i's are the corresponding Chunk Servers. This plan indicates S_4 would aggregate partial results from downstream peers S_2 and S_3. Therefore, S_4 sends requests $< x_2{:}C_2 >$ and $< x_3{:}C_3 >$ to these servers indicating S_2 and S_3 would return results after reading their local chunks C_2 and C_3. Before returning the results, servers S_2 and S_3 also multiply chunks C_2 and C_3 by their corresponding coefficients x_2 and x_3, respectively. As part of the *same* repair plan, S_4 also receives a request $< x_4{:}C_4 >$ from its upstream peer (in this case the repair site). Thus S_4 schedules a local disk read for chunk C_4, which is then multiplied by the coefficient x_4. S_4 waits for results from S_2 and S_3 and performs incremental XORs before replying to its upstream peer with an aggregated result.

The repair site aggregates the results by XORing all the results coming from the downstream Chunk Servers to reconstruct the missing chunk and writes back to the disk at the end of the operation. Finally, this destination Chunk Server sends a message to the RM indicating a successful completion of the repair operation.

Reconstruction during degraded reads: If a degraded read operation triggers the PPR-based reconstruction, a Client acts as the repair site and informs the RM about a missing chunk. Then the RM distributes a repair plan with the *highest priority*.

Tail completion: The number of flows, as well as the number of nodes involved in PPR, is exactly the same as in traditional repair. It is equal to k. Since k is small in practice (between 6 and 12), the probability of encountering a relatively slow

node is small in both traditional repair and PPR. Nevertheless, the RM uses node usage statistics (CPU and I/O counters collected with the Heartbeat messages) to de-prioritize the slow nodes before creating the repair plan. If reconstruction does not complete within a certain time threshold (because of unpredictable congestion or failures), the RM reschedules the reconstruction process with a new repair plan.

4.6.3 IO pipelining, caching, and efficient use of memory

Overlapping disk IO with network transfer: Disk IO (read/write) time is another dominant component in the overall reconstruction time. Aggregation Chunk Servers that had posted downstream requests (*e.g.*,, S_2, S_4, S_6), read *different* chunks from disk and wait[8] for data transfer from their downstream peer Chunk Servers to complete. Then they apply the aggregating XOR operation and send the result to further upstream servers in the tree. To increase parallelism, aggregation Chunk Servers schedule IO-reads in parallel with data transfer from network.

Caching: We attempt to further reduce the impact of IO-read time by introducing an *in-memory* caching mechanism in Chunk Servers, as described in Section 4.4.4. When choosing k out of the remaining $k+m-1$ Chunk Servers for a reconstruction operation in m-PPR protocol, RM gives higher priority to hot chunks but tries to avoid *very-hot* chunks in order to minimize the impact on application performance. However, for multiple simultaneous reconstructions, we found that making sure that these reconstructions use disjoint servers has a greater benefit than cache-aware server assignment, since in general data centers are constrained by network resources.

4.6.4 Implementation details

The choice of a codebase: We implemented our technique with QFS [95] written in C++. Among several alternatives, we chose QFS because of its simpler architec-

[8]Because network transfer of a chunk usually takes longer than IO time.

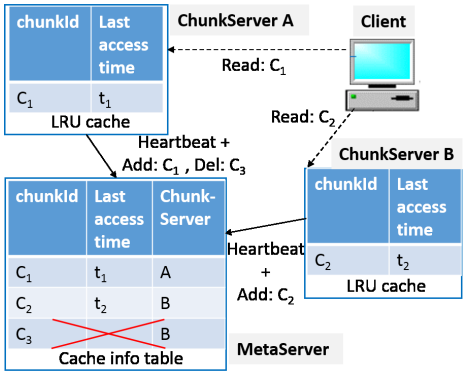


Fig. 4.6.: Protocol for LRU cache. Updates are piggybacked with heartbeat messages

ture and reasonable popularity in the community. However, our PPR technique is general enough to be applicable to other widely used erasure coded storage systems. Specifically the architecture of HDFS with erasure coding [96] is almost identical to that of QFS, and therefore PPR is directly applicable. In addition, our technique can also be applied to Ceph [68], another popular distributed storage system that supports erasure coding. In Ceph, clients use a pseudo-random mapping function called CRUSH [97] to place and access data chunks, rather than relying on a centralized meta server. Nonetheless, it does have a centralized entity, called *ceph monitor* (`ceph-mon`) that knows the layout of Object Storage Devices (OSDs) (equivalent to Chunk Servers in QFS). `ceph-mon` is responsible for checking the health of each OSD, letting the newly joined OSDs know the topology, etc. Thus, we can augment such an entity with RM to enable PPR. Moreover, we can also augment *any* OSD with RM function, since all OSDs know where a given chunk is (or will be) located based on the pseudo-random mapping function.

Changes made to the codebase: To implement PPR, we have made the following changes to the QFS codebase. First, we extended the QFS code to make the chunk size configurable; QFS uses a fixed chunk size of 64MB. Second, we implemented PPR decoding operations using Jerasure and GF-Complete [98] libraries, which were not the defaults in QFS. Jerasure allows a configurable set of coding parameters, while the

default in QFS only supports the (6, 3) code. Third, we augmented the Meta-Server with the RM to calculate decoding coefficients, create a repair plan, and distribute it to Chunk Servers. The RM also keeps track of the cached chunks at Chunk Servers. Fourth, the Chunk Server's state machine was modified to incorporate the PPR protocol to communicate with the peers and the RM, and search for a chunk in its memory cache before attempting to perform disk IO. Lastly, it is worthwhile to note that our implementation of PPR-based reconstruction is fully transparent to the end user.

4.7 Evaluation

In this section we evaluate our implementation of PPR on top of QFS and compare the repair performance with QFS's traditional Reed-Solomon-based reconstruction technique. Our primary metric is the reduction in repair time. We also layer PPR on top of two other popular and practical erasure codes, namely LRC [73] and Rotated RS [83], and evaluate the effectiveness of PPR when used with these codes.

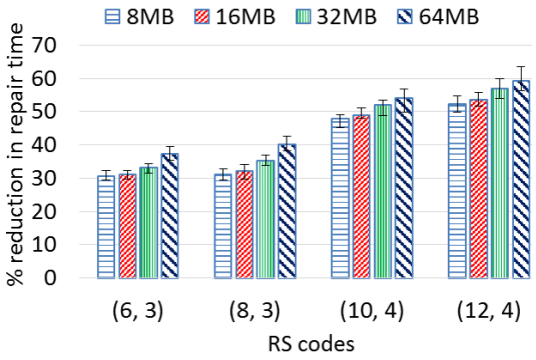
Experimental setup: We use two OpenStack [99] clusters— a 16 host lab cluster (SmallSite) and an 85 host production cluster (BigSite), to demonstrate the scalability advantages of PPR. In SmallSite, each machine belongs to one rack and has 16 physical CPU cores with 24GB RAM. Each core operates at 2.67GHz. They are connected to a 1Gbps network. Each VM instance runs Ubuntu 14.04.3 with four vcpus, 8GB memory, and 80GB of storage space. In BigSite, the machines have dual 10-core 2.8GHz CPUs and are connected by two bonded 10G NICs, with each NIC going to an independent ToR (Top-of-Rack) switch. However, an `iperf` test showed an average bandwidth of about 1.4Gbps between any two VMs (such lower than expected bandwidth is due to the well-know VxLAN issues, which we do not discuss here for brevity). For both proactive repair and degraded read experiments on the SmallSite, we kill a single Chunk Server, which affects a small number of chunks. For each experiment, we report the mean values computed from 20 runs. We

measure repair time on the RM as the difference between the time when it starts a repair process and the time when it is notified by a completion message sent from the repair site. For degraded reads, we measure the latency as the time elapsed from the time instant when a client posts a read request to the time instant when it finishes reconstructing the lost chunk(s).

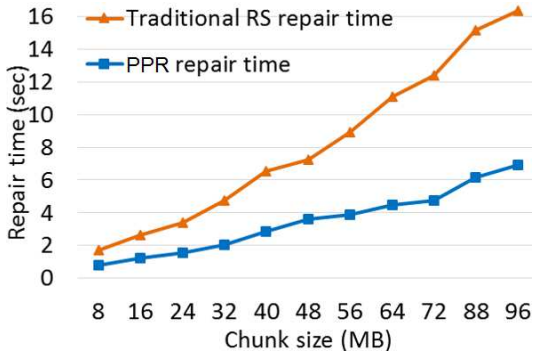
4.7.1 Performance improvement with main PPR

Improving regular repair performance

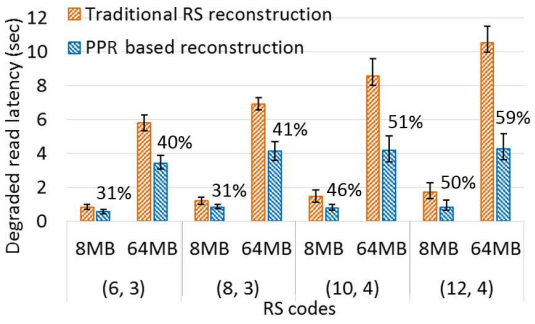
Fig. 4.7a illustrates the percentage reduction in the repair time achieved by PPR compared to the baseline traditional RS repair technique, for four different codes: (6, 3), (8, 3), (10, 4), and (12, 4), each with chunk sizes of 8MB, 16MB, 32MB, and 64MB. PPR reduces the repair time quite significantly. For a higher value of k the reduction is even higher and reaches up to 59%. This is mainly because in PPR the network transfer time increases with $log(k)$, as opposed to increasing linearly in k as in the traditional RS repair (Sec. 4.4.2). Another interesting observation is that PPR becomes more attractive for higher chunk sizes. To investigate this further, we performed an experiment by varying the chunk size from 8MB to 96MB for the (12, 4) RS code. Fig. 4.7b illustrates that the benefit of PPR is higher at higher chunk sizes, *e.g.*,, 53% at 8MB while 57% at 96MB. This is because as the chunk size grows, it increases the network pressure on the link connected to the repair site, leading to a higher delay. PPR can alleviate such a situation through its partial and parallel reconstruction mechanism. It should be noted that many practical storage systems use big chunks so that relevant objects (e.g., profile photos in a social networking applications) can be contained within a single chunk, thereby avoiding the need to fetch multiple chunks during user interaction.



