PROGRAMMING SYSTEMS FOR SAFE AND ACCURATE PARALLEL PROGRAMS IN THE FACE OF UNCERTAINTY

BY

VIMUTH FERNANDO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Assistant Professor Sasa Misailovic, Chair
Professor Josep Torrellas
Professor Sayan Mitra
Professor Micheal Carbin, Massachusetts Institute of Technology

Abstract

Many emerging distributed applications operate on inherently noisy data or produce approximate results. Emerging application domains, including IoT, self-driving cars, and precision agriculture, routinely need to deal with noise from their sensors and unreliable communication mediums. Furthermore, increased volume of data, and the rise of highly parallel and often heterogeneous systems have brought forth new challenges in overcoming bottlenecks in both computation and communication between processing units. Many prominent systems adopted approximation in communication to address these challenges.

Developing software in the presence of these novel architectures, optimizations, and approximations can be a challenging task. As these systems get deployed in safety critical situations, it is important to verify that they behave in a predictable and safe manner, even in situations where the outcomes are uncertain. Developers need to ensure that the programs operating with noisy data do not result in unexpected crashes and produce acceptable results with high reliability. In recent years, researchers have designed several analyses for verifying these program properties in the presence of uncertainty. These prior works had stayed away from parallel programming models, in part due to the complexities involved with reasoning about parallel interactions.

This dissertation presents an ecosystem of several programming language tools and techniques across the computational stack that provides foundations for safety and accuracy analyses of parallel programs that deal with uncertainty. First, the dissertation will present a software infrastructure that enables simple and efficient use of a novel architecture that speeds up communication in a manycore processor using a wireless network. Next, the dissertation will show how to use programming language techniques to reduce the complexity of verifying the correctness of a subset of asynchronous message passing parallel programs in Parallely. We show how to lift many existing analyses that are designed for sequential programs to the domain of parallel programs. Next, the dissertation presents how to further extend verification to bigger programs and newer error models using runtime monitoring in Diamont. Finally, the dissertation presents several case studies that look at extending runtime verification to recovery mechanisms, algorithmic fairness analysis, and a novel architecture with potentially erroneous wireless communication.

To my family, with love.

Acknowledgments

I was fortunate to have the help and support of many people during my Ph.D. journey.

I thank my advisor Sasa Misailovic for his kindness, patience, and advice throughout these six years. Sasa always encouraged me and motivated me to believe in myself. In addition to helping me immensely with my research, he was always available and ready with a joke to cheer me up when I was feeling down. I will never forget his help while navigating some of my life's most challenging periods.

I would like to thank my committee, Professors Josep Torrellas, Sayan Mitra and Mike Carbin for their support and guidance. I also greatly appreciate all the help and advice from Professor Darko Marinov.

My excellent collaborators taught me a lot and helped me make progress in my work. Keyur was an invaluable part of many of my research projects. It was a pleasure to work with such an intelligent and helpful colleague. I also thank Antonio for letting me be a part of his fun and exciting research projects and for all his help. I thank everyone who worked with me and gave me feedback on my work. I learned a lot from them during these years.

Everyone in my research group was a pleasure to work with. I was very fortunate to somehow fall into this group of brilliant and friendly people. I look forward to seeing all the great things they will accomplish in the future. Special thanks to Saikat for being a good friend and for all his feedback on my work. I had a wonderful time working and traveling the country with him.

I would also like to thank all the people in the Computer Science Department for making my time here enjoyable. Everyone from the faculty, my fellow students, to the administrative staff has been great. I also thank all my friends in the area. They have always been there when I needed help. I enjoyed all of our get-togethers and fun activities.

I owe a debt of gratitude to the people of Sri Lanka, who funded a large portion of my education before I joined the Ph.D. program. I also thank all the funding agencies who supported my research during my Ph.D.

Finally, I am fortunate to have a family who supports me and cares for me. I would like to thank my mother, Lalani, who works harder than anyone else I know to take care of us, my father, Spelman, who has always been there for me and taught me a lot, and my brother, Hasith, who always encouraged me and pushed me forward.

Contents

Chapter 1: Introduction

Programs today have to deal with increasing levels of uncertainty in their execution. Uncertainty is inherent in many application domains, including big-data analytics, multimedia processing, probabilistic inference, and sensing [1, 2, 3, 4, 5, 6]. Noise can be introduced to programs operating in these domains from many different sources. Programs can operate on noisy inputs, they can be executing in environments that can cause data corruption, or they can be running on low-energy/unreliable hardware. Furthermore, programmers can intentionally use approximation techniques that reduce computational bottlenecks in program components at the cost of introducing some uncertainty in the result.

In the modern world, the increased volume of data and the emergence of novel heterogeneous processing systems dictate the need to focus our attention on the uncertainty in the results of parallel and distributed programs. With the growth of data volumes, programs today consist of more and more parallel processes that need to coordinate together to speedup computations. Furthermore, in recent years the number of cores integrated into processor chips has grown significantly, allowing programs to distribute workloads among more and more processes. The wide availability of networking and compute resources have driven the spread of domains such as Internet-of-Things (IoT) and precision agriculture. The Internet of Things provides the ability for billions of things including sensors, actuators, services, and other Internet-connected objects to communicate and interact with each other [7]. In precision agriculture, large numbers of sensors are used to monitor and control the health of plants in large fields [8, 9]. In all these domains, programs consist of many distributed components sharing data and synchronizing with each other in challenging environments.

In many of these settings, data communication and synchronization consumes a large amount of energy. For instance, in the MIT RAW microprocessor, the communication infrastructure consumes 36% of the overall system power [10], and in the Intel TeraFLOPS processor, communication costs up to 28% energy [11]. A recent study [12] predicts that the fraction of time spent in communication will consume the majority of runtime (over 50%) for many applications as the number of processes goes to thousands. Modern applications require higher high bandwidth, low latency and high throughput.

The same sources of uncertainty in sequential programs are present in parallel programs but are combined with new sources of noise and uncertainty. For example, communications among parallel components using messages can result in the following errors that inject uncertainty into program results:

- Message corruption: silent bit flips in data can change the content of the message. Such errors can occur due to noise in the environment and crosstalk [13, 14, 15], process

variations [16, 17], or from using low power communication links[18, 19, 20], among other reasons. Detecting and recovering from these silent errors is costly [21, 22]. To protect against data corruption, people use error correction codes or re-transmission. Both these techniques cost energy and sometimes require specialized hardware to be efficient.

- Message delays and drops: Unpredictability in message delivery can cause computations to stall or get stuck and affect the results of programs. The impact of such errors is high in time-sensitive applications.

- Order of execution: Different inter-leavings among parallel programming components can also lead to uncertain outcomes, especially in programs with race conditions. Detecting and fixing these errors has been a challenging research area for many years [23, 24, 25].

A lot of energy and resources are spent in safety-critical programs to detect communication errors and recover from them. Programs operating in challenging environments such as remote agricultural fields or urban centers are forced to use unreliable networks with such errors to share data [26]. Error in these networks can result in problems such as incoherence in data among different parallel processes and deadlocks. Identifying safe program components that can execute even in the presence of errors can reduce the cost of error detection and recovery mechanisms.

Furthermore, the increase in parallel program components increases the amount of data that needs to be communicated, resulting in communication bottlenecks that slow down many parallel programs. To handle these communication bottlenecks, program developers and hardware designers propose techniques across the computation stack that trade-off the accuracy of computations to reduce the communication load. Many systems contain such approximations. For instance, Hogwild! has significantly improved machine learning tasks by eschewing synchronization in stochastic gradient descent computation [27], TensorFlow can reduce precision of floating-point data by transferring only some bits [28], Hadoop can sample inputs to reductions [5], MapReduce can drop unresponsive tasks [29]. Researchers also proposed various other techniques for approximating parallel computations in software [30, 31, 32, 33, 34, 35, 36, 37, 38]. A recent survey [39] studied over 17 different communication-related transformations such as *compression* (e.g., reducing numerical precision), *selective communication skipping* (e.g., dropping tasks or messages), *value prediction* (e.g., memoization), and *relaxed synchronization* (e.g., removing locks from shared memory).

Hardware base approximation techniques have also been introduced that intentionally inject uncertainty into programs to increase performance. For example, approximate networks-

on-chips [18, 19, 40] use various techniques such as selective message dropping to reduce communication load. Furthermore, in some settings, uncertainty in communication is unavoidable.

Despite the wide variety of novel architectures/frameworks for parallel programs, and approximation techniques, they have been justified only empirically. As these systems get deployed in safety-critical situations, it is essential to verify that they behave in a predictable and safe manner even in situations where the outcomes are uncertain. Developers need to ensure that the programs operating with uncertain data do not result in unexpected crashes and produce acceptable results with high reliability. As programs combine techniques to reduce both *computation* and *communication* bottlenecks in programs, analyzing the impact of noise and uncertainty of the program results becomes challenging.

Therefore, providing foundations of safety and accuracy analyses for parallel programs that deal with uncertainty remains an intriguing and challenging research problem. Developing safe and accurate parallel and distributed programs in the presence of uncertainty from multiple possible sources require programming systems that can represent various sources of uncertainty, and efficient tools that simplify the process of verifying important safety and accuracy properties.

The goal of my dissertation is to answer the following question:

Can programming systems help us develop safe, accurate, and efficient parallel programs in the presence of uncertainty?

There are several important properties that developers want to check in parallel programs containing uncertainty

- Type Safety: Programs typically contain *critical* regions (which must execute without errors) and approximate regions (which can execute acceptably even in the presence of uncertainty). But, variables containing uncertainty should not affect the values of safety-critical variables (array indices, conditionals, etc). Variables that can handle uncertainty can be annotated (for example, `approx int x` can indicate that the variable `x` may have uncertain data). Type systems can then show that `approx` variables do not affect other data directly, via assignment, or indirectly, by affecting control flow [41].

- Quantitative Reliability: the probability with which the computation produces a correct result when its approximate regions are affected by uncertainty should be high [42]. For example, we can use a specification of the form $[0.99 \leq \mathcal{R}(\text{result})]$ to specify that the calculated value of a variable named `result` is the same as the *correct* value (from an execution without any noise or error) with at least 99% probability.

- Accuracy: In addition to high reliability, the program results should be close to the correct result, with error magnitudes being low [43]. To specify this type of specifications, we can extend the notation to the form $[0.99 \leq \mathcal{R}^*(0.01 \geq \mathcal{D}(\text{result}))]$ to specify that the calculated value of a variable named `result` is within ± 0.01 of the *correct* value with at least 99% probability.

- Deadlock freedom: A deadlock is any situation in which no parallel components of a program can proceed, possibly as the result of competition for resources. Deadlocks can occur due to bugs in the program (Ex: a developer adding a receive statement that waits for a message that is never sent), or from messages being unexpectedly dropped and delayed. Writing deadlock-free programs and verifying their correctness is challenging.

- Relative safety: If an approximate program fails to satisfy some assertion, then there exists a path in the original program that would also fail this assertion [44]. This type of analysis can be used to prove that transformations do not affect important safety properties in a program. For example, consider a program that has been verified to contain no divide by zero errors. To show approximations preserve this property, it is enough to show that the approximate transformation never assigns zero to the divisor.

In recent years, researchers have designed several static analyses for verifying these program properties in the presence of uncertainty, but only for sequential programs. Previous works include safety analyses, such as the EnerJ type system for *type safety* [41] and Relaxed RHL for relational safety [44], analysis of quantitative *reliability* in Rely [42], and *accuracy* analysis for programs running on unreliable cores in Chisel [43].

These prior works had stayed away from parallel programming models, in part due to the complexities involved with reasoning about arbitrary interleavings and execution changes due to transformations of the communication primitives.

This dissertation presents an ecosystem of several programming language tools and techniques that investigate the thesis statement across the computational stack (summarized in Figure 1.1). In our first section, I will present a hardware architecture containing a network with uncertainty in message delivery. I will discuss how to use empirical techniques to optimize the utilization of the unreliable communication network and develop safe applications with acceptable results. Next, I will discuss how to use programming language techniques to reduce the complexity of verifying the correctness of a useful subset of parallel programs in Parallely. I will present the technique we developed to lift many existing analyses designed for sequential programs to the domain of parallel programs. Next, the dissertation

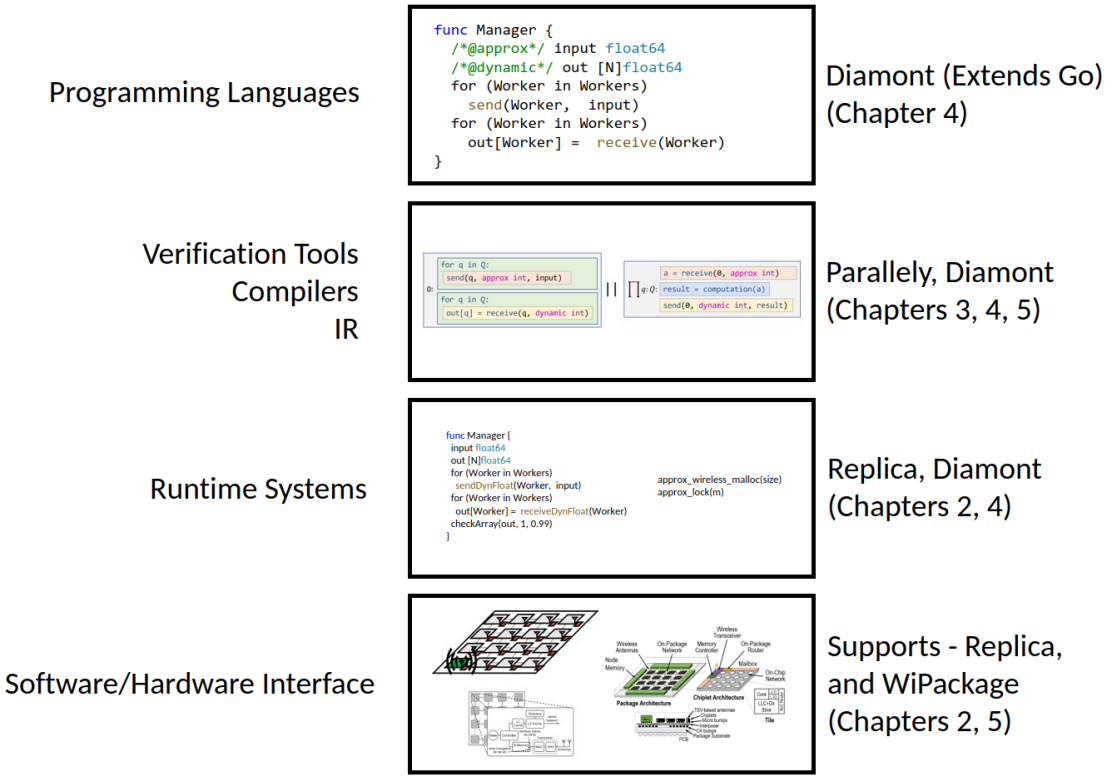


Figure 1.1: Overview of the dissertation

presents how to further extend verification to bigger programs and newer error models using runtime monitoring in Diamont. Finally, I will present several case studies that look at extending runtime verification to recovery mechanisms, algorithmic fairness analysis, and novel architectures. These works are further explained below:

Chapter 2 - Software Interface for Replica. Replica is a manycore that uses wireless communication for communication-intensive data. Data access patterns that involve fine-grained sharing, multi-casts, or reductions have proved to be hard to scale in shared- memory platforms. Recently, wireless on-chip communication has been proposed as a solution to this problem. Wireless communication provides low-latency, and is broadcast-friendly compared to a conventional on-chip network.

But, using wireless communication for communication-intensive data faces challenges due to bounded memory resources and the limited bandwidth of the wireless communication channel. To deliver high performance, Replica supports an adaptive wireless protocol and selective message dropping. Replica provides hardware support for selectively dropping packets if they carry certain types of data and if the sender encounters a certain level of channel contention.

In this dissertation, I will describe the computational patterns that can leverage wireless communication, programming techniques to restructure applications to fully utilize the potential of wireless communication, and a detailed evaluation of the proposed architecture. Our results show that wireless communication is effective for ordinary data. For 64 cores, Replica obtains a mean speed-up of 1.76x over a conventional machine. The mean speed-up reaches 1.89x if approximate-computing transformations are enabled.

Chapter 3 - Parallely Parallely is a programming language and a system for verifying approximations in parallel message-passing programs. Despite a wide variety of parallel approximations, they have been justified only empirically. Researchers designed several static analyses for verifying program approximation, but only for sequential programs. Parallely is a step towards extending these analyses for a subset of parallel programs.

Parallely's language can express various software and hardware level approximations. To support safety and quantitative accuracy analyses, Parallely presents an approximation-aware version of canonical sequentialization. Canonical sequentialization is a recently proposed verification technique that generates sequential programs that capture the semantics of well-structured parallel programs. We show that the sequential programs that are generated using sequentialization can be used with existing analysis techniques to develop safe and accurate parallel programs.

We demonstrate the effectiveness of Parallely on eight benchmark applications from the domains of graph analytics, image processing, and numerical analysis. Our results show that Parallely is both effective and efficient: it verifies type safety and reliability/accuracy of all kernels in under a second, and all programs within 3 minutes (on average, in 47.3 seconds).

Chapter 4 - Diamont Diamont is a system for *dynamic* monitoring of uncertainty properties in distributed programs. Verifying accuracy specifications at runtime can provide increased precision compared to static analysis techniques. This increase in precision comes at the cost of runtime performance overheads. Therefore, such runtime verification systems need to be optimized to reduce overheads. However, developers who try to manually implement these runtime systems and optimizations that span multiple processes can easily make subtle errors.

Diamont provides tools to efficiently verify important properties at runtime with minimal developer effort. Diamont provides a simple front-end that extends the Go programming language. Diamont programs are then converted to the IR of Parallely for static analysis. Diamont includes data types that monitor uncertainty in data at runtime, and provides support for checking uncertainty specification at runtime and reacting to excessive uncertainty

safely. We prove the soundness of the Diamont runtime and optimizations. Soundness of a Diamont program means that if the execution passes a variable uncertainty check, then the uncertainty of the variable is within the bound specified in the check statement.

We implemented Diamont for a subset of the Go language and evaluated 8 programs from precision agriculture, graph analytics, and media processing. We show that Diamont can prove important end-to-end properties on program outputs for significantly larger inputs compared to prior work, with modest execution time overhead – on average 3% and a maximum of 16.3%.

Chapter 5 - Diamont Case Studies. We present several case studies. First, we look at techniques to check for and recover from excessive uncertainty in program data. Next, we look at how to use Diamonts constructs to encode algorithmic fairness properties. Finally, we present how to develop a runtime verification system for a novel architecture *WiPackage* that extends Replica. We discuss how to extend Diamont to a new front-end for MPI programs and show how programs can be verified using Diamont.

Chapter 2: Replica: A Wireless Manycore for Communication-Intensive and Approximate Data

2.1 INTRODUCTION

Data access patterns where multiple threads interleave reads and writes to the same set of variables in a fine-grained manner and without much per-thread locality do not scale well in shared-memory multiprocessors. They create many network messages, inducing communication bottlenecks. To alleviate this problem, commercial vendors (e.g., [45, 46, 47, 48]) and researchers (e.g., [49, 50, 51, 52, 53, 54, 55, 56]) have proposed various hardware techniques. They include new synchronization and cache coherence protocol improvements, special networks, and new communication technologies such as optics and transmission lines.

Recently, on-chip wireless communication has emerged as a promising alternative that supports fine-grained data sharing with low-latency, and is broadcast-friendly [57, 58, 59, 60]. In this environment, broadcasting a short message of 80 bits takes about 4 ns, which is about two orders of magnitude lower than in conventional on-chip networks. For example, the recent WiSync manycore [57] augments each core with a small antenna and a transceiver. It supports low-latency implementations of synchronization primitives, such as locks and barriers. WiSync stores the state of synchronization variables in a small, per-core Broadcast Memory (BMem) that has identical contents in all of the cores. Writes to the BMem are broadcasted, updating all the BMems at the same time, while reads are satisfied from the local BMem.

While WiSync shows the attractiveness of on-chip wireless communication, it is only tailored to speed-up synchronization operations. An intriguing question is whether the wireless communication and BMem support can be used to speed-up transfers of ordinary data.

Using wireless communication for ordinary data faces two fundamental challenges: the bounded size of BMem and the limited bandwidth of the wireless communication channel. WiSync does not completely experience these challenges, as the synchronization variables typically fit in the 16KB BMem and do not consume much of the wireless channel bandwidth. In contrast, ordinary data does not fit in BMem, and its frequent updates may cause contention in the wireless channel. It is therefore necessary to judiciously select the subset of the data that will benefit the most from the wireless communication, and place it in BMem.

In this chapter we present *Replica*, a manycore architecture and software interface that enables efficient use of wireless communication for ordinary data. We tailor Replica to speed-up *communication-intensive shared data* – whose accesses typically induce substantial overheads

in standard cache hierarchies. Our analysis presents several common communication-intensive patterns. They include broadcasts, regular many-to-many interactions, irregular many-to-many interactions, and reductions. To handle these patterns, we present: (i) a software API that exposes BMem to the software developer, and (ii) transformations and tools for selecting communication-intensive data and restructuring applications for improved BMem and wireless channel use. Replica also provides hardware support for selectively dropping packets if they carry certain types of data and if the sender encounters a certain level of channel contention. A software developer can use two operations, *approximate locks* and *approximate stores*, to optimize applications that can tolerate noise. Further, they can combine these operations with existing approximation techniques that trade accuracy for reduced communication and/or data size. Together, these techniques have a greater impact on Replica than on standard architectures, due to the limited BMem size and the limited wireless channel bandwidth.

Our results show that Replica effectively uses wireless communication for ordinary data. We evaluated Replica with 10 applications from graph analytics, vision, and numerical simulation. For 64-core executions, Replica speeds-up the applications over a conventional machine by a geometric mean of 1.76x for exact computation and 1.89x for approximate computation. Further, Replica substantially reduces the average energy consumption by 34% (or 38% with approximate computation). Finally, the area increase is modest.

2.2 BACKGROUND

2.2.1 WiSync

WiSync [57] augments every core of a manycore with a *Broadcast Memory* (BMem), a wireless transceiver, and two antennas (of which we will only consider one). The transceiver has two main modules, namely the physical layer (PHY) and the Medium Access Control (MAC). The PHY module serializes and modulates the data to transmit, detects collisions, and demodulates and deserializes data at reception. The MAC module manages the access to the channel by scheduling transmissions and handling collisions [61].

The BMem is a direct-mapped memory of a size similar to an L1 cache. The BMems of all the cores contain the exact same variables that are kept coherent through wireless updates. A core accesses its BMem with plain loads and stores. Based on the physical address of the location accessed, a load or store request is sent either to the L1-L2 hierarchy or to the BMem.

When a core writes to a BMem location, it generates a message to be broadcasted through

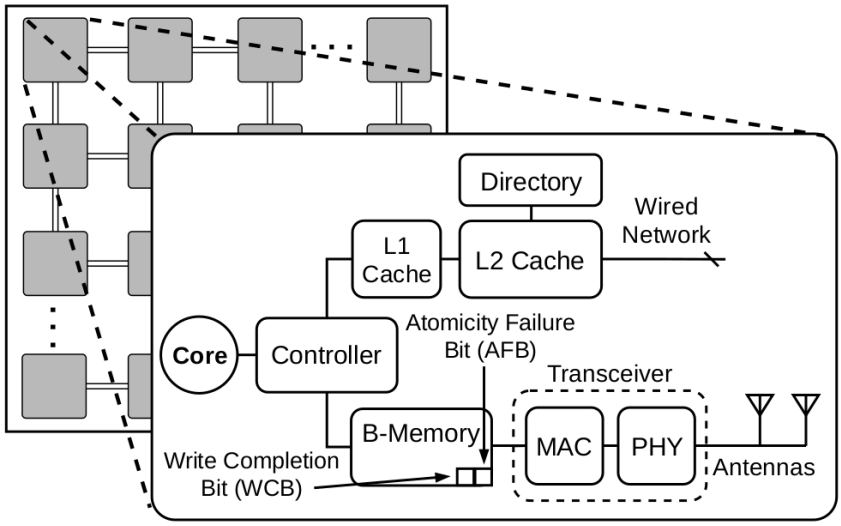


Figure 2.1: Replica manycore.

the wireless network. All BMems (including the local one) are updated simultaneously. This design ensures a total order of writes to BMems across all cores. It also ensures that, at all times, all cores have the same values in their BMems. Loads that access the BMem read the local copy of the data.

WiSync uses 5 cycles to transmit a 77-bit packet, which corresponds to a 64-bit write. In the second cycle, the transceiver listens if there was a collision with another packet in the first cycle. If there was no collision, in the next three cycles it sends the rest of the packet with guaranteed no collision. Otherwise, the transfer is aborted, and the senders will retry sending their packets after a randomized, exponentially-increasing number of cycles. This carrier-sensing protocol with exponential backoff adapted to the on-chip scenario is called Broadcast Reliability Sensing (BRS) [62].

2.2.2 Replica Architectural Extensions

Figure 2.1 shows the Replica architecture. Replica extends the WiSync architecture in several ways, including the ability to store ordinary (i.e., non-synchronization) *and* synchronization data in the BMem. Similar to WiSync, Replica contains two different networks: the regular wired network that provides high latency and low throughput, and the wireless network that provides low latency and high throughput. The software interface for the architecture hides the complexity of choosing the underlying communication networks using the memory location of data. By changing the data allocation site, developers can change the network being used for communication. Developers can request the use of the wireless network by allocating data in the *BMem*.

Broadcast Memory. Replica provides an API to allocate data in BMem. To store an array `a` in BMem and use the wireless network for communicating updates to the array, the developer only needs to change the allocation site to `float* a = wireless_malloc(n*sizeof(float));`

All accesses to the array elements are automatically directed to the BMem, and writes use the wireless channel. Programs do not require any additional developer or compiler interventions, as the BMem is memory mapped. A call to `wireless_free` deallocates the memory and makes the BMem locations available.

Since the amount of communication-intensive data may exceed the size of the BMem, it is essential to restructure communication-intensive data structures to fit as much as possible in BMem. We present transformations that allow the flexible storage of a fraction of communication-intensive data in BMem. Our approach rests on two observations: (i) in many applications, the size of communication-intensive data increases at a much slower rate than the full input data size, and (ii) since BMem is memory-mapped, we can transform the data structure layout with little performance penalty.

Adaptive Wireless Protocol. In Replica, the wireless network utilization varies across applications and even within an application. For applications with sparse transmissions, the carrier-sensing protocol from WiSync is sufficient. However, applications with high or bursty load perform better with a token-passing protocol, in which only the node that owns the token can transmit. Replica's MAC module supports both protocols, and automatically switches between the two to adapt to the characteristics of the application.

Approximate Broadcast Memory. To further reduce the wireless channel contention, Replica uses a specially designed section of BMem for approximate data. In this section, the messages for data updates and locking operations may occasionally be dropped, if the latency to perform the access exceeds a certain threshold. Approximate BMem supports two operations that selectively drop packets:

- **Approximate Store:** It assigns a value `val` to a variable `var` if the write succeeds within a specified latency threshold [1]. Approximate stores can be *unchecked* or *checked*. In the former, if the message is dropped, the computation silently continues without informing the software. In the latter, software can use the call `approx_stac(var, val)` (for store approximate checked) to find out if the write succeeded. Replica uses a register to store the outcome of the packet transmission. If the packet was dropped, the register is updated and the software interface generates opcode to read the register value and respond

[1]Replica keeps track of the time a packet spends in the send buffer. When the time in the buffer exceeds a specified threshold, the packet is removed from the buffer. As the local data is updated only if the packet is successfully sent, all copies of the data remains consistent

accordingly. Unchecked stores use the same opcode as standard stores. Checked stores use a different opcode.

- **Approximate Lock:** `approx_lock(m)` attempts to obtain the lock `m` within a specified latency threshold. If it succeeds, it returns a success code. If it does not succeed, either because it spins for too long on an already taken lock, or because it takes too long to obtain the wireless network to send the lock acquire update, it returns a failure code. In this case, the software skips the critical section and the unlock operation.

  More details about the Replica architecture are available in [63].

2.3 SOFTWARE ADAPTATION

In this section, we describe the software infrastructure that we developed to leverage the Replica architecture. We start by describing the key access patterns that we target, and then discuss our transformations.

2.3.1 Communication-Intensive Access Patterns

There are several parallel access patterns that are hard to support in conventional shared-memory multiprocessors. They involve multiple (or all) cores reading from and writing to a particular shared address. They cause communication bottlenecks in current machines. Fortunately, the wireless channel of Replica is especially suited to support *broadcast*, *many-to-many* and *reduction* communication patterns efficiently.

```
while (!converge ( total_cost )) {
  if (thread_id == MASTER) shuffle(feasible);
  BARRIER_WAIT(barrier);
  for (int i=0; i < numfeasible; i++)
    global_cost += local_cost( feasible [ i ], thread_id);
}
```

Figure 2.2: An example of a broadcast communication pattern

Broadcast. One thread (possibly referred to as the master) writes to a shared address that is subsequently read by many (or all) of the other threads (referred as the workers). An example of such computation can be found in Streamcluster (PARSEC [64]). As shown in Figure 2.2, the main loop uses fine-grained parallel section to calculate which cluster centers

to open. At the beginning of each iteration, the master thread reorders the set of clustering centers (array `feasible`):

Regular Many-to-Many Interactions. It occurs in codes where different threads operate on sets of overlapping shared addresses, and the communication has regular patterns. Common examples are simulations and numerical applications.

We illustrate this communication pattern with the computation from Water (SPLASH [65]) in Figure 2.3. This program calculates forces applied on water molecules. Each step of the parallel program calculates the pairwise forces between two molecules and updates their state. In the next time step, each thread reads the forces of the molecules computed in the previous step and calculates new forces. The neighbors remain fixed, therefore the communication is regular:

```
for (mol1 = start_id; mol1 < end_id; mol1++){
  for (mol2 = mol + 1; mol2 < mol+MAX_MOL/2; mol2++){
    f = local_force_calc (molecules, mol1, mol2);
    LOCK(locks[mol1]); molecules[mol1].forces += f;
    UNLOCK(locks[mol1]);
    LOCK(locks[mol2]); molecules[mol2].forces += f;
    UNLOCK(locks[mol2]);
  }
}
```

Figure 2.3: An example of a regular many-to-many interaction

Irregular Many-to-Many Interactions. This pattern is like the previous one except that the inter-thread communication follows irregular patterns. A common example is graph-processing algorithms.