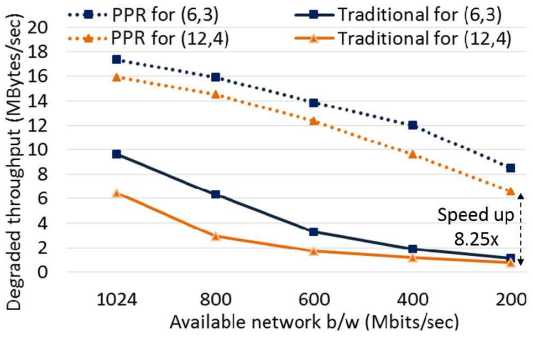
(a) Percentage reduction in repair time with PPR over baseline Reed-Solomon code for different chunk sizes and coding parameters



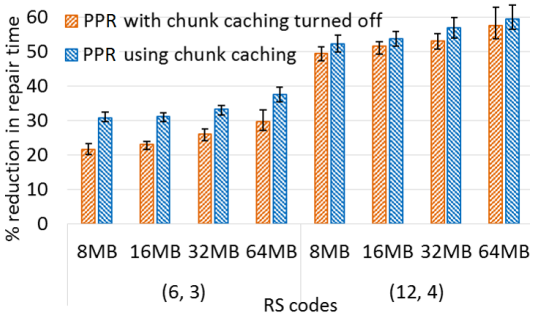
(b) Traditional repair vs. PPR using RS (12, 4) code. PPR's benefit becomes more obvious as we increase the chunk size



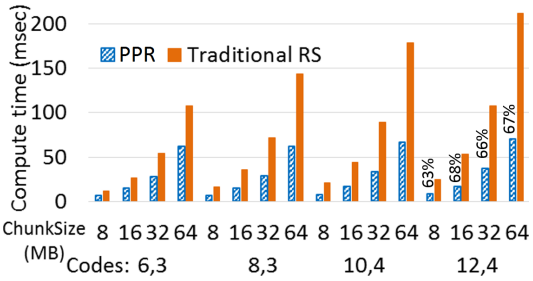
(c) Improvement in degraded read latency



(d) Degraded read throughput under constrained bandwidth



(e) Percentage reduction: PPR without chunk caching vs. PPR with chunk caching. The baseline is standard RS code.



(f) Improved computation time during reconstruction

Fig. 4.7.: Performance evaluation on SmallSite with a small number of chunk failures

Improving degraded read latency

Recall that a degraded read happens when a user submits a read request for some data that is currently unavailable. As a result, the requested chunk must be

reconstructed on the fly at the client before the system replies to the user request. Fig. 4.7c illustrates how PPR can drastically reduce the degraded read latency for four common RS coding parameters: (6, 3), (8, 3), (10, 4), and (12, 4), and for two different chunk sizes: 8MB and 64MB. Fig. 4.7c shows that the reduction in the degraded read latency becomes more prominent for the codes with higher values of k. Moreover, it is also noticeable that at a higher chunk size PPR provides even more benefits because of the reason discussed in Section 4.7.1.

4.7.2 Improving degraded reads under constrained bandwidth

PPR not only reduces the reconstruction time but also reduces the *maximum amount of data* transferred to any Chunk Server or a Client involved in the reconstruction process. In a PPR-based reconstruction process, a participating Chunk Server needs to transfer only $\lceil(log_2(k+1))\rceil$ number of chunks over its network link, as opposed to k number of chunks in a traditional repair. This becomes a desirable property when the network is heavily loaded or under-provisioned. In the next experiment, we use the Linux traffic control implementation (tc) to control the network bandwidth available to all the servers and measure the degraded read throughput. As shown in Fig. 4.7d, as we decrease the available bandwidth from 1Gbps to 200Mbps, the degraded read throughput with the traditional RS reconstruction rapidly drops to 1.2MB/s and 0.8MB/s for RS(6, 3) and RS(12, 4), respectively. Since, network transfers are distributed in PPR, it achieves higher throughput—8.5MB/s and 6.6MB/s for RS(6, 3) and RS(12, 4), respectively. With a relatively well-provisioned network (1Gbps), the gains of PPR are $1.8X$ and $2.5X$, while with the constrained bandwidth (200Mbps), the gains become even more significant, almost $7X$ and $8.25X$.

4.7.3 Benefit from caching

In this section we evaluate the individual contribution of the distributed reconstruction technique and caching mechanism to the overall benefit of PPR. The former

reduces the network transfer time, while the latter reduces the disk IO time. Fig. 4.7e shows that chunk caching is more useful for lower values of k (*e.g.,*, (6, 3) code). For higher values of k or for higher chunk sizes, the benefit of caching becomes marginal because the improvement in the network transfer time dominates that of the disk IO time. For instance, for $k = 12$ and 64MB chunk size, the caching mechanism provides only 2% additional savings in the total repair time. However, the caching mechanism can reduce the demand on disk IO, making it available for other workloads. Knowing the exact access patterns of data chunks will help us identify better caching strategies and choose the right cache size. We leave such exploration in realistic settings for future work.

4.7.4 Improvement in computation time

Now we compare PPR's computation to the serial computation in a traditional RS reconstruction, *i.e.,*, a default QFS implementation with the Jerasure 2.0 library [98]. Note that during reconstruction, either *decoding* (when a data chunk is lost) or *encoding* (when a parity chunk is lost) can happen (Fig. 4.3b). The amounts of computation required by RS encoding and decoding are almost identical [100]. The only difference is the extra matrix inversion involved in decoding. During our experiments we randomly killed a Chunk Server to create an erasure. Since, for the codes we used, there are more data chunks than parity chunks ($k > m$), decoding happens with higher probability than encoding. We report the average numbers and do not explicitly distinguish based on the type of the lost chunk. Fig. 4.7f shows that PPR can significantly speed up the computation time using its parallelism. These gains are consistent across different chunk sizes. Moreover, the gain is higher for higher values of k because the critical path in PPR needs fewer multiplications and XOR operations compared to traditional decoding. Existing techniques to reduce computation time for erasure codes using GPUs [101] or hardware acceleration techniques [102,103] are complementary to PPR. They can serve as drop-in replacements to the current

Jerasure library used by PPR. However, it should be noted, repair in erasure-coded storage is not a compute-bound task, but a network-bound task. Nevertheless, PPR helps to reduce the overall computation time.

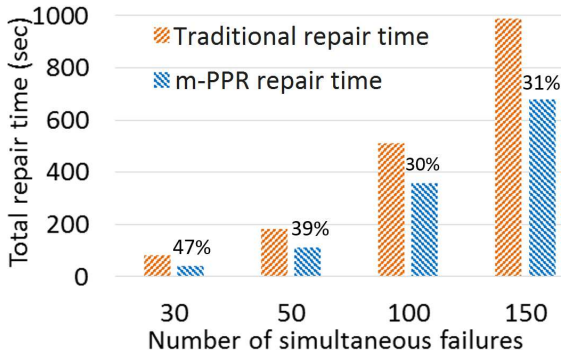


Fig. 4.8.: Comparison of total repair time for simultaneous failures triggered by Chunk Server crash

4.7.5 Evaluation with simultaneous failures (*m*-PPR)

In this section we evaluate the effectiveness of m-PPR in scheduling multiple repairs caused by simultaneous chunk failures. We control the number of simultaneous chunk failures by killing the appropriate number of Chunk Servers. We performed this experiment in the BigSite, where we placed the Meta-Server and the Client on two hosts and ran 83 Chunk Servers on the rest. The coding scheme was RS(12, 4) with 64MB chunks. Fig. 4.8 shows that our technique provides a significant reduction (31%–47%) in total repair time compared to the traditional RS repair. However, the benefit seems to decrease with a higher number of simultaneous failures. This is because, in our testbed configuration, the network links to the host servers that are shared between multiple repairs tend to get congested for large number of failures. Consequently m-PPR has less flexibility in choosing the repair servers. If the testbed has more resources (more hosts, higher network capacity, etc.), m-PPR will perform much better for the scale of simultaneous failures considered in our experiments. However, it should be noted that the main PPR technique does not reduce

the total amount of data transferred over the network during repair. Rather it distributes the network traffic more uniformly across network links. For a large number of simultaneous failures, if the repair sites are spread across the data center, m-PPR would provide reduced benefit compared to the single failure case. This is because the simultaneous repair processes on multiple nodes already spread the network traffic more evenly compared to the case of a single failure. Overall, the result validates that m-PPR can effectively handle multiple repairs and minimizes the competition for shared resources (*e.g.,*, network and disk) for a moderate number of simultaneous failures.

4.7.6 Scalability of the Repair-Manager

The Repair-Manager (RM) creates and distributes a repair plan to a few Chunk Servers that are selected as the aggregators. We investigate if the RM can become the bottleneck at large scale. As detailed in Sec. 4.5, the m-PPR scheduling algorithm has a time complexity of $O(Nlog(N))$ for scheduling each repair, where N is the number of possible destination servers. N is usually a small fraction of the total number of machines in the data center. Additionally, to handle a data chunk failure, RM computes the decoding coefficients, which involves a matrix inversion. Following this, RM sends $(1 + \frac{k}{2})$ messages to distribute the plan to the aggregation Chunk Servers. Not surprisingly, we observe that the time for coefficient calculation is negligible. Specifically for RS (6, 3) and (12, 4) codes, we measured the time period between the instant when the plan is created to the instant when the RM finishes distributing the plan for a single repair. It took on average 5.3ms and 8.7ms respectively. Thus for the two coding schemes, one instance of the RM is capable of handling 189 repairs/sec and 115 repairs/sec, respectively. Further, as discussed in Sec. 4.5, the planning capability can be easily parallelized by using multiple RM instances, each of which can handle disjoint sets of repairs.

4.7.7 Compatibility with other repair-friendly codes

PPR is compatible with most of the existing erasure coding techniques. Its applicability is not limited to only RS codes. We demonstrate its compatibility by applying it on top of two popular erasure coding techniques—Local Reconstruction Code (LRC) [73] and Rotated RS code [83]. These are the state-of-the-art codes targeted for reducing the repair time.

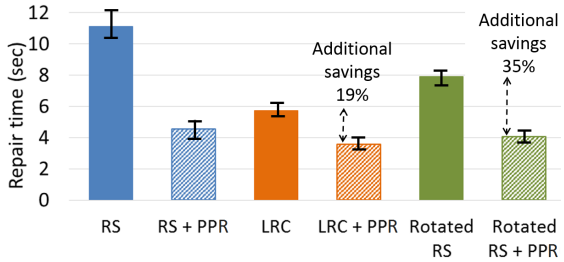


Fig. 4.9.: PPR repair technique can work with LRC and Rotated RS and can provide additional improvement in repair time

Improvements over LRC code: Huang *et al.* introduced Local Reconstruction Code (LRC) in Windows Azure Storage to reduce the network traffic and disk IO during the reconstruction process [73]. LRC stores additional local parities for subgroups of chunks, thereby increasing the storage overhead for comparable reliability. For example, a (12, 2, 2) LRC code uses two global parities and two local parities, one each for a subgroup of six chunks. If one chunk in a subgroup fails, LRC needs only six other chunks to reconstruct the original data compared to 12 in RS (12, 4) code. Papailiopoulos *et al.* [74] and Sathiamoorthy *et al.* [75] also proposed Locally Repairable Codes that are conceptually similar. For our experiments, we emulated a (12, 2, 2) LRC code that transfers six chunks over the network, in the best case, to one Chunk Server in order to reconstruct a missing chunk. Then we applied PPR-based reconstruction technique for LRC to create `LRC+PPR`.

In `LRC+PPR` only three chunks are transferred over any particular network link. In Fig. 4.9, for a 64MB chunk size, PPR-based reconstruction on (12, 4) RS code was

faster than a (12, 2, 2) LRC code reconstruction because the maximum number of chunks that must go through any particular network link is only $4C$ for PPR as opposed to $6C$ in case of LRC, where C is the chunk size. More interestingly, `LRC+`PPR version performs even better resulting in 19% additional reduction, compared to using LRC alone. Even in the worst case, for the `LRC+`PPR only three chunks are transferred over any particular network link.

Improvements over Rotated RS code: Khan *et al.* [83] proposed Rotated RS code that modifies the classic RS code in two ways: a) each chunk belonging to a single stripe is further divided into r sub-chunks and b) XOR on the encoded data fragments are not performed within a row but across adjacent rows. For Rotated RS code, the repair of r failed chunks (called "fragments" in [83]), requires exactly $\frac{r}{2}(k+\lceil(\frac{k}{m})\rceil)$ other symbols when r is even, compared to $r \times k$ data fragments in the RS code. On an average, for a RS(12, 4) code and $r = 4$ (as used by the authors [83]), the reconstruction of a single chunk requires approximately nine other chunks, as opposed to 12 chunks in traditional RS codes. However, the reconstruction is still performed after gathering *all* the necessary data on a single Chunk Server. As can be observed from Fig. 4.9, PPR with RS code outperforms Rotated RS. Moreover, the combined version `Rotated RS+`PPR performs even better and results in 35% additional reduction compared to the traditional RS repair.

4.7.8 Discussion

It is worthwhile to discuss whether emerging technologies, such as the zero-copy-based high throughput networks (e.g., Remote Direct Memory Access (RDMA)), would remove the network bottleneck. However, it should be noted that other system components are also getting better in performance. For example, Non-Volatile Memory Express (NVMe) and hardware-accelerator-based EC computation have the potential to make the non-network components to be even faster. Moreover, application data is likely to grow exponentially putting even more pressure on the future

data center network. Thus, techniques like PPR that attempt to reduce the network bottleneck would still be relevant.

4.8 Related Work

Quantitative comparison studies have shown that EC has lower storage overhead than replication while providing better or similar reliability [104, 105]. TotalRecall [106] dynamically predicts the availability level of different files and applies EC or replication accordingly. Publications from Facebook [107] and Microsoft [73] discuss the performance optimizations and fault tolerance of their EC storage systems.

A rich body of work targets the reconstruction problem in EC storage. Many new codes have been proposed to reduce the amount of data needed during reconstruction. They achieve this either by increasing the storage overheads [73,74,108,109], or under restricted scope [83,110–112]. We have already covered the idea behind Local Reconstruction Codes [73] and the conceptually identical Locally Repairable Codes [74,75] when presenting our evaluation of PPR coupled with these codes. The main advantage of our technique compared to these is that PPR neither requires additional storage overhead nor mandates a specific coding scheme. Moreover, our technique is fully compatible with these codes and can provide additional benefits if used together with them, as shown in our evaluation. Another body of work suggests new coding schemes to reduce the amount of repair and IO traffic, but comes with restricted settings. Examples are Rotated RS [83] and Hitchhiker [82]. Yet another class of optimized recovery algorithms are EVEN-ODD [111] and RDP codes [112]. However, they support only two parities, making them less useful for many systems [82]. In contrast, PPR can work with any EC code.

In a different context, Silberstein *et al.* [80] proposed that delaying repairs can lead to bandwidth conservation and marginally increases the performance of degraded reads as well. However, such a policy decision will not be applicable to many scenarios because it puts the reliability of the data at risk. Xia *et al.* [88] proposed a

hybrid technique using two different codes in the same system, *i.e.,*, a fast code and a compact code. They attempted to achieve faster recovery for frequently accessed files using the fast code, and to get lower storage overhead for the less frequently accessed files using the compact code. This technique is orthogonal to our work, and PPR can again be used for *both* fast and compact codes to make reconstruction faster. In the context of reliability in replicated systems, Chain Replication [113] discusses how the number of possible replica sets affects the data durability. Carbonite [114] explores how to improve reliability while minimizing replica maintenance under transient failures. These are orthogonal to PPR. Lastly, several papers evaluate advantages of deploying EC in distributed storage systems. OceanStore [115, 116] combines replication and erasure coding for WAN storage to provide highly scalable and durable storage composed of untrusted servers.

4.9 Conclusion

In this section we present a distributed reconstruction technique called PPR, for erasure coded storage. This achieves reduction in the time needed to reconstruct missing or corrupted data chunks, without increasing the storage requirement or lowering data reliability. Our technique divides the reconstruction into a set of partial operations and schedules them in parallel using a distributed protocol that overlays a reduction tree to aggregate the results. We introduce a scheduling algorithm called m-PPR for handling concurrent failures that coordinates multiple reconstructions in parallel while minimizing the conflict for shared resources. Our experimental results show PPR can reduce the reconstruction time by up to 59% for a (12, 4) Reed-Solomon code and can improve the degraded read throughput by 8.25X, which can be directly perceived by the users. Our technique is compatible with many existing codes and we demonstrate how PPR can provide additional savings on latency when used with other repair-friendly codes.

5. IMPROVING APPLICATION PERFORMANCE THROUGH PHASE-AWARE APPROXIMATION

5.1 Introduction

Approximate computing is a new computing paradigm that trades accuracy of computation for savings in execution time and/or energy by leveraging approximation opportunities across the computing stack, including programming languages [117–120], compilers [121–124], runtime systems [125–127], and hardware [128–131]. In addition to data-parallel and streaming applications [132–134] researchers proposed approximation techniques suitable for iterative numerical computations, such as iterative solvers, large scale numerical models, and sparse matrix calculations [135–137].

Approximate computing techniques typically introduce inexactness and/or approximation by transforming compute-intensive kernels, which we call *approximable blocks* (ABs). Furthermore, many approximation techniques expose *knobs* to calibrate the approximation levels (ALs), which control the error or speedup. For instance, *loop perforation* [121, 123] skips a fraction of a loop's iterations, and exposes this fraction as a knob to control the accuracy/speedup tradeoff.

Outer-Loop Pattern. Many applications follow a computation pattern in which the majority of computation is performed inside a main loop (we refer to it as the *outer loop*) encompassing all the ABs. Examples of outer loops include *timestep* loops in scientific simulations, *convergence* loops in iterative solvers, or *enumerator* loops for processing a series of information blocks (e.g., video frames). Configurations of the ALs in the internal ABs can often affect the number of iterations of the outer loop, ultimately impacting the application-level speedup and quality of service (QoS).

For large applications with multiple ABs, the trade-off between speedup and error becomes complex. Often the optimum configuration of ALs in the various ABs are

not obvious, especially if the approximation of internal ABs influences the number of iterations of the outer loop. Off-line exhaustive search can be possible only for a small number of approximate program configurations [123], and the majority of the previous approaches used various heuristic search strategies based on representative inputs [118,120–122], or dynamically tuned the ALs based on the observed errors from the approximated regions [120,122,138,139]. While many of these techniques identify and leverage properties of specific code patterns in ABs, they typically apply the same transformation for the entire execution and/or input. Such fixed optimization choices may lead to rigid transformed programs that miss fine-grained optimization opportunities.

Phase-Aware Optimization. For many iterative computations, the outer loop controls the precision of the final solution. Here, the iterations of the outer loop naturally segment the overall application execution into multiple *phases*. We define a phase as *a segment of execution that has distinct speedup or error characteristics*. For example, a numerical solver execution can go through an initialization phase, a maturity phase, and a convergence phase.

Our experimental results show that two different phases of the computation may generate different amounts of error for the same level of approximation. This exposes a new opportunity for optimization algorithms – they can select not just *how much* to approximate, but also *in which phase* to approximate. We find empirically that for some applications (such as LULESH [25]), approximating one phase may induce almost 8x less error than applying the same approximation in another phase of the execution.

Our Solution Approach. We present OPPROX, a novel system for phase-aware optimization of approximate programs. OPPROX takes as inputs: a program with tunable approximation and a user-provided *accuracy specification*, which consists of (1) a set of representative inputs, which exercise the application's desired functionality, (2) an accuracy metric, which tells how to compute the difference between the results of the exact and the approximate execution, and (3) an acceptable error bud-

get e_b, which specifies how much imprecision in the final output the user is ready to tolerate. OPPROX can work with any approximation technique that exposes multiple ALs.

OPPROX operates in four conceptual steps. First, OPPROX identifies different phases during the program execution. Second, OPPROX models the speedup and error generated due to different levels of approximation in the individual ABs *and* in different execution phases using representative inputs. Third, OPPROX compares the benefits of using various approximation settings in different phases and splits the overall error budget e_b into phase-specific error budgets in proportion to that predicted benefits. Finally, OPPROX formulates phase-specific trade-off space exploration as a numerical optimization problem and finds the most profitable approximation settings for each phase using the phase-specific error budgets as the constraints.

Our results show that for many applications, *both* the approximation level and the phase in which approximation is performed, have significant contributions towards the final error. Hence, phase-specific optimal approximation settings can provide good speedup (which we express here using the amount of instructions executed) even under constrained error budget. When compared to an oracle but phase-agnostic version from prior works [123, 139], our approach on average provides 42% speedup compared to 37% from the oracle version for an error budget of 20% and for a small error budget of 5% provides on average 14% speedup compared to only 2% achieved by the phase-agnostic oracle version.

Contributions. This section makes following contributions:

1. We introduce the concept of phase-specific approximation for controlling the approximation error and improve application speedup.
2. We introduce modeling of application speedup and approximation error based on polynomial regression that captures (a) the dependency on input-parameter and phase of the application and (b) impact of approximation levels on the number of iterations of enclosing loops.

3. We define the phase-specific approximation space exploration as a numerical optimization problem and present an algorithm to find profitable configurations for multiple approximations under a given error budget.
4. We evaluate OPPROX on five benchmarks and four existing approximations. The results show that phase-aware approximation becomes very attractive for improving speedup (especially when operating under low error budget) compared to phase-agnostic approximation that was used by prior works [123, 139].

5.2 Example

We explain the motivation for OPPROX and how it works using LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) as an example. LULESH [25] is a widely used hydrodynamic application that simulates the Sedov blast wave problem [140] in three dimensions. It represents a typical hydrodynamics code that solves the hydrodynamics equations by partitioning the equations spatially. Fig.5.1 gives an abstract representation of the main computation part in LULESH. In the main computation, LULESH iterates through an outer loop until the simulation reaches a stable state (*i.e.*,, until a condition called the *Courant condition* is met). Inside this outer loop, LULESH computes several physical quantities.

At the end, LULESH reports the final energy for each of the elements it simulated.

```
1  elements = A set of elements to simulate
2  While(state->stable ==false)
3  {
4        increment_simulation_time();
5        forces_on_elements();
6        acceleration_of_elements();
7        velocity_of_elements();
8        position_of_elements();
9        strain _of_elements();
10       calculate_timeconstraints();
11       state = get_current_state();
   }
```

Fig. 5.1.: Abstract computation pattern for LULESH. The `while` loop iterates until the simulation achieves stable state.