Graphs can have arbitrary shapes. Thus, graph algorithms typically have irregular memory access patterns. For example, Pagerank (CRONO [66]) computes a set of weights that quantify connectedness of the vertices in a graph using the code presented in Figure 2.4. The benchmark stores the ranks for all vertices in the shared `pagerank` array. For each vertex `v`, the array `inedge_cnt[v]` contains the number of incoming edges, and `neighbors[v][j]` is the vertex with an incoming edge. The computation calculates the local page ranks (`loc_pr`) of the vertices. The shared array is accessed between the barriers. Each node accesses a different number of neighbors and this can change by iteration, therefore the communication is irregular.

```
while (!stop) {
  for(Node v=startid; v<stopid; v++) {
    loc_pr[v] = pagerank[v];
    for(int j=0; j<inedge_cnt[v]; j++)
    loc_pr[v] += pagerank[neighbors[j][v]]*const;
  }
  BARRIER_WAIT(barrier);
  for(v=i_start;v<i_stop;v++) pagerank[v] = loc_pr[v];
  BARRIER_WAIT(barrier);
}
```

Figure 2.4: An example of an irregular many-to-many interaction

Reduction. Many (or all) threads read and write to a single shared address, aggregating their local contributions. Such a computation pattern exists the Single-Source Shortest-Path benchmark as shown in Figure 2.5. In each step, the computation updates the path weights for the nodes connected by the outgoing edges. The contributions to each `Weight[j]` may be aggregated from different cores.

```
for(v=startid; v<stopid; v++) {
  for(int i = 0; i < outedge_cnt[v]; i++) {
    neighbor = neighbors[v][i];
    if( /* distance check */ ) {
      LOCK(locks[neighbor]);
      weight[neighbor] += local_update(v, neighbors[v]);
      UNLOCK(lock[neighbor])
    }
} }
BARRIER(barrier);
```

Figure 2.5: An example of a reduce computation

All these access patterns are easily supported using an address in the BMem. A write by a processor automatically broadcasts the update to all BMems. Since reads are always to the local BMem and writes are observed by all processors quickly, regular and irregular many-to-many interactions are supported trivially. Reductions simply require that processors read and write atomically to the single shared address.

2.3.2 Transformations to Optimize BMem Utilization

We present several program transformations that enable the BMem to store the most important data, or to store a larger amount of important data.

Data Splitting. This transformation partitions a data structure into important data, which is allocated in BMem, and less important data, which is allocated in regular memory. This allows Replica to deliver high performance even for large data structures that do not completely fit in BMem.

We describe two variants of the transformation. The first variant sets up an ***indirect data structure*** and then partitions the original structure into two parts. For example, consider an array of records as shown Figure 2.6. This transformation creates an indirection array with as many pointers as the records, where each pointer points to a record. The latency-critical records are allocated in BMem, while the less important ones in regular memory. All accesses to the original array are then replaced with indirect references. The indirection array (which after the initialization remains read-only) is allocated in the regular memory. In the following code block the `parr` array is used for the indirection.

```
T** parr = (T**) malloc(Size*sizeof(T*));
T* warr = (T*) wireless_malloc(WSize*sizeof(T));
T* narr = (T*) malloc((Size−WSize)*sizeof(T));

for (i=0; i<WSize; i++) parr[i]=&warr[i];
for (i=WSize; i<Size; i++) parr[i]=&narr[i−WSize];
```

Figure 2.6: An example of splitting data using the indirection array

The part of the array in the wireless memory, `warr`, has a size `WSize`, while the remaining part (`narr`) is stored in the regular memory. Each access of the original array, such as `x=arr[i]` is replaced with the indirect reference to `parr`, e.g., `x=(*parr)[i]`. The approach directly extends to multidimensional arrays.

The second variant involves ***mapping some of the pages*** of the data structure in the BMem, and mapping the rest in regular memory. For example, we can map the first set of pages of the structure into BMem, or the last set of pages, or an arbitrary set of pages. Compared to the first variant, this approach does not add additional indirections or cause cache evictions. However, it is less flexible, as the grain size of allocation is a page.

Data Reduction. These transformations, inspired by other ones from literature, enable BMem to store data more efficiently. As a result, the BMem logically stores a larger amount of important data, potentially reducing program accuracy.

- *Lock Coarsening:* reduces the number of locks needed to access a given data structure, by making multiple elements of the structure share the same lock [67]. This change reduces the data in the BMem (since only a subset of locks is required) and the inter-core communication, but at the expense of false contention. For instance, let the original computation have a single lock for each element as shown in Figure 2.7.

```
Mutex * locks = malloc( NumElem * sizeof(Mutex))
for (int i = 0; i<NumElem; i++) {
  mutex_lock(locks[i])
  data[i] = // ...
  mutex_unlock(locks[i])
}
```

Figure 2.7: An example of a data access protected by a lock

The transformed computation in Figure 2.8 reduces the number of locks, for instance by a factor `K`.

```
Mutex * locks = wireless_malloc( NumElem/K * sizeof(Mutex))
for (int i = 0; i<NumElem; i++) {
  mutex_lock(locks[i/K])
  data[i] = // ...
  mutex_unlock(locks[i/K])
}
```

Figure 2.8: Using lock coarsening

- *Cyclic Collection Update:* sets an upper bound on the memory footprint of a collection, such as a list or a set. If we need to add a new element to the collection that would require an increase in the collection footprint, the new element is dropped or it replaces an existing element. This transformation is inspired by cyclic memory allocation from program repair [68]. For instance consider a program that allocates `NumElem` items and update their value as shown in Figure 2.9.

```
Mutex * locks = malloc( NumElem * sizeof(Mutex))
for (int i = 0; i<NumElem; i++) {
  data[i] = // ...
}
```

Figure 2.9: An example of updating a large list

If the wireless memory is limited to `NumElemLimit`, we can change the pattern to replace an existing element in the array for the extra data as shown in Figure 2.10.

```
Mutex * locks = wireless_malloc( NumElemLimit * sizeof(Mutex))
for (int i = 0; i<NumElem; i++) {
  data[i  % NumElemLimit] = // ...
}
```

Figure 2.10: Using the cyclic collection update transformation

- ***Numerical Precision Reduction*** changes the type of the variables stored in the BMem, reducing the size of the variables at the expense of precision. For example, we can change 64-bit `double` types to 32-bit `float` types.

2.3.3 Transformations to Reduce Communication

Some of these transformations leverage Replica's approximate locks and stores to reduce communication in the wireless channel.

Skipping Negligible Updates. This transformation skips updates to a shared variable when the contribution of the update to the value is below a specified threshold. In the following example in Figure 2.11, the original code adds the variable `upd` to the variable `shared`.

```
upd = local_res();
lock(m);
shared += upd;
unlock(m);
```

Figure 2.11: An example of a data update that results in a message

In the transformed code below (Figure 2.12), if `upd` is smaller than `Threshold`, the update is skipped.

```
upd = local_res();
if (upd > Threshold){
   lock(m);
   shared += upd;
   unlock(m);
}
```

Figure 2.12: Program transformed to skip negligible updates

This transformation reduces the wireless communication if `shared` is allocated in BMem. It is applicable when small updates do not contribute much to the overall solution. However, it changes the computation and its result.

Skipping Critical Sections. We use Replica's approximate locks to occasionally skip critical sections as shown in Figure 2.13:

```
if (approx_lock(m)==0) { // acquired lock
  // execute critical section
  unlock(m);
}  // else skip
```

Figure 2.13: Using approximate locks to skip updates

In the example, the code tries to acquire the lock. When using an approximate lock, if the software unsuccessfully spins for more than a certain number of attempts, or the write packet in a read-modify-write instruction is queued for a certain number of cycles, `approx_lock` returns a non-zero code. In this case, the code skips the critical section. This transformation reduces the communication between cores. It is motivated by a software-only transformation from [37].

Skipping Updates with Compensation. This transformation skips updates but later tries to compensate for the contribution of the missed updates. For instance, in the example shown in Figure 2.14, variable `var` should receive the sum of all the elements of array `val`. Since the stores use `approx_stac`, they may be dropped. However, if `approx_stac` returns a non-zero status because the contribution of `var[i]` is dropped, subsequent iterations will attempt to add multiple times their contribution to compensate. Finally, if the loop completed without adding the final element(s), they will be aggregated after. In the code, a local variable `fcnt` counts the number of consecutive failed attempts.

```
int fcnt = 0; // failcount
for (i=0;i<MAX;i++){
  if (approx_stac(var, var+val[i]*(1+fcnt))) fcnt++;
  else fcnt = 0;
}
if (fcnt > 0) do {
   lastf = approx_stac(var, var+val[MAX-1]*fcnt);
} while (lastf);
```

Figure 2.14: Program transformed to skip negligible updates with compensation

This transformation reduces communication in the wireless channel. It relies on the fact that, in many programs, the consecutive updates have similar values. A similar transformation can also be applied to approximate locks.

2.3.4 Tool Support

To ease program adaptation, we implemented tools that help the developer identify shared data and tune transformations.

Profiler. We developed a memory profiler that detects the shared data in a program. The profiler instruments the memory instructions of the program to record a trace of the memory activity. It then identifies shared variables that are written to by a thread before being read by multiple other threads. It then sorts data structures based on the percentage of addresses that exhibit such patterns, and based on the number of threads that access such addresses. Finally, it presents the report to the developer.

Automated Transformations. We implement the compiler transformations discussed in the previous sections within Clang/LLVM. For instance, for the data splitting transformations in Section 2.3.2, the developer only needs to write a pragma in the code, and the compiler then generates the code with the structures that best fit in the provided BMem.

Autotuner. The Replica architecture and the program transformations expose parameters that can be tuned to optimize performance. An example of such parameters is the frequency of dropped messages. To explore the space of parameter values and find those that maximize performance subject to accuracy specifications, we develop an autotuner. The autotuner uses the OpenTuner framework [69].

Table 2.1: Summary of the applications.

Name	Description	Input
Water [65]	Simulation of water molecules	1000 molecules, 10 steps
BFS [66]	Breadth-first search	p2p-gnutella31 [70]
SSSP [66]	Single source shortest path	p2p-gnutella31 [70]
Pagerank [66]	Compute pagerank for nodes in a graph	p2p-gnutella31 [70]
CC [66]	Compute connected components of a graph	p2p-gnutella31 [70]
Bodytrack [64]	Track a body pose through images	4 frames, 1000 models
Streamcluster [64]	Cluster streams of points	4096 pts, 20 centers
Volrend [65]	Render a 3D object	head
Community [66]	Compute modularity of a graph	p2p-gnutella31 [70]
Canneal [64]	Find optimal routing for gates on a chip	10000 elements

Table 2.2: Summary of the transformations for different configurations.

Name	Shared Vars (Beyond Synchronization)	Optimization (O, WO)
Water	`molecules`, `gl_memory`	Data splitting
BFS	`D`	Data splitting
SSSP	`D`	Data splitting
Pagerank	`PageRank`	Data splitting
CC	`D`	Data splitting
Bodytrack	`mParticles, mWeights, valid`	Command line knob
Streamcluster	`feasible`, `work_mem`, `clusterCenters`	Command line knob
Volrend	`shading_table`, `out_image`	Data splitting
Community	`comm`	Data splitting
Canneal	`Array of locks`	Lock coarsening

2.4 METHODOLOGY

To evaluate Replica, we perform cycle-level architectural simulations using Multi2sim [72]. We run a variety of applications from SPLASH-2 [65], PARSEC [64], and the CRONO [66] graph suite.

2.4.1 Applications

Table 2.1 lists the 10 applications, what they do, and the inputs we use in the evaluation.

Data Sharing Patterns. The benchmark applications have different data-sharing patterns. Water has broadcast communication. The graph applications (BFS, Pagerank, SSSP, CC, and Community) have irregular, mostly many-to-many communication. Volrend mainly contains communication between neighbors, but also has broadcast communication. Canneal has an irregular communication pattern, due to locks. Bodytrack and Streamcluster have one-to-many communications and reductions.

Table 2.3: Summary of the application approximation parameters.

Name	Approximation (A, WA)	Accuracy Metric
Water	Precision reduction and skipping critical sections with compensation in function `INTERF`	Difference in average energies
BFS	Approximate stores with T_{drop}=75 cycles	Fraction of unvisited nodes
SSSP	Approximate stores with T_{drop}=40 cycles	Fraction of nodes with different distances
Pagerank	Skipping negligible updates with Threshold=0.01	Average difference in pagerank
CC	Approximate stores with T_{drop}=350 cycles	Fraction of nodes with wrong component
Bodytrack	Approximate stores with T_{drop}=750 cycles	Average relative difference of poses
Streamcluster	Cyclic collection update in function `copycenters`	B^3 clustering metric [71]
Volrend	Approximate stores with T_{drop}=1000 cycles	Peak Signal to Noise Ratio (PSNR)
Community	Approximate stores with T_{drop}=2500 cycles	Average difference in calculated value
Canneal	Skipping critical sections in function `swap_locations`	Relative difference in routing length

Inputs and Metrics. For the SPLASH-2 and PARSEC applications (except Streamcluster), we use the same input sets as WiSync. For the graph applications, we use input sets from SNAP [70]. The input set sizes were chosen to allow detailed simulation runs that ranged between 4 and 48 hours per run. For the autotuning and profiling runs, we use different, training inputs. These training inputs are as follows. For the graph applications, they are different graphs of the same size and connectivity. For Streamcluster, we generate three new data sets with existing ground truth cluster centers [73]. For the other applications, we use alternative input data sets provided by the application suite. The last column of Table 2.1 shows the metrics that we use We use metrics that have been previously proposed in the literature to compute the accuracy loss of the computations when we use approximation optimizations.

Other Programs. We also analyzed other applications from the SPLASH-2 and PARSEC suites. As noted in previous characterizations [74], most of the remaining programs are data-parallel (e.g., blackscholes and swaptions) or implement regular algorithms with limited sharing, typically among neighbors (e.g., fluidanimate and raytrace). Since we do not expect Replica to improve performance for such computational patterns, we do not evaluate such applications.

2.4.2 Architecture Configurations

We analyze three configurations of Replica:

- Wireless-Locks (**WL**): it allocates only synchronization variables in BMem. It extends WiSync with the adaptive wireless protocol, and a BMem size that holds all the synchronization variables.
- Wireless-Optimized (**WO**): it extends WL by allocating some ordinary data in the BMem

Table 2.4: Architecture modeled. RT means round trip.

General Parameters	
Architecture	Manycore with 32–64 cores at 22nm technology
Core	Out of order, 2-issue wide, 1GHz, x86 ISA
ROB; ld/st queue	64 entries; 20 entries
L1 I+D caches	Private 32KB WB, 2-way, 2-cycle RT, 64B lines
L2 cache	Shared with per-core 512KB WB banks
L2 bank	8-way, 6-cycle RT (local), 64B lines
Cache coherence	MOESI directory based
On-chip network	2D-mesh, 4 (default), 2 or 1 cycles/hop, 128-bit links
Off-chip memory	Connected to 4 mem controllers, 110-cycle RT
Replica Parameters	
Per-core BMem	Up to 512KB, in 32KB chunks (Table 2.6) 6-cycle RT, 64-bit wide line
Wireless channel	20Gb/s; 1 cycle for collision detection
MAC Protocols	BRS (exponential backoff), token passing in ring
MAC Thresholds	$T_{BRS} = 0.4$, $T_{token} = 15$
T_{drop}	40–2500 cycles (Table 2.2)
Transceiv+Anten	Area: 0.4mm^2; TX/RX/idle: 39.4/39.4/26.9mW
Power gating	Analog amplif. (transient: 1.14 pJ), unused BMem

and applying the Data Splitting and Lock Coarsening transformations (Section 2.3.2). These transformations preserve the program semantics.

- Wireless-Approximate (**WA**): it extends WO by applying approximation transformations, including Cyclic Collection Update and Numerical Precision Reduction (Section 2.3.2), the transformations from Section 2.3.3, and approximate stores.

We compare these configurations to a conventional architecture without BMem or wireless network in three configurations: Baseline (**B**) runs the original application, Optimized (**O**) augments B with the transformations in WO, and Approximate (**A**) augments O with the transformations in WA except those that need hardware support (e.g., approximate stores).

Table 2.2 shows the transformations for each application. The shared variables column lists the non-synchronization variables stored in the BMem in Replica. The Optimization column presents the semantics-preserving transformations in WO and in O. Table 2.3 presents the approximation transformations in WA and, if applicable, in A.

Tuning Approximation Parameters. The Approximation column shows different values of T_{drop} and Threshold (for skipping negligible updates). To select these values for an application, we used the autotuner and executed the application multiple times on a set of different inputs. Our goal was to find the minimum T_{drop} and the maximum Threshold such

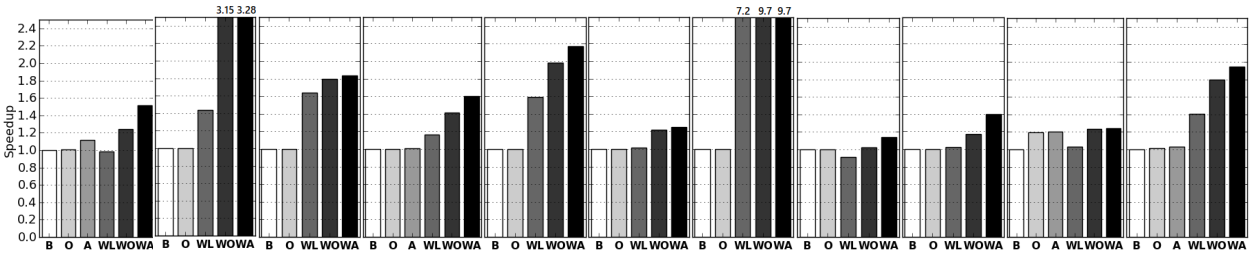


Figure 2.15: Speedups of the different configurations over Baseline (B) for 64 cores.

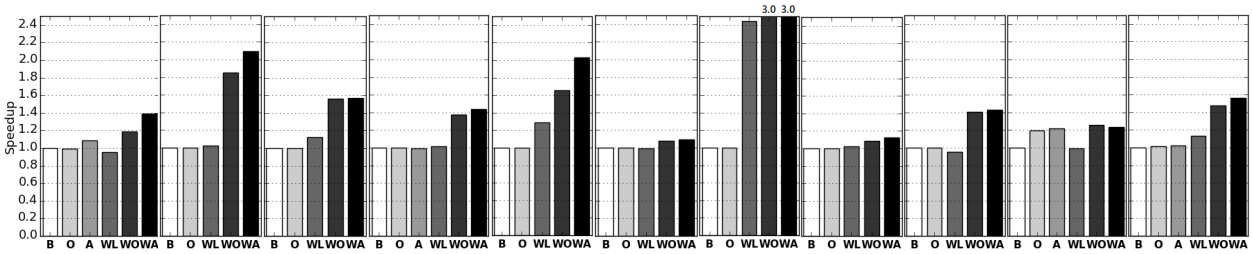


Figure 2.16: Speedups of the different configurations over Baseline (B) for 32 cores.

that the accuracy of the result was acceptable. We present the details in Section 2.5.3.

2.4.3 Simulator Implementation

We use cycle-level execution-driven simulations using the Multi2sim [72] simulator. We model a manycore with 32–64 cores at 22nm technology running at 1GHz. Each tile has a 2-issue out-of-order core, 32KB of private L1 instruction and data caches, and a 512KB bank of shared L2. The NoC is a 2D mesh. The per-core BMem is as large as an L2 bank, but we power-gate unused 32KB chunks as directed by the application. We will present the used fraction of BMem in the next section. The wireless network has a data rate of 20 Gb/s, enough to transmit a BMem line and its address (about 80 bits) in 4 cycles (plus one cycle for collision detection). We do not consider missing packets due to noise, since the error rate is below 10^{-16}. We augment Multi2sim with an on-chip wireless network that accurately models transmissions, collision handling, transceiver power-gating, and packet dropping.

2.5 EVALUATION

We discuss three main questions about the benefits of Replica:

- **RQ1.** Does placing ordinary data in broadcast memory improve performance of the applications? (Section 2.5.1)
- **RQ2.** Do approximate program configurations produce outputs of acceptable accuracy? (Section 2.5.2)
- **RQ3.** How sensitive is the application accuracy to various approximation parameters? (Section 2.5.3)
- **RQ4.** How difficult is it to concert applications to use the architecture? (Section 2.5.4)
- **RQ5.** How sensitive are the performance improvements to architectural parameters? (Section 2.5.5)

2.5.1 Analysis of Performance

Figures 2.15 and 2.16 present the speedup of the different architecture configurations over Baseline (B) for 64 and 32 core architectures, respectively. The X-axis of the plots lists the configurations: **B**aseline, **O**ptimized, **A**pproximate (when applicable), **W**ireless-**L**ocks, **W**ireless-**O**ptimized, and **W**ireless-**A**pproximate. The Y-axis is the speedup, computed as the ratio of the execution times of the B configuration and the other configuration.

From Baseline (B) to Baseline Replica (WL). The difference between these two bars is the effect of using wireless communication for synchronization variables and the support for the adaptive wireless protocol. As the figure shows, the average speedup of WL is 1.40x for 64 cores and 1.13x for 32 cores. For the applications in common with the WiSync paper, the numbers are largely similar, except for Streamcluster, which uses a different input set.

From Baseline Replica (WL) to Optimized Replica (WO). WO improves performance over WL for all the applications. On average, these improvements translate into an average speedup of 1.27x for 64 cores and 1.30x speedup for 32 cores. This shows the benefits of wireless transfers of optimized ordinary data. BFS and Streamcluster are communication heavy and thus benefit the most from these transformations. Most of the other applications have large improvements as well. Even Volrend, the application with the smallest gains, still manages to obtain speedups of about 10%.

From Optimized Replica (WO) to Approximate Optimized Replica (WA). Allowing approximations further increases the speedups in six applications, while in four applications there is practically no change. The average speedup of WA over WO is 1.08x for 64 cores and 1.06x for 32 cores (but can go up to 1.27x for CC on 32 cores).

Summary of speedups. Overall, the speedup of the exact version of Replica (WO) over

Table 2.5: Output accuracy.

Benchmark	A	WA
Water	0.083	0.0004
BFS	-	0.0002
SSSP	-	0.046
Pagerank	0.024	0.024
CC	-	0.0007
Bodytrack	-	0.099
Streamcluster	-	1.00
Volrend	-	37.2 dB
Community	-	0.07
Canneal	0.0001	0.0004

the optimized baseline (O) is 1.76x. If we add the approximations, the speedup of the approximate Replica (WA) over the approximate baseline (A) is 1.89x.

2.5.2 Analysis of Accuracy

Table 2.5 presents the accuracy losses of the A and WA executions analyzed in Section 2.5.1 for 64 cores. The accuracy losses are based on the accuracy metrics from past literature. Configuration A does not exist in some applications because the optimizations applied are not supported in conventional architectures (e.g., the approximate stores) or are not useful (e.g., the cyclic collection updates). It can be shown from past literature [35, 75, 76] that the levels of accuracy loss presented are considered acceptable.

Six applications use approximate stores. On average, 4% of all stores were dropped in our applications. The graph benchmarks implement iterative algorithms where each iteration improves on the results. Approximate stores may cause skipping an update to a node's value. However, the computation for that node will be redone in a future iteration, reducing the final error. In Volrend, approximate stores cause a small effect on the PSNR of the output image. Our inspection shows that only 0.8% of the pixels differ by 10% or more from the Baseline (7% of the pixels differ by more than 5%). In Bodytrack, approximate stores cause the model calculations to be done on stale data. Because Bodytrack aggregates a large number of models, errors in a few models have a small impact on accuracy.

The approximate version of Pagerank skips updates to the shared state if the value is below a given threshold. With a threshold value of 0.01, the approximate version produces the same top 10 and top 100 elements. Water is a simulation that can typically handle the small loss of precision from double to float conversion. While skipping updates can cause errors to amplify across time steps, using the compensation significantly reduces this effect. In Canneal, lock

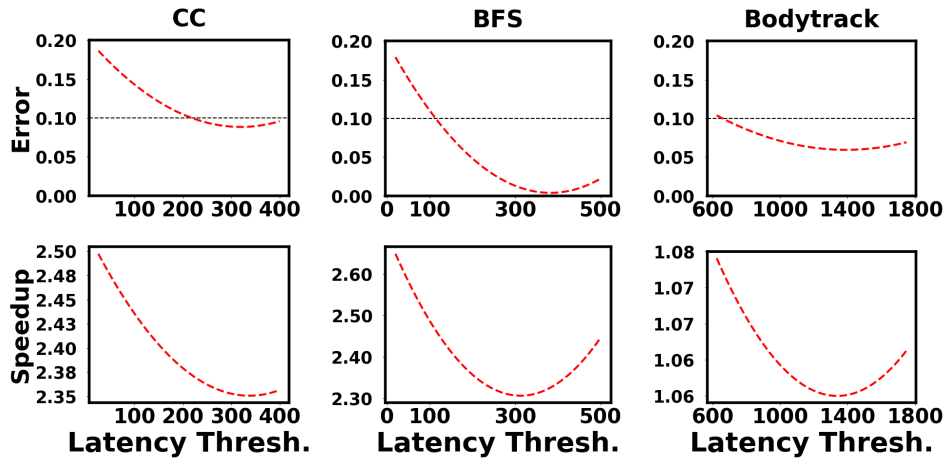


Figure 2.17: Autotuning latency thresholds (T_{drop}).

coarsening and skipping critical sections cause minimal changes in the generated netlist. In Streamcluster, the approximation overwrites cluster centers if the allocated list of centers is already full. For the provided input, all intermediate centers fit into the list without the approximation. Section 2.5.3 shows the impact on accuracy for larger inputs.

2.5.3 Profiles of Tunable Approximations

We study the relationship between the accuracy of the computation, the approximation parameters, and different inputs.

Approximate Stores. Since the latency threshold for dropping packets in approximate stores (T_{drop}) needs to be specified by the software, we used our autotuner to identify good T_{drop} values. Our goal is to attain a given level of accuracy (i.e., more than 40 dB for Volrend and less than 10% error for other applications).

Figure 2.17 shows the results of autotuning experiments for CC, BFS, and Bodytrack. The figure shows how the error and speedup change with T_{drop} values. Typically, as the autotuner reduces T_{drop}, the application accuracy changes a little, until the point where the error rate dramatically increases. The remaining applications exhibit similar behavior. Using these models, the autotuner selected the T_{drop} values that we used with the benchmarks.

Streamcluster. The number of cluster centers controls the size of the memory allocated in the BMem. The Streamcluster input provided by PARSEC consists of uniformly-distributed centers, and is not suitable for accuracy analysis. We therefore used alternative inputs (Section 2.4.1). Figure 2.18 presents the accuracy as a function of the size of the data structure that contains the intermediate cluster centers. Each line is generated by a different

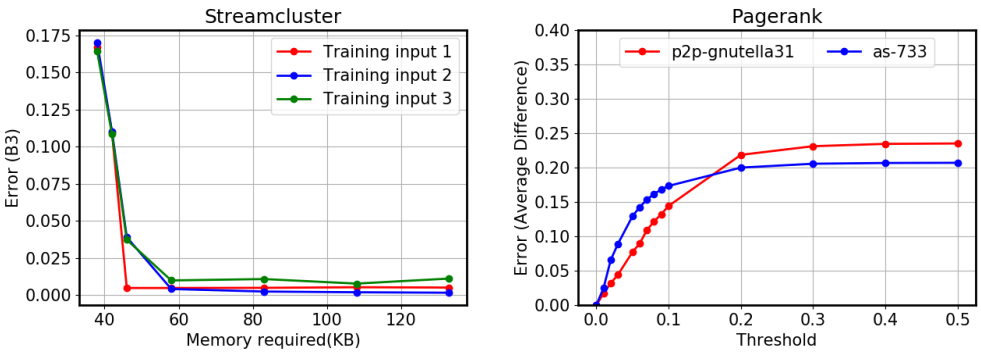


Figure 2.18: Tunable accuracy profiles.

training input. At 60 KB, the number of intermediate centers is around 200, which is more than enough to contain all the actual cluster centers. We see that, for 60 KB or higher, the error is negligible.

Pagerank. In Pagerank, we conditionally skip updates if they are below a certain threshold. We analyze the impact of using different thresholds on accuracy for two inputs. Figure 2.18 presents the results for inputs p2p-gnutella31 and as-733 (from [70]). For both inputs, when the threshold values are small (i.e., fewer updates are dropped), the error is low. As the threshold increases, more messages are dropped and the error increases.

2.5.4 Adapting Applications to Replica

Table 2.6 shows how we adapt applications for Replica. It shows the lines of code in the program (Column 2), the number of lines affected by Replica's transformations (Column 3), the size of the data we place in BMem (Column 4), the fraction of application's data in BMem (Column 5), and the number of data structures allocated in BMem vs. the number of structures that the profiler identified as shared among all threads, including synchronization data structures (Column 6).

The results show that the changes to the code are typically small. Moreover, the fraction of the application's data that is placed in BMem is typically very small – only Water and SSSP are exceptions. Also, the size of such data is typically only 100–300KB. In each application, we power-up as many 32KB chunks of BMem as needed to hold this data.

Profiler. In all applications except Bodytrack, the profiler identified all the data structures shared by all the threads. This includes synchronization data structures, such as barriers. We allocated these in the BMem. In Bodytrack, the profiler could not instrument the C++

Table 2.6: Statistics on how programs are adapted for Replica.

Name	LOC	Affected Lines	Data in BMem	% Data in BMem	Allocated vs. Profiled
Water	1641	10	352 KB	26.0%	2 vs. 2
BFS	475	10	245 KB	0.0%	2 vs. 2
SSSP	351	30	245 KB	23.2%	2 vs. 2
Pagerank	375	20	66 KB	0.8%	2 vs. 2
CC	557	10	245 KB	1.4%	2 vs. 2
Bodytrack	8672	24	121 KB	8.7%	n/a
Streamcluster	1660	8	137 KB	16.4%	3 vs. 4
Volrend	2604	4	147 KB	0.6%	3 vs. 3
Community	580	15	245 KB	0.0%	2 vs. 2
Canneal	2886	50	39 KB	0.1%	2 vs. 2

Table 2.7: Speedups for different cycles per hop (C/H) in the wired network.

	64 cores			32 cores		
Speedup Metric	C/H=4	C/H=2	C/H=1	C/H=4	C/H=2	C/H=1
A/WA	1.89	1.51	1.41	1.52	1.37	1.32
O/WO	1.76	1.39	1.31	1.45	1.29	1.23
WL/WO	1.27	1.12	1.12	1.30	1.17	1.17
WO/WA	1.08	1.09	1.08	1.06	1.06	1.07

`std::vector` allocator.

Data Scaling. We also studied how the size of the data that we want to place in BMem scales with input data size. For the graph applications, such data consists of nodes with many neighbors. We studied 20 graphs with 100K–3M nodes from the popular SNAP dataset of graphs [70]. In 16 of these graphs, all nodes with high sharing (at least 8 neighbors) do fit inside the BMem for our applications. Even some graphs of size 10M nodes will fit, if we limit the storage in BMem to nodes with at least 32 neighbors.

For the other applications, the size of the data that we want to place in BMem scales as follows. For Water, it scales linearly with the number of molecules, but independently of the number of steps; for Bodytrack, with the number of models used, but independently of the size or number of frames; for Volrend, with the size of the image, but independently of the number of rendering steps; for Canneal, with the number of locks, but independently of the number of circuit gates; and for Streamcluster, with the number of intermediate centers, but independently of the total number of data points.

2.5.5 Sensitivity to Architectural Parameters

Latency of the Wired Network. Our default wired NoC has a latency of 4 cycles per hop (Table 2.4). In this section, we re-evaluate Replica with wired NoCs that have a latency of 2 or 1 cycles per hop. Table 2.7 shows the resulting values of various speedups for different cycles per hop and different core counts. Each number is the geometric mean of all the applications. The table shows that, as the wired network becomes faster, the Replica speedups (A/WA, O/WO, and WL/WO) decrease. However, even for the fastest, 1-cycle per hop NoC, the speedups are considerable. The speedups due to approximations (WO/WA) remain unchanged.

Bigger L2 Cache. We have increased the size of the L2s of the Baseline (B) architecture from 512KB to 1MB per core, to use the same storage as a worst-case Replica – although Replica only uses a fraction of its BMem (Table 2.6). We find that this change only speeds-up Baseline by 1.04x for 64 cores.

2.5.6 Related Work

Wireless Architectures. We described WiSync [57] in Section 2.2. Duraisamy et al. [58] accelerate graph analytics using an NoC augmented with wireless links to better support irregular communication patterns. In their case, the application is oblivious of the underlying architecture, and the routing mechanism of each node decides whether to use the wireless links or the regular wire lines, based on the destination address. Their work is also different from ours in that the wireless links are only used to unicast packets between distant cores, irrespective of their criticality, and just as a way to shorten the propagation time of the packets through the network. Later, Duraisamy et al. [59] propose to accelerate graph analytics by bypassing certain updates. Their approximation is exclusively software-based and reduces both the computation and the volume of data lookups, specific to a particular community detection graph algorithm. In contrast, Replica presents hardware-supported, general approximate store and approximate lock mechanisms, which we applied across multiple application domains.

Scratchpads. While both BMem and scratchpads [77] have a finite size, BMems are automatically coherent. They do not rely on the compiler to keep them coherent, which is a major reason for the difficulty of using scratchpads. In Replica, the programmer and/or compiler just allocates the data in BMem and Replica transparently handles coherence in hardware.

Lossy NoCs. Prior work has proposed to apply lossy compression techniques to messages before sending them to the network [19]. The approximation occurs in the (wired) network interface, but could be potentially applied to wireless too. Although bufferless networks [78, 79] drop or deflect packets to undesired paths when there is contention at the switches, they are not approximate, since delivery is ensured through retransmissions.

Approximate Parallelization. Relaxed synchronization optimizations intentionally give up some synchronization for faster execution (e.g., [1, 31, 33, 35, 75, 80, 81, 82, 83, 84, 85, 86]). The previous works mainly show the potential of many computations to successfully continue execution with relaxed synchronization and random errors on commodity hardware. This chapter presents an approximate BMem architectural abstraction that is specialized for packet dropping. We show the efficiency of our hardware and software co-design and develop a toolchain to automate program adaptation.

2.6 CONCLUSION

Data access patterns that involve fine-grained sharing, multicasts, or reductions have proved to be hard to scale in shared-memory platforms. Recently, wireless on-chip communication has been proposed as a solution to this problem, but a previous architecture has used it only to speed-up synchronization. An intriguing question is whether wireless communication can be widely effective for ordinary shared data.

This chapter presented *Replica*, a manycore that uses low latency wireless communication for communication-intensive ordinary data. To deliver high performance, Replica supports an adaptive wireless protocol and selective message dropping. We described the computational patterns that leverage wireless communication, programming techniques to restructure applications, and tools that help with automation.

Our results showed that Replica effectively uses wireless communication for ordinary data. For 64-core executions, Replica sped-up applications over a conventional machine by a geometric mean of 1.76x for exact computation and 1.89x for approximate computation. Further, Replica substantially reduced the average energy consumption by 34% (or 38% with approximate computation).

In this chapter we presented how to use empirical techniques to develop safe programs in the presence of uncertainty. Our results show that careful co-design of hardware and software can help programs to be safe and accurate. Our evaluation also shows the need for new programming language support to ensure safety and accuracy of parallel applications that use these various approximations.

Chapter 3: Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization

3.1 INTRODUCTION

Approximation is inherent in many application domains, including machine learning, big-data analytics, multimedia processing, probabilistic inference, and sensing [1, 2, 3, 4, 5, 6]. The increased volume of data and emergence of heterogeneous processing systems dictate the need to trade accuracy to reduce both *computation and communication bottlenecks*. For instance, Hogwild! has significantly improved machine learning tasks by eschewing synchronization in stochastic gradient descent computation [27], TensorFlow can reduce precision of floating-point data by transferring only some bits [28], Hadoop can sample inputs to reductions [5], MapReduce can drop unresponsive tasks [29]. Researchers also proposed various techniques for approximating parallel computations in software [30, 31, 32, 33, 34, 35, 36, 37, 38], and networks-on-chips [18, 19, 40]. A recent survey [39] studied over 17 different communication-related transformations such as *compression* (e.g., reducing numerical precision), *selective communication skipping* (e.g., dropping tasks or messages), *value prediction* (e.g., memoization), and *relaxed synchronization* (e.g. removing locks from shared memory).

Despite a wide variety of parallel approximations, they have been justified only empirically. Researchers designed several static analyses for verifying program approximation, but only for sequential programs. Previous works include safety analyses, such as the EnerJ type system for type safety [41] and Relaxed RHL for relational safety [44], analysis of quantitative reliability (the probability that the approximate computation produces *the same* result as the original) in Rely [42], and *accuracy* (the frequency and magnitude of error) analysis for programs running on unreliable cores in Chisel [43]. These prior works had stayed away from parallel programming models, in part due to the complexities involved with reasoning about arbitrary interleavings and execution changes due to transformations of the communication primitives. Providing foundations of safety and accuracy analyses for parallel programs is therefore an intriguing and challenging research problem.

Parallely Language. We present Parallely, the first approach for rigorous reasoning about the safety and accuracy of approximate parallel programs. Parallely's language supports programs consisting of distributed processes that communicate via asynchronous message-

passing. Parallely is a high level intermediate language for modeling approximations. In Chapter 4 we will show how to target this language from a general programming language front end. Each process communicates with the others using strongly-typed *communication channels* through the common `send` and `receive` communication primitives. We identify three basic statements that are key building blocks for a variety of approximation mechanisms and include them in Parallely:

- **Conditional Send/Receive:** The `cond-send` primitive sends data only if its boolean argument is set to true. Otherwise, it informs the matching `cond-receive` primitive to stop waiting. It can be used to implement selective communication skipping transformations.
- **Probabilistic Choice:** The probabilistic choice statement $x = e_{orig}\ [p]\ e_{approx}$ will evaluate the expression from the original program (e_{orig}) with probability p, or otherwise the approximate expression (e_{approx}). It can be used to implement transformations for selectively skipping communication or computation, and value prediction.
- **Precision Conversion:** The conversion statement $x = (t')\ y$ allows reducing the precision of data that has primitive numeric types (e.g., double to float). It can be used to implement approximate data compression.

We used these statements to represent five approximations from literature. We studied three software-level transformations that trade accuracy for performance: precision reduction, memoizing results of maps, and sampling inputs of reductions. We also studied two approximations that resume the execution after run-time errors: dropping the contribution of failed processes running on unreliable hardware, and ignoring corrupted messages transferred over noisy channels.

Verification. Our verification approach starts from the observation that many approximate programs implement well-structured parallelization patterns. For instance, [30] present a taxonomy of parallel patterns amenable to software-level approximation including map, partition, reduce, scan, scatter-gather, and stencil. We show that all these parallel patterns satisfy the *symmetric nondeterminism* property – i.e., each receive statement must only have a unique matching send statement, or a set of symmetric matching send statements.

Approximation-Aware Canonical Sequentialization. Our safety and accuracy analyses rest on the recently proposed approach for *canonical sequentialization* of parallel programs [87]. This approach statically verifies concurrency properties of asynchronous message passing programs (with simple send and receive primitives) by exploiting the symmetric nondeterminism of well-structured parallel programs. It generates a simple sequential program

that over-approximates the semantics of the original parallel program. Such a *sequentialized program* exists if the original parallel program is deadlock-free. Moreover, a safety property proved on the sequentialized program also holds on the parallel program.

We present a novel version of canonical sequentialization that supports approximation statements. We prove that the safety and deadlock-freeness of the sequentialized program implies safety of the parallel approximate program. We then use this result to support the type and reliability analyses.

Approximation-Aware Canonical Sequentialization. Our safety and accuracy analyses rest on the recently proposed approach for *canonical sequentialization* of parallel programs [87]. This approach statically verifies concurrency properties of asynchronous message passing programs (with simple send and receive primitives) by exploiting the symmetric nondeterminism of well-structured parallel programs. It generates a simple sequential program that over-approximates the semantics of the original parallel program. Such a *sequentialized program* exists if the original parallel program is deadlock-free. Moreover, a safety property proved on the sequentialized program also holds on the parallel program.

We present a novel version of canonical sequentialization that supports approximation statements. We prove that the safety and deadlock-freeness of the sequentialized program implies safety of the parallel approximate program. We then use this result to support the type and reliability analyses.

Type System and Relative Safety. We propose typed *approximate channels* for (1) communicating approximate data (computed by the processes) or (2) representing unreliable communication mediums [18, 19, 40]. Every variable in the program can be classified as `approx` or `precise`. The design of our type system is inspired by EnerJ [41], as we enforce that approximate data does not interfere with precise data. For instance, approximate data can be sent only through an approximate channel and received by the `receive` primitive expecting an approximate value. We prove the *type system safety* and *non-interference* properties between the approximate and precise data. The type checker operates in two steps: it first checks that each process is locally well-typed, and then checks for the agreement between the corresponding sends and receives by leveraging canonical sequentialization.

We also studied *relative safety*, another important safety property for approximate programs. It states that if an approximate program fails to satisfy some assertion, then there exists a path in the original program that would also fail this assertion [88]. We show that if a developer can prove relative safety of the sequentialized approximate program with respect to the sequentialized exact program, this proof will also be valid for the parallel programs.

Reliability and Accuracy Analysis. Rely [42] is a probabilistic analysis that verifies that a computation running on unreliable hardware produces a correct result with high probability. Its specifications are of the form $r \leq \mathcal{R}(\texttt{result})$ and mean that the exact and approximate results are the same with high probability (greater than the constant r). It has two approximation choices: arithmetic instructions that can fail and approximate memories. Chisel [43] extends Rely to support joint frequency and magnitude specifications. For error magnitude, it computes the absolute error intervals of variables at the end of the execution of a program with approximate operations. It adds specifications of the form $d \geq \mathcal{D}(\texttt{result})$ that mean that the deviation of the approximate result from the exact result should be at most the constant d.

We extend the reliability analysis to support the more general probabilistic choice, allowing Rely to reason about general software-level transformations in addition to the previously supported approximate hardware instructions (such as unreliable add or multiply). We prove that verifying the reliability of the sequentialized program implies the reliability of the original parallel program, given some technical conditions on the structure of the parallel programs. We do the same with Chisel's error-magnitude analysis.

Contributions. The chapter makes the following contributions:

- **Language.** Parallely is a strongly-typed message-passing asynchronous language with the statements that allow implementing various program approximations.
- **Verification of Parallel Approximations.** Parallely is the first approach for verifying the safety and accuracy of approximate parallel programs using a novel approximation-aware canonical sequentialization technique.
- **Safety Analysis.** We present type analysis for parallel approximate programs and give conditions for relative safety of parallel programs.
- **Reliability and Accuracy Analysis.** We present reliability and error magnitude analyses that leverage the approximation-aware canonical sequentialization.
- **Evaluation.** We evaluate Parallely on eight kernels and eight real-world computations. These programs implement five well-known parallel communication patterns and we apply five approximations. We show that Parallely is both effective and efficient: it verifies type safety and reliability/accuracy of all kernels in under a second and all programs within 3 minutes (on average in 47.3 seconds).

3.2 EXAMPLE

Figure 3.2 presents an implementation of an image scaling algorithm. A Parallely program consists of *processes*, which execute in parallel and communicate over *typed channels*.

The program divides the image to be scaled into horizontal slices. The master process, denoted as α, sends the image to the worker processes. The left side of Figure 3.2 presents the code for α. We denote each worker process as β, and the set of worker processes as $Q = \{\beta_1, \beta_2, \ldots\}$. The right side of Figure 3.2 presents the code for β. The notation $\Pi.\beta : Q$ states that each worker process in Q shares the same code, but with β replaced by β_i inside the code for the i^{th} worker process.

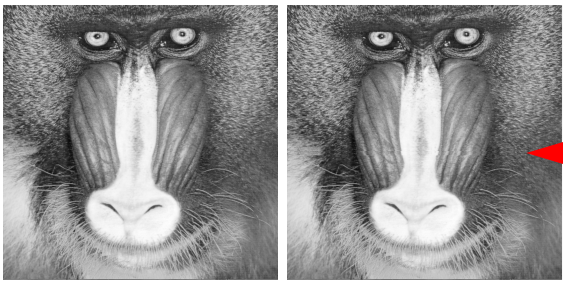


Figure 3.1: Exact (left) and Approximate (right) Scaled Images

Each worker process scales up its assigned slice and returns it to the master process, that then constructs the complete image. To transfer the data between the processes, a developer can use the statement `send` (Line 6, process α), which should be matched with the corresponding `receive` statement (Line 4, process β).

Types of the variables in Parallely can be precise (meaning that no approximation is applied to the values) and approximate. Parallely supports integer and floating-point scalars and arrays with different precision levels (e.g., 16, 32, 64 bit). Since the channels are typed, data transfers require the developer to specify the data type. Transferring large data structures may incur a signficant time overhead. Large arrays (e.g. the image array) are prime candidates for applying various approximate communication techniques to reduce the overhead. We assume that processes have disjoint sets of variable names and write `pid.var` to refer to variable `var` of process `pid`.

```
1  precise int[] src[10000];
2  precise int[] dst[40000];
3  precise int[] slice[4000];
4  precise int i, idx;
5  for β in Q do {
6    send(β, precise int[], src)
7  };
8  for β in Q do {
9    slice = receive(β, precise int[]);
10   i = 0;
11   repeat 4000 {
12     idx = β*4000+i;
13     dst[idx] = slice[i];
14     i = i+1;
15 } }
```

α

$|| \Pi.\beta : Q$

```
1  precise int[] src[10000];
2  precise int[] slice[4000];
3
4  src = receive(α, precise int[]);
5  //scales up the assigned slice
6  slice = scaleKernel(src,β);
7  send(α, precise int[], slice);
```

β

Figure 3.2: Parallely: Scale Calculation

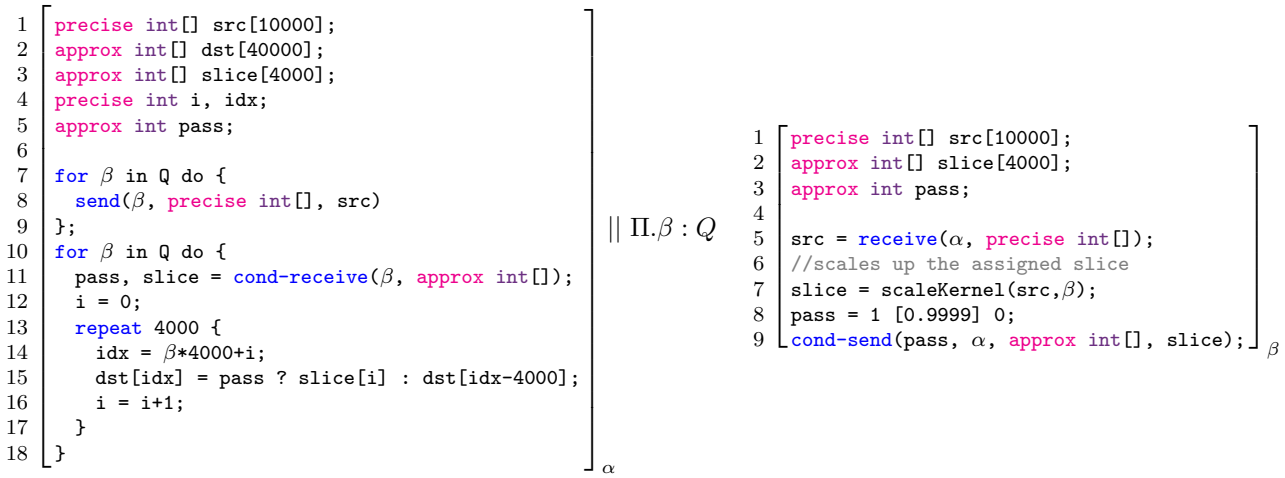


Figure 3.3: Parallely: Scale with Random Task Failures

3.2.1 Approximate Transformation

We consider the scenario in which there exists a small chance (here 0.01%) that a Scale worker process will fail due to a hardware error. We model this source of unreliability by setting `pass` to 1 with probability 0.9999 and 0 otherwise. Figure 3.3 shows an approximate version of the Scale computation. The worker processes use a special version of send, `cond-send`, to send the result to the master process. `cond-send` sends an empty acknowledgement if its first argument is 0. Otherwise, it sends the message. This statement needs to be matched with a corresponding `cond-receive` statement in the master process. The `slice` array is only updated if the message is received, and the `pass` variable is also set to 1 or 0 accordingly.

The developer may implement additional functionality to rectify the execution from such failures. While a conventional fault-recovery strategy would be re-sending the data and re-executing the failed tasks, such a strategy can incur a significant overhead of additional computation or communication. Instead, a significantly less-expensive and approximate recovery from error in this domain would be to reuse the previously computed pixels from the adjacent image positions. In the master process from Figure 3.3, we implemented a simple version of this strategy. If the master process receives the data (`pass` is `true`), it copies the received `slice` into `dest`. If it does not receive the data (`pass` is `false`), the previous `slice` is duplicated.

We developed a translator for Parallely, which does source-to-source translation to the Go language. Figure 3.1 shows the images produced by the exact version (left) and approximate version (right). The red triangle on the right side indicates the region in which the pixels have been approximated. The peak-signal-to-noise ratio is 38.8dB, indicating an acceptable approximation.

```
precise int[] α.src[10000];
approx int[] α.dst[40000];
approx int[] α.slice[4000];
precise int α.i, α.idx;
approx int α.pass;
precise int[] β1.src[10000], β2.src[10000], ...;
approx int[] β1.slice[4000], β2.slice[4000], ...;
approx int β1.pass, β2.pass, ...;

for β in Q do {
  β.src = α.src;
};
for β in Q do {
  //scales up the assigned slice
  β.slice = scaleKernel(β.src,β);
  β.pass = 1 [0.9999] 0;
  α.pass = β.pass ? 1 : 0;
  α.slice = β.pass ? β.slice : α.slice;
  α.i = 0;
  repeat 4000 {
    α.idx = β*4000+α.i;
    α.dst[α.idx] = α.pass ? α.slice[α.i]
                          : α.dst[α.idx-4000];
    α.i = α.i+1;
  };
}
```
α

Figure 3.4: Parallely: Scale Sequential Code.

3.2.2 Properties

We wish to verify the following properties about this program:

- **Type Safety:** approximate variables cannot affect the values of precise variables, either directly, via assignment, or indirectly, by affecting control flow;
- **Deadlock-Freeness:** the execution of the approximate program is deadlock-free; and
- **Quantitative Reliability:** the result will be calculated correctly with probability at least 0.99. We encode this requirement in Parallely as $0.99 \leq \mathcal{R}(\text{dst})$, using the notation from Rely.

Next, we show how Parallely's static analyses verify these properties.

3.2.3 Verification

Parallely verifies that approximate variables do not interfere with precise variables via a type checking pass in two steps. In the first step, it checks that the code in the master process and the worker process has the correct type annotations, i.e., an approximate value cannot be assigned to the precise variable. In the second step, it uses program sequentialization (which is sensitive to the types in the send and receive primitives) to ensure that precise sends are consumed by precise receives.

n	$\in \mathbb{N}$	*quantities*		
`m`	$\in \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$	*values*	$S \rightarrow$	
r	$\in [0, 1.0]$	*probability*	`skip`	*empty program*
x, b, X	$\in$ Var	*variables*	$\mid x$ `=` *Exp*	*assignment*
a	$\in$ ArrVar	*array variables*	$\mid x$ `=` *Exp* $[r]$ *Exp*	*probabilistic choice*
α, β	$\in$ Pid	*process ids*	$\mid x$ `=` b`?` *Exp* `:` *Exp*	*conditional choice*
			$\mid S$`;`S	*sequence*
Exp	$\rightarrow m \mid x \mid f(Exp^*) \mid$	*expressions*	$\mid x$ `=` a`[`Exp^+`]`	*array load*
	$(Exp) \mid Exp\ op\ Exp$		$\mid a$`[`Exp^+`]` `=` *Exp*	*array store*
			$\mid$ `if` $x\ S\ S$	*branching*
q	$\rightarrow$ `precise` $\mid$ `approx`	*type qualifiers*	$\mid$ repeat n {S}	*repeat n times*
t	$\rightarrow$ `int<n>` $\mid$ `float<n>`	*basic types*	$\mid x$ `=` (T)*Exp*	*cast*
T	$\rightarrow q\ t \mid q\ t$ []	*types*	$\mid$ `for` $i : [Pid^+]\{S\}$	*iterate processes*
D	$\rightarrow$ T x $\mid$ T a$[n^+]$ $\mid$	*variable*	$\mid$ send(α, T, x)	*send message*
	D`;`D	*declarations*	$\mid x$ `=` receive(α, T)	*receive a message*
			$\mid$ cond-send(b, α, T, x)	*conditionally send*
P	$\rightarrow [D; S]_\alpha \mid$	*process*	$\mid b, x$ `=` cond-receive(α, T)	*receive from a*
	$\Pi.\alpha : X\ [D; S]_\alpha \mid$	*process group*		*cond-send*
	$P \parallel P$	*process composition*		

Figure 3.5: Parallely Syntax

To ensure that the approximate and precise channels are matched, and to prove that the program is deadlock free, Parallely converts the program to an equivalent sequential program, shown in Figure 3.4. It does so by matching each `send` or cond-send with the corresponding `receive` or cond-receive and converting the message transmission operation to an assignment operation. This conversion is achieved by applying a selection of rewrite rules that perform syntactic transformations on statements in the parallel program. One rewrite rule replaces each `send` statement with a skip statement (no operation) and stores the value being sent in the context of the rewrite system. A second rewrite rule replaces the matching `receive` statement with an assignment, where the assigned value is the value that was sent by the matching `send` statement. Line 11 and line 18 in Figure 3.4 show how the rewrite rules sequentialize the communication from our example into assignments. Successful sequentialization guarantees that there are no deadlocks in the program *and* that the precise sends (resp. approximate sends) are matched with precise receives (resp. approximate receives).

Finally, Parallely performs a reliability analysis pass on the sequentialized program from Figure 3.4 to verify that the reliability of the `dest` array at the end of execution is at least 0.99. The sequential Rely analysis requires a finite bound on the number of loop iterations. Here, it is the number of the worker processes, $|Q|$. Our reliability analysis on the sequentialized program results in the constraint: $0.9999^{|Q|} \geq 0.99$. The formula is satisfied

for every $|Q| \leq 100$. This chapter shows that if the reliability predicate is valid for the sequentialized program, it will also be valid for the parallel program.

3.3 VERIFYING SAFETY AND ACCURACY OF TRANSFORMATIONS

Figure 4.3 presents the Parallely syntax. Parallely is a strongly typed imperative language with primitives for asynchronous communication. The `send` statement is used to asynchronously send a value to another process using a unique process identifier. The receiving process can use the blocking `receive` statements to read the message. In addition, Parallely supports array accesses, iteration over a set of processes, conditionals, and precision manipulation via casting. We present the precise semantics of the Parallely language in Section 3.4.

Given an approximate version (P^A) of a program (P), Parallely first type checks each individual process using the rules defined in Section 3.6. Then, it converts the approximate program to its canonical sequentialization (P^A_{seq}) using the procedure described in Section 3.5.1, which proves deadlock freedom. Finally Parallely performs the reliability and accuracy analysis on the sequentialized program (Section 3.7).

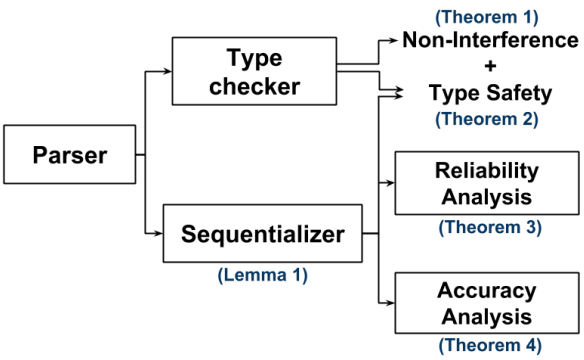


Figure 3.6: Overview of Parallely

Figure 3.6 presents the overview of the modules in our implementation of Parallely. The type checker and sequentializer modules work together to provide the safety guarantees. The sequentialization module outputs a sequential program, which can then be used with the reliability/accuracy analysis. Figure 3.6 also highlights the relevant lemma or theorem in this chapter for each aspect. We next present several popular parallel patterns and approximate transformations from literature, all of which can be represented and verified using Parallely. For each pattern, we present a code example, sequentialized versions of the code examples, transformation, and discuss verification challenges. We also present the analysis time for these patterns in Section 3.8.

3.3.1 Precision Reduction

Pattern and Transformation: It reduces the precision of approximate data being transferred between processes by converting the original data type to a less precise data type (e.g. doubles to floats). Precision reduction is a common technique for approximate data compression (e.g. as used in TensorFlow [28]).

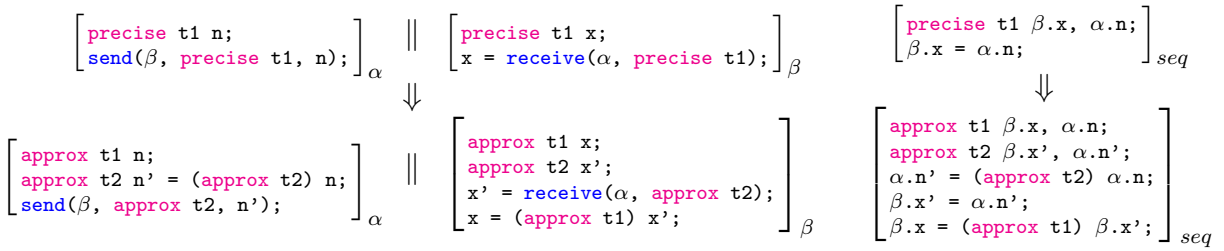


Figure 3.7: Precision reduction code in Parallely and its sequentialization

Safety: As precision reduction is an approximate operation, the type of the reduced-precision data must be an approximate type. The type must be changed in both the sender and receiver process to the same type. Further, there must not already be messages of the less precise type being sent between the processes, else the converted code may affect the order of the messages and violate the symmetric nondeterminism property necessary for sequentialization. **Accuracy:** The developer specifies the domain of the transmitted value as an interval (e.g., $[0, 32]$). Precision reduction introduces an error to this transmitted value that depends on the original and converted types (e.g., 10^{-19} when converting from double to float). The interval analysis in Section 3.7 can then calculate the maximum absolute error of the result.

3.3.2 Data Transfers over Noisy Channels

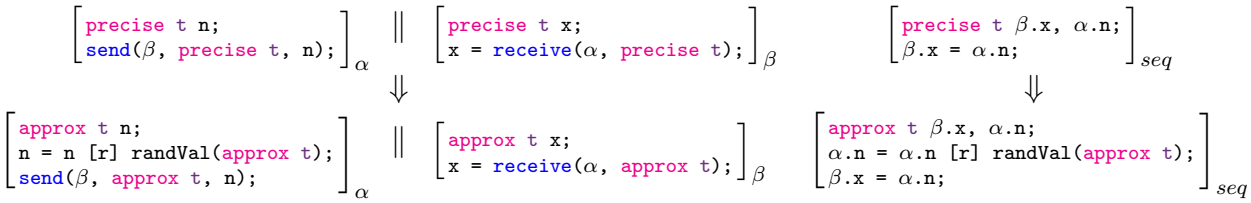


Figure 3.8: Code for communication over a noisy channel in Parallely and its sequentialization

Pattern and Transformation: The code in Figure 3.8 models the transfer of approximate data over an unreliable communication channel. The channel may corrupt the data and the receiver may receive a garbage value with probability $1 - r$. If the received message

is corrupted, the approximate program may still decide to continue execution (instead of requesting resend).