Accuracy Specification. The quality metric for LULESH is the difference in the final energy from the approximate run compared to that from the accurate execution and averaged across all the elements.

Application Profiling. Given an error budget provided by the user, OPPROX tries to find the best configuration to maximize the speedup of LULESH. We represent the speedup in terms of the number of instructions executed in the program. At first, we profile LULESH to find six most compute intensive kernels (lines 5-10 in Fig.5.1). Then, OPPROX applied three approximation techniques – loop perforation, loop truncation, and memoization (we review the techniques in Sec. 5.3.2).

Ultimately, we found four of these loops inside the logical functions: *forces_on_elements*, *position_of_elements*, *strain_of_elements*, and *calculate_timeconstraints*, can sustain such approximations and did not lead to either a hang or crash or unusable QoS degradation. These 4 kernel loops form the approximable blocks (ABs) for LULESH. This process of finding ABs is analogous to the one in [121]. The approximation levels (ALs) corresponding to these loops were exposed as environment variables and each one can be set to different levels from 0 to 5 where 0 being the accurate run and 5 is the run with maximum approximation.

Phase-Specific Behavior of Approximable Blocks. For some ABs, as we increase the approximation level, as expected, we observe application speedup as well as an increase in QoS degradation, as shown in Fig.5.2. However, while running LULESH with different combinations of ALs corresponding to different ABs, we observed that the number of iterations in the outer loop also varies significantly as can be seen in Fig.5.3 — when run without any approximation the outer loop iterates 921 times, with some combinations of ALs it increases to 965 times and as a result *slows down* the application instead of speeding it up. The maximum speedup we observed was 1.34 but at a cost of 38% QoS degradation. Further, we explored whether approximating only during some selected duration of the execution would help us to have a better control over QoS degradation and speedup. In this example, we divided the outer loop iterations into 4 phases—each phase comprises an equal number of

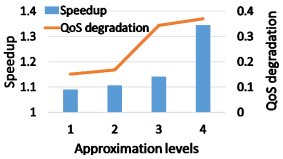


Fig. 5.2.: Both speedup and error increase with approximation levels of the blocks in LULESH.

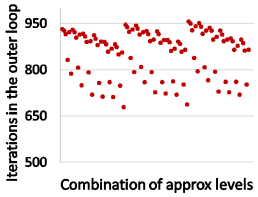


Fig. 5.3.: Variation in the number of iterations made by the outer loop in LULESH.

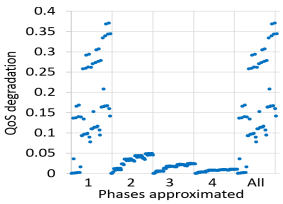


Fig. 5.4.: LULESH phase-specific QoS

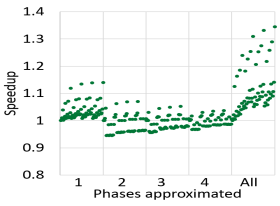


Fig. 5.5.: LULESH phase specific speedup

outer loop iterations— and selectively approximated only in one phase. We show the results in Fig.5.4 and Fig.5.5, where the different points within one phase correspond to different combinations of the ALs. We see that approximating in phase-1 can provide speedup but also drastically degrades the QoS level. This behavior can be explained from the intrinsic nature of the algorithms involved in LULESH. Approximation during the initial phases makes it difficult to meet the stable condition leading to QoS degradation. Approximation in later phases reduces the impact of such errors because the accurate execution of the first phase already took the simulation much closer to the golden values. For example, approximation only in phase-4, generates negligible error but still can provide some speedup with different approximation settings. Therefore, phase-specific approximation gives better opportunity to find a suitable combination for speeding up the application, *especially when operating under low error budget.*

Phase-Aware Optimization of Approximate Blocks. OPPROX takes several steps to build such phase specific speedup and QoS degradation model for LULESH as illustrated in Fig.5.6. At first, OPPROX instruments the LULESH code by adding log messages to capture the call-context corresponding to the ABs. Then LULESH is run with different combinations of its input parameters (length of cube mesh and number of regions) and the sequence of unique AB call-context are extracted from the logs. These sequences are used to classify control-flows based on input parameters

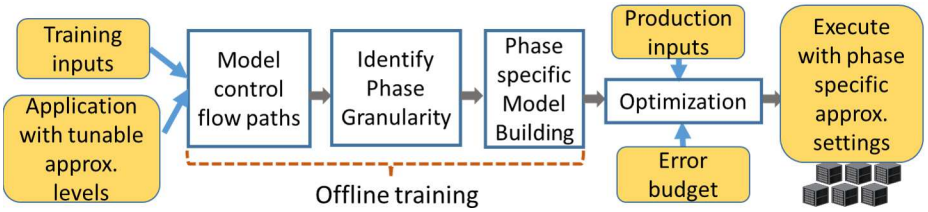


Fig. 5.6.: Workflow of OPPROX

and build separate models for each distinct control-flows to capture input-parameter-dependent behavior.

For LULESH, OPPROX automatically divides the execution in to 4 phases. For each phase, it builds polynomial regression models for speedup and QoS degradation by using random samples of different combinations of input parameter combinations and ALs for that phase. For LULESH, the R^2 score corresponding to the final QoS degradation and the speedup model were 0.94 and 0.99 respectively.

Application-level Optimization. Now, for a user provided QoS budget, OPPROX uses optimization to find the best phase specific ALs. At first, it allocates sub-budget (a portion of the QoS degradation budget) to each phase in proportion to the mean value of the ratios of how much speedup is gained from that phase to the amount of QoS degradation. Then OPPROX finds the best combination of approximation for that phase subjected to the allocated sub-budget. Any unused sub-budget from one phase is reallocated to the other phases. For LULESH, initially sub-budget allocated to the 4 phases are in proportion to 0.166, 0.17, 0.265, and 0.399 of the full budget e_b. Using the optimized settings suggested by OPPROX, for error budgets (e_b) of 20%, 10%, and 5%, the approximate versions of LULESH achieved speedups of 1.28, 1.21, and 1.17 respectively.

5.3 Opprox

We illustrate the overall workflow of OPPROX in Fig.5.6. OPPROX performs an offline training phase using execution logs collected from multiple runs. Then at

runtime, it finds phase-specific approximation for a user provided error budget, by solving a numerical optimization problem.

5.3.1 Opprox Inputs

Application With Tunable Approximation Levels. OPPROX requires that user has already identified the ABs and has implemented suitable approximation techniques that provides multiple approximation levels and can be *tuned* by OPPROX to achieve a sweet spot in the speedup vs. QoS degradation curve. In general, for compute intensive kernels the computation time decreases (*i.e.,* speedup increases) with an increase in the AL. At the same time, the QoS degrades with an increase in the AL due to the inaccuracies introduced by the technique, as illustrated in Fig.5.2 for LULESH.

To choose the ABs, we follow the sensitivity profiling procedure presented in prior work [121]. In particular, these techniques filter out the blocks where approximation makes the program to crash, introduces memory leaks, or results in unacceptable-quality output. As part of the sensitivity profiling, we try different approximation techniques on the compute intensive blocks and finally choose a set of blacks that are both compute intensive and can withstand certain levels of approximation.

QoS Metric. The Quality of Service (QoS) is an application specific metric that captures how different are the results from approximate computing when compared to results produced by an *exact* computation, and denote it as δQoS. Some applications have common domain-specific metrics, e.g., for image or video processing applications the QoS can be the value of Peak Signal to Noise Ratio (PSNR). For the applications that do not have a domain-specific QoS metric, we use a default *distortion* [117], which computes the relative scaled difference between the outputs generated from an approximate computation and an exact computation.

5.3.2 Examples of Approximation Techniques

There are many available techniques [123, 128, 130] that can be used to approximate an AB. The concept of OPPROX is generic and can be applied with *any* approximation technique that provides multiple ALs for each AB. Here, we assume that the approximation exposes the variable `approx_level`, which controls the approximation level for each of the techniques. In this paper, we analyze four previously proposed techniques:

Loop perforation: In loop perforation [121, 123], the computation time is reduced by skipping some iterations, as shown below. The behavior essentially samples the result space.

```
for (i = 0; i < n; i = i + approx_level)
  { result = compute_result(); }
```

Loop truncation: In this technique [121, 123], we simply drop last few iterations as shown in the following example:

```
for (i = 0; i < (n − approx_level); i++)
  { result = compute_result(); }
```

Memoization: In this technique [141], for some iterations inside a loop we compute the result and cache it. For other iterations we use the previously cached results.

```
for (i = 0; i < n; i++)
  { if (0 == i % approx_level)
        cached_result = result = compute_result();
    else result = cached_result; }
```

Parameter tuning: In some applications, there exist some input parameter which can directly be used to control the accuracy of the computation [125]. For example, in Bodytrack, the parameters `min-particles` and the *number of annealing layers* is suitable for this purpose. Overall, users can provide a list of the parameter names and the set of values that the parameter may take.

5.3.3 Sampling for Training Data

OPPROX collects training data by profiling the instrumented application with different combinations of ALs for each AB and a variety of representative input parameters provided by the user. During each run it collects the number of instructions executed by each AB in each iteration of the outer-loop, the sequence of ABs executed and the QoS degradation w.r.t the golden output obtained from corresponding accurate execution. OPPROX first builds local models for each AB, hence for each AB, it exhaustively covers the corresponding AL-space, while executing all other ABs accurately. Then, to capture the interaction due to approximations in multiple ABs, random sparse samples are collected where approximation levels in *all* the ABs are arbitrarily set between zero (accurate computation) and the maximum approximation level. We assume the number of discrete ALs in each AB are not high (usually between 4-8, in our case), hence to build good local models, exhaustive sampling is required. In case, number of discrete ALs are high, sparse sampling can also be used for the local models. We collect phase-aware training data by doing such sampling for each phase.

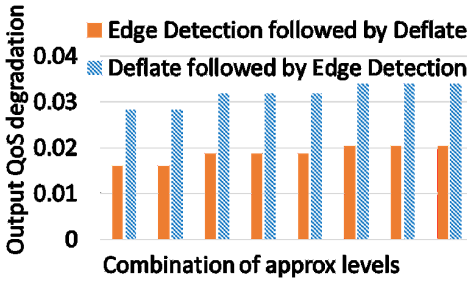


Fig. 5.7.: FFmpeg: Swapping the order of two filters: Deflate and Edge Detection, results in significant change in the QoS degradation.

5.3.4 Predicting Control Flows

Application control flow, *i.e.*,, the ABs that would be executed and their execution sequence can change depending on the inputs parameters resulting in different

speedup and QoS degradation characteristics. As shown in Fig.5.7, we found in FFmpeg, if we swap the order of approximated *edge detection* and *deflate* filters, QoS degradation drastically changes. OPPROX uses the training data to learn these AB-level control flow paths from the call-context logs. The ideal training set should consist of variations of inputs parameters to capture all the control flow paths that can be encountered in the production. Using these training data OPPROX uses a *Decision Tree* to classify which control-flow the application would take, *i.e.,*, which ABs and what sequence would be executed for a given combination of input parameters.