Safety: The variable that may potentially be corrupted must have an approximate type. Consequently, the developer must send and receive the data over an approximate typed channel.

Accuracy: We use a reliability specification $r \leq \mathcal{R}(\beta.x)$ which states that the variable being transferred (x) must reach the destination intact with probability at least r. This specification can be directly proved on the sequentialized version of the program, with a single statement (the probabilistic choice modeling the occasional data corruption) affecting this reliability condition.

3.3.3 Failing Tasks

Pattern and Transformation: the code in Figure 3.9 models the execution of tasks that can fail with some probability $1-r$ due to hardware or software errors. For instance, MapReduce ignores a task result if the task experiences a failure [29]; Topaz returns an error if a task running on an unreliable core fails [89]. We model such scenarios by conditionally transferring data (with cond-send and cond-receive), based on the random chance of task success, r.

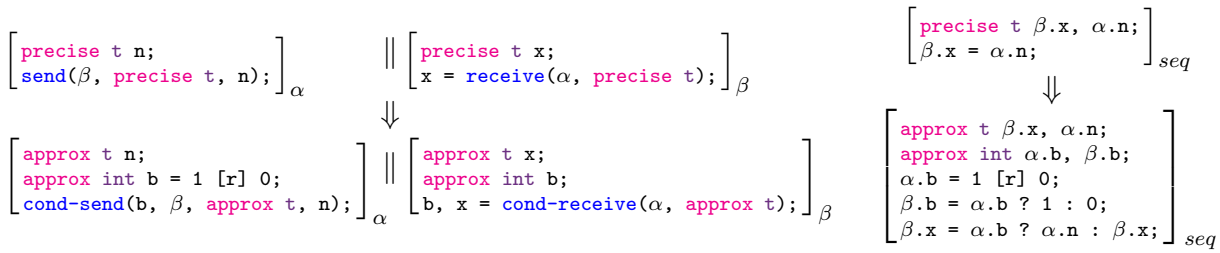


Figure 3.9: Code for modeling a failing task in Parallely and its sequentialization

Safety: Like in the noisy-channel pattern, the developer needs to assign approximate types to the data being sent and the channels over which the data is sent. In addition, the developer must use cond-send and cond-receive calls.

Accuracy: Similar to the noisy-channel pattern, we again use a Rely style specification, e.g. $r \leq \mathcal{R}(\beta.x)$, which can be proved on the sequentialized version of the program.

3.3.4 Approximate Reduce (Sampling)

Pattern and Transformation: This pattern approximates an aggregation operation such as finding the maximum or sum. To implement sampling, the worker process only computes and sends the result with probability r. Otherwise, it only sends an empty message to the master. The master process adjusts the aggregate based on the number of received results (but only if it received *some* data). Figure 3.10 presents the Parallely code for the pattern.

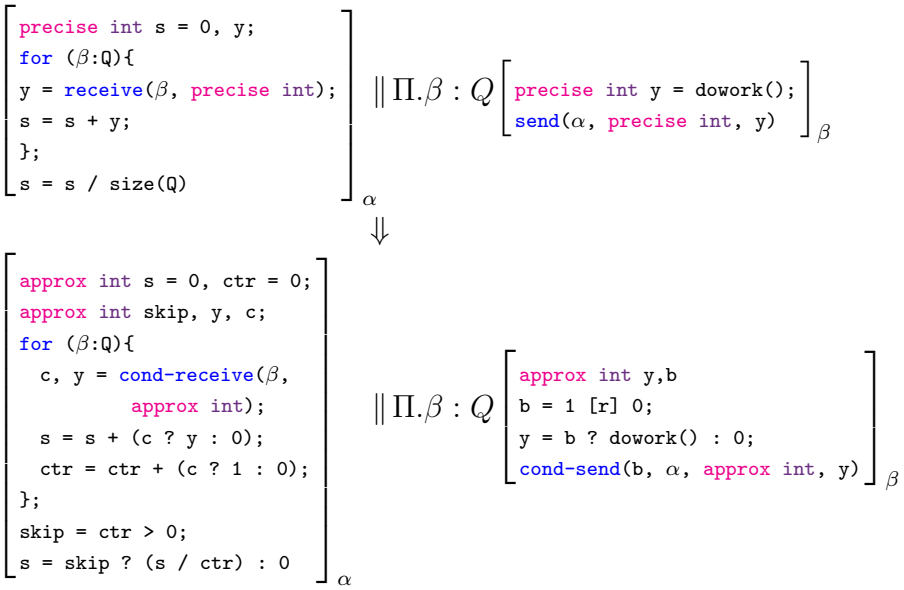


Figure 3.10: Code implementing approximate reduction in Parallely

Safety: Figure 3.13 shows the sequentialized version of the code. To successfully sequentialize, the transformed program's master task gathers the results from all symmetric workers tasks. To ensure that divide-by-zero cannot happen, we need an additional check for `ctr`.

Accuracy: We can automatically prove two properties: the reliability (as before), and the interval bound of the result. If the inputs are in the range $[a, b]$, then the error of the average will also be in the same range. Further formal reasoning about this pattern (e.g., [90]) may provide more interesting probabilistic bounds. However, application of such analyses is outside of the scope of this work.

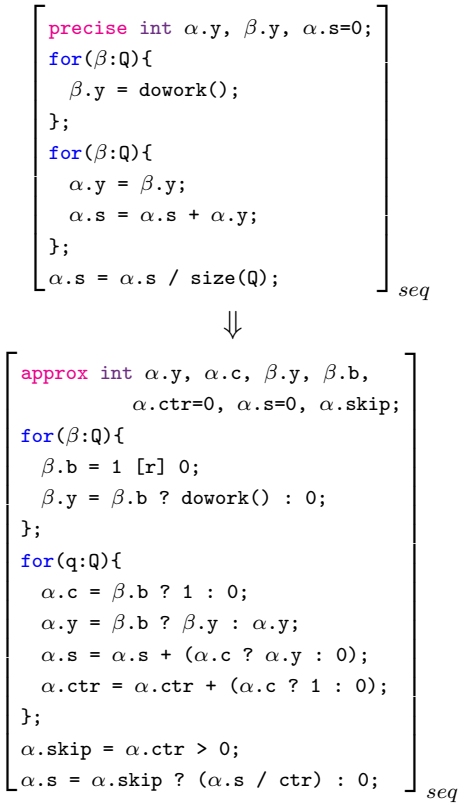


Figure 3.11: Sequentialization of approximate reduction

3.3.5 Skipping Negligible Updates

This transformation drops packets if the value being sent is a scalar below a certain threshold. The receiver must be adding the received value to a sum variable. This transformation saves energy by not sending small updates to the receiver, which will cause an insignificant change in the sum. This transformation requires that the received value is used to update some other variable via addition. The result message type must be scalar to allow thresholding.

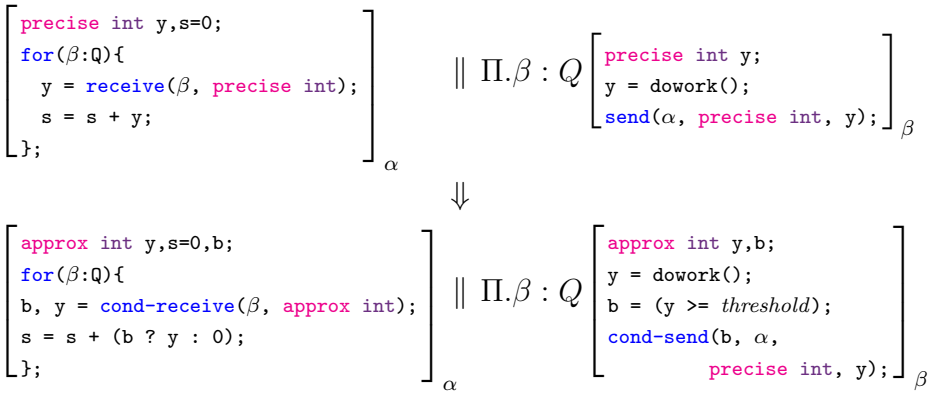


Figure 3.12: Skipping negligible updates in Parallely

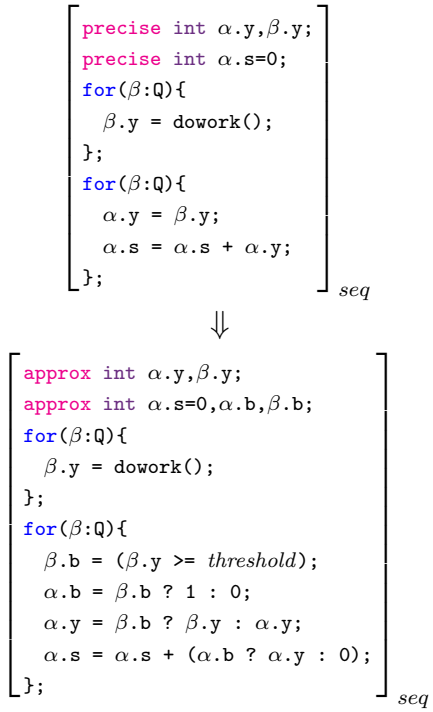


Figure 3.13: Skipping negligible updates in sequentialized form

3.3.6 Approximate Map (Approximate Memoization)

Pattern and Transformation: A map task computes on a list of independent elements. Reduction of the number of tasks in conjunction with approximate memoization [30, 91] can reduce communication and improve energy efficiency. If the master process decides not to send a task to a worker process, then that worker process will return an empty result. Upon receiving an empty result, the master process uses the previously received result in its place. There is no need for additional code to use the most recently received result, as cond-receive does not update the variable that stores the received value when an empty value is received.

The example in Figure 3.3 uses this pattern. If a task fails to return a slice to the master task then the previous slice is used, as described in Section 4.2.

Safety: Even if no work is sent to some workers, the master task must still receive an empty message from each worker to ensure symmetric nondeterminism.

Accuracy: We use a reliability specification $r \leq \mathcal{R}(\alpha.results)$ which states that the reliability of the entire `results` array is at least r. If even one element of the array is different from the precise version, then the entire array is considered incorrect for the purposes of measuring reliability. This reliability depends on the probability of sending a job to a worker task in the master task.

3.3.7 Other Verified Patterns and Transformations

Multiple other computation patterns can be expressed in Parallely in a manner that satisfies symmetric nondeterminism. These include the Scatter-Gather, Stencil, Scan, and Partition patterns. These patterns are similar to the map and reduce patterns, but distribute data slightly differently to the worker tasks. We can apply transformations such as precision reduction, failing tasks, sampling, etc. to these patterns and prove their sequentializability and type safety. We can also calculate their reliability and accuracy. These patterns are presented along with several code examples in the Appendix.

3.3.8 Approximate Hardware

We also support Rely's approximate instructions and approximate memories. For example, we can model arithmetic instructions as : `z = (x op y) [r] randVal()`, which models an instruction that can produce an error with probability `r`. Similarly, approximate memories can be modeled through the corresponding read and write operations, e.g., as `x = x [r] randVal()`, which corrupts the memory location storing a variable `x` with probability `r`.

3.3.9 Unsafe Patterns and Transformations

Runtime Task Skipping. Certain approximations are not safe, as they can introduce deadlocks or violate relative safety properties. For example, if the approximate reduce transformation is implemented by simply not sending some data back from the workers, it may cause the master process to wait for data that it will never receive, violating the symmetry requirement necessary for sequentialization.

Timed Receives. Another possible type of approximate receive operation is the timed receive operation, which times out if no value is received within a specified time bound. Such timed receives will not work with our approach, as they introduce the possibility of sending a value that is not received. However, we anticipate that recent approaches like [92] (which support this type of timed communication using some simplifying assumptions), could extend the reach of our analysis to support timed receive operations.

Iterative Fixed-Point Computations. A common computation pattern is to repeat a calculation until the errors are small. This pattern does not satisfy the property of non-interference in Parallely even though it is a safe computation. In addition, if there is communication within the loop body, the loop cannot be sequentialized, as it uses the loop

carried state for termination. Sequentialization requires that the decision only depends on values computed in the current iteration. Figure 3.14 shows an example of this pattern.

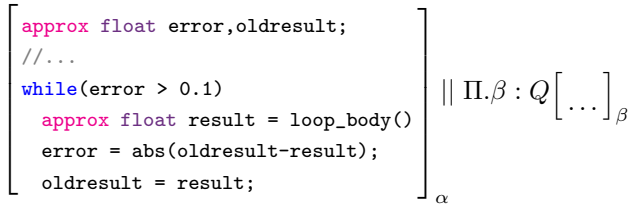


Figure 3.14: An example iterative fixed point computation

3.4 SEMANTICS OF PARALLELY

Figure 3.15 and Figure 3.17 present the Parallely's rules for the small-step expression and statement semantics.

References. A *reference* is a pair $\langle n_b, \langle n_1, \ldots, n_k \rangle\rangle \in \texttt{Ref}$ that consists of a base address $n_b \in \mathit{Loc}$ and a dimension descriptor $\langle n_1, \ldots, n_k \rangle$. References describe the location and the dimension of variables in the heap.

Frames, Stacks, and Heaps. A *frame* σ is an element of the domain $\mathrm{E} = \mathrm{Var} \to \mathrm{Ref}$ which is the set of finite maps from program variables to references. A *heap* $h \in H = \mathbb{N} \to \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ is a finite map from addresses (integers) to values. Values can be an Integer, Float or the special *empty message* ($\varnothing$).

Processes. Individual processes execute their statements in sequential order. Each process has a unique process identifier (Pid). Processes can refer to each other using the process identifier. We do not discuss process creation and removal. We assume that the processes have disjoint variable sets of variable names. We write `pid.var` to refer to variable `var` of process `pid`. When unambiguous, we will omit `pid` and just write `var`.

Types. Types in Parallely are either precise (meaning that no approximation can be applied to them) and approximate. Parallely supports integer and floating-point scalars and arrays with different levels of precision.

Typed Channels and Message Orders. Processes communicate by sending and receiving messages over a typed *channel*. There is a separate subchannel for each pair of processes further split by the *type* of message. $\mu \in \mathit{Channel} = \mathit{Pid} \times \mathit{Pid} \times \mathit{Type} \to \mathit{Val}^*$. Messages

$$\text{E-VAR-C}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x)}{\langle x,\sigma,h\rangle \rightharpoonup_\psi \langle h(n_b),\sigma,h\rangle} \qquad \text{E-VAR-F}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x)}{\langle x,\sigma,h\rangle \overset{1}{\rightharpoonup}_\psi \langle n_f,\sigma,h\rangle}$$

$$\text{E-IOP-R1}\quad \frac{\langle e_1,\sigma,h\rangle \overset{p}{\rightharpoonup}_\psi \langle e_1',\sigma,h\rangle}{\langle e_1\ op\ \ e_2,\sigma,h\rangle \overset{p}{\rightharpoonup}_\psi \langle e_1'\ op\ \ e_2,\sigma,h\rangle}$$

$$\text{E-IOP-R2}\quad \frac{\langle e_2,\sigma,h\rangle \overset{p}{\rightharpoonup}_\psi \langle e_2',\sigma,h\rangle}{\langle n\ op\ \ e_2,\sigma,h\rangle \overset{p}{\rightharpoonup}_\psi \langle n\ op\ \ e_2',\sigma,h\rangle} \qquad \text{E-IOP-C}\quad \frac{}{\langle n_1\ op\ \ n_2,\sigma,h\rangle \overset{1}{\rightharpoonup}_\psi \langle op(n_1,n_2),\sigma,h\rangle}$$

Figure 3.15: Dynamic Semantics of Expressions

on the same subchannel are delivered in order but there are no guarantees for messages sent on separate (sub)channels.

Programs. We define a program as a parallel composition of processes. We denote a program as $P = [P]_1 \parallel \cdots \parallel [P]_i \parallel \cdots \parallel [P]_n$. Where $1, \ldots, n$ are process identifiers. An approximated program executes within *approximation model*, ψ, which in general may contain the parameters for approximation (e.g., probability of selecting original or approximate expression). We define special reliable model 1_ψ, which evaluates the program without approximations.

Scheduler Distributions. $P_s(i \mid \langle P, \epsilon, \mu\rangle)$ models the probability that the thread with id i is scheduled next. We define it history-less and independent of ϵ contents. For reliability analysis we assume a fair scheduler that in each step has a positive probability for all threads that can take a step in the program.

We make the following assumptions for the reliability analysis to ensure that the scheduler is fair. (The remaining analyses do not take into account this distribution).

1. $\forall \epsilon, \mu.\ \sum_{\alpha \in Tid} P(\alpha|(P,\epsilon,\mu)) = 1$
2. $\forall P, \epsilon, \mu.\ \forall \alpha.\ P(\alpha|(P,\epsilon,\mu)) > 0$ iff $\exists\ P', \epsilon'\mu'$ s.t. $\quad (\epsilon,\mu,P) \xrightarrow{\alpha,p}_\psi (\epsilon',\mu',P')$

Global and Local Environments. Each process works on its private environment consisting of a frame and a heap, $\langle \sigma^i, h^i\rangle \in \Lambda = H \times \mathrm{E}$. We define a global configuration as a triple $\langle P, \epsilon, \mu\rangle$ of a program, global environment, and a channel. The global environment is a map from the process identifiers to the local environment $\epsilon \in Env = Pid \mapsto \Lambda$.

Expressions. Figure 3.15 presents the dynamic semantics for expressions. The labeled small-step evaluation relation of the form $\langle e,\sigma,h\rangle \overset{1}{\rightharpoonup}_\psi \langle e',\sigma,h\rangle$ states that from a frame σ and a heap h, an expression e evaluates in one step with probability 1 to an expression e' without

$$\textsc{Dec-Var}\quad \frac{\langle n_b,\ h'\rangle = \texttt{new}(h, \langle 1\rangle)}{\langle \text{T x}, \sigma :: \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma[x \mapsto \langle n_b, \langle 1\rangle m\rangle] :: \sigma, h', \mu\rangle}$$

$$\textsc{Dec-Array}\quad \frac{\forall i.0 < n_i \qquad \langle n_b,\ h'\rangle = \texttt{new}(h, m, \langle n_1 ... n_k\rangle) \qquad \sigma' = \sigma[x \mapsto \langle n_b, \langle n_1 .. n_k\rangle m\rangle]}{\langle \text{T x}[n_1 ... n_k], \sigma :: \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma' :: \sigma, h', \mu\rangle}$$

Figure 3.16: Semantics of Declarations

any changes to the frame σ and heap h. Parallely supports typical integer and floating point operations. We allow function calls and inline them as a preliminary step.

Statements. The small-step relation of the form $\langle s, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s', \sigma', h', \mu'\rangle$ defines a single process in the program evaluating in its local frame σ, heap h, and the global channel μ. Figure 3.16 defines the semantics for declaration statements. Figure 3.17 defines the semantics for statements. Individual processes can only access their own frame and heap. We unroll `repeat` statements as a preliminary step. We use `h::t` to denote accessing the head (`h`) of a queue and `h++t` to denote adding `t` to the end of the queue. Figure 3.18 defines the semantics for statements that interact with arrays.

$$\textsc{E-Array-Load-Idx}\quad \frac{\langle e_i, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e_i', \sigma, h\rangle}{\langle x \texttt{ = } a\texttt{[}n_1, ..., e_i, ..., e_k\texttt{]}, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x \texttt{ = } a\texttt{[}n_1, ..., e_i', ..., e_k\texttt{]}, \sigma, h, \mu\rangle}$$

$$\textsc{E-Array-Load-C}\quad \frac{\langle n_b,\ \langle l_1, ..., l_k\rangle\rangle = \sigma(x) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(n_b + n_o)}{\langle x \texttt{ = } a\texttt{[}n_1, ..., n_k\texttt{]}, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x \texttt{ = } n, \sigma, h, \mu\rangle}$$

$$\textsc{E-Array-Store-Idx}\quad \frac{\langle e_i, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e_i', \sigma, h\rangle}{\langle a\texttt{[}n_1, ..., e_i, ..., e_k\texttt{]} \texttt{ = } x, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle a\texttt{[}n_1, ..., e_i', ..., e_k\texttt{]} \texttt{ = } x, \sigma, h, \mu\rangle}$$

$$\textsc{E-Array-Store-C}\quad \frac{\begin{array}{c}\langle n_b,\ \langle l_1, ..., l_k\rangle\rangle = \sigma(x) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \\ \langle n_b',\ \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b'] = v \qquad \psi(wr(m)) = 1\end{array}}{\langle a\texttt{[}n_1, ..., n_k\texttt{]} \texttt{ = } x, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle skip, \sigma, h[(n_b + n_o) \mapsto v], \mu\rangle}$$

Figure 3.18: Process-Level Dynamic Semantics of Arrays

E-Assign-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e', \sigma, h\rangle}{\langle x \texttt{ = } e, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x \texttt{ = } e', \sigma, h, \mu\rangle}$$

E-Assign-C

$$\frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x)}{\langle x \texttt{ = } n, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto n], \mu\rangle}$$

E-Assign-Prob-True

$$\frac{}{\langle x \texttt{ = } e_1\ [r]\ e_2, \sigma, h, \mu\rangle \xrightarrow{r}_{\psi} \langle x \texttt{ = } e_1, \sigma, h, \mu\rangle}$$

E-Assign-Prob-False

$$\frac{}{\langle x \texttt{ = } e_1\ [r]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1-r}_{\psi} \langle x \texttt{ = } e_2, \sigma, h, \mu\rangle}$$

E-Assign-Approx-True

$$\frac{\langle l,\ \langle 1\rangle\rangle = \sigma(b) \qquad h[l] \neq 0}{\langle x \texttt{ = } e_1\ [b]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle x \texttt{ = } e_1, \sigma, h, \mu\rangle}$$

E-Assign-Approx-True

$$\frac{\langle l,\ \langle 1\rangle\rangle = \sigma(b) \qquad h[l] = 0}{\langle x \texttt{ = } e_1\ [b]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle x \texttt{ = } e_2, \sigma, h, \mu\rangle}$$

E-Seq-R1

$$\frac{\langle s_1, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s_1', \sigma', h', \mu'\rangle}{\langle s_1 \texttt{;}\ s_2, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s_1' \texttt{;}\ s_2, \sigma', h', \mu'\rangle}$$

E-Seq-R2

$$\frac{}{\langle \texttt{skip;}\ s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_2, \sigma, h, \mu\rangle}$$

E-If-True

$$\frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] \neq 0}{\langle \texttt{if}\ x\ s_1\ s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_1, \sigma, h, \mu\rangle}$$

E-If-False

$$\frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] = 0}{\langle \texttt{if}\ x\ s_1\ s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_2, \sigma, h, \mu\rangle}$$

E-Send

$$\frac{\begin{array}{c}\text{isPid}(\beta) \qquad \langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \\ h[n_b] = n \qquad \mu[\langle \alpha, \beta, t\rangle] = m\end{array}}{\begin{array}{l}\langle [\text{send}(\beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle\end{array}}$$

E-Receive

$$\frac{\begin{array}{c}\mu[(\beta = w) \vee (\beta \in w) \qquad \langle \beta, \alpha, t\rangle] = m :: n \\ \langle n_b,\ \langle 1\rangle\rangle = \sigma(x)\end{array}}{\begin{array}{l}\langle [x \texttt{ = } \text{receive}(w, t)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto v], \mu[\langle \beta, \alpha, t\rangle \mapsto n]\rangle\end{array}}$$

E-CondSend-True

$$\frac{\begin{array}{c}\langle l,\ \langle 1\rangle\rangle = \sigma(b) \qquad h[l] \neq 0 \qquad \text{isPid}(\beta) \\ \langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = v \qquad \mu[\langle \alpha, \beta, t\rangle] = m\end{array}}{\begin{array}{l}\langle [\text{cond-send}(b, \beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle\end{array}}$$

E-CondSend-False

$$\frac{\begin{array}{c}\langle l,\ \langle 1\rangle\rangle = \sigma(b) \\ h[l] = 0 \qquad \text{isPid}(\beta) \qquad \mu[\langle \alpha, \beta, t\rangle] = m\end{array}}{\begin{array}{l}\langle [\text{cond-send}(b, \beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +\varnothing]\rangle\end{array}}$$

E-CondReceive-True

$$\frac{\begin{array}{c}\mu[\langle \beta, \alpha, t\rangle] = m :: n \qquad (\beta = w) \vee (\beta \in w) \\ \langle n_1,\ \langle 1\rangle\rangle = \sigma(x) \qquad \langle n_2,\ \langle 1\rangle\rangle = \sigma(b)\end{array}}{\begin{array}{l}\langle [b, x \texttt{ = } \text{cond-receive}(\beta, t)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_1 \mapsto v][n_2 \mapsto 1], \mu[\langle \beta, \alpha, t\rangle \mapsto n]\rangle\end{array}}$$

E-CondReceive-False

$$\frac{\begin{array}{c}\mu[\langle \beta, \alpha, t\rangle] = \varnothing :: m \\ (\beta = w) \vee (\beta \in w) \qquad \langle n_b,\ \langle 1\rangle\rangle = \sigma(b)\end{array}}{\begin{array}{l}\langle [\text{b, x } \texttt{ = } \text{cond-receive}(\beta, t)]_{\alpha}, \sigma, h, \mu\rangle \\ \quad \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto 0], \mu[\langle \beta, \alpha, t\rangle \mapsto m]\rangle\end{array}}$$

E-Cast-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e', \sigma, h\rangle}{\langle x \texttt{ = } (\text{T})e, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x \texttt{ = } (\text{T})e', \sigma, h, \mu\rangle}$$

E-Cast-C

$$\frac{n' = \texttt{convert}(T, n) \qquad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x)}{\langle \text{x } \texttt{ = } (\text{T})n, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto n'], \mu\rangle}$$

E-Par-Iter

$$\frac{}{\begin{array}{l}\langle \texttt{for}\ i : [\alpha_1 ... \alpha_k]\{S\}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \\ \quad \langle S[\alpha_1/i]; \ldots; S[\alpha_k/i], \sigma, h, \mu\rangle\end{array}}$$

Figure 3.17: Process-Level Dynamic Semantics of Statements

E-Assign-Prob-Exact

$$\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \rightarrow^{1}_{1_\psi} \langle x = e_1, \sigma, h, \mu\rangle$$

E-Cast-Exact

$$\langle x = (\mathrm{T})e, \sigma, h, \mu\rangle \rightarrow^{1}_{1_\psi} \langle x = e, \sigma, h, \mu\rangle$$

Figure 3.19: Exact Execution Semantics of Statements

Approximate Statements. In addition to the usual statements, we include probabilistic choice and Boolean choice. A probabilistic choice expression $x = e_1\ [r]\ e_2$ evaluates to e_1 with probability r (e_2 with $1 - r$) if r is float, or as a deterministic if-expression when r is an integer, selecting e_1 if $r \geq 1$ (e_2 if $r = 0$). We use the *cast* statement to perform precision reduction (but only between the values of the same general type, int or float).

Parallely also contains `cond-send` statements that use an additional *condition variable* and only sends the message if it evaluates to 1. If the message is not sent, an empty message ($\varnothing$) is sent to the channel as a signal. `cond-receive` acts similar to `receive` but only updates the variables if the received value is not $\varnothing$.

In Parallely non-deterministic differences in evaluations arise only from the probabilistic choice expressions. These can be used to model a wide range of approximate transformations. We use the approximation model to differentiate between exact and approximate executions of the program. We use 1_ψ to specify exact, precise execution and present program semantics of exact execution in Figure 3.19. Under exact execution, probabilistic choice statements always evaluate to the first option, casting performs no change, and all declarations allocate the memory required to store the full precision data.

For reliability analysis to be valid we require that the approximate program under exact evaluation be the same as the program without any approximations.

Global Semantics. Small step transitions of the form $(\epsilon, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha,p}_\psi (\epsilon', \mu', P'_\alpha \parallel P_\beta)$ define a single process α taking a local step with probability p. Figure 4.9 defines the global semantics. The distribution $P_s(i \mid \langle P, \epsilon, \mu\rangle)$ models the probability that the process with id i is scheduled next. We define it history-less and independent of ϵ contents. For reliability analysis we assume a fair scheduler that in each step has a positive probability for all threads that can take a step in the program. The global semantics consists only of individual processes executing using the statement semantics in their local environment and the shared μ.

GLOBAL-STEP

$$\frac{p_\alpha = P_s[\alpha \mid (\epsilon, \mu, P_\alpha \parallel P_\beta)] \qquad \epsilon[\alpha] = \langle \sigma, h\rangle \qquad \langle P_\alpha,\ \sigma, h, \mu\rangle \xrightarrow{p}_\psi \langle P'_\alpha,\ \sigma', h', \mu'\rangle \qquad p' = p \cdot p_\alpha}{(\epsilon, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha,p'}_\psi (\epsilon[\alpha \mapsto \langle \sigma', h'\rangle], \mu', P'_\alpha \parallel P_\beta)}$$

Figure 3.20: Global Dynamic Semantics

3.4.1 Big-Step Notations

Since we are concerned only with the halting states of processes and analysis of deadlock free programs, we will define big-step semantics as follows for parallel traces that begin and end with an empty channel:

Definition 3.1 (Trace Semantics for Parallel Programs)**.**

$$\langle \cdot,\ \epsilon \rangle \overset{\tau, p}{\Longrightarrow}_{\psi} \epsilon' \equiv \langle \epsilon,\ \varnothing,\ \cdot \rangle \overset{\lambda_1, p_1}{\longrightarrow}_{\psi} \ \ldots \ \overset{\lambda_n, p_n}{\longrightarrow}_{\psi} \langle \epsilon',\ \varnothing,\ \texttt{skip} \rangle \\ \text{where } \tau = \lambda_1, \ldots, \lambda_n,\ p = \prod_{i=1}^{n} p_i \tag{3.1}$$

This big-step semantics is the reflexive transitive closure of the small-step global semantics for programs and records a *trace* of the program. A trace $\tau \in T \to \cdot | \alpha :: T$ is a sequence of small step global transitions. The probability of the trace is the product of the probabilities of each transition.

Definition 3.2 (Aggregate Semantics for Parallel Programs)**.**

$$\langle \cdot,\ \epsilon \rangle \overset{p}{\Longrightarrow}_{\psi} \epsilon' \text{ where } p = \sum_{\tau \in \mathrm{T}} p_\tau \text{such that } \langle \cdot,\ \epsilon \rangle \overset{\tau, p_\tau}{\Longrightarrow}_{\psi} \epsilon' \tag{3.2}$$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program starts in an environment ϵ and terminates in an environment ϵ'. It accumulates the probability over all possible interleavings that end up in the same final state.

Termination and Errors. Halted processes are those processes that have finished executing permanently. A process α is a halted process, i.e. $\alpha \in \text{hprocs}(\epsilon, \mu, P)$ if any of the following hold: (1) α's remaining program is `skip` (correct execution), (2) α is stuck waiting for a message, when its next statement is a receive or cond-receive, but there is no matching send or cond-send in the rest of the program (error state).

3.5 APPROXIMATION-AWARE CANONICAL SEQUENTIALIZATION

Canonical sequentialization by [87] is a method for statically verifying concurrency properties of asynchronous message passing programs. It leverages the structure of the parallel program to derive a representative sequential execution, called a *canonical sequentialization*. Verifying the properties on this sequential version would imply their validity in parallel execution too.

$$\textsc{R-Send}\quad \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +y]}{\Gamma, \Delta, [\text{send}(x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

$$\textsc{R-Receive}\quad \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\beta, \alpha, t] = m :: n \qquad \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \text{y = m}}{\Gamma, \Delta, [y \texttt{ = } \text{receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

$$\textsc{R-CondSend}\quad \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +(b : y)]}{\Gamma, \Delta, [\text{cond-send}(b, x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

$$\textsc{R-CondReceive}\quad \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\beta, \alpha, t] = (b' : m) :: n \qquad \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \text{b = } b'? \ 1 : 0; \text{y = } b'? \ m : x}{\Gamma, \Delta, [b, y \texttt{ = } \text{cond-receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

$$\textsc{R-Context}\quad \frac{\Gamma, \Delta, A \rightsquigarrow \Gamma', \Delta', A'}{\Gamma, \Delta, \text{A;B} \rightsquigarrow \Gamma', \Delta', \text{A';B}}$$

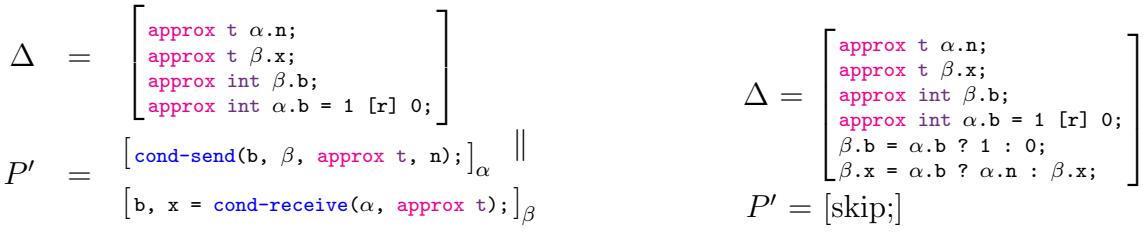
Figure 3.21: A Selection of the Rewrite Rules

One major requirement for sequentialization is that the parallel program must be *symmetrically nondeterministic* – each receive statement must only have a unique matching send statement, or a set of symmetric matching send statements. Further, there must not be spurious send statements that do not have a matching receive statement. The procedure for checking that a program is symmetrically nondeterministic is discussed by [87, Section 5].

3.5.1 Sequentialization of Parallely Programs

We define rewrite rules of the form $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$ which consist of a context (Γ), sequential prefix (Δ), and the remaining program to be rewritten (P). The prefix Δ contains the part of the program that has already been sequentialized. The context Γ consists of a symbolic set of messages in flight – *variables* being sent, but their matching receive has not already been found and sequentialized, and assertions about process identifiers.

A selection of the rewriting rules are available in Figure 4.13. They aim to fully sequentialize the program, i.e., reach $(\varnothing, \Delta_{prog}, \texttt{skip})$. The rules aim to gradually replace statements from the parallel program with statements in the sequential program Δ_{prog}. The sequential program is equivalent to the parallel program (as further discussed in Lemma 3.1). We define $\rightsquigarrow^*$ to be the transitive closure of rewrite rules. The sequentialization process starts from an empty context and sequential prefix, along with the original program, and applies the rewrite steps until the program is rewritten to `skip` with a context having empty message buffers.



(a) An intermediate step in the rewriting process (b) Final sequentialized code

Figure 3.22: An Example of the Rewriting Process.

Preliminaries. To support the rewrite process in our imperative language and avoid side effects we ensure that only variables can appear in `send` statements and that the program is in a Single Static Assignment (SSA) form before the rewriting process. This ensures that our rewrite context correctly represents the state of the variables that are being communicated. We also provide syntactic sugar to represent processing sending entire arrays. We de-sugar such statements to be a set of sequential `send` statements for each array location. In addition, we rename all variables in the program to ensure that the variable sets used in individual processes are disjoint. We place some restrictions on Parallely programs to simplify the rewriting process: we do not allow communication with external processes and do not allow communication inside conditional statements.

Example. Figure 4.11 illustrates the sequentialization process for an example program that models the execution of tasks that can fail with some probability. We reach the intermediate step in Figure 4.11(a) by applying the R-Context rule multiple times on statements that do not perform communication. Next, we apply the R-CondSend rule, which is the only applicable rule in this step. This rule saves the message being sent as a guarded expression $n : b$ in the context (i.e. $\Gamma(\alpha, \beta, \texttt{approx t}) = n : b$). Finally, we apply the R-CondReceive rule, which retrieves the guarded expression from the context and assigns to the variables in the receiver process. The final sequentialized program is shown in Figure 4.11(b).

Rewrite Soundness. Programs are in a *normal form* if they are parallel compositions of statements from distinct processes, i.e. statements from the same process are not composed in parallel. For two programs P_1 and P_2 in normal form, we define $P_1 \circ P_2$ as the process-wise sequencing of P_2 after P_1, i.e., for each process p present in both P_1 and P_2, the statements for p in P_1 are executed before those in P_2. We define $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ to indicate that ϵ and μ are a store and message buffer consistent with states reachable by executing Δ and assumptions in Γ. Let $\epsilon|_P$ denote the store restricted to variables local to processes in P. Let the set of permanently halted processes in a global configuration with an environment ϵ and

a channel μ be halted(ϵ, μ, P).

Lemma 3.1 states that the sequentialized program is an over-approximation of the parallel program with respect to halted processes. Intuitively, the lemma states that the sequentialized program can reach the same environment as the parallel program restricted to halted processes ($\rightarrow^*$ is the transitive closure over Global Semantics).

Lemma 3.1 (Rewrite Soundness)**.** Let P be a program in normal form. If

- $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$
- $P \circ P_0$ is symmetrically nondeterministic for some extension P_0
- $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ such that $(\epsilon, \mu, P \circ P_0) \longrightarrow^* (\epsilon_F, \mu_F, F)$,

there exists $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]$ such that $(\epsilon', \mu', P' \circ P_0) \rightarrow^* (\epsilon_{F'}, \mu_{F'}, F')$ and $\epsilon_F|_H = \epsilon_{F'}|_H$ where $H = \text{halted}(\epsilon_F, \mu_F, F)$

The proof of Lemma 3.1 builds up on the proof of Theorem 4.1 in [87]. The main additions in our proof are the additional cases for the R-CondSend and R-CondReceive rewrite rules. Unlike the proof for R-Send and R-Receive, our proof must show that the sequentialized code can mirror the behavior of two different semantic rules depending on whether or not the message was successfully sent.

Next we provide several important definitions followed by the proof of several lemmas that show important sub-properties in Parallely. Then, we use these definitions and lemmas to prove Lemma 3.1.

3.5.2 Definitions

Definition 3.3 (Transitive closure of global semantics)**.** $\rightarrow^*$ is defined as the transitive closure over global semantics rules.

Definition 3.4 (Process-wise composition)**.** $A \ltimes B$ is the process-wise composition of A and B. For each process α in B, $A \ltimes B$ sequences the statements belonging to α in A before those in B. This definition is from [93].

Definition 3.5 (Interpretation of stores)**.** $\epsilon \in [\![\Delta]\!]_{\epsilon_0}$ if and only if $(\epsilon_0, \varnothing, \Delta) \rightarrow^* (\epsilon, \varnothing, \texttt{skip})$. This definition is from [93].

Definition 3.6 (Interpretation of contexts)**.** $\mu \in [\![\Gamma]\!]_\epsilon$ if and only if $\forall(\alpha, \beta, t) \in \text{dom}(\Gamma)$. $\mu(\alpha, \beta, t) = [\![\Gamma(\alpha, \beta, t)]\!]_\epsilon$. This definition is from [93].

Definition 3.7 (Interpretation of stores and contexts)**.** $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_{\epsilon_0}$ if and only if

- $\epsilon \in [\![\Delta]\!]_{\epsilon_0}$
- $\mu \in [\![\Gamma]\!]_{\epsilon}$
- for all constraints $\{x \in X\}$ in Γ,$\epsilon(x) \in \epsilon(X)$
- for all constraints $\{\varnothing \subset X \subseteq Y\}$ in Γ,$\varnothing \subset \epsilon(X) \subseteq \epsilon(Y)$

This definition is from [93].

Definition 3.8 (Preorder on stores and buffers)**.**

$$\begin{aligned} \epsilon \preceq \epsilon' &\leftrightarrow \mathrm{dom}(\epsilon) \subseteq \mathrm{dom}(\epsilon') \wedge \forall x \in \mathrm{dom}(\epsilon).\epsilon'(x) = \epsilon(x) \\ \mu \preceq \mu' &\leftrightarrow \mathrm{dom}(\mu) \subseteq \mathrm{dom}(\mu') \wedge \forall x \in \mathrm{dom}(\mu).\exists m.\mu'(x) = \mu(x) + +m \end{aligned} \tag{3.3}$$

This definition is from [93].

Definition 3.9 (Halted processes)**.** α is a halted process, i.e. $\alpha \in \mathrm{hprocs}(\epsilon, \mu, P)$ if any of the following hold:

- α's remaining program is `skip` or an error.
- α's next statement is a receive or cond-receive, but there is no matching send or cond-send in the rest of the program.

This definition is from [93].

Definition 3.10 (Restriction of program stores and buffers)**.** $\epsilon|_X$ is the projection of ϵ to the set of variables local to the processes in X. $\mu|_X$ is the projection of μ to the subchannels whose destination process is a process in X. This definition is from [93].

Definition 3.11 (Preorder on halted environments)**.**

$$(\epsilon, \mu, P) \preceq (\epsilon', \mu', P') \leftrightarrow \epsilon|_H \preceq \epsilon'|_H \wedge \mu|_H \preceq \mu'|_H \tag{3.4}$$

Where $H = \mathrm{hprocs}(\epsilon, \mu, P)$. This definition is from [93].

Definition 3.12 (Simulation on environments)**.** $(\epsilon, \mu, P) \sqsubseteq (\epsilon', \mu', P')$ if and only if, for all $(\epsilon, \mu, P) \rightarrow^* (\epsilon_f, \mu_f, P_f)$, there exists $(\epsilon'_f, \mu'_f, P'_f)$, such that $(\epsilon', \mu', P') \rightarrow^* (\epsilon'_f, \mu'_f, P'_f)$ and $(\epsilon_f, \mu_f, P_f) \preceq (\epsilon'_f, \mu'_f, P'_f)$. This definition is from [93].

Definition 3.13 (Simulation on rewrite rules)**.** $\Gamma, \Delta, P \sqsubseteq \Gamma', \Delta; \Delta', P'$ if and only if, for all P_x such that $P \ltimes P_x$ is symmetrically nondeterministic,

$$\forall(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_{\varnothing}.\exists(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_{\varnothing}.(\epsilon, \mu, P \ltimes P_x) \sqsubseteq (\epsilon', \mu', P' \ltimes P_x) \tag{3.5}$$

This definition is from [93].

Definition 3.14 (Left movers)**.** s_1 is a left mover in $(\epsilon, \mu, P|||[s_1; s]_\alpha)$ if and only if

- If s_1 is enabled in $(\epsilon, \mu, P|||[s_1; s]_\alpha)$, and $(\epsilon, \mu, P|||[s_1; s]_\alpha) \rightarrow^* (\epsilon', \mu', P'|||[s_1; s]_\alpha)$, then s_1 is still enabled in $(\epsilon', \mu', P'|||[s_1; s]_\alpha)$.
- If $(\epsilon, \mu, P|||[s_1; s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P'|||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon', \mu', P'|||[s]_\alpha)$ then there exists ϵ_1 and μ_1 such that $(\epsilon, \mu, P|||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon_1, \mu_1, P|||[s]_\alpha) \xrightarrow{\beta} (\epsilon', \mu', P'|||[s]_\alpha)$. That is, s_1 commutes to the left.

This definition is from [93].

3.5.3 Left Movers

Lemma 3.2. cond-send(b, x, t, m) is a left mover in $(\epsilon, \mu, P|||[\text{cond-send}(b, x, t, m); s]_\alpha)$, for all $\epsilon, \mu, P, \alpha, s$.

Proof. The proof is by definition of left movers. cond-send is always enabled when it is the first statement in a process. This satisfies the first condition in the definition of left movers.

Suppose $(\epsilon, \mu, P|||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P_0|||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\alpha} (\epsilon_1, \mu_1, P_0|||[s]_\alpha)$ and $[s_1]_\beta$ is the first statement in β. Statement $[s_1]_\beta$ is enabled at the start. Since cond-send can only push to message queues where the source is α and does not affect any variables, $[s_1]_\beta$ cannot be disabled if cond-send is run first instead.

Let $(\epsilon, \mu, P|||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\alpha} (\epsilon_2, \mu_2, P|||[s]_\alpha) \xrightarrow{\beta} (\epsilon_3, \mu_3, P_0|||[s]_\alpha)$. Now we need to prove that $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Statement $[s_1]_\beta$ can only access and modify variables that are not local to α. cond-send does not modify any variables. $[s_1]_\beta$ may push messages into message queues whose source is not α and may pop messages from queues whose destination is not α. cond-send may only push messages to queues whose source is α. In short, the actions performed by $[s_1]_\beta$ and cond-send do not interfere with each other. Therefore, the changes made by $[s_1]_\beta$ to convert ϵ to ϵ_0 and μ to μ_0 are the same changes as those made by $[s_1]_\beta$ to convert ϵ_2 to ϵ_3 and μ_2 to μ_3. Also, the changes made by cond-send to convert ϵ_0 to ϵ_1 and μ_0 to μ_1 are the same

changes as those made by cond-send to convert ϵ to ϵ_2 and μ to μ_2. Therefore, $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Lemma 3.3. For all $\epsilon, \mu, P, \alpha, s,\ b, y = \text{cond-receive}(x, t)$ is a left mover in $(\epsilon, \mu, P||[b, y = \text{cond-receive}(x, t); s]_\alpha)$ if the subchannel $\mu(\epsilon(x), \alpha, t)$ is not empty.

Proof. The proof is by definition of left movers. Only a statement in process α can pop from the message queue $\mu(\epsilon(x), \alpha, t)$. Running statements from other processes does not affect the message currently at the head of this queue. Therefore cond-receive is enabled even if P is run first. This satisfies the first condition in the definition of left movers.

Suppose $(\epsilon, \mu, P||[b, y = \text{cond-receive}(x, t); s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P_0||[b, y = \text{cond-receive}(x, t); s]_\alpha)$ $\xrightarrow{\alpha} (\epsilon_1, \mu_1, P_0||[s]_\alpha)$ and $[s_1]_\beta$ is the first statement in β. Statement $[s_1]_\beta$ is enabled at the start. Since cond-receive can only affect variables local to α and can only pop from message queues where the destination is α, $[s_1]_\beta$ cannot be disabled if cond-receive is run first instead.

Let $(\epsilon, \mu, P||[b, y = \text{cond-receive}(x, t); s]_\alpha) \xrightarrow{\alpha} (\epsilon_2, \mu_2, P||[s]_\alpha) \xrightarrow{\beta} (\epsilon_3, \mu_3, P_0||[s]_\alpha)$. Now we need to prove that $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Statement $[s_1]_\beta$ can only access and modify variables that are not local to α. cond-receive may only modify variables local to α. $[s_1]_\beta$ may push messages into message queues whose source is not α and may pop messages from queues whose destination is not α. cond-receive may only pop messages from queues whose destination is α. In short, the actions performed by $[s_1]_\beta$ and cond-receive do not interfere with each other. Therefore, the changes made by $[s_1]_\beta$ to convert ϵ to ϵ_0 and μ to μ_0 are the same changes as those made by $[s_1]_\beta$ to convert ϵ_2 to ϵ_3 and μ_2 to μ_3. Also, the changes made by cond-receive to convert ϵ_0 to ϵ_1 and μ_0 to μ_1 are the same changes as those made by cond-receive to convert ϵ to ϵ_2 and μ to μ_2. Therefore, $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Lemma 3.4. If s_1 is a left mover in $(\epsilon, \mu, P||[s_1; s]_\alpha)$ then $(\epsilon, \mu, P||[s_1; s]_\alpha) \sqsubseteq (\epsilon, \mu, s_1; P||[s]_\alpha)$

Proof. The proof analogous to the proof of the same in [93].

3.5.4 Rewrite Rule Soundness

Lemma 3.5. If $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$ then $\Gamma, \Delta, P \sqsubseteq \Gamma', \Delta; \Delta', P'$

Proof. The proof is by induction on the derivation of $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$. Each rewrite rule has a separate case. The proof for all rewrite rules except R-Cond-Send and R-Cond-Receive is analogous to the proof of the same in [93]. The remaining proof is given here.

Case R-Cond-Send: Let $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_{\varnothing}$ and assume $(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_{\alpha} \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$.

since cond-send is a left mover, $(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_{\alpha}; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Suppose $\epsilon(x) = \beta$. By the R-Cond-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +(n : b)]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]_{\epsilon}$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m]$ where m is $\epsilon(n)$ when $\epsilon(b) \neq 0$ or $\varnothing$ when $\epsilon(b) = 0$.

Suppose $\epsilon(b) \neq 0$. Then by semantic rule E-Cond-Send-True,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\epsilon(n)], P_x) \rightarrow^* (\epsilon_f, \mu_f, H) \tag{3.6}$$

Suppose $\epsilon(b) = 0$. Then by semantic rule E-Cond-Send-False,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\varnothing], P_x) \rightarrow^* (\epsilon_f, \mu_f, H) \tag{3.7}$$

that is, $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Therefore, $(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_{\alpha} \ltimes P_x) \sqsubseteq (\epsilon', \mu', P_x)$.

Case R-Cond-Receive: Let $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_{\varnothing}$ and assume

$$(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_{\beta} \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H) \tag{3.8}$$

since cond-receive is a left mover, $(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_{\beta}; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Suppose $\epsilon(x) = \alpha$. By the R-Cond-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.b = \alpha.b'?\ 1 : 0; \beta.y = \alpha.b'?\ \alpha.n : \beta.y]_{\beta}$ when $\text{head}(\Gamma(\alpha, \beta, t)) = (n : b')$. Suppose $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]_{\epsilon}$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$. Further, either $\epsilon' = \epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$ or $\epsilon' = \epsilon[\beta.b \mapsto 0]$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$. Then by semantic rule E-Cond-Receive-True,

$$(\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \rightarrow^* (\epsilon_f, \mu_f, H) \tag{3.9}$$

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$. Then by semantic rule E-Cond-Receive-False,

$$(\epsilon[\beta.b \mapsto 0], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \rightarrow^* (\epsilon_f, \mu_f, H) \tag{3.10}$$

that is, $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Therefore, $(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_{\alpha} \ltimes P_x) \sqsubseteq (\epsilon', \mu', P_x)$.

QED.

3.5.5 Deadlock Freedom

The sequentialized program obtained using the rewrite rules in Figure 4.13 has a single process and does not contain any communication with other processes. As a result, it

$$\text{TR-Val}\ \frac{\texttt{type(n)} = t}{\Theta \vdash n : \text{precise } t} \qquad \text{TR-Var}\ \frac{\Theta(x) = T}{\Theta \vdash x : T} \qquad \text{TR-Iop}\ \frac{\Theta \vdash e_1 : T \qquad \Theta \vdash e_2 : T}{\Theta \vdash e_1\ op\ e_2 : T} \qquad \text{TR-Iop-approx}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t}{\Theta \vdash e_1\ op\ e_2 : \text{approx } t}$$

Figure 3.23: Types for Integer Expressions

cannot deadlock. It follows from Lemma 3.1 that if a parallel program can be completely sequentialized, then it is also deadlock free. Using Lemma 3.1, we get the following proposition:

Proposition 3.1 (Transformations do not introduce deadlocks)**.** If P is the original program, P^A is the approximated program, $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$, and $\varnothing, \varnothing, P^A \rightsquigarrow^* \varnothing, \Delta', \texttt{skip}$, then the approximation itself does not introduce deadlocks in to the program.

3.6 SAFETY ANALYSIS OF PARALLEL PROGRAMS

We show that our system has safety properties that ensure isolation of precise computations and type soundness. We use the sequentialization from Section 3.5 as an important building block for these analyses.

3.6.1 Approximate Type Analysis

We use similar type annotations as in EnerJ [41]. We use type qualifiers to explicitly specify data that may be subject to approximations. We use a static type environment $\Theta : Var \mapsto T$ that maps variables to their type to check type-safety of statements. We use two type judgments – (1) expressions are assigned a type, $\Theta \vdash e : T$ (Figure 3.23), and (2) statements update the type environment, $\Theta \vdash S : \Theta'$ (Figure 3.24).

3.6.2 Non-Interference

We show that the Parallely type system ensures non-interference between approximate data and precise data. Non-interference states that the results of precise data that need to be isolated from approximations will not be affected by changes in the approximate data. The *approx* type qualifiers are used to mark the data that can handle approximations and the type system guarantees non-interference.

The type system rules ensure that this property holds. Parallely only allows `cond-receive` statements to update approximate type variables (TR-CondReceive), therefore all `cond-send`

TR-Skip
$\Theta \vdash \texttt{skip} : \Theta$

TR-Var
$$\frac{\Theta \vdash x : T \;\; \Theta \vdash e : \Theta}{\Theta \vdash \text{x = e} : \Theta}$$

TR-Var2
$$\frac{\Theta \vdash x : \text{approx } t \qquad \Theta \vdash e : q\ t}{\Theta \vdash \text{x = e} : \Theta}$$

TR-Prob
$$\frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \text{approx } t}{\Theta \vdash \text{x = } e_1\ [p]\ e_2 : \Theta}$$

TR-ApproxAssign
$$\frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \text{approx } t \qquad \Theta \vdash b : q''\ \texttt{int}}{\Theta \vdash \text{x = } e_1\ [b]\ e_2 : \Theta}$$

TR-SEQ
$$\frac{\Theta \vdash s_1 : \Theta \qquad \Theta \vdash s_2 : \Theta}{\Theta \vdash s_1; s_2 : \Theta}$$

TR-If
$$\frac{\Theta \vdash b : \text{precise } \texttt{int}}{\Theta \vdash \texttt{if}\ b\ s_1\ s_2 : \Theta}$$

TR-Array-load
$$\frac{\Theta \vdash e_1 : \text{precise } \texttt{int} \;...\; \Theta \vdash e_k : \text{precise } \texttt{int} \qquad \Theta \vdash a : q\ t[] \qquad \Theta \vdash x : q\ t}{\Theta \vdash x = a\texttt{[}e_1...e_k\texttt{]} : \Theta}$$

TR-Array-load2
$$\frac{\Theta \vdash e_1 : \text{precise } \texttt{int} \;...\; \Theta \vdash e_k : \text{precise } \texttt{int} \qquad \Theta \vdash a : q\ t[] \qquad \Theta \vdash x : \text{approx } t}{\Theta \vdash x = a\texttt{[}e_1...e_k\texttt{]} : \Theta}$$

TR-Array-store
$$\frac{\Theta \vdash e_1 : \text{precise } \texttt{int} \;...\; \Theta \vdash e_k : \text{precise } \texttt{int} \qquad \Theta \vdash a : q\ t[] \qquad \Theta \vdash e : q\ t}{\Theta \vdash a\texttt{[}e_1...e_k\texttt{]} = \text{e} : \Theta}$$

TR-Array-store2
$$\frac{\Theta \vdash e_1 : \text{precise } \texttt{int} \;...\; \Theta \vdash e_k : \text{precise } \texttt{int} \qquad \Theta \vdash a : \text{approx } t[] \qquad \Theta \vdash e : q\ t}{\Theta \vdash a\texttt{[}e_1...e_k\texttt{]} = \text{e} : \Theta}$$

TR-Send
$$\frac{\Theta \vdash y : T}{\Theta \vdash \text{send}(q, T, y) : \Theta}$$

TR-Receive
$$\frac{\Theta \vdash x : T}{\Theta \vdash x = \text{receive}(q, T) : \Theta}$$

TR-CondSend
$$\frac{\Theta \vdash b : \text{precise } \texttt{int} \qquad \Theta \vdash y : \text{approx } t \qquad T = \text{approx } t}{\Theta \vdash \text{cond-send}(b, q, T, y) : \Theta}$$

TR-CondReceive
$$\frac{\Theta \vdash x : \text{approx } t \qquad T = \text{approx } t \qquad \Theta \vdash b : \text{approx } \texttt{int}}{\Theta \vdash b, x = \text{cond-receive}(q, T) : \Theta}$$

TR-Cast
$$\frac{\Theta \vdash x : \text{approx } t \qquad \Theta \vdash e : q'\ t}{\Theta \vdash \text{x = (q t) e} : \Theta}$$

Figure 3.24: Type Rules for Statements

statements can only communicate approximate type data (TR-CondSend). In addition, probabilistic choice statements can only update approximate type data (TR-Prob). Remaining rules are similar to those in EnerJ.

To prove this property for parallel programs, we start by defining non-interference for individual processes using the small-step semantic relation similar to EnerJ. As all communication channels in Parallely are typed, only precise data from a process can affect precise data in another process through communication. Therefore, by proving non-interference in each individual process, we can prove global non-interference.

We use $\cong$ to denote equivalence of frames, heaps, and channels limited to precise typed sections. For primitive values, $v \cong v'$ iff they have the same type $q\ T$ and either q is approx or $v = v'$. For heaps $h \cong h'$ iff they have the same set of addresses M and $\forall m \in M.\ h(m) \cong h'(m)$ (Similarly for the frames, stacks). We use the same definition for channels, $\mu \cong \mu'$ if $\text{domain}(\mu) = \text{domain}(\mu')$ and $\forall (p, q, \text{precise } t) \in \text{domain}(\mu)\ \mu[(p, q, \text{precise } t)] = \mu'(p, q, \text{precise } t)$

Non-interference property states that starting at equivalent environments and executing a program will lead to equivalent final states regardless of differences in approximate data.

Theorem 3.1 (parallel non-interference)**.** Suppose $\Theta \vdash P_i \parallel P_j : T.\ \forall \epsilon, \epsilon', \epsilon_f \in Env$ and $\mu, \mu', \mu_f \in Channel$, s.t., $\langle \epsilon, \mu \rangle \cong \langle \epsilon', \mu' \rangle$, if $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f, \mu_f, P_i' \parallel P_j)$, then there exists $\epsilon_f' \in Env$ and $\mu_f' \in Channel$ such that $(\epsilon', \mu', P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f', \mu_f', P_i' \parallel P_j)$, and $\langle \epsilon_f, \mu_f \rangle \cong \langle \epsilon_f', \mu_f' \rangle$.

The proof of this property is analogous to the proofs of non interference in information flow security for parallel programs [94]. We first show that the property holds for each process and then we extend the property to the distributed setting.

3.6.3 Sequential Non-Interference

If $\Theta \vdash s : \Theta,\ \langle s,\ \sigma_s, h_s, \mu_s \rangle \rightharpoonup_\psi \langle s',\ \sigma_f, h_f, \mu_f \rangle$, $\sigma_s \cong \sigma_s'$, $h_s \cong h_s'$, and $\mu_s \cong \mu_s'$ Then there exists, $(\sigma_f',\ h_f',\ \mu_f')$ s.t. $\langle s,\ \sigma_s', h_s', \mu_s' \rangle \rightharpoonup_\psi \langle s',\ \sigma_f', h_f', \mu_f' \rangle$ and $\sigma_f \cong \sigma_f'$, $h_f \cong h_f'$, and $\mu_s \cong \mu_s'$

Proof. We will use rule induction on the semantics. The proof is broken into several cases. Case E-ASSIGN-R: The environment is not modified. So, $h_s = h_f$ and $h_s' = h_f'$. As, $h_s \cong h_s'$ we can trivially say $h_f \cong h_f'$. Same argument holds for the stack and mail box.

Case E-ASSIGN-C: The stack and the channel does not change. From assumption $\Theta \vdash x = n : \Theta$, therefore either both x and n both have the same type, or $\Theta \vdash x : \text{approx } t$ and $\Theta \vdash n : q\ t$. In the first case, If both x and n are approx then, $h_s \cong h_f$ by definition as only

the approximate value changes and the property holds. If both values are precise they will be the same in h_s and h'_s and we can take the same step. In the second case, x is approx and n is precise. Again in this case only approx values in the heap will change.

Case E-Assign-Prob-True and E-Assign-Prob-False: The type rule `TR-Prob` ensures that the assigned variable is approximate. Therefore only the approximate regions of the environment changes and the property holds.

Case E-SEQ-R1, E-SEQ-R2: Follows directly from the inductive hypothesis.

Case E-If, E-If-True, E-If-False: In the case of rule `E-If`, The environment does not change and the property is satisfied trivially and since $\Theta \vdash$ `if` $b\ s_1\ s_2 : \Theta$ we know $\Theta \vdash b :$ precise `int` therefore, the guard evaluates to the same value in both $\langle \sigma_s, h_s, \mu_s \rangle$ and $\langle \sigma'_s, h'_s, \mu'_s \rangle$ and takes the same branch. In the case of rules `E-If-True` and `E-If-False` the property follows from the inductive hypothesis.