5.3.5 Identifying Phase Granularity

The QoS degradation of a variety of applications depends not only on the ALs in the ABs but also on the phase of application's execution in which the approximation was done. In majority of the cases, we found application suffers low QoS degradation if the approximation is turned on during later phases of the execution.

Algorithm 6 Finding proper phase granularity

```
1: app ← Application under test
2: thresh ← Phase sensitivity threshold for QoS
3: N ← 2 # Number of phases
4: maxDiffPrev = getMaxQoSDiff(app,N)
5: while True do
6:     newN=N*2
7:     maxDiffNew = getMaxQoSDif(app,newN)
8:     if abs(maxDiffPrev - maxDiffNew) > thresh then
9:         N ← newN
10:    else
11:        Break
12:    end if
13: end while
```

To find the best set of phases for approximation, OPPROX progressively divides the overall execution of the application into N logical phases. Then it measures whether the maximum difference between the mean QoS degradation or mean speedup between two consecutive phases would change more than a user-provided threshold value if number of phases are further increased as illustrated in Algo 6 in the context of QoS degradation. The `getMaxQoSDiff` function runs the application with N phases

and multiple approximation settings and also calculates the above mentioned maximum difference in QoS degradation between two consecutive phases. While a large value of N can capture the relationship between phases and QoS degradation at finer details, the size of the search space (and the training time) increase exponentially.

5.3.6 Performance and Error Models

Estimating Iteration Counts. We define the speedup in terms of the computation or amount of work done, *i.e.,*, the total number of instructions executed as follows:

$$S = \frac{\#(\text{instructions executed in accurate run})}{\#(\text{instructions executed in approximate run})}$$

As a result of approximation, each AB gets some speedup. However, the final speedup of an application depends not only on the local speedups gained at the ABs, but also the number of iterations of the outer loops encompassing those ABs. Recall, the number of iterations of these outer loops can be constant (*e.g.,*, in a predefined timestep based simulation), input parameter dependent (*e.g.,*, in FFmpeg, depends on the number of frames in a video) or can depend on internal approximations (*e.g.,*, in LULESH).

OPPROX extracts the number of iterations made along the outer loop by calculating how many times a call-context sequence of ABs has repeated in the execution log. OPPROX uses polynomial regression to build estimators for the number of outer loop iterations using the approximation settings of the internal ABs and input parameters as the modeling input.

Modeling Speedup and QoS Degradation. Approximation levels at the ABs and the corresponding phases dictate how much speedup can be obtained and how much QoS degradation would be incurred. For each unique control flow path, OPPROX builds separate models to capture the application behavior. Phase specific speedup and QoS degradation models are built in two steps: first, local models are built to capture the speedup or QoS degradation when only one AB is approximated at a

time. Each AB will have such local speedup and QoS degradation approximation models per phase. These models take as input the ALs for that particular AB and input parameters provided to the application. The second step builds phase specific models to capture the combined effect on overall speedup or QoS degradation when multiple ABs are approximated simultaneously. The models in this second step uses the prediction (speedup or QoS degradation) from the models from the first step as inputs.

Now, the final QoS degradation or speedup also explicitly depends on how many times each AB is called *i.e.,*, how many times the outer loop executes. This is because, as for the QoS, the more number of times an AB is called, the more QoS degradation it is likely to create due to approximation error. For speedup, it is slightly more complex, an approximation of an inner AB may actually increase the number of iterations of the outer loop and thus *decrease* the speedup. Let us take an example of an outer loop with only one AB inside. Let W be the amount of work done by this block per iteration. After approximation the amount of work performed by this block becomes w resulting in a local speedup of: x=W/w. Say, at the same time the number of iteration of the outer loop changed from I to i. due to approximation. Then, the overall speedup of the application would be: $(I*W)/(i*w)$ or $I*x/i$. Thus the overall speedup would depend not only on x but also on the change in the number of iterations in the outer loop i. Hence, we explicitly use our estimated number of outer loop iterations as an input feature while building our overall speedup and QoS models to ensure at least one input variable will closely dictate the output. This technique generalizes to any loops whose iteration may vary due to approximations in the internal ABs.

Confidence Analysis of Models. There might be errors in these machine learning based models itself because the training data does not exhaustively capture all the possible scenarios due to combinatorial explosion. To address this problem, OPPROX calculates a confidence interval of its models by adapting the approach from [142].

For example, if OPPROX predicts Q as the QoS degradation value for a particular approximation combination and p% of the time modeling error remains within e%, then it interprets that actual QoS degradation for that approximation settings can be anywhere between $[Q-e, Q+e]$ which form the confidence interval. Here p is a means of controlling the *confidence* in the prediction. To remain conservative, OPPROX use the upper limit of the p=100 confidence interval as the effective QoS degradation and use the lower limit in case of speedup estimation. This ensures that we avoid the risk of going over the QoS degradation budget.

5.3.7 Improving Modeling Accuracy

To reduce noise during modeling, OPPROX uses Maximal Information Coefficient (MIC) [143], to determine if an association exists between any given input feature, which could be an input parameter to the application or the approximation level of any AB, and the target output of the model, which could be the number of iterations of the outer loop, the degradation of QoS, or the speedup of the AB. Features not having an association with the output are filtered out.

After this filtering, OPPROX gradually increases the degree of the polynomial regression until it finds a good R^2 score with 10-fold cross validation. In 10-fold cross-validation, as per standard practice, the original training data is randomly partitioned into 10 equal size subsets. Of the 10 subsets, 9 subsets are used for training and the remaining single subset is used for testing the model. This process is then repeated 10 times, with each of the 10 subsets used exactly once as the test data. The 10 results from the folds are then averaged to produce a final estimation. Use of cross-validation avoids overfitting.

OPPROX checks if the models achieve a target accuracy (*i.e.,*, a good R^2 score). The value of this target accuracy is a design choice, *e.g.,*, a value greater than 0.9 may denote a good model. If OPPROX finds that the models are not accurate enough for the entire input data set, it breaks the input into smaller subcategories and attempts

to build a model for each subcategory. To create the sub-models, OPPROX takes one input feature, splits its training values put in magnitude order into k subsets, and learns separate models for each subset. It repeat this for other features until achieving a good modeling accuracy.

5.3.8 Optimization Framework

The final goal of OPPROX is to find the optimal settings for the ALs for each phase of the application that would maximize the speedup of the application, for a given QoS degradation budget QoS_b specified by the user. The overall optimization algorithm framework is shown in Algo. 7.

Phase Specific Allocation of QoS Degradation. By analyzing various applications we found that the same approximation in different phases of the execution achieves different speedups, and causes different levels of QoS degradation. For the purpose of our optimization we define a metric called *return on investment* (ROI) of QoS degradation budget for a phase ph as follows:

$$roi_{ph} = \frac{1}{m} \sum_{i=1}^{m} \frac{S_i}{\delta QoS_i} \tag{5.1}$$

Here, m is the number of available training data points for the phase ph. S_i is the speedup for the i_{th} data point and δQoS_i is the corresponding QoS degradation. Intuitively, the ROI value for a phase gives a statistical measure of how much benefit we are likely to get at the expense of certain amount of QoS degradation for that phase.

OPPROX divides the overall QoS degradation budget across all the phases of execution *in proportion* to their corresponding ROI values. Thus, for a given a QoS degradation budget QoS_b, the share of the budget allocated to the phase ph would be: $normROI_{ph} * QoS_b$, where $normROI_{ph}$ is the ROI of this phase normalized by the sum of the ROIs of all the phases. This is a policy decision of how to divide the

overall QoS degradation and OPPROX can accommodate other policies than the one described above.

Algorithm 7 Finding phase specific approximation settings

```
1: QoS_b ← Total QoS degradation budget
2: models ← Phase specific approximation models
3: sortedPhases ← sortPhasesBasedOnROI()
4: for all phase in sortedPhases do
5:     normROI ← calculateNormalizedROI()
6:     phaseQoSBudget ← QoS_b*normROI
7:     phaseModel ← models[phase]
8:     consumedQoS = optimizePhase(phaseModel, phaseQoSBudget)
9:     QoS_b ← QoS_b − consumedQoS
10: end for
```

Finding the Optimal Settings for Each Phase. OPPROX uses the QoS degradation budget allocated to each phase to find the optimum settings for approximation for that phase that would maximize the speedup.

Assume there are M, ABs and $A = (A_1, \ldots, A_M)$ denotes the *configuration* of the ALs for these blocks for a phase ph. The values each A_i can take are the discrete approximation levels for the corresponding block. Let $S(A)$ be the speedup of the application, and $\delta QoS(A)$ be the QoS degradation as a result of these approximations. Thus, OPPROX's goal is to find the optimum value of A for phase ph that will maximize the speedup while keeping the overall QoS degradation within the budget:

$$\begin{aligned} \underset{A}{\text{maximize}} \quad & S(A) \\ \text{subject to} \quad & \delta QoS(A) \leq normROI_{ph} * QoS_b \end{aligned}$$

Using the previously described techniques in Sec. 5.3.6, OPPROX estimates the value of S and δQoS for each A by solving a (polynomial) numerical optimization problem as depicted in Algo.7 as the function *optimizePhase* (which we inlined). OPPROX searches the configuration space among the phases in the decreasing order of their ROI values and any QoS budget left over are redistributed among the other phases (Algo. 7). Left over QoS budget exists when even the most aggressive approximation cannot cause the budgeted error.

5.4 Experimental Methodology

For evaluation, we use five representative applications and benchmarks from a wide variety of domains. Here, we describe these applications and implementation of OPPROX.

5.4.1 Description of the Applications

LULESH: We provided details for LULESH in Sec. 5.2.

CoMD: CoMD [43] is a representative application for a broad class of molecular dynamics(MD) simulations. In general, the method of MD simulation involves the evaluation of the force acting on each atom due to all other atoms in the system and the numerical integration of the Newtonian equations of motion for each of those atoms.

[QoS Metric:] At the end of the simulation, the energy of the system is expressed in terms of the potential and kinetic energy of the atoms. As the QoS metric, we use the difference in potential and kinetic energy compared to the accurate execution and averaged across all the atoms.

[Computation Pattern:] CoMD's main computation is surrounded by an outer loop which iterates for the number of simulation timesteps provided as the input. This outer loop internally calls several compute intensive functions. CoMD outer loop represents a classic timestep loop in scientific computations where the number of timesteps is an input parameter. The outer loop iteration for CoMD does not depend on any other input parameters or the ALs of the internal ABs.

FFmpeg: FFmpeg [44] is a widely used video processing toolkit which provides a large number of filters to process a video, like edge detection filter, blur, color balance, deshake etc. These can be combined in various ways for a specific type of processing.

[QoS Metric:] As the QoS metric for FFmpeg, we use the standard PSNR (peak signal to noise ratio).