Case E-ARRAY-LOAD-IDX, E-ARRAY-LOAD-C: In E-ARRAY-LOAD-IDX the environment does not change. As $\Theta \vdash$ x `=` a`[`$e_1 ... e_k$`]` $: \Theta$. All e_i has precise type. Therefore all array indices will evaluate to the same value in $\langle \sigma'_s, h'_s, \mu'_s \rangle$. In addition if $\Theta \vdash x :$ approx t then $h_f \cong h_s$ and the property holds.
Similarly, if $\Theta \vdash x :$ precise t then $\Theta \vdash a :$ precise t and the resultant value n will be the same in both h_s and h'_s resulting in the same update to heap.

Case E-ARRAY-STORE-IDX, E-Array-Store-C: As $\Theta \vdash$ a`[`$e_1 ... e_k$`]` `=` x $: \Theta$. As in the previous case, all e_i has precise type. Therefore all array indices will evaluate to the same value in $\langle \sigma'_s, h'_s, \mu'_s \rangle$. In addition if $\Theta \vdash x :$ approx t, then $\Theta \vdash a :$ approx t and $h_f \cong h_s$ and the property holds.
Similarly, if $\Theta \vdash x :$ precise t then $\Theta \vdash x :$ precise t and the resultant value n will be the same in both h_s and h'_s resulting in the same update to heap.

Case E-SEND: As $\Theta \vdash \text{send}(Pid, T, v) : \Theta$, v has the type T. If T is a approx type only the approximate part of the mailbox changes and the property holds. If T is precise, type safety also ensures that v has precise type and will evaluate to the same value under $\langle \sigma'_s, h'_s, \mu'_s \rangle$ resulting in a equivalent environment.

Case E-Receive: As $\Theta \vdash \text{receive}(Pid, T) : \Theta$, x has some type T. If x is an approx type only the approximate part of the mailbox and environment changes and the property holds. If T is precise, the precise part of the mailbox is accessed, and the value in $\langle \sigma'_s, h'_s, \mu'_s \rangle$ will be the same, therefore the final environment will be the same for any equivalent start environment.

Case E-CONDRECEIVE-TRUE, E-CONDRECEIVE-FALSE: The behavior of conditional communication through the E-CONDRECEIVE-TRUE rule is similar to E-Receive. But since $\Theta \vdash x$ = cond-receive$(Pid, T) : \Theta$, $T =$ approx t. Therefore the only change in the channel is to $\mu(\alpha, \beta,$ approx $t)$ therefore $\mu_s \cong \mu_f$. Similarly, since $\Theta \vdash x :$ approx$t$, $h_s \cong h_f$. Same argument for E-CONDRECEIVE-FALSE. In this case, the content in the heap and the stack does not change.

Case E-Cast-R and E-Cast-E: The type rule `TR-Cast` ensures that the assigned variable is approximate. Therefore only the approximate regions of the environment changes and the property holds.

We have shown that the property is satisfied by all possible statements.

3.6.4 Distributed Noninterference

While type checking individual processes allows us to prove non-interference between approximate and precise data, we need further checks to show that inter-process interactions don't cause deadlocks. Consider the two processes in Figure 3.25. While each process would pass our type checker (i.e., demonstrate non-interference), the program would deadlock as the two messaging channels do not match. The type of variable `x` in process β needs to be approximate for this program to function correctly. By incorporating canonical sequentialization to our safety analysis we can catch such bugs as the program would fail to sequentialize.

```
[ approx t n;                                  ]_α  ||  [ precise t x;                          ]_β
[ n = n [r] randVal(approx t);                 ]       [ x = receive(α, precise t);             ]
[ send(β, approx t, n);                        ]
```

Figure 3.25: An Example Program with Incorrect Type Annotations

Concurrent Non-interference says that if each individual process in the program was well typed then the parallel program has similar noninterference property guaranteed.

Theorem 3.2 (Parallel Non-Interference)**.** Suppose $P_i \parallel P_j$ is well typed under Θ and $\forall \epsilon, \epsilon', \epsilon_f \in Env$ and $\mu, \mu, \mu_f \in Channel$, such that, $\langle \epsilon, \mu \rangle \cong \langle \epsilon', \mu' \rangle$,
if $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f, \mu_f, P_i' \parallel P_j)$, then there exists $\epsilon_f' \in Env$ and $\mu_f' \in Channel$ such that $(\epsilon', \mu', P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f', \mu_f', P_i' \parallel P_j)$ and $\langle \epsilon_f, \mu_f \rangle \cong \langle \epsilon_f', \mu_f' \rangle$

Proof: If $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f, \mu_f, P_i' \parallel P_j)$ then from the semantics of global execution we can see that there exist i such that $\epsilon[i] = \langle \sigma, h \rangle$ and $\langle P_i,\ \sigma, h, \mu \rangle \rightharpoonup \langle P_i',\ \sigma_f, h_f, \mu_f \rangle$.

From sequential non-interference we know that For any $\sigma' \cong \sigma$, $h' \cong h$, and $\mu' \cong \mu$ there exists $(\sigma'_f,\ h'_f,\ \mu'_f)$ s.t. $\langle s,\ \sigma', h', \mu' \rangle \rightharpoonup \langle s',\ \sigma'_f, h'_f, \mu'_f \rangle$ and $\sigma_f \cong \sigma'_f$, $h_f \cong h'_f$, and $\mu \cong \mu'_f$ So we can consider ϵ', where $\epsilon' = \epsilon[i \mapsto \langle \sigma', h' \rangle]$ Consider the same transition as before, We will end up at $\epsilon'_f = \epsilon_f[i \mapsto \langle \sigma'_f, h'_f \rangle]$. $\sigma'_f \cong \sigma_f$, $h'_f \cong h_f$, therefore $\epsilon'_f \cong \epsilon_f$

3.6.5 Type Soundness

Lemma 3.6 (The type system is sound for individual processes.)**.** For a single process, assuming there are no deadlocks, if $\Theta \vdash s : \Theta'$, then either $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle \texttt{skip}, \sigma', h', \mu' \rangle$ or $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle s', \sigma', h', \mu' \rangle$ and $\Theta' \vdash s' : \Theta''$

(proof sketch). We prove this property by induction on the typing rules. Since we assume there are no deadlocks all processes would be eventually scheduled and the statement will be executed. Most statements evaluate to skip in a single step and the proof is straightforward. We will present several cases the proof of the remaining cases are similar.

Case: $x = e$: If $\langle x = e, \sigma, h, \mu \rangle \rightharpoonup \langle x = e', \sigma, h, \mu \rangle$, we know from the subject reduction lemma for expressions that $\Theta \vdash e' : T$. Therefore if $\Theta \vdash s : \Theta$, then either $\Theta \vdash x : T$ and $\Theta \vdash e : T$ in which case the property holds as $\Theta \vdash e' : T$ or $\Theta \vdash x : \text{approx } t$ and $\Theta \vdash e : q\ t$ which again will be satisfied as $\Theta \vdash e' : q\ t$.

Case: send(q, T, x): Consider send statements send(q, T, x), send statements are always enabled and will evaluate to `skip`.

Case: x= receive(q, T): Receive statements will eventually get enabled as we assume there are no deadlocks and evaluate to `skip`.

Case: x=e_1 $[r]$ e_2: Probabilistic choice statements are always enabled as we assume there are no deadlocks and evaluate to either $x = e_1$ or $x = e_2$. From the assumption we know that $\Theta \vdash \text{x = } e_1\ [p]\ e_2 : \text{approx } t$ based on type rule `TR-Prob`. Therefore, $\Theta \vdash x : \text{approx } t$ and we can say that $\Theta \vdash \text{x = } e_1 : \text{approx } t$ and $\Theta \vdash \text{x = } e_2 : \text{approx } t$ based on `TR-Var2`

The remaining cases are similar. QED.

Theorem 3.3 (The type system is sound.)**.** If $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$ and $\Theta \vdash P : \Theta'$, then either $(\cdot, \cdot, P) \longrightarrow_\psi (\cdot, \cdot, \texttt{skip})$ *or* $(\cdot, \cdot, P) \longrightarrow_\psi (\cdot, \cdot, P')$ and $\Theta \vdash P' : \Theta'$

Proof. (Sketch) As the program P can be sequentialized, there are no deadlocks (Lemma 3.1). Therefore, there exists at least one individual process that is enabled and can take a step (i.e. the program makes progress). From Lemma 3.6 we know that this step will preserve the type of the statement and therefore the entire program will remain well typed. QED.

3.6.6 Relative Safety

Finally, we show how our approach can be used to prove relative safety of approximations. Relative safety allows us to transfer the reasoning about the safety of the original program to the approximated program [44]. If the approximate transformation maintains relative safety, and the original program satisfies a property, then the transformed program also satisfies that property. For instance, if the original program has no array out of bound errors, and the approximation satisfies relative safety, then the approximate program is also has no array out of bound errors.

Definition 3.15 (Process-Local Relative safety [88])**.** Let P be a program and P^A be the approximate program obtained by transforming P. The programs are relatively safe if when $(\epsilon, \mu, P^A) \rightarrow^* (\epsilon_t, \mu_t, \text{assert}(e); \cdot)$ there exists ϵ_o s.t. $(\epsilon, \mu, P) \rightarrow^* (\epsilon_o, \mu_o, \text{assert}(e); \cdot)$ and $\forall x \in \text{free}(e) \cdot \epsilon_t(x) = \epsilon_o(x)$.

It states that if the approximate program satisfies (or does not satisfy) the property e, then there must exist an execution in the original program that satisfies (does not satisfy) the same property at the same program point. Therefore, if the assert statement is valid in the original program (i.e., its condition always evaluates to true), then it must also be valid in the transformed program. We can extend it to parallel computations: if any process-local safety properties are satisfied by the sequentialized program in its halting states, then they are also satisfied by the parallel program in its halting states, since according to Lemma 3.1 the sequentialized program is an over-approximation of the parallel program with respect to halted processes. We can immediately state the following proposition as a consequence of Lemma 3.1.

Proposition 3.2 (Relative safety of transformations via sequentialization)**.** If P is the original program, P^A is the program after applying some approximation, $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$, $\varnothing, \varnothing, P^A \rightsquigarrow^* \varnothing, \Delta', \texttt{skip}$, and a process-local safety property holds on the halting states of both Δ and Δ', then the same safety property holds on the halting states of P and P^A.

Therefore, we can use the sequentialized programs to prove the relative safety of approximations and that the original parallel programs will also satisfy relative safety.

3.7 RELIABILITY AND ACCURACY ANALYSIS OF PARALLEL PROGRAMS

In this section we define syntax for specifying reliability and accuracy requirements and show that we can generate guarantees on the parallel program by analyzing the canonical sequentializations.

3.7.1 Reliability Analysis – Semantic Foundations and Conditions

Reliability Predicates. Parallely can generate reliability predicates that characterize the reliability of a approximate program. A reliability predicate Q has the following form:

$$Q := R_f \leq R_f \mid Q \wedge Q \tag{3.11}$$

$$R_f := r \mid \mathcal{R}(O) \mid r \cdot \mathcal{R}(O) \tag{3.12}$$

A predicate can be a conjunction of predicates or a comparison between *reliability factors*. A reliability factor is either a rational number r, a *joint reliability factor*, or a product of a number and a *joint reliability factor*. A reliability factor represents the probability that an approximate execution has the same values as the original execution for all variables in the set $O \subseteq \text{Var}$. By definition, $\mathcal{R}(\{\}) = 1$ (i.e., an empty set of variables has a reliability of 1).

For example, we can specify the constraint that the reliability of some variable x be higher than the constant 0.99 using the reliability predicate $0.99 \leq \mathcal{R}(\{x\})$. Intuitively, $\mathcal{R}(\{x\})$ refers to the probability that a approximate execution of the program has the same value for variable x as the exact execution of the program. A *joint reliability factor* such as $\mathcal{R}(\{x, y\})$ refers to the probability that both x and y have the same value.

We now define the semantics of reliability factors by following the exposition in [42]. The denotation of a reliability factor $[\![R_F]\!] \in \mathcal{P}(Env \times \Phi)$ is the set of environment and environment distribution pairs that satisfy the predicate. An environment distribution $\phi \in \Phi = Env \mapsto \mathbb{R}$ is a probability distribution over possible approximate environments. For example, $[\![r]\!](\epsilon, \varphi) = r$. The denotation of $\mathcal{R}(O)$ is the probability that an environment ϵ_a sampled from Φ has the same value for all variables in O as the environment ϵ:

$$[\![\mathcal{R}(O)]\!](\epsilon, \varphi) = \sum_{\epsilon_u \in \mathcal{E}(O,\epsilon)} \varphi(\epsilon_u) \tag{3.13}$$

where, $\mathcal{E}(O, \epsilon)$ is the set of all environments in which the values of O are the same as in ϵ (which we express with the predicate *equiv*, formally defined in [42, Section 5]):

$$\mathcal{E}(O, \epsilon) = \{\epsilon' \mid \epsilon' \in Env \wedge \forall v. v \in O \Rightarrow \mathit{equiv}(\epsilon, \epsilon', v)\} \tag{3.14}$$

Paired Execution Semantics. For reliability and accuracy analysis we define a *paired execution semantics* that couples an original execution of a program with an approximate execution, following the definition from Rely.

Definition 3.16 (Paired Execution Semantics [42]). $\langle\ s\ ,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle$ such that $\langle\ s\ ,\ \epsilon\rangle \Longrightarrow_{1_\psi} \epsilon'$ and $\varphi'(\epsilon'_a) = \sum_{\epsilon_a \in \mathrm{E}} \varphi(\epsilon_a) \cdot p_a$ where $\langle\ s\ ,\ \epsilon_a\rangle \overset{\cdot, p_a}{\Longrightarrow}_\psi \epsilon'_a$

This relation states that from a configuration $\langle \epsilon,\ \varphi\rangle$ consisting of an environment ϵ and an *environment distribution* $\varphi \in \Phi$, the paired execution yields a new configuration $\langle \epsilon',\ \varphi'\rangle$. The execution reaches the environment ϵ' from the environment ϵ with probability 1 (expressed by the deterministic execution, 1_ψ). The environment distributions φ and φ' are probability mass functions that map an environment to the probability that the execution is in that environment. In particular, φ is a distribution on environments before the execution of s whereas φ' is the distribution on environments after executing s.

Reliability Transformer. Reliability predicates and the semantics of programs are connected through the view of a program as a reliability transformer.

Definition 3.17 (Reliability Transformer Relation [42]).

$$\psi \models \{Q_{pre}\}\ s\ \{Q_{post}\} \equiv \forall \epsilon, \varphi, \epsilon', \varphi'.\ (\epsilon, \varphi) \in [\![Q_{pre}]\!] \implies \langle s,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!] \tag{3.15}$$

Similar to the standard Hoare triple relation, if an environment and distribution pair $\langle \epsilon,\ \varphi\rangle$ satisfy a reliability predicate Q_{pre}, then the program's paired execution transforms them into a new pair $\langle \epsilon',\ \varphi'\rangle$ that satisfy a predicate Q_{post}.

Conditions for Analysis. For our reliability analysis to work, we require that the transformed approximate program P^A evaluated under exact execution semantics (1_ψ) is equivalent to the original program execution. We also need to ensure that the sequentialized program is equivalent (not just an over-approximation) in behavior to the parallel program. To achieve this property for our rewrite rules we removed sources of over-approximation from the language (e.g., Parallely does not support communication with external processes or wildcard receives from [87]).

We perform the following transformations to simplify the analysis process: (1) we transform the program into its Single Static Assignment form; (2) we ensure that variable sets used in individual processes are disjoint by simple renaming; (3) we unroll finite loops; (4) we do conditional flattening as in Rely; and (5) we do not allow send and receive operations inside conditionals.

3.7.2 Reliability Analysis – Precondition Transformer

Given a reliability predicate that should hold after the execution of the program, the precondition generation produces a predicate that should hold before the execution of the program. The precondition generator starts at the end of the sequential program and successively builds preconditions, traversing the program backwards, and finally builds the precondition at the start of the program.

Substitution. We define substitution for reliability predicates the same way as [42]. A substitution $e_0[e_2/e_1]$ replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. The substitution matches set patterns. For instance, the pattern $R(\{x\} \cup X)$ represents a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. The substitution $r_1 \cdot \mathcal{R}(\{x, z\})[\mathcal{R}(\{y\} \cup X)/\mathcal{R}(\{x\} \cup X)]$ results in $r_1 \cdot \mathcal{R}(\{y, z\})$.

Precondition Generator. The reliability precondition generator function $C \in S \times Q \mapsto Q$ takes as inputs a statement and a postcondition and produces a precondition as output. The analysis rules for instructions are the same as in Rely. We add the following rules to handle the new probabilistic choice and cast statements in Parallely:

$$C(\, x = e,\, Q\,) = Q\,[\,\mathcal{R}(\,\rho(e) \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \tag{3.16}$$

$$C(\, x = e_1\ [r]\ e_2,\, Q\,) = Q\,[\,r \cdot \mathcal{R}(\,\rho(e_1) \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \tag{3.17}$$

$$C(\, x = b?\ e_1 : e_2,\, Q\,) = Q\,[\,\mathcal{R}(\,\rho(e_1) \cup \{b\} \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \tag{3.18}$$

$$C(\, x = (T)\ y,\, Q\,) = Q\,[\,0\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \tag{3.19}$$

where $\rho(e)$ specifies the set of variables referred to by e. Intuitively, for simple assignment of an expression, any reliability specification containing x is updated such that x is replaced by the variables occurring in e. For probabilistic assignment, the reliability of x is equal to r times the reliability of variables occurring in e_1. For conditional assignment, where b is an integer variable, the reliability of x is equal to the reliability of variables occurring in e_1 and b; recall that e_1 is expected to be equivalent to the expression from the original program. Casting, which changes the precision of the variables causes the reliability of any variable to be 0, since in general reduced-precision values are not equal to the original values. The remaining rules are analogous to those from [42, Section 5].

Figure 3.26 presents the results of the precondition generation for the sequential program given in Figure 4.11 (sequentialization of the example program from Section 3.3.3). Given the

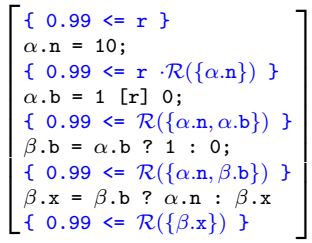


Figure 3.26: An Example of the Reliability Precondition Generation.

post condition $0.99 \leq \mathcal{R}(\{\beta.x\})$, which states that the reliability of β.x needs to be higher than 0.99, the precondition generation results in $0.99 \leq$ r. Solving these constraints is done via a subsumption-based decision procedure from Rely.

Our goal is to analyze the sequential version of the program and know that the generated reliability constraints are valid for the parallel program. In the following section we will discuss how to accomplish this goal through sequentialization.

3.7.3 Reliability Analysis via Canonical Sequentialization

In this section we will prove the following theorem stating that reliability transformer relations defined on a sequentialized approximate program (P^A_{seq}) will also hold on the original parallel approximate program (P^A).

Theorem 3.4 (Reliability Analysis Soundness)**.** If a program with approximation P^A (obtained by transforming the program P) can be sequentialized then the reliability analysis of the sequentialized program P^A_{seq} will also be valid for the parallel approximate program. Therefore, if $\varnothing, \varnothing, P^A \rightsquigarrow \varnothing, P^A_{seq}, \texttt{skip}$ then $\psi \models \{Q_{pre}\}\ P^A_{seq}\ \{Q_{post}\} \implies \psi \models \{Q_{pre}\}\ P^A\ \{Q_{post}\}$

We prove the theorem above in several steps. First, we show that any environment reached by the sequentialized Parallely program can be reached in the original parallel program (Lemma 3.7). Second, we use that result to prove that the original parallel program and the canonical sequentialization are equivalent (Lemma 3.9). Third, we show that approximate executions in the parallel program have equivalent executions in the sequential program (Lemma 3.10). Finally, we show that any paired execution of the parallel program has an equivalent execution in the sequentialized version (Lemma 3.11).

Our first step proves that any environment reached by the sequentialized program can be reached in the original parallel program. This is the converse of Lemma 3.1. Together these two lemmas allow us to establish that the two programs are equivalent.

Lemma 3.7. Let P be a program in normal form. If

- $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$
- $P \circ P_0$ is symmetrically nondeterministic for some extension P_0
- $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]$ such that $(\epsilon', \mu', P' \circ P_0) \rightarrow^* (\epsilon_{F'}, \mu_{F'}, F')$

there exists $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ such that $(\epsilon, \mu, P \circ P_0) \rightarrow^* (\epsilon_F, \mu_F, F)$ and $\epsilon_F|_H = \epsilon_{F'}|_H$ where $H = \text{halted}(\epsilon'_F, \mu'_F, F')$.

We prove Lemma 3.7 by induction on the derivation of Δ from P, splitting into multiple cases based on the rewriting rule. We will first prove the following lemma, which will be used in the proof of Lemma 3.7.

Lemma 3.8. If s_1 is a left mover in $(\epsilon, \mu, P||[s_1; s]_\alpha)$ then $(\epsilon, \mu, P||[s_1; s]_\alpha) \sqsupseteq (\epsilon, \mu, s_1; P||[s]_\alpha)$

Suppose $(\epsilon, \mu, s_1; P||[s]_\alpha) \rightarrow^* (\epsilon', \mu', P')$. We need to show that there exists (ϵ'', μ'', P'') such that $(\epsilon, \mu, P||[s_1; s]_\alpha) \rightarrow^* (\epsilon'', \mu'', P'')$ and $(\epsilon', \mu', P') \preceq (\epsilon'', \mu'', P'')$. The first statement that must be executed from $(s_1; P||[s]_\alpha)$ is s_1. Let $(\epsilon, \mu, s_1; P||[s]_\alpha) \xrightarrow{\alpha} (\epsilon_{s_1}, \mu_{s_1}, P||[s]_\alpha)$. If s_1 is also the first statement executed from $(P||[s_1; s]_\alpha)$, then we get $(\epsilon, \mu, P||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon_{s_1}, \mu_{s_1}, P||[s]_\alpha)$. From this point, both programs have the exact same behavior, hence the lemma is proved. QED.

Proof of Lemma 3.7: the proof is split into multiple cases depending on the rewrite rule.

Case R-Send: Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$ By the R-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +n]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n]$.
Therefore, $(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n], P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
By semantic rule E-Send, $(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha; P_x) \xrightarrow{\alpha} (\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n], P_x)$
Therefore, $(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
Since send is a left mover, $(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Case R-Receive: Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$ By the R-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.y = \alpha.n]_\beta$ when $\text{head}(\Gamma(\alpha, \beta, t)) = n$.
Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon[\beta.y \mapsto \alpha.n]$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$.
Therefore, $(\epsilon[\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$.
From E-Receive, $(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon[\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x)$.
Therefore, $(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Since receive is a left mover, $(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Case R-Cond-Send: Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
By the R-Cond-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +(n : b)]$ and $\Delta' = \texttt{skip}$.
Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m]$ where m is $\epsilon(n)$ when $\epsilon'(b) \neq 0$ or $\varnothing$ when $\epsilon'(b) = 0$.
Therefore, $(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m], P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
by semantic rule E-Cond-Send-True or E-Cond-Send-False (depending on m),

$$(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha; P_x) \xrightarrow{\alpha} (\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m], P_x) \tag{3.20}$$

Therefore, $(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$.
Since cond-send is a left mover, $(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$

Case R-Cond-Receive: Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
By the R-Cond-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and
$\Delta' = [\beta.b = \alpha.b'?\ 1 : 0; \beta.y = \alpha.b'?\ \alpha.n : \beta.y]_\beta$ when $\text{head}(\Gamma(\alpha, \beta, t)) = (n : b')$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$.
Further, either $\epsilon' = \epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$ or $\epsilon' = \epsilon[\beta.b \mapsto 0]$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.
Suppose $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$,
Then,$(\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
Suppose $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$. Then, $(\epsilon[\beta.b \mapsto 0], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$. by semantic rule E-Cond-Receive-True or E-Cond-Receive-False,
$(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x)$
Therefore, $(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$
Since cond-receive is a left mover, $(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, H)$.

Case R-Context: Proof is by application of the inductive hypothesis over rewrite rules.

Case R-Congruence: By definition, $A \equiv B$ if and only if the set of program traces in A is equivalent to the program traces in B.

Case R-If-Then and R-If-Else: Since we do not allow communication inside conditionals, the rewritten program is congruent to the original program.
We have shown that the property is satisfied by all possible statements. QED.

Lemma 3.9 (equivalence of sequentialized program for halted states)**.** If $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$ then $(\varnothing, \varnothing, P) \rightarrow^* (\epsilon_H, \varnothing, H)$ if and only if $(\varnothing, \varnothing, \Delta) \rightarrow^* (\epsilon'_H, \varnothing, H')$ such that $\epsilon_H = \epsilon'_H$ where all processes are permanently halted in $(\epsilon_H, \varnothing, H)$.

Proof. The proof of Lemma 3.9 is by using Lemma 3.1 and Lemma 3.7, which state that the sequentialized program is an over-approximation of the parallel program and that the parallel program is an over-approximation of the sequential program respectively (with respect to halted processes). Thus, the sequentialized program is equivalent to the parallel program with respect to halted processes.

By applying Lemma 3.1, we know that $\forall(\epsilon_0, \mu_0) \in [\![\varnothing, \varnothing]\!]_\varnothing, \exists(\epsilon_1, \mu_1) \in [\![\Delta, \varnothing]\!]_\varnothing$ such that whenever $(\epsilon_0, \mu_0, P) \rightarrow^* (\epsilon, \mu_f, P_0)$ then there exists $(\epsilon', \mu'_f, P'_0)$ such that $(\epsilon_1, \mu_1, \texttt{skip}) \rightarrow^* (\epsilon', \mu'_f, P'_0)$ and $\epsilon|_H = \epsilon'|_H$ where $H = \text{halted}(\epsilon, \mu_f, P_0)$. By unifying variables, we get that if $(\varnothing, \varnothing, P) \rightarrow^* (\epsilon, \varnothing, \texttt{skip})$ then $(\varnothing, \varnothing, \Delta) \rightarrow^* (\epsilon, \varnothing, \texttt{skip})$.

Similarly, By applying Lemma 3.7, we know that the inverse holds true.

In the following lemma we show that approximations have the same behaviors on the parallel and the sequentialized programs:

Lemma 3.10 (Aggregate Semantics equivalence)**.** If $\varnothing, \varnothing, P \rightsquigarrow \varnothing, \Delta, \texttt{skip}$, $\langle P,\ \epsilon\rangle \overset{p}{\Longrightarrow}_\psi \epsilon'$ if and only if $\langle \Delta,\ \epsilon\rangle \overset{p}{\Longrightarrow}_\psi \epsilon'$

Proof. (Sketch) From rewrite soundness lemma (Lemma 3.9), we know that $(S, \varnothing, \epsilon) \rightarrow^*_\psi (\texttt{skip}, \varnothing, \epsilon')$ if and only if $(\Delta, \varnothing, \epsilon) \rightarrow^*_\psi (\texttt{skip}, \varnothing, \epsilon')$. Probabilistic differences in executions only appear in Parallely programs in the form of probabilistic choice statements and cast statements. All such statements are sequentialized without any change and added straight to the sequential prefix through the `R-Context` rule.

In addition, all of the probabilistic transitions we model are independent of the execution environment. Therefore, aggregated over all possible schedules, the probability of reaching the same final environment will be the same for the two versions of the program. QED.

Finally, the next lemma states that sequentialization preserves the paired execution relation.

Lemma 3.11 (Rewrites preserve paired executions)**.** If $\varnothing, \varnothing, P \rightsquigarrow \varnothing, \Delta, \texttt{skip}$ then
$\langle\ \Delta\ ,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle \iff \langle\ P\ ,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle$

Proof. From Lemma 3.9, $\langle \epsilon,\ \varnothing,\ \ \Delta\ \rangle \overset{1}{\Longrightarrow}_{1_\psi} \epsilon'$ iff $\langle \epsilon_a,\ \varnothing,\ \ P\ \rangle \overset{1}{\Longrightarrow}_{1_\psi} \epsilon'$. In addition, from Lemma 3.10, $\langle \epsilon_a,\ \varnothing,\ \ \Delta\ \rangle \overset{p_a}{\Longrightarrow}_\psi \epsilon'_a$ iff $\langle \epsilon_a,\ \varnothing,\ \ P\ \rangle \overset{p_a}{\Longrightarrow}_\psi \epsilon'_a$. Therefore, $\sum_{\epsilon_u \in \mathcal{E}(O,\epsilon)} \varphi(\epsilon_u) \cdot p_a$ is the same for both versions of the program, leading to the same distributions. QED.

We can now use these lemmas to prove our main theorem:

Proof of Theorem 3.4. Since $\psi \models \{Q_{pre}\}\ P^A_{seq}\ \{Q_{post}\}$, we know that, $\forall \langle \epsilon,\ \varphi\rangle \in [\![Q_{pre}]\!]$, $\langle P^A_{seq},\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$.
Lemma 3.11 states that both P^A and P^A_{seq} have the same paired execution behavior.
Consequently, $\langle\ P^A_{seq}\ ,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle$ if and only if $\langle\ P^A\ ,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle$.
It follows that if $\langle P^A_{seq},\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$, then,
$\langle P^A,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle$ and $(\epsilon', \varphi') \in [\![Q_{post}]\!]$.
Therefore, we conclude, $\forall \langle \epsilon,\ \varphi\rangle \in [\![Q_{pre}]\!],\ \langle P^A,\ \langle \epsilon,\ \varphi\rangle\rangle \Downarrow_\psi \langle \epsilon',\ \varphi'\rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$.
QED.