Table 5.1.: Application specific input parameters, approximation techniques used and number of combinations explored

Apps	**Input parameters**	**Approx. techniques used**	**Search space** (# approx. settings)
LULESH	length of cube mesh, # regions	loop perforation, loop truncate, memoization	699,840
FFmpeg	frames per second, video duration, bitrate, filters	loop perforation, memoization	207,360
Bodytrack	# annealing layers, # particle, # frames	loop perforation, input-tuning	1,966,080
PSO	Swarm size, dimension	loop perforation, memoization	14,400
CoMD	# unit cells, lattice parameter, # timestep	loop perforation, loop truncate	229,500

[Computation Pattern]: FFmpeg passes encoded video to a decoder which produces uncompressed frames (raw video). Inside an outer loop, FFmpeg applies a series of filters on each frame to process the video in various ways. After filtering, the frames are re-encoded and passed to a multiplexer, which writes the encoded packets to the output file. FFmpeg outer loop represents typical streaming analytics loops. The number of iterations depends on the input parameter, the number of video frames, and not on the ALs.

Bodytrack: Bodytrack [144] is a computer vision application that uses an annealed particle filter and videos from multiple cameras to track the movement of a human through a scene.

[QoS Metric]: QoS metric is the distortion of the vectors that represent the position of the body parts. The weight of each vector component is proportional to its magnitude. Vector components which represent larger body components (*e.g.,*, torso) therefore have a larger influence on the QoS metric than vectors that represent smaller body components (*e.g.,*, forearms).

[Computation Pattern]: For every frame of the input videos, the application extracts the image features and computes the likelihood of a given pose in a annealed particle filter. The main computation is inside an outer convergence loop. Inside the loop, the likelihood weight for each particle is calculated and if that results in an invalid model, particles are removed. Bodytrack's outer loop is also a type of conver-

gence loop. The number of iterations depend on the number of annealing layers and not on the internal ALs. However, when the value of `min-particles` is small, the iteration count also depends on the ALs.

Particle Swarm Optimization: Particle swarm optimization (PSO) [145] is a population-based stochastic approach for solving continuous and discrete optimization problems. We used an implementation for continuous functions (called the objective functions). PSO has similarity to evolutionary computation where the algorithm is expected to move the swarm of particles toward the best solutions.

[QoS Metric]: QoS metric is the average difference in the final value of the best fitness vector calculated for each particle in the swarm.

[Computation Pattern]: PSO starts with a population of candidate solutions, also called particles. PSO's main computation part is inside an outer loop which in each of its iteration improves a candidate solution until the convergence criterion is met. In each iteration, the computation computes new positions and velocities of the particles in the search space. The number of iterations depends on the internal ALs, and varied between 1,612 to 5,257 in our experiments.

5.4.2 Implementation Details

In this section we briefly discuss some of the design choices and implementation details. Table 5.1 mentions which of these approximation techniques were used for each application. We had 4 ABs for LULESH and Bodytrack, and 3 ABs for CoMD, PSO and FFmpeg. Depending on the application and the AB, we used between 4 to 8 different approximation levels and up to 27 different input combinations. While trying to find optimal number of phases for dividing the application execution, we explored up to N=8 phases. Table 5.1 summarizes the total number of approximation combinations we collected. For regression models, we found the polynomial degrees varied between 2 to 6 for all the models in our applications corresponding to an R^2 score greater than 0.9.

What happens at the runtime. For each application, the trained models are

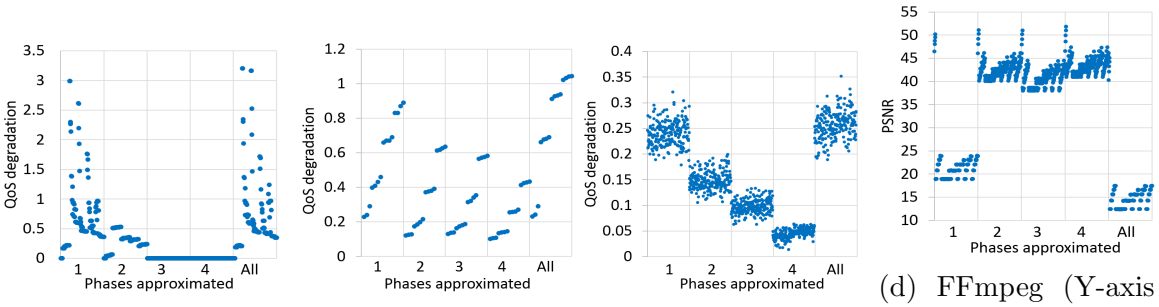
stored as Python's serialized `pickle` format in designated locations. User submits the job with a target error budget in a configuration-file. Then a runtime-script loads the corresponding models and finds the best phase-specific approximation settings for that error budget using OPPROX's optimizer on the trained-models and invokes the `SLURM` native scheduler. The phase-specific approximation settings are passed to the job via environment variables; exactly specifying what would be the approximation level for each AB during each phase of the execution.

5.5 Evaluation

We now present the experimental results. We performed all evaluations on 64-bit Intel Xeon Phi machines with 64GB of RAM running RHEL 6.6, 64-bit OS. Following the same approach as [123, 125, 138, 139], we ran all the applications in serial mode using one thread. Applications were compiled with `gcc` version 4.8.4 and with `O3` optimization.

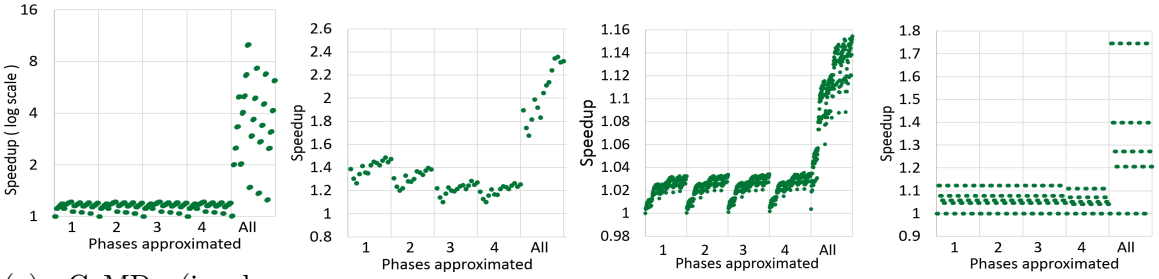
5.5.1 Phase Specific Behavior

First, we show how the QoS degradation and speedup varies as we turn on approximation in different phases. To show a visually comparable phase-specific behavior, for all the applications we divide the main computation into 4 phases of equal length. Here a phase is defined in terms of the number of iterations in the main outer loop. Fig.5.8 presents QoS degradation and Fig.5.9 presents the speedup characteristics resulting from different combination of approximation levels in the ABs. Corresponding results for LULESH is in Fig.5.4 and Fig.5.5. Each point in the plots represents a distinct approximation setting, *i.e.*,, a different configuration of ABs. The X-axis is divided in segments showing the QoS degradation and speedup characteristics when approximations were applied only to that phase, letting all other phases run accurately. The last segment (marked as "*All*") shows the behavior when approximation was turned on for the entire duration of the application execution. For all the appli-



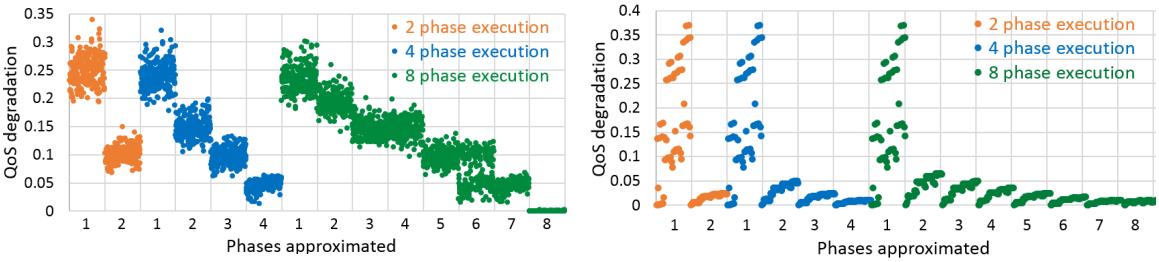
(a) CoMD (b) PSO (c) Bodytrack (d) FFmpeg (Y-axis is PSNR, higher is better)

Fig. 5.8.: Phase specific QoS degradation



(a) CoMD (in log scale) (b) PSO (c) Bodytrack (d) FFmpeg

Fig. 5.9.: Phase specific speedup



(a) Bodytrack (b) LULESH

Fig. 5.10.: Characteristics of QoS degradation for execution divided in to 2, 4, and 8 phases

cations except FFmpeg, Y-axis shows the percentage of degradation in QoS (lower is better). For FFmpeg, Y-axis is the value of PSNR (Peak Signal to Noise Ratio) and a higher value represents lower approximation error. The Y-axis in Fig.5.9a for CoMD speedup is in log scale.

Error Characterization

As can be observed, for all the applications we studied, approximation in the first phase introduces maximum approximation error resulting in significant QoS degradation. For LULESH and CoMD, error introduced in the first phase of the execution is so significant that its effect on QoS degradation is comparable to the execution where approximation is turned on for the entire duration. It can also be observed that as we turn on the approximation in the later execution phases, its impact on QoS degradation diminishes. Approximation during the 4th phase of the execution creates almost insignificant QoS degradation. This behavior can be explained from the intrinsic nature of the algorithms involved in these applications. We have already discussed this behavior in the context of LULESH in Sec. 5.2.

CoMD. Approximation during the initial phases puts the particles further from their accurate positions with vastly inaccurate values of kinetic and potential energies. Inaccurate positions and energy values create a ripple effect during the rest of the simulation, such that QoS never recovers. The magnitude of the inaccuracies and the scope for their propagation is reduced if OPPROX applies approximations only during the later phases.

PSO. PSO iteratively converges towards the best solution starting from a set of initial candidate solutions (particles). The quality of the solution set being explored in the current iteration depends on the accuracy of the solutions from the previous iterations.

Bodytrack. Bodytrack's QoS degradation is less affected if approximation is turned on at later phases.

FFmpeg. The outer loop iterates over the video frames applying multiple approximated filters on each frame. Although this application runs the same set of filters on each frame, the approximations in the first phase significantly reduce PSNR because the encoding procedure (which follows the filter execution) induces the dependency between the neighboring frames. For example, the second encoded frame only keeps

the information relative to the first frame. Therefore, any error introduced in the first few frames propagated throughout the remaining frames (out of 150 frames in total) leading to a phase-dependent PSNR degradation.

Performance Characterization

From Fig.5.9, we see that phase-specific behavior for speedup have two distinct patterns. Either the speedup drops if we trigger approximation in later phases (*e.g.,*, in Fig.5.5 for LULESH) or speedup remains almost unaffected with respect to which phase is being approximated (*e.g.,*, for CoMD, Bodytrack, FFmpeg in Fig.5.9a, Fig.5.9c,and Fig.5.9d, respectively). Thus, for applications that fall in the first category, it is most beneficial to approximate in the later phases, rather than uniformly, because the speedup remains the same while the QoS degradation is lower in the later phases. Therefore the application benefits from our phase-aware approximation technique, as opposed to prior works based on phase-agnostic approximation.

Changing Phase Granularity

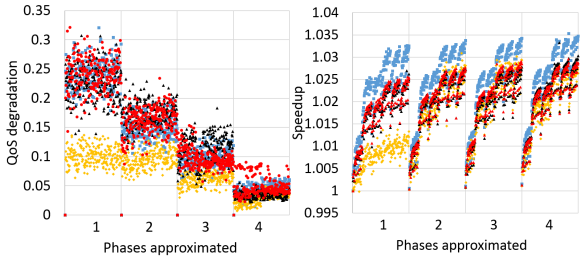
Fig.5.10 presents how changing the number of phases affects the QoS degradation for Bodytrack and LULESH. We uniformly divide the application's execution in to 2, 4 and 8 phases to meaningfully compare the increase of phase numbers. For both applications, when we divide execution into 2 phases, it is preferable to use aggressive approximation in phase-2 instead of phase-1 (especially when operating under low QoS degradation budget). The behavior is similar when we divide the execution in 4 phases and it provides more fine granularity for controlling QoS degradation. However, when we divide the execution in 8 phases, the distinction between the QoS degradation coming from different phases becomes blurry – e.g., in the cases of Bodytrack (phase-3 and phase-4 have almost the same QoS degradation) and LULESH (phases 5 to 8). This highlights that it is important for the performance of analysis to precisely (and automatically) identify phases.

Fig.5.11a and Fig.5.11b present how phase specific QoS degradation and speedup characteristic varies for four different input parameter combinations (described in Sec. 5.4) for Bodytrack and LULESH, when the execution is divided into four phases. Each point represents a particular approximation setting for that phase and the color denotes the corresponding input parameter combinations. For both the applications, for all the four input combinations, we see a consistent trend in the behavior of QoS degradation and speedup with respect to various phase-specific approximations. This validates that the benefits of phase-aware approximation is not tied to any particular input parameter combination.

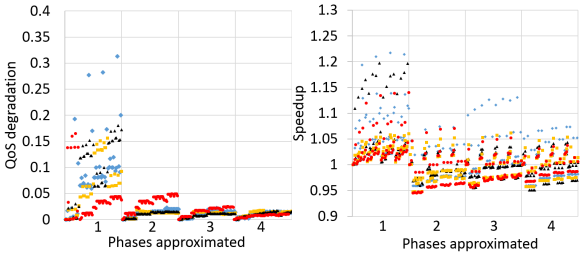
5.5.2 Evaluation of Optimization Framework

To show the effectiveness of our proposed *phase-aware* optimization technique, we compare OPPROX with a *phase-agnostic* optimization through *exhaustive search*. Such phase-agnostic exhaustive search was used previously [123, 139] as an idealized oracle technique. Thus, essentially we compare OPPROX with the *best* achievable result by the phase-agnostic optimization. Phase-agnostic exhaustive search goes over *all* combinations of approximation settings to find which setting provides maximum possible speedup while keeping the corresponding QoS degradation within the budget. However, such phase-agnostic search does not consider any phase-specific approximations and applies the chosen approximation setting through the entire execution. Fig.5.12 presents evaluation using 3 levels of the QoS degradation budget: *large-budget* (20% degradation), *medium-budget* (10% degradation) and *small-budget* (5% degradation). Since for FFmpeg, QoS is calculated in terms of PSNR where a higher value signifies less error, we use target PSNR values of 10, 20, and 30 as the large-budget, medium-budget, and small-budget respectively.

Small Error Budget (5%). Phase-specific approximation improves performance for all benchmarks, while the baseline (phase-agnostic) approximation is unable to obtain any speedup in 4 of the 5 applications. For example, in case of FFmpeg,



(a) Bodytrack



(b) LULESH

Fig. 5.11.: Phase specific characteristics of QoS degradation and speedup for different inputs. Each point represents an approximation setting. Points from different input combinations have different colors.

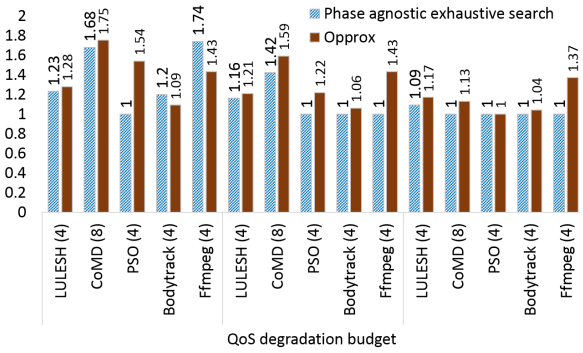


Fig. 5.12.: For different QoS budgets, comparison between OPPROX and phase-agnostic exhaustive search used by prior works [123, 139] as the idealized or oracle scheme.

OPPROX achieved 37% speedup while phase-agnostic search achieved nothing. as it could not find any approximation setting that would create lower than user-specified QoS degradation. On average, OPPROX gave 14% speedup while phase-agnostic search gave only 2%.

Medium Error Budget (10%). OPPROX improves performance for all benchmarks because OPPROX can perform search at a finer (phase) granularity, while phase-agnostic search was able to provide speedup only for LULESH and CoMD. Approximating CoMD during phase-3 and phase-4 creates insignificant QoS degradation (Fig.5.8a) but the share of speedup achieved is similar to phase-1 and phase-2 (Fig.5.9a). Thus, OPPROX can set a higher approximation level for phase-3 and phase-4 to increase speedup and choose a lower approximation level for phase-2 or phase-1 to keep the QoS degradation within budget.

Large Error Budget (20%). OPPROX can provide significant speedup (up to 75% for CoMD) for all the applications. However, for Bodytrack and FFmpeg, phase-agnostic search is able to find a better setting that gives higher speedup. For FFmpeg, the large budget is large enough to accommodate all possible approximation settings for the entire execution of the application. For Bodytrack, our model for QoS degradation computes a less precise prediction, which is a consequence of conservative confidence intervals that OPPROX computes around the predicted values.

These results jointly show that OPPROX can successfully use the concept of phase-specific approximation to fine-tune and control the error budget giving better speedup compared to phase-agnostic approximation method.

5.6 Related Work

We discuss related software-based approximations.

Software Systems for Approximation. Researchers have presented programming language support, including static analyses [119,146–149] and dynamic analyses [121, 150,151] that quantify the effects of approximation. Researchers also proposed various (phase-agnostic) compiler transformations [121, 123, 125, 132, 141].

PetaBricks autotuner [152, 153], automatically finds configurations of alternative function implementations for a given QoS budget. An extension in [154] identifies classes of similar inputs using two-level clustering and applies different approximations for each input class. These are complementary to our approach as OPPROX can

learn the control-flow of the input-optimized program generated by PetaBricks and then apply its phase-specific optimization.

Models for Input-Aware Optimization. In an early work, Rinard [117, 155] presented an approach that builds linear-regression models for various values of the accuracy knobs, but for individual inputs. More recently, Capri [139] constructs generalized models of performance and accuracy of the computation using M5 estimation algorithm [156]. The main difference between Capri and OPPROX is that they do not exploit the additional control coming from execution-phase-specific approximation levels. Furthermore, Carpi builds tree-based linear models for input features while OPPROX uses polynomial regression for individual execution phases.

Laurenzano et al. [138] derive and runs canary inputs (smaller versions of the full inputs) to determine the approximation level based on input content, while focusing on streaming and data-parallel applications. In contrast, OPPROX uses input-parameters to predict how control-flow variations impacts performance and accuracy. However, OPPROX can also benefit from using such canary inputs to more accurately model the phase-specific behaviors.

Runtime Systems and Middleware for Approximation: Existing adaptive techniques support on-line monitoring of accuracy [122] and latency [125,157]. Approximation-aware runtime systems have also been used to improve resilience [158, 159], guide the execution of data-parallel [120, 127, 132], and reduce communication cost in parallel programs [126]. While adaptive mechanisms track program execution and actions may get activated during specific program phases, these approaches do not use specialized phase-aware models or incur runtime overhead to dynamically build models. In contrast, OPPROX does not incur runtime overhead and instead predicts the phase-behavior of the program from input features.

5.7 Conclusion

We introduce phase-aware approximation for finding the best approximation settings for different phases of the execution. We present OPPROX, a system that

models speedup and QoS degradation corresponding to phase-specific approximation settings and maximize the speedup subject to an acceptable QoS degradation. OPPROX is compatible with many prior approximation techniques. Our evaluations show that, compared to oracle phase-agnostic baseline used by prior works, OPPROX can significantly improve performance, especially for tight QoS degradation budgets.

6. SUMMARY

I looked at various kinds of performance problems in large scale distributed and parallel programs as well as in some popular serial applications. These performance problems often create slowdown in the system causing wastage of valuable time and money. Through my research, I developed techniques to diagnose, detect, and mitigate such performance problems. My proposed performance problem diagnosis technique targets large-scale supercomputing applications. It finds the root-cause of the problem by comparing the logical progress of various parallel processes. It uses some sophisticated techniques to compare progress inside complex loop structures. I proposed a performance problem detection technique that first characterizes the prediction error of the underlying model and then can automatically calibrate the detection threshold based on the inputs received in the production. My proposed performance problem mitigation techniques target two kinds of applications: (1) erasure-coded distributed storage systems, and (2) error-resilient applications. Both of the techniques leverage our understanding of the underlying algorithm to improve performance. For erasure-coded distributed storage I proposed a distributed and parallel repair technique that can significantly reduce the repair time for the missing or corrupted data. For error-resilient applications, I proposed an application's execution phase-aware approximate computation technique that can greatly increase the speedup but effectively controls the resulting error introduced by inexact computation.

REFERENCES

REFERENCES

[1] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate application progress analysis for large-scale parallel debugging," in *PLDI*, 2014.

[2] "Debugging Tool based on Statistical Analysis," https://github.com/scalability-llnl/AutomaDeD.

[3] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi, "Dealing with the unknown: Resilience to prediction errors," in *PACT*, 2015.

[4] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage," in *EuroSys*, 2016.

[5] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware optimization in approximate computing," in *CGO*, 2017.

[6] "TotalView Debugger," http://www.roguewave.com/products/totalview.aspx.

[7] "DDT - Debugging tool for parallel computing," http://www.allinea.com/products/ddt/.

[8] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.

[9] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical debugging using latent topic models," in *ECML*, 2007.

[10] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*, 2002.

[11] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. d. Supinski, D. H. Ahn, and M. Schulz, "Automaded: Automata-based debugging for dissimilar parallel tasks," in *DSN*, 2010.

[12] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, "Large scale debugging of parallel tasks with automaded," in *SC*, 2011.

[13] The MPI Forum, "MPI: A Message Passing Interface," https://http://www.mpi-forum.org/, 1993.

[14] D. H. Ahn, B. R. d. Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *SC*, 2009.

[15] I. Laguna, D. H. Ahn, B. R. d. Supinski, S. Bagchi, and T. Gamblin, "Probabilistic diagnosis of performance faults in large-scale parallel applications," in *PACT*, 2012.