3.7.4 Accuracy Analysis

Parallely can generate an accuracy predicate that characterizes the magnitude of error of an approximate program. An accuracy predicate A has the following form ([43]):

$$A := D \leq D \mid A \wedge A \tag{3.21}$$

$$D := r \mid \mathcal{D}(x) \mid r \cdot \mathcal{D}(x) \tag{3.22}$$

An accuracy predicate can be a conjunction of predicates or a comparison between two error factors. An error factor can be a rational number r or the maximum error associated with a program variable x. The user can use an accuracy predicate to specify the maximum allowed error of various program variables at the end of execution. Parallely's analysis generates a precondition accuracy predicate that must hold at the start of the program for the user specified accuracy predicate to hold at the end of the program. To assist with error calculation, the user must specify the intervals of values that each input program variable can take. For a program variable x, this interval is specified as $x[a, b]$, indicating that the variable x can take any value in the range $[a, b]$ at the start of the program. Parallely then performs interval analysis similar to Chisel [43] to generate the accuracy precondition.

Theorem 3.5. If the program with approximation P^A (obtained by transforming the program P) can be canonically sequentialized, then the accuracy analysis of the sequentialized program P^A_{seq} will also be valid for the parallel approximate program. If, $\varnothing, \varnothing, P^A \rightsquigarrow \varnothing, P^A_{seq}, \texttt{skip}$ then $\psi \models \{A_{pre}\}\ P^A_{seq}\ \{A_{post}\} \implies \psi \models \{A_{pre}\}\ P^A\ \{A_{post}\}$

The proof follows directly from Lemma 3.11 and is analogous to the proof of Theorem 3.4.

Table 3.1: Accuracy and Performance of Approximated Programs

Benchmark	Parallel Pattern	Approximation	LoC	LoC Changed	Types Changed	Accuracy Property
PageRank	Map	Failing Tasks	49	12	4	$0.99 \leq \mathcal{R}(\text{result})$
Scale	Map	Failing Tasks	82	10	4	$0.99 \leq \mathcal{R}(\text{result})$
Blackscholes	Map	Noisy Channel	115	6	3	$0.99 \leq \mathcal{R}(\text{result})$
SSSP	Scatter-Gather	Noisy Channel	70	14	6	$0.99 \leq \mathcal{R}(\text{result})$
BFS	Scatter-Gather	Noisy Channel	70	14	6	$0.99 \leq \mathcal{R}(\text{result})$
SOR	Stencil	Precision Reduction	50	16	6	$10^{-6} \geq \mathcal{D}(\text{result})$
Motion	Map/Reduce	Approximate Reduce	43	13	6	-
Sobel	Stencil	Precision Reduction	46	16	6	$10^{-6} \geq \mathcal{D}(\text{result})$

3.8 EVALUATION

To evaluate Parallely expressiveness and efficiency, we implemented a set of benchmarks from several application domains. These benchmarks exhibit diverse parallel patterns, can tolerate error in the output and have been studied in the approximate computing literature:

- *PageRank:* Computes PageRank for nodes in a graph [95]. The parallel version is derived from the CRONO benchmark suite [96]. We verify the reliability of the calculated PageRank array.
- *Scale:* Computes a bigger version of an image. The version is derived from a Chisel benchmark [43]. We verify the reliability of the output image.
- *Blackscholes:* Computes the prices of a portfolio of options. The version is derived from PARSEC suite [97]. We verify the reliability of the array of calculated option prices.
- *SSSP:* Single Source Shortest Path in a graph. The version is derived from CRONO benchmark suite [96]. We verify the reliability of the calculated distances array.
- *BFS:* Breadth first search in a graph. The version is also derived from the CRONO suite [96]. We verify the reliability of the array of visited vertices.
- *SOR:* A kernel for computing successive over-relaxation (SOR). The version is derived from Chisel benchmark [43]. We verify the accuracy of the resultant 2D array.
- *Motion:* A pixel-block search algorithm from the x264 video encoder. The version is derived from Rely benchmark [42]. Although the accuracy specification is out of Parallely's reach, we still verify type safety, non-interference, and deadlock-freeness.
- *Sobel:* Sobel edge-detection filter calculation. We use the version from AxBench [98]. We verify the accuracy of the output image.

Table 3.1 presents the summary of the benchmarks and their specifications. For each benchmark, it presents the parallel pattern and approximation, the lines of Parallely code, the

Table 3.2: Performance of Parallely Analysis for Kernels (left) and Benchmarks (right)

Kernels	T_{Seq}	T_{Safety}	$T_{Rel/Acc}$
Simple (precision)	0.4 ms	0.2 ms	15 ms
Simple (noisy)	0.3 ms	0.2 ms	14 ms
Simple (failing tasks)	0.4 ms	0.2 ms	17 ms
Map (memoization)	0.9 ms	0.4 ms	48 ms
Reduce (sampling)	0.9 ms	0.5 ms	53 ms
Scan (noisy)	1.9 ms	0.9 ms	76 ms
Partition (failing tasks)	1.2 ms	0.6 ms	17 ms
Stencil (precision)	1.4 ms	0.7 ms	73 ms
Average	0.9 ms	0.5 ms	39 ms

Benchmarks	T_{Seq}	T_{Safety}	$T_{Rel/Acc}$
PageRank	1.8 s	1 ms	168 s
Scale	6.5 s	3 ms	7.4 s
Blackscholes	0.2 s	1 ms	12 s
SSSP	9.6 s	6 ms	9.6 s
BFS	8.9 s	6 ms	9.2 s
SOR	8.3 s	4 ms	53 s
Motion	2.9 s	1 ms	–
Sobel	0.2 s	6 ms	72 s
Average	4.8 s	3 ms	47.3 s

number of lines and type declarations affected by approximation, and the accuracy property we check. We omit the accuracy specification for Motion since it returns an index. For the accuracy analysis we assumed that the inputs to SOR and Sobel are in the range [0, 1].

For a set of simple parallel kernels and benchmark listed above, we measured the real time required to perform sequentialization, type checking, and reliability/accuracy checking. We ran the experiments on an Intel Xeon E5-1650 v4 processor with 32 GB of RAM. We implemented our parser using ANTLR and the other modules in Python.

3.8.1 Efficiency of Parallely

Table 3.2 presents a summary of Parallely's analysis for the real world programs and the kernels. For each benchmark, Column 2 presents the time for sequentialization, Column 3 presents the time for type safety analysis, and Column 4 presents the time for reliability analysis.

Overall, Parallely was efficient for both kernels and benchmark applications. The sequentialization analysis took only a few seconds, even for more complex benchmarks. Type checking was instantaneous – showcasing the benefits of our design to first do per-process type checking and then sequentialization (which is more expensive). Therefore, if there are simple type errors, they can be easily discovered and reported to the developer. The time for the reliability analysis is proportional to the amount of computation in each benchmark. The PageRank benchmark is an outlier, with 168 seconds. Most of this time was spent on analyzing unrolled loops in the sequentialized version of the program.

Table 3.3: Experimental Setup for Evaluation

Benchmark	Approximation	Input
PageRank	Failing Tasks with 1×10^{-6} probability	10 Iterations, random graph with 1000 nodes
Scale	Failing Tasks with 0.0001 probability	512×512 pixel image (baboon.ppm)
SOR	Precision Reduction (Float64 to Float32)	10 iteration on a 1000×1000 array
Sobel	Precision Reduction (Float64 to Float32)	1000×1000 array in the range [0,1]
Motion	Approximate Reduce (Skipping 90% of tasks)	10 blocks with 1600 pixels each

3.8.2 Benefits of Approximations

To analyze the benefits of the approximate transformations we translated the programs in Parallely to the Go language and measured the speedup. We only looked at the effect on runtime for three approximations: failing tasks, precision reduction, and approximate reduce. We selected representative inputs and present the average results over 100 runs (for statistical significance). Table 3.3 shows the parameters used in each of the benchmark's approximation (Column 2), number of processes we used (Column 3), and the size of the inputs (Column 4). For each benchmark, we verified the accuracy or reliability property specified in Table 3.1. Noisy channel evaluation requires detailed hardware simulation, which is out of our scope.

For each benchmark, Table 3.4 shows the approximations applied (Column 2), the metric used to compare the accuracy of the final results (Column 3), the speedup obtained using approximations, and the errors calculated using the relevant metric (Column 4). We next describe how we simulated each approximation.

- **Failing Tasks**. We simulate this approximation by letting child processes fail with a small random chance. If a child process fails, it sends a null signal to the main process. Many calculations can tolerate a small number of incorrect calculations without a significant loss of accuracy, such as the PageRank and Scale benchmarks. In the precise version, if the main process observes that a child process has failed, the child process is restarted.

- **Precision Reduction**. We reduce the precision of data sent to the worker processes from 64 bit floats to 32 bit floats. Likewise, the workers send the results as 32 bit floats. The results are converted back into 64 bit floats in the main process. Reducing precision only affects the least significant bits of the variables, so the accuracy degradation is acceptable for many calculations. In the precise version, this reduction in precision is not performed.

- **Approximate Reduce**. We simulate this approximation by letting worker processes decide not to do any work and return a null signal to the main process. The main process aggregates the received (non-null) results and adjusts the aggregate according to the number of returned results. In our evaluation each worker process only does work 10% of

Table 3.4: Performance Benefits from Approximations

Benchmark	Approximation	Accuracy metric	Speedup	End-to-End Error
PageRank	Failing Tasks	Avg. difference in PageRank	1.09x	2.87×10^{-8}
Scale	Failing Tasks	PSNR	1.35x	38.80 dB
SOR	Precision Reduction	Avg. sum of squared diffs. (SSD)	1.76x	3.9×10^{-16}
Motion	Approximate Reduce	Avg. difference from best SSD	1.20x	0.09
Sobel	Precision Reduction	Avg. sum of squared diffs. (SSD)	1.70x	1.56×10^{-16}

the time. When a large number of similar calculation results are aggregated by a reduction operation (such as sum, minimum, or maximum), the result of performing the calculations on a subset of data is often very close to the result of performing the calculation on the entire dataset. In the exact version, worker processes always return a result.

Results. Table 3.4 shows the speedup and error obtained as a result of the approximations. Column 1 shows the benchmark, Column 2 shows the approximation applied, Column 3 shows the error metric, Column 4 shows the speedup with respect to the precise version, and Column 5 shows the average measured error. For benchmarks with failing tasks (PageRank and Scale), the average is taken over runs that experienced at least one task failure. For Motion, SOR, and Sobel, the average is taken over all runs.

The results show that Parallely can be used to generate approximate versions of programs with significant performance improvements, while providing important safety guarantees. For PageRank and Scale, Parallely verified a reliability specification. However, even when a task fails, the error in the final result is very low. This result shows that programs with high reliability also often have high accuracy. For Motion, even when skipping most of tasks, the calculated minimum SSD is very close to the actual minimum SSD. For SOR and Sobel, the actual error is significantly lower than the worst case bound of 10^{-6} verified by Parallely.

3.9 RELATED WORK

Approximate Program Analyses. While approximations have been justified predominantly empirically (e.g., [1, 3, 30, 32, 34, 69, 89, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110]), several sound static analyses emerged in recent years. EnerJ [41, 111] presents an information-flow type system that separates approximate and precise data. As this *non-interference* constraint is restrictive, EnerJ allows approximate data to influence precise data through unsound type conversion (*endorse*). While we did not use such conversions, a general approach that more rigorously reasons about type conversions is an interesting topic for future work.

The frameworks presented in [44], [88], [112], and [113] provide reasoning about relational safety properties and relative safety. For accuracy and reliability, researchers have proposed quantitative analyses [43, 88, 90, 91, 114, 115, 116, 117, 118, 119]. Some (e.g., [90, 91, 116]) focus on specific sequential transformations and code patterns. For general programs, Rely [88], Chisel [43], and Decaf [120] analyze quantitative reliability and/or accuracy when running on unreliable hardware. Approaches such as [117], [121] and [122] reason about floating point errors in programs.

All these approaches have been developed for sequential programs. Parallely generalizes several key analyses for message-passing programs and presents a methodology for proving the correctness of the analysis via canonical sequentialization. Parallely's language further presented general approximation constructs and quantitative reasoning about both software and hardware approximations, in contrast to the previous approaches that were closely coupled with the hardware-specific error models [43, 88, 120].

Mechanized Proofs. Proofs in Parallely are manual. To increase our confidence in them they can be mechanized using proof assistants such as Coq [123] or Isabelle/HOL [124]. Various systems have been proposed to embed probabilities in Coq and to prove theories about probabilistic properties [125, 126, 127]. But, use of these formalisms continue to require a lot of manual effort to correctly capture the behavior of our systems, therefore such mechanization is left for future work.

Analysis of Parallel Programs. Previous work discusses the verification of programs using message passing interfaces, e.g., [128, 129, 130, 131]. Various tools are available to do model checking for Erlang, a popular actor language [132, 133, 134]. However, actor languages often have only one incoming message queue, making it difficult to prove properties about them via canonical sequentialization. Alternatively, one can reduce complex parallel programs into relatively simpler programs, or use representative program traces that are sufficient for reasoning about the properties of the original parallel program [135, 136, 137, 138, 139]. Sequentialization approaches such as [140, 141] reduce parallel programs to sequential versions to provide bounded guarantees. Other approaches [142, 143] allow developers to annotate a sequential program and verify that automated parallelizations are equivalent.

Our work primarily draws inspiration from Canonical Sequentialization by Bakst et. al. [87], which enables verification of general safety properties of parallel functional programs. We show that canonical sequentialization is a solid foundation for reasoning about approximate programs. We anticipate that future progress on sequentialization such as [92], can provide new opportunities for precisely modeling and analyzing various approximations.

Session Types. Session types [144, 145] provide a formalism for expressing the order and type of messages exchanged by concurrently executing processes. Session type systems require the developers to provide a global type specification. This specification encodes the allowed sequences of messages that processes may exchange in a given session, as well as the types of these messages. An algorithm then generates the allowed behavior for each participant process based on the global specification. Session types can be used with Parallely's type extensions to prove properties about process interactions, such as deadlock-freeness and non-interference. But, they require extensive developer annotations and cannot be used to verify functional properties such as reliability and accuracy of local data.

Foundations of Parallel Programming. Researchers have defined various formalisms for representing parallel computation, e.g., Actor model [146, 147], Pi calculus [148], Petri nets [149] and others. To make an approximation-aware analyses both tractable and easier to express/implement, we aimed for a modular approach that separates reasoning about concurrency from reasoning about quantitative properties. Canonical sequentialization proved to be a good match in this regard, while offering a good level of generality and reusing the formalizations of the analyses for sequential programs. An alternative route would be to define individual analyses, such as EnerJ, Rely, or Chisel directly on a parallel calculus. While such an approach would be equally fruitful for the property in question, it would require reasoning about interleavings and shared data in an ad-hoc manner and new proof would need to be derived for every other analysis, making it hard to support multiple analyses.

3.10 CONCLUSION

We presented Parallely, a language and system for verification of approximations in parallel message-passing programs. It is the first approach tailored to systematically represent and analyze approximate parallel computations. In this chapter, we have presented how to leverage a large body of techniques for verification of approximate sequential programs to the parallel setting, while allowing us to generalize those and increase their reach (as in the case of Rely). Our experimental results on a set of approximate computational kernels, representative transformations, and full programs show that Parallely's analysis is both fully automatable and efficient.

Our approximation-aware canonical sequentialization is particularly promising: in addition to the accuracy analyses that we studied in this work, we anticipate that other existing and future accuracy analyses of sequential programs can directly leverage sequentialization to analyze parallel computations. We anticipate that our theoretical results will also be useful

to reason about other quantitative properties of parallel programs, such as differential privacy or fairness.

While our results show that static analysis techniques can show the correctness of many useful programs, to extend these techniques to bigger programs and dynamic error models we have to use runtime verification methods. The next chapter shows how to develop a runtime system that can soundly verify similar safety and accuracy properties.

Chapter 4: Diamont: Dynamic Monitoring of Uncertainty for Distributed Asynchronous Programs

4.1 INTRODUCTION

Many emerging distributed applications operate on inherently noisy data or produce approximate results [150]. Emerging edge applications, including autonomous robotics and precision agriculture, routinely need to deal with noise from their sensors. Machine learning applications regularly encounter datasets that contain a high degree of noise, or other irregularity. Furthermore, the rise of highly-parallel and often heterogeneous systems have brought forth new challenges in overcoming bottlenecks in computation and communication between processing units. Many prominent systems adopted approximation in communication, e.g., MapReduce's task dropping [29], TensorFlow's precision reduction [28], or Hogwild's synchronization-eschewing stochastic gradient descent [27]. Also, researchers explored various non-conventional architectures and networks-on-chip [18, 19, 40, 151].

To cope with different kinds of uncertainty, researchers developed several static and run-time analyses that quantify the level of noise, reliability, or accuracy. We survey the existing techniques in Section 4.7. These existing techniques suffer from one or more of the following problems: 1) they have been developed *only for sequential* programs, 2) they are either *imprecise* (static analyses) or *lack guarantees* on result quality and soundness of monitoring code (empirical analyses), or 3) their applicability is *limited* – a single analysis is defined exclusively for a *specific* source of uncertainty (e.g., an unreliable instruction or a noisy sensor) and cannot be combined with others. Directly extending and generalizing the existing frameworks to a distributed setting can lead to subtle problems and/or run-time inefficiencies. An intriguing question is how to design a general analysis framework that will overcome these challenges, thus enabling a flexible and precise uncertainty analysis for parallel computations.

Our Work. We present Diamond, the first system for *sound, precise and efficient* runtime monitoring of uncertainty in *distributed applications*. Diamond offers a flexible runtime system for specifying and verifying uncertainty bounds in the face of *various sources of uncertainty*. Diamond supports programs consisting of distributed processes that communicate via asynchronous message-passing. Each process communicates with the others using strongly-typed communication channels through the common `send` and `receive` communication primitives. Diamond includes multiple language constructs for dynamic monitoring:

- **Dynamic types and data channels:** The developer specifies the variables that need to be dynamically monitored by annotating them using the `dynamic` type qualifier. In addition, Diamond introduces dynamic channels that use specialized communication primitives to reliably transfer the monitoring information.
- **Runtime Monitoring of Uncertainty:** Diamond maintains *uncertain intervals* for `dynamic` typed variables – these map variables to a maximum error bound and a probability that the error is within the bound. Diamond propagates the interval through computations. It can precisely do so even for individual array elements and unbounded loops – factors that reduce precision of existing analyses like Parallely [152] and DECAF [120].
- **Checkers:** Diamond's `check` statement evaluates logical predicates over the program state and the monitored uncertainty to report violations. For example, the check can verify whether the magnitude of a variable's error is less than a developer-defined threshold. Using Diamond's checks, developers can decide if further attention should be given to the results. If the uncertainty of a result is acceptable at runtime, developers can avoid costly error checking and correction mechanisms.

We implemented Diamond for a distributed fragment of the Go language, extended with the dynamic type and check statements. Diamond performs static analysis at the level of an intermediate representation (IR) extracted from the Go code. It generates instrumented Go code with dynamic monitoring implemented via a Go library.

Diamond also presents a set of optimizations to reduce the runtime overhead arising from the monitoring of uncertain intervals throughout and across processes. These optimizations include: 1) combining static analysis with dynamic monitoring 2) approximating dynamically monitored uncertainty of arrays, 3) moving check statements across processes, and 4) using compiler techniques such as constant propagation and dead-code elimination. These optimizations give Diamond a significant advantage over direct extensions of systems like Decaf [120] or AffineFloat [153] to parallel programs. Developers who try to manually implement such run-time system optimizations that span multiple processes can easily make subtle errors.

Verified Runtime and Optimizations. We prove the soundness of the Diamond runtime and optimizations. Soundness of a Diamond program means that if the execution passes a variable uncertainty check, then the uncertainty of the variable is within the bound specified in the check statement. An optimization is sound if all check failures in a program are also guaranteed to occur in its optimized version.

Diamond's runtime system is sound for programs that satisfy the *symmetric nondeterminism* property [87] – i.e., each receive statement must have a unique matching send statement, or a

set of symmetric matching send statements. Many common parallel patterns in data analytics applications [30, 152] satisfy this property. We use *canonical sequentialization* [87, 152], which rewrites a symmetrically nondeterministic parallel program to an equivalent sequential program. We can then prove soundness of runtime monitoring on the sequentialized program. Lastly, we show that this soundness proof also applies to the original parallel program.

Through sequentialization, Diamond can also automatically verify type safety and the absence of deadlocks of programs caused by approximations, the runtime system, or optimizations that change communication patterns.

Results. We applied Diamond on eight parallel applications. These real-world applications come from the domains of graph analytics, precision agriculture, and media processing. We modeled four sources of uncertainty: noisy communication, precision reduction (compression), noisy inputs, and timing errors.

We showed that Diamond can verify important end-to-end properties for all applications. In particular, we looked at four error probability predicates of end results, three error magnitude predicates, and one predicate on both error probability and magnitude. These properties cannot be validated by existing static techniques [42, 43, 152].

Our optimizations reduced the runtime overhead of Diamond with respect to the unmonitored program. Directly extending existing sequential runtime analyses to parallel settings leads to overheads between 30-80%. Our optimizations reduced the overhead to a geomean of 3% and maximum of 16.3% while satisfying strict predicates. We show that these overheads remain low and the communication of monitoring data is minimized even when the input size increases, especially for applications that implement intensive communication. These results demonstrate that even in the face of both uncertainty and significant parallelism, runtime monitoring is still practical.

Contributions. The chapter makes several contributions:

- **Diamond.** Diamond is a system for dynamically monitoring uncertainty properties in strongly-typed, message-passing, asynchronous programs. We show that Diamond can soundly monitor uncertainty (error probability and magnitude).
- **Optimizations for reducing overhead.** We present several optimizations that reduce the overhead of performing runtime monitoring across processes.
- **Implementation.** We implement Diamond's analysis and runtime system with optimizations for a subset of Go.
- **Evaluation.** We evaluate Diamond on 8 benchmarks. We show that Diamond can verify important correctness properties with small runtime overheads.

```
ovar Q = [NUMSENSORS] process
var R = [NUMWORKERS] process

type point struct {
    /*@dynamic*/ temperature, humidity float64
}

func IoTDevice {
  /*@dynamic*/ var temperature, humidity float64
  tempVal, tempErr, tempConf := readTemperature()
  humidVal, humidErr, humidConf := readHumidity()
  temperature = track(tempVal, tempErr, tempConf)
  humidity = track(humidVal, humidErr, humidConf)
  send(Manager, point{temperature, humidity})
}

func Worker {
  var data [NUMSENSORS] point
  var centers, newcenters [NUMCENTERS] point
  /*@dynamic*/ var assign [PERTHREAD] int
  data = receive(Manager)
  for iter:=0; iter<ITERATIONS; iter++ {
    centers = receive(Manager)
    newcenters = kmeansKernel(data, centers,
                              assign)
    send(Manager, newcenters)
        }
}

func Manager {
  // setup skiped to preserve space
  for i, IoTDevice := range(Q) {
    data[i] = receive(IoTDevice)
  }
  centers = // randomly pick some nodes
  for i, Worker := range(R) {
    send(Worker, data)
  }
  for j:=0; j<ITERATIONS; j++ {
    for _, Worker := range(R) {
      send(Worker, centers)
    }
    for i, Worker := range(R) {
      newcenters[i] = receive(Worker)
    }
    centers = AverageOverThreads(newcenters)
  }
  checkArr(centers, 1, 0.99, 4, 0.99)
}
```

Figure 4.1: GoLang: Smart Agriculture Setup

4.2 EXAMPLE

We consider a scenario from precision agriculture [154]. Multiple low-power embedded systems with sensors are distributed across a field to monitor changes in the environment. Each embedded system (e.g., Raspberry Pis) can read the temperature, humidity, or other properties using their sensors. It can perform limited local processing of the readings, and periodically sends those results to a server for further (typically more expensive) analysis.

Figure 4.1 shows an implementation of the application in Go. The program has multiple parallel processes that communicate over typed channels using the Diamond API using matched `send` and `receive` statements (E.g., Lines 33, 14). The `Manager` process coordinates the computation.

The process group `Q` is of a set of processes $\texttt{IoTDevice}_{1,\ldots,\text{NUMSENSORS}}$ that runs on embedded systems and read sensor values and communicate the data to the `Manager`. Each `IoTDevice` gathers and stores datapoints using the struct `point` from Line 5. The `/*@dynamic*/` annotation indicates that the fields of `point` are of `dynamic` type. Diamond monitors the uncertainty of `dynamic` variables at runtime.

The `Manager` process first gathers sensor data (Line 33) from each `IoTDevice`. Then it performs a distributed k-means clustering analysis using the processes in the group `R`. The `Manager` picks a set of random points as the initial cluster centers (Line 8). Next, over `ITERATIONS` iterations, it updates the cluster centers (Lines 39-47). Each `Worker` process from the group `R` processes a subset of the data points to calculate new cluster centers (Lines 22-26) for that subset. The `Manager` combines the partial results from each `Worker` and redistributes them (Line 46).

4.2.1 Sources of Uncertainty

Approximate sensors. Sensors are often noisy (e.g., the AM2302-DH22 relative humidity and temperature sensor has an error range of $\pm 0.5°$F for temperature and $\pm 2\%$RH for humidity reading [155]). Each process in `Q` calculates the error of its sensors while reading the value at Lines 10 and 11. This error calculation can come from the sensor specification (e.g. [155]). Next, Lines 12 and 13 initialize `dynamic` variables using the sensor value and error.

Approximate Communication. We also consider the impact of communication over noisy channels (Line 37, 21), prevalent in situations where sensors are deployed in remote areas (E.g., [26]). Messages in such channels can be corrupted with a small probability [156].

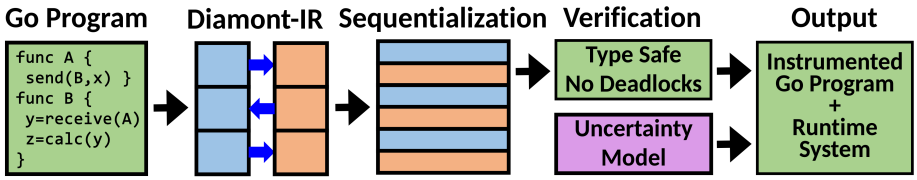


Figure 4.2: Diamond Workflow

Instead of implementing costly error correction mechanisms, a developer may choose to deal with potentially incorrect data to save resources.

An uncertainty model ψ provides parameters such as the probability of message corruption. For example, $\psi(\texttt{Manager}, \texttt{Worker}, \texttt{dynamic float<64>}) = 1 - 10^{-7}$ indicates that the probability of corruption of a `dynamic float<64>` type message from `Manager` to `Worker` is 10^{-7}. The specification is modeled after the ones from [41, 42, 113].

4.2.2 Verification

Properties. We wish to verify that the final values of `centers` are close to the true cluster centers with high probability. We encode this requirement in the `checkArr` statement in Line 48. This check specifies a maximum error magnitude and probability for each `dynamic` field in the struct. This program has features that make static verification using tools such as Parallely [152] challenging:

- The error specification of the sensors may not be known a priori. Additionally, prior static verification techniques require worst-case bounds for the number of loop iterations and the number of processes. Using worst-case estimates for these in a static analysis will invalidate many correct programs.
- Parallely treats entire arrays as single variables, and thus array analysis accumulates errors even across two different array locations. Consequently, the conservative static estimate of uncertain intervals quickly expands to unusable levels for any sufficiently large number of sensors for our example.

Workflow. Diamond combines static and dynamic analyses to verify safety and accuracy properties at *runtime*. Figure 4.2 shows the workflow for generating an instrumented program in Diamond. Given a Go program, Diamond 1) translates it to Diamond-IR, 2) sequentializes the program to statically verify type safety, deadlock-freeness, and the applicability of the runtime analysis, and 3) produces an instrumented version of the original Go program with an *uncertainty map* for each process. The sequentialized version of the code in Figure 4.1 is in the Appendix.

The *uncertainty map* of a process maintains a conservative uncertain interval for each `dynamic` local variable. Uncertain intervals are stored as pairs $\langle d, r \rangle$ indicating that the maximum error of the associated variable is $\leq d$ with probability $\geq r$. The default uncertain interval is $\langle 0, 1 \rangle$ (no error with 100% confidence). Developers can use `track` statements (E.g., Line 12) to use external error specifications within Diamond. When a dynamic variable is updated, Diamond also updates the uncertain interval. Diamond's instrumentation 1) initializes the uncertain interval of the data in `IoTDevice`, 2) communicates the uncertain interval across process boundaries, 3) propagates this uncertainty through computations, and 4) checks the uncertain interval of the array at the end of the program against a developer-specified bound.

We verified this system for a setting with 128 sensors and a set of 8 workers performing the k-means computation over 10 iterations. As more and more computations containing unreliable values affect the `centers` array, the uncertain interval of individual elements widens. However, the specification is still satisfied.

Overhead. Diamond's instrumentation adds runtime overhead. To reduce overhead, Diamond applies optimizations such as constant propagation, dead code elimination, and simplification of monitoring uncertainty in arrays. To reduce overhead when transmitting arrays, Diamond transmits the maximum uncertainty among the elements of the array as the uncertainty of every element of the array. This allows Diamond to only communicate one uncertain interval across processes, while maintaining high analysis precision in other parts of the program. These optimizations reduce Diamond's overhead from 42% to 3.2%. Increasing the number of sensors does not significantly increase overhead (Section 4.6.3). Even for 2-8x larger data, the overhead remains below 5%.

4.3 DIAMOND SYSTEM

Diamond takes as input a Go program and an uncertainty model. Diamond first converts the program to the Diamond-IR and verifies important safety properties necessary to ensure that the runtime system will be sound. Finally, Diamond generates instrumented Go code.