[16] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[17] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[18] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[19] U. Banerjee, "Loop transformations for restructuring compilers: The foundations," 1993.

[20] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, 1976.

[21] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, pp. 77–84, 1975.

[22] "PRODOMETER source code," https://computation-rnd.llnl.gov/automaded/.

[23] C.-K. Luk, R. Cohn, and et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[24] "Sequoia Benchmarks," https://asc.llnl.gov/sequoia/benchmarks/.

[25] "LULESH https://codesign.llnl.gov/lulesh.php."

[26] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, pp. 345–420, Dec. 1994.

[27] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[29] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *ICML*, 2006.

[30] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," in *POPL*, 2005.

[31] M. Jose and R. Majumdar, "Cause clue clauses:error localization using maximum satisability," in *PLDI*, 2011.

[32] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. d. Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for mpi programs," in *SC*, 2009.

[33] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose, "Assertion based parallel debugging," in *CCGRID*, 2011.

[34] A. Mirgorodskiy, N. Maruyama, and B. Miller, "Problem diagnosis in large-scale computing environments," in *SC*, 2006.

[35] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *SC*, 2007.

[36] W. Haque, "Concurrent deadlock detection in parallel programs," *International Journal of Computers and Applications*, pp. 19–25, 2006.

[37] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "Collective error detection for mpi collective operations," *Recent Advances in Parallel Virtual Machine and Message Passing Interface Lecture Notes in Computer Science*, pp. 138–147, 2005.

[38] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," in *ICS*, 1997.

[39] N. Ahmed, N. Mateev, and K. Pingali, "Tiling imperfectly-nested loop nests," in *SC*, 2000.

[40] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *CF*, 2007.

[41] G. Bronevetsky, I. Laguna, B. R. de Supinski, and S. Bagchi, "Automatic fault characterization via abnormality-enhanced classification," in *DSN*, 2012.

[42] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGPLAN Notices*, vol. 41, no. 11. ACM, 2006, pp. 185–194.

[43] "CoMD http://www.exmatex.org/comd.html."

[44] "FFmpeg https://www.ffmpeg.org/."

[45] "SpMV Benchmark http://bebop.cs.berkeley.edu/spmvbench/."

[46] "LINPACK Benchmark http://www.netlib.org/benchmark/hpl/."

[47] A. N. Langville and C. D. Meyer, "Deeper inside pagerank," *Internet Mathematics*, 2004.

[48] "PARSEC: Black-Scholes http://parsec.cs.princeton.edu/."

[49] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," *Science*, pp. 1518–1524, 2011.

[50] "FFmpeg and H.264 Encoding Guide," https://trac.ffmpeg.org/wiki/Encode/H.264.

[51] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[52] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *ICDCS*, 2009.

[53] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[54] J. D. Rodriguez, A. Perez, and J. A. Lozano, "Sensitivity analysis of k-fold cross validation in prediction error estimation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pp. 569–575, 2010.

[55] A. G. Wilson and R. P. Adams, "Gaussian process kernels for pattern discovery and extrapolation," in *ICML*, 2013.

[56] A. Wilson, E. Gilboa, J. P. Cunningham, and A. Nehorai, "Fast kernel learning for multidimensional pattern extrapolation," in *NIPS*, 2014.

[57] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *ACM SIGOPS Operating Systems Review*, 2007.

[58] W. Pfeiffer and N. J. Wright, "Modeling and predicting application performance on parallel computers using hpc challenge benchmarks," in *IPDPS*, 2008.

[59] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI*, 2012.

[60] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *PLDI*, 2012.

[61] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software." in *OSDI*, 2012.

[62] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *EuroSys*, 2010.

[63] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic problem localization via multi-dimensional metric profiling," in *SRDS*, 2013.

[64] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[65] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*, 2002.

[66] "Big Data and What it Means: http://www.uschamberfoundation.org/bhq/big-data-and-what-it-means."

[67] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST)*, 2010.

[68] "Ceph http://ceph.com/."

[69] "OpenStack Object Storage (Swift): http://swift.openstack.org."

[70] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, 2010.

[71] B. Calder, J. Wang, A. Ogus, N. Nilakantan *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *SOSP*, 2011.

[72] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth," in *FAST*, 2015.

[73] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *ATC*, 2012.

[74] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Transactions on Information Theory*, 2014.

[75] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *VLDB*, 2013.

[76] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *VLDB*, 2013.

[77] "Google Colossus File System: http://static.googleusercontent.com/media/research.google.com /en/university/relations/facultysummit2010/ storage_architecture_and_challenges.pdf."

[78] "Yahoo Cloud Object Store: http://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at."

[79] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," in *HotStorage*, 2013.

[80] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, "Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage," in *SYSTOR*, 2014.

[81] J. Li and B. Li, "Beehive: erasure codes for fixing multiple failures in distributed storage systems," in *HotStorage*, 2015.

[82] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," in *SIGCOMM*, 2014.

[83] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads." in *FAST*, 2012.

[84] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems." in *OSDI*, 2010.

[85] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields." *SIAM*, 1960.

[86] F. J. MacWilliams and N. J. A. Sloane, "The theory of error correcting codes," *North-Holland*, 1977.

[87] k. Björck and V. Pereyra, "Solution of vandermonde systems of equations," *Mathematics of Computation*, 1970.

[88] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *FAST*, 2015.

[89] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, 2005.

[90] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate application progress analysis for large-scale parallel debugging," in *PLDI*, 2014.

[91] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," 2008.

[92] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *SIGCOMM*, 2009.

[93] A. Akella, T. Benson, B. Chandrasekaran, C. Huang, B. Maggs, and D. Maltz, "A universal approach to data center network design," in *ICDCN*, 2015.

[94] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *IEEE International Symposium on Network Computing and Applications (NCA)*, 2006.

[95] "Quantcast File System http://quantcast.github.io/qfs/."

[96] "Erasure Coding Support inside HDFS: https://issues.apache.org/jira/browse/HDFS-7285."

[97] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *ACM/IEEE conference on Supercomputing*, 2006.

[98] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," Technical Report CS-08-627, University of Tennessee, Tech. Rep., 2008.

[99] "OpenStack: Open source software for creating private and public clouds: http://www.openstack.org/."

[100] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn *et al.*, "A performance evaluation and examination of open-source erasure coding libraries for storage." in *FAST*, 2009.

[101] X. Chu, C. Liu, K. Ouyang, L. S. Yung, H. Liu, and Y.-W. Leung, "Perasure: a parallel cauchy reed-solomon coding library for gpus," in *IEEE ICC*, 2015.

[102] H. M. Ji, "An optimized processor for fast reed-solomon encoding and decoding," in *IEEE ICASSP*, 2002.

[103] L. Atieno, J. Allen, D. Goeckel, and R. Tessier, "An adaptive reed-solomon errors-and-erasures decoder," in *ACM/SIGDA FPGA*, 2006.

[104] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *Peer-to-Peer Systems*. Springer, 2005.

[105] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Peer-to-Peer Systems*. Springer, 2002.

[106] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker, "Total recall: System support for automated availability management." in *NSDI*, 2004.

[107] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "F4: Facebooks warm blob storage system," in *OSDI*, 2014.

[108] K. S. Esmaili, L. Pamies-Juarez, and A. Datta, "Core: Cross-object redundancy for efficient data repair in storage systems," in *IEEE International Conference on Big Data*, 2013.

[109] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction," *IEEE Transactions on Information Theory*, 2011.

[110] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "Nccloud: applying network coding for the storage repair in a cloud-of-clouds." in *FAST*, 2012.

[111] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in rdp code storage systems," in *SIGMETRICS*, 2010.

[112] Z. Wang, A. G. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *GLOBECOM Workshop*, 2010.

[113] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, 2004.

[114] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems." in *NSDI*, 2006.

[115] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, "Oceanstore: An architecture for global-scale persistent storage," in *ASPLOS*, 2000.

[116] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The oceanstore prototype." in *FAST*, 2003.

[117] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *ICS*, 2006.

[118] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency," in *DAC*, 2010.

[119] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[120] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *MICRO*, 2013.

[121] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[122] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[123] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[124] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report UW-CSE-15-01*, 2015.

[125] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

[126] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix-up: Relaxing program semantics to unleash parallelization," in *CGO*, 2015.

[127] Í. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015.

[128] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[129] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," *ACM TOCS*, 2014.

[130] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: Synthesis of approximate sequential circuits," in *DATE*, 2014.

[131] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *ETS*, 2013.

[132] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[133] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "Incapprox: A data analytics system for incremental approximate computing," in *WWW*, 2016.

[134] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: an online quality management system for approximate computing," in *ISCA*, 2015.

[135] P. D. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. V. Palem, and T. Palmer, "On the use of inexact, pruned hardware in atmospheric modelling," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2018, 2014.

[136] Q. Zhang, Y. Tian, T. Wang, F. Yuan, and Q. Xu, "Approxeigen: An approximate computing technique for large-scale eigen-decomposition," in *ICCAD*, 2015.

[137] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "Approxit: An approximate computing framework for iterative methods," in *DAC*, 2014.

[138] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsive approximation: Using canary inputs to dynamically steer software approximation," in *PLDI*, 2016.

[139] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *ASPLOS*, 2016.

[140] M. Rogers, "Analytic solutions for the blast-wave problem with an atmosphere of varying density." *The Astrophysical Journal*, vol. 125, p. 478, 1957.

[141] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *FSE*, 2011.

[142] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi, "Dealing with the unknown: Resilience to prediction errors," in *PACT*, 2015.

[143] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," *Science*, pp. 1518–1524, 2011.

[144] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[145] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of machine learning*, 2011, pp. 760–766.

[146] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *POPL*, 2012.

[147] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, 2012.

[148] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[149] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[150] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *ISSTA*, 2010.

[151] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *ASPLOS*, 2015.

[152] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *PLDI*, 2009.

[153] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *CGO*, 2011.

[154] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. OReilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *PLDI*, 2015.

[155] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *OOPSLA*, 2007.

[156] J. R. Quinlan *et al.*, "Learning with continuous classes," in *Australian joint conference on artificial intelligence*, 1992, pp. 343–348.

[157] H. Hoffmann, "Jouleguard: energy guarantees for approximate applications," in *SOSP*, 2015.

[158] S. Achour and M. C. Rinard, "Approximate computation with outlier detection in topaz," in *OOPSLA*, 2015.

[159] V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos, "A programming model and runtime system for significance-aware energy-efficient computing," in *PPoPP*, 2015.

VITA

VITA

Subrata Mitra is advised by Prof. Saurabh Bagchi at the Purdue University, West Lafayette. His research is focused on performance problem detection, diagnosis and mitigation in distributed and parallel applications. He has published his works in various conferences such Programming Language Design and Implementation (PLDI), Parallel Architectures and Compilation Techniques (PACT), European Conference on Computer Systems (EuroSys), Symposium on Reliable and Distributed Systems (SRDS) etc. Subrata received his MS in Computer Engineering from University of Florida, Gainesville and BE in Electronics and Telecommunication Engineering from Jadavpur University, India.