4.3.1 Syntax

Go Language. Diamond supports a subset of the Go Programming Language (matching the features of Diamond-IR along with external functions that do not perform communication)

$m, v \in \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$	*values*	$S \to$ T x $\mid$ T a[n^+]	*declarations*
$Exp \to m \mid \langle m, v \rangle \mid x \mid Exp\ op\ Exp$	*expressions*	$\mid x$ = *Exp*	*assignment*
$AExp \to d \mid d \cdot x \mid d \cdot a[Exp^+]$	*affine*	$\mid x$ = *Exp* [r] *Exp*	*probabilistic choice*
$\mid AExp \pm AExp$	*expressions*	$\mid$ dyn-send(α, T, x)	*send dynamic*
$q \to$ `precise` $\mid$ `approx` $\mid$ `dynamic`	*type qualifiers*	$\mid x$ = dyn-receive(α, T)	*receive dynamic*
$t \to$ `int<n>` $\mid$ `float<n>`	*basic types*	$\mid x$ = rdDyn(y)	*read dynamic map*
$T \to q\ t \mid q\ t\ [] \mid$ struct T^+	*types*	$\mid x$ = endorse(y)	*cast to precise*
$P \to [S]_\alpha$	*process*	$\mid x$ = track(y, $\langle$d, r$\rangle^+$)	*initiate monitoring*
$\mid \Pi.\alpha : X\ [S]_\alpha$	*process group*	$\mid x$ = *Exp*? *Exp* : *Exp*	*conditional choice*
$\mid P \parallel P$	*parallel comp*	$\mid$ check(*AExp*, $\langle$d, r$\rangle^+$)	*check error*
		$\mid$ checkArray(a, $\langle$d, r$\rangle^+$)	*check array error*

Figure 4.3: Diamond-IR Syntax Extensions (full language contains conditionals, loops and function calls)

extended with an API for distributed communication and annotations in comments for type qualifiers.

Diamond-IR. Diamond's intermediate representation supports a strongly typed imperative language with primitives for asynchronous communication. Diamond extends the syntax of Parallely with support for the additional `dynamic` type. Figure 4.3 defines the subset of Diamond syntax dealing with `dynamic` data. Remaining instructions follow directly from Parallely (Chapter 3) and has no impact on the runtime monitoring system. Here, d refers to reals, r to probabilities, n to positive integers, x, y to variables, and a to array variables. The full syntax includes conditionals, loops, operations on arrays, and structs.

Types. Diamond's type qualifiers explicitly split data into either `precise` (no uncertainty), `dynamic` (uncertainty monitored at runtime), or `approx` (uncertain but unmonitored). Diamond's type system ensures that uncertainties in executions do not cause errors in critical program sections and ensures that the dynamic monitoring is sound by avoiding control flow divergence. Using type inference, Diamond automatically annotates some variables as dynamic to reduce programmer burden.

Communication. Processes communicate by sending and receiving messages over typed channels. For each pair of processes, Diamond provides a set of logical sub-channels for communication, further split by message type (μ). A `send` statement asynchronously sends a value to another process using a unique process identifier. The receiving process uses the blocking `receive` statement to read the message. Messages on the same sub-channel are delivered in order but there are no guarantees for messages sent on separate (sub)channels. Diamond supports communication of `dynamic` type data through `dyn-send` and `dyn-recv` statements, which also send the monitored uncertainty using reliable channels.

S-Assign-Dyn

$$\frac{(x,.,.) \in D \quad \langle e, \sigma, h\rangle \Downarrow v \quad d = \langle \texttt{calc-eps}(e, D),\ \texttt{calc-del}(e, D)\rangle \quad D' = D[x \mapsto d] \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v]}{\langle x \texttt{ = e},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-DynSend

$$\frac{\mu[\langle \alpha, \beta, D_t\rangle] = m_d \quad \mu' = \mu[\langle \alpha, \beta, D_t\rangle \mapsto m_d + +D[y]]}{\langle [\text{dyn-send}(\beta, t, y)]_\alpha,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle [\text{send}(\beta, t, y)]_\alpha,\ \langle \sigma, h\rangle,\ \mu',\ D\rangle}$$

S-DynReceive

$$\frac{\mu[\langle \beta, \alpha, D_t\rangle] = d :: m_d \quad \mu' = \mu[\langle \beta, \alpha, D_t\rangle \mapsto m_d] \quad d_b = \langle d.\epsilon,\ d.\delta \times \psi(\beta, \alpha, t)\rangle \quad D' = D[x \mapsto d_b]}{\langle [x \texttt{ = } \text{dyn-receive}(\beta, t)]_\alpha,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle [x \texttt{ = } \text{receive}(\beta, t)]_\alpha,\ \langle \sigma, h\rangle,\ \mu',\ D'\rangle}$$

S-Cast

$$\frac{\langle n'_b,\ \langle 1\rangle\rangle = \sigma(y) \quad h[n'_b] = m \quad m' = \texttt{cast}(\mathrm{T}, m) \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto m'] \quad d = \langle \texttt{cast-eps}(x, y, D),\ D[y].\delta\rangle \quad D' = D[x \mapsto d]}{\langle x \texttt{ = } (\text{dynamic T})y,\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-Prob-True

$$\frac{x \in D \quad \langle e_1, \sigma, h\rangle \Downarrow v_1 \quad d = \langle \texttt{calc-eps}(e_1, D),\ \texttt{calc-del}(e_1, D) \times \psi(r_f)\rangle \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v_1] \quad D' = D[x \mapsto d]}{\langle x \texttt{ = } e_1\ [r_f]\ e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{\psi(r_f)}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-Prob-False

$$\frac{x \in D \quad \langle e_2, \sigma, h\rangle \Downarrow v_2 \quad d = \langle \texttt{calc-eps}(e_1, D),\ \texttt{calc-del}(e_1, D) \times \psi(r_f)\rangle \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v_2] \quad D' = D[x \mapsto d]}{\langle x \texttt{ = } e_1\ [r_f]\ e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1-\psi(r_f)}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-check-pass

$$\frac{\texttt{calc-eps}(ae, D) \leq d \wedge \texttt{calc-del}(ae, D) \geq r}{\langle \text{check(ae, d, r)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle}$$

S-Check-Fail

$$\frac{\texttt{calc-eps}(AExp, D) > d \quad \vee \quad \texttt{calc-del}(AExp, D) < r}{\langle \text{check}(AExp\text{, d, r}),\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \bot, \mu, D\rangle}$$

S-RdDyn

$$\frac{D[y] = v \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v]}{\langle x \texttt{ = } \text{rdDyn}(y),\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D\rangle}$$

S-array-check-true

$$\frac{\forall i.\ D[y, i].\epsilon \leq d \wedge D[y, i].\delta \geq r}{\langle \text{checkArray(y, p)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle}$$

S-array-check-false

$$\frac{\forall i.\ D[y, i].\epsilon > d \wedge D[y, i].\delta < r}{\langle \text{checkArray(y, p)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip}, \bot, \mu, D\rangle}$$

S-track

$$\frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad \langle n'_b,\ \langle 1\rangle\rangle = \sigma(y) \quad v = h[n'_b] \quad h' = h[n_b \mapsto v] \quad D' = D[x \mapsto \langle d,\ r\rangle]}{\langle x \texttt{ = } \text{track(y, d, r)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-endorse

$$\frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad \langle n'_b,\ \langle 1\rangle\rangle = \sigma(y) \quad v = h[n'_b] \quad h' = h[n_b \mapsto v] \quad D' = D[x \mapsto \langle 0,\ 1\rangle]}{\langle x \texttt{ = } \text{endorse(y)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

Figure 4.4: Semantics of Dynamic Monitoring

Type conversion. To explicitly convert a variable to `dynamic` type, the developer or compiler can use a `track` statement (x = track(y, $\langle d,\ r\rangle$)), which sets the uncertain interval to $\langle d,\ r\rangle$. `track` statements can be used to initiate monitoring for variables updated by external functions, or to incorporate informal specifications (e.g., from a datasheet) into Diamond. Similarly, the `endorse` statement (x = endorse(y)) converts an `approx` or `dynamic` variable to a `precise` variable, usually after a user-defined check (similar to EnerJ [41]). The `rdDyn` intrinsic (rdDyn(x)) can be used to read the monitored uncertainty of a dynamic variable.

Uncertainty Model (ψ). The reliability/accuracy of program components (e.g., the probability of message corruption or the probability that a sensor fails) are provided to the runtime using the uncertainty model.

Specifications. Diamond exposes the following statements to check specifications of dynamically monitored variables.

- check(*AExp*, ⟨d, r⟩): It checks if an affine expression *AExp* has a maximum error $\leq d$ with probability $\geq r$. If not, the check fails and creates an error.
- checkArray(a, ⟨d, r⟩): It checks if the dynamically monitored uncertainty for *each* element in array a satisfies the specification.

While this version of Diamond stops the execution if a check fails, it can be extended to trigger a recovery mechanism instead [89, 157, 158]. Aloe [158] represents recoverable computations with blocks of the form `try {...} check (...) recover {...}`. Using this construct, Diamond can recover the execution if a check fails, and calculate the effect of (possibly imperfect) checks and recovery mechanisms on uncertainty.

Structs. The programmer can specify the uncertainty of each field of a struct in a `track` statement by using multiple ⟨d, r⟩ pairs. The programmer can check each field of a struct in `check` and `checkArr` statements in a similar manner.

4.3.2 Diamond Semantics

Semantics for `precise` and `approx` data in Diamond are the same as those from Parallely (Chapter 3). For `dynamic` data, the compiler adds instructions to monitor their uncertain intervals alongside the original program instructions.

References, Frames, Stacks, and Heaps. A *reference* is a pair $\langle n_b, \langle n_1, ..., n_k \rangle\rangle \in \texttt{Ref}$ that contains a base address $n_b \in Loc$ and dimension descriptor $\langle n_1, ..., n_k \rangle$ denoting the location and dimension of variables in the heap. A *frame* $\sigma \in \mathrm{E} = \mathrm{Var} \rightarrow \mathrm{Ref}$ maps program variables to references. A *heap* $h \in H = \mathbb{N} \rightarrow \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ is a finite map from addresses to values (Integers, Floats or the special *empty message* $[\varnothing]$). Each process i maintains its own private environment consisting of a frame and a heap $\langle \sigma^i, h^i \rangle \in \Lambda = \{H \times \mathrm{E}\} \cup \perp$, where $\perp$ is considered to be an error state.

Programs. Diamond defines a program as a parallel composition of processes. We denote a program as $P = [P]_1 \parallel \cdots \parallel [P]_n$, where $1 \ldots n$ are process identifiers. Individual processes

$$\texttt{calc-eps}(e, D) =$$

$$\begin{cases} 0 & e \text{ is a constant} \\ D[x].\epsilon & e \text{ is a variable } x \\ D[x].\epsilon + D[y].\epsilon & e \text{ is } x \pm y \\ |x|D[y].\epsilon + |y|D[x].\epsilon + D[x].\epsilon \times D[y].\epsilon & e \text{ is } x \times y \\ \infty & e \text{ is } x \div y \text{ and } 0 \in [y \pm D[y].\epsilon] \\ \frac{(|x|D[y].\epsilon + |y|D[x].\epsilon)}{(|y|(|y| - D[y].\epsilon))} & e \text{ is } x \div y \text{ and } 0 \notin [y \pm D[y].\epsilon] \end{cases}$$

$$\texttt{calc-del}(e, D) = \max\,(\,0,\ (\Sigma_{x \in \rho(e)} D[x].\delta) - (|\rho(e)| - 1)\,)$$

$$\begin{aligned} \texttt{cast-eps}(x, v, D) \ = \max(&\max(x + D[x].\epsilon, v + D[x].\epsilon) - v, \\ &v - \min(x - D[x].\epsilon, v - D[x].\epsilon))) \end{aligned}$$

Figure 4.5: Runtime for Dynamic Monitoring of Uncertainty

execute their statements sequentially. Each process has a unique process identifier (Pid). Processes can refer to each other using Pids. We write ⟨pid⟩.⟨var⟩ to refer to variable ⟨var⟩ of process ⟨pid⟩. When unambiguous, we will omit ⟨pid⟩ and just write ⟨var⟩.o

Uncertainty Map. For each process, Diamond defines an *uncertainty map* (D) to attach each variable with a uncertain interval, consisting of a maximum absolute error (ϵ), and a probability/confidence (δ) that the true error is below ϵ.

Local Semantics. The small-step relation $\langle s,\, \langle \sigma, h\rangle,\, \mu,\, D\rangle \xrightarrow{p}_{\psi} \langle s',\, \langle \sigma', h'\rangle,\, \mu',\, D'\rangle$ defines a process in the program evaluating in its local frame σ, heap h, uncertainty map D, and the global channel set μ. Figure 4.4 presents the semantics for the statements that deal with `dynamic` typed data. The semantics for the remaining statements are the same as Parallely(with the new signature). Those statements do not affect the uncertainty map.

- **Initialization:** Each `dynamic` variable is initialized by setting the maximum error ϵ to 0 and the confidence δ to 1.
- **Expressions:** The `S-Assign-Dyn` rule in Figure 4.4 is applied when a `dynamic` variable is updated by assigning it an expression `e`. We use a big-step evaluation relation of the form $\langle e, \sigma, h\rangle \Downarrow v$ to compute the result of the expression. Diamond supports typical integer and floating point operations.

  For `dynamic` variables, in addition to the assigned variable, Diamond updates its interval using the uncertain interval arithmetic defined in Figure 4.5. The `calc-eps` function is used to calculate an expression's maximum error. The confidence in this maximum error is then

$$\textsc{Dec-Array}\quad \frac{\forall i.n_i > 0 \qquad \langle n_b,\ h'\rangle = \text{new}(h, \langle n_1 ... n_k\rangle) \qquad D' = \text{init}(D, \langle n_1 ... n_k\rangle, \texttt{init-track}()) \qquad \sigma' = \sigma[x \mapsto \langle n_b, \langle n_1 .. n_k\rangle\rangle]}{\langle \text{T x}[n_1 ... n_k],\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma', h'\rangle,\ \mu,\ D'\rangle}$$

$$\textsc{S-Array-Load}\quad \frac{\forall i.\langle e_i, \sigma, h\rangle \Downarrow l_i \qquad \langle n_b,\ \langle l_1, ..., l_k\rangle\rangle = \sigma(a) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \qquad v = h(n_b + n_o) \qquad D' = D[x \mapsto D[\langle a,\ n_o\rangle]] \qquad \langle n'_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h' = h[n'_b \mapsto v]}{\langle x \texttt{ = } a\texttt{[}e_1, ..., e_i, ..., e_k\texttt{]},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

$$\textsc{S-Array-Store}\quad \frac{\forall i.\langle e_i, \sigma, h\rangle \Downarrow l_i \qquad \langle n_b,\ \langle l_1, ..., l_k\rangle\rangle = \sigma(a) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \qquad D' = D[\langle a,\ n_o\rangle \mapsto D[x]] \qquad \langle n'_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad n = h(n'_b) \qquad h' = h[(n_b + n_o) \mapsto v]}{\langle a\texttt{[}n_1, ..., e_i, ..., e_k\texttt{]} \texttt{ = } x,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

Figure 4.6: Semantics of Arrays

computed using `calc-del` ($\rho(e)$ returns the list of variables used in an expression e.) To avoid any assumptions about the independence of the uncertainties (prior approaches such as [120] restrictively assumed all the operations and probability of failures are independent) Diamond uses the conservative union bound.

- **Communication:** When sending `dynamic` variables of type T to another process (rule `S-DynSend`), Diamond uses special channels (D_T) that are assumed to be fully reliable to communicate the relevant uncertain intervals before sending the data. ++ denotes adding an element to the end of the message queue. At the receiver (rule `S-DynReceive`), Diamond updates the local uncertainty map. Diamond assumes the channel failure rate is independent of the message content and reduces the confidence based on the failure rate defined in the Uncertainty Model.

- **Precision Manipulation:** Diamond monitors the errors introduced to programs through *cast* statements that change the precision of values of the same general type (int or float). In the rule `S-Cast`, the added error is calculated using the `cast-eps(`x`,`v`,D)` function using the casted value v and the original variable x. Confidence remains the same.

- **Arrays:** To increase the precision of array analysis, when `dynamic` type arrays are declared, we allocate an entry in D for each array element. When array elements are updated, we

also update their corresponding dynamically monitored value. The array semantics are available in Figure 4.6.

- **Conditionals:** For branching on `dynamic` values, Diamond uses x `= cond`? $e_1 : e_2$ (conditional choice) operator where `cond` compares a `dynamic` value against a threshold. In these situations the maximum error and error confidence of the assigned variable x need to be calculated with care based on the uncertainty interval. If the entire interval satisfy a condition we can confidently calculate the maximum error form the resultant expression. If not we must assume the worst case whereby we could have chosen the wrong branch, thus the maximum error depends on the difference between Exp_1 and Exp_2. Figure 4.7 defines the precise semantics for conditionals.

S-Assign-Cond-Dyn-True

$$\frac{(x,\epsilon,\delta) \in D \quad x - \epsilon > r \quad d = \texttt{get-dyn-choice}(e_1, \delta, D) \quad \langle e_1, \sigma, h\rangle \Downarrow v_1 \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \quad h' = h[n_b \mapsto v_1] \quad D' = D[y \mapsto d]}{\langle y = x > r?\ e_1 : e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-Assign-Cond-Dyn-False

$$\frac{(x,\epsilon,\delta) \in D \quad x + \epsilon < r \quad d = \texttt{get-dyn-choice}(e_2, \delta, D) \quad \langle e_2, \sigma, h\rangle \Downarrow v_2 \langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \quad h' = h[n_b \mapsto v_2] \quad D' = D[y \mapsto d]}{\langle y = x > r?\ e_1 : e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-Assign-Cond-Dyn-unknown-True

$$\frac{(x,\epsilon,\delta) \in D \quad x + \epsilon > r \quad x - \epsilon < r \quad x > r \quad d1 = \texttt{get-dyn-exp}(e_1, D) \quad d2 = \texttt{get-dyn-exp}(e_2, D) \quad \langle e_1, \sigma, h\rangle \Downarrow v_1 \quad \langle e_2, \sigma, h\rangle \Downarrow v_2 \quad \epsilon' = |v_1 - v_2| \times \max(d1.\epsilon, d2.\epsilon) \quad D' = D[y \mapsto \langle \epsilon',\ \min(d1.\delta, d2.\delta)\rangle] \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v_1]}{\langle y = x > r?\ e_1 : e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

S-Assign-Cond-Dyn-unknown-False

$$\frac{(x,\epsilon,\delta) \in D \quad x + \epsilon > r \quad x - \epsilon < r \quad x < r \quad d1 = \texttt{get-dyn-exp}(e_1, D) \quad d2 = \texttt{get-dyn-exp}(e_2, D) \quad \langle e_1, \sigma, h\rangle \Downarrow v_1 \quad \langle e_2, \sigma, h\rangle \Downarrow v_2 \quad \epsilon' = |v_1 - v_2| \times \max(d1.\epsilon, d2.\epsilon) \quad D' = D[y \mapsto \langle \epsilon',\ \min(d1.\delta, d2.\delta)\rangle] \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v_2]}{\langle y = x > r?\ e_1 : e_2,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle}$$

Figure 4.7: Semantics of Dynamic Conditionals

- **Checks:** If a check fails, the Diamond program transitions into an error state (Figure 4.4 rule `S-Check-Fail`). To prevent such check failures, the user can implement error recovery mechanisms.

- **Functions.** Diamond can also support functional specifications that bound the error of the output. We support Chisel style specifications. When such a function call is reached, if the requirements for the specification is satisfied (rule `S-FUNCTION`), the uncertainty interval is updated by taking into account the error confidence in the input parameters along with the guarantee provided by the specification. If the requirements are not satisfied (rule `S-FUNCTION-FAIL`), the system throws an error.

S-Function

$$\frac{\begin{array}{c}\psi(f) = \langle d, r * R(d_1 \geq \Delta(y_1), \ldots, d_n \geq \Delta(y_n))\rangle \\ \forall j = 1 \ldots, n.\, \texttt{calc-eps}(y_j, D) \leq d_i \\ \langle f(y_1, \ldots, y_n), \sigma, h\rangle \Downarrow v \quad r' = r \times \texttt{calc-del}(f(y_1, \ldots, y_n), D) \\ \sigma(x) = \langle n_b,\ \langle 1\rangle\rangle \qquad h' = h[n_b \mapsto v] \quad D' = D[y \mapsto \langle d, r'\rangle]\end{array}}{\begin{array}{l}\langle [x = f(y_1, \ldots, y_n)]_\alpha,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \\ \qquad \xrightarrow{1}_\psi \langle [\texttt{skip}]_\alpha,\ \langle \sigma, h'\rangle,\ \mu,\ D'\rangle\end{array}}$$

S-Function-Fail

$$\frac{\begin{array}{c}\psi(f) = \langle d, r * R(d_1 \geq \Delta(y_1), \ldots, d_n \geq \Delta(y_n))\rangle \\ \exists j = 1 \ldots, n.\, \texttt{calc-eps}(y_j, D) \geq d_i\end{array}}{\begin{array}{l}\langle [x = f(y_1, \ldots, y_n)]_\alpha,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \\ \qquad \xrightarrow{1}_\psi \langle [\texttt{skip}]_\alpha, \bot, \mu, D'\rangle\end{array}}$$

Figure 4.8: Semantics of statically verified functions

Global Semantics. We define a global configuration as $\langle \epsilon, \mu, \omega, P\rangle$, consisting of a global environment $\epsilon \in Env = Pid \mapsto \Lambda$, a set of typed channels $\mu \in Channel = Pid \times Pid \times Type \to Val^*$, global uncertainty map $\omega \in Pid \mapsto D$, and the program P. As shown in Figure 4.9, small step transitions of the form $(\epsilon, \omega, \mu, P) \xrightarrow{\alpha, r}_\psi (\epsilon', \omega', \mu', P')$ define a process α taking a step and thus changing the global configuration. Inter-process communication happens using typed channels – though processes adding to and reading from the relevant queue.

$$\frac{\begin{array}{c}\epsilon[\alpha] = \langle \sigma, h\rangle \quad \omega[\alpha] = D \quad \langle P_\alpha,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{p} \langle P'_\alpha,\ \langle \sigma, h'\rangle,\ \mu',\ D'\rangle \\ p_s = P_s[\alpha \mid (\epsilon, \mu, P_\alpha \parallel P_\beta)] \quad p' = p \cdot p_s\end{array}}{(\epsilon, \omega, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha, p'}_\psi (\epsilon[\alpha \mapsto \langle \sigma', h'\rangle], \omega[\alpha \mapsto D'], \mu', P'_\alpha \parallel P_\beta)}$$

$$\frac{\begin{array}{c}\epsilon[\alpha] = \langle \sigma, h\rangle \quad \omega[\alpha] = D \\ \langle P_\alpha,\ \sigma, h, \mu, D\rangle \xrightarrow{p} \langle P'_\alpha, \bot, \mu', D'\rangle \\ p_s = P_s[\alpha \mid (\epsilon, \mu, P_\alpha \parallel P_\beta)] \quad p' = p \cdot p_s\end{array}}{(\epsilon, \omega, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha, p'}_\psi (\bot, \omega, \mu', \texttt{skip})}$$

Figure 4.9: Diamond Global Semantics

4.3.3 Runtime Monitoring Soundness

Diamond's runtime system works across distributed processes. We use *Canonical Sequentialization* [87] to simplify our reasoning about the soundness of the runtime system. Canonical

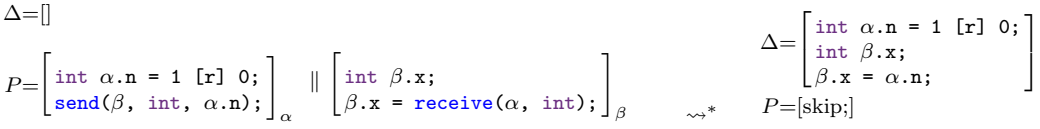


Figure 4.10: Canonical Sequentialization: An Example of the Rewriting Process.

sequentialization uses the assumption that correct programs tend to be well-structured to generate a sequential program that over-approximates the semantics of a parallel program. If such a sequentialized program can be generated, then the parallel program is deadlock-free, and local safety properties that hold for the sequentialized program also hold for the parallel program.

To be sequentializable, the parallel program must be *symmetrically nondeterministic* – each receive statement must only have a single matching send statement, or a set of symmetric matching send statements[1]. We use a set of rewrite rules of the form $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$ to rewrite a parallel program P to a sequential program Δ' step by step (the rules are available in [159, (A)]). The *context* Γ is used as a symbolic set of messages in flight, and P' is the part of the parallel program that remains to be rewritten. The sequentialization process applies the rewrite steps until the entire program is rewritten to Δ'. We extend the results from prior work [87, 152] to show that rewrite rules maintain equivalent behavior between the original parallel program and the generated sequential program, i.e., they both produce the same environment and uncertainty map at the halting states of the programs.

Figure 4.11 shows a small program with inter-process communication (P) and its canonical sequentialization (Δ) generated using the rewrite rules. We show that the existence of a canonical sequentialization guarantees that uncertain intervals are not affected by the different possible interleavings of processes during execution, allowing us to generate correct monitoring code.

In contrast, consider the following program where the process α has a receive statement that receives from two other processes:

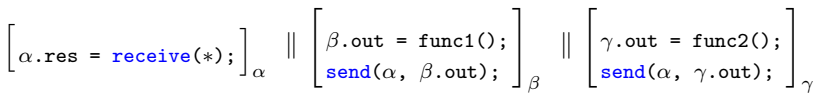


Figure 4.11: A program where execution order affect the results.

The final value of `res` depends on the runtime interleavings and it is difficult to generate monitoring code at compilation time that soundly calculates an uncertain interval combining

[1]Many popular parallel application patterns (e.g. Map, Reduce, Scatter-Gather, Stencil) exhibit symmetric non-determinism [87, 152]. Further, programs satisfying this property can be less error-prone [87].

all possible interleavings. Therefore, we limit our analysis only to programs with canonical sequentializations and prove that the runtime is sound.

We prove the following soundness theorem for Diamond programs with a canonical sequentialization.

Theorem 4.1 (Soundness of dynamic monitoring)**.** For programs not containing `track` and `endorse` statements, for all statements s, and for all x s.t. $\Theta \vdash x : \text{dynamic } t$, $\Theta \vdash s : \Theta'$ and $\langle s,\ \langle \sigma, D,\ \varphi\rangle\rangle \Downarrow \langle s',\ \langle \sigma', D',\ \varphi'\rangle\rangle \implies [\![\mathcal{R}^*(\ D'[x].\epsilon \geq \Delta(x))\,]\!](\sigma', \varphi') \geq D'[x].\delta$

Recall that Diamond's runtime monitors two properties for each `dynamic` variable x: (1) the maximum possible error magnitude ($D[x].\epsilon$) and (2) a probability ($D[x].\delta$) that the *precise* value of x is within $x \pm D[x].\epsilon$. The notation $\Delta(x)$ denotes the *true error* of a variable x, and $[\![\mathcal{R}^*(E)]\!](\sigma, \varphi)$ denotes the *true probability* that an environment σ sampled from the *environment distribution* φ satisfies the error comparison E.

4.3.4 Background and Definitions

In the following section we will define the terms used in Theorem 4.3 using the notation developed in Chisel [43]. We will define how to quantify *true error* and *true reliability* (probability of being within the error bound) in programs using *paired* execution semantics.

Definition 4.1 (Partial Trace Semantics for Parallel Programs)**.**

$$\langle s,\ \epsilon, \omega\rangle \overset{\tau, p}{\Longrightarrow}_{\psi} \langle s',\ \epsilon', \omega'\rangle \equiv \langle \epsilon, \omega, .,\ s\rangle \overset{\lambda_1, p_1}{\longrightarrow}_{\psi} \ \ldots \ \overset{\lambda_n, p_n}{\longrightarrow}_{\psi} \langle \epsilon', \omega', .,\ s'\rangle \tag{4.1}$$

This big-step semantics is a reflexive transitive closure of the small-step global semantics for programs and records a *trace* of the program. A trace $\tau \in T \to \cdot \mid \alpha :: T$ is a sequence of small step global transitions. The probability of the trace is the product of the probabilities of each transition. We only consider the environment and ignore differences in the message channels for this definition as we are concerned about differences in environment for programs. This semantics defines the probability of the program reaching the final state following one possible execution path. In the next definition, we aggregate the probabilities of all such traces that reach the same final state.

Definition 4.2 (Aggregate Semantics for Parallel Programs)**.**

$$\langle s,\ \epsilon, \omega\rangle \overset{p}{\Longrightarrow}_{\psi} \langle s',\ \epsilon', \omega'\rangle \text{ where } p = \sum_{\tau \in \mathrm{T}} p_\tau \text{ such that, } \langle s,\ \epsilon, \omega\rangle \overset{\tau, p_\tau}{\Longrightarrow}_{\psi} \langle s',\ \epsilon', \omega'\rangle \tag{4.2}$$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program starts in an environment ϵ and terminates in an environment ϵ'. This accumulates the probability over all possible interleavings that end up in the same final state.

Paired Execution Semantics. To define *true error* and *true reliability* we define a *paired execution semantics* that pairs an original (without uncertainty) execution of a program with an execution that contain errors, expanding the definition from Rely.

Definition 4.3 (Paired Execution Semantics)**.** $\langle s,\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s',\ \langle \epsilon', \omega',\ \varphi'\rangle\rangle$ such that,

$$\langle s,\ \epsilon, \omega\rangle \overset{\tau, p}{\Longrightarrow}_{1\psi} \langle s',\ \epsilon', \omega'\rangle \text{ and } \varphi'(\epsilon'_a) = \sum_{\epsilon_a \in\ Env} \varphi(\epsilon_a) \cdot p_a \text{ where } \langle s,\ \epsilon_a, \omega\rangle \overset{\cdot, p_a}{\Longrightarrow}_{\psi} \langle s',\ \epsilon', \omega'\rangle$$

This relation states that from a configuration $\langle \epsilon, \omega,\ \varphi\rangle$ consisting of an environment ϵ, dynamic map ω and an *environment distribution* $\varphi \in \Phi$, the paired execution yields a new configuration $\langle \epsilon', \omega',\ \varphi'\rangle$. The environments ϵ and ϵ' and the dynamic maps ω and ω' are related by the fully deterministic execution (1_ψ). The distributions φ and φ' are probability mass functions that map an environment to the probability that the execution is in that state. In particular, φ is a distribution on states before the execution of s whereas φ' is the distribution on states after executing s.

The *true error* of a variable x ($\Delta(x)$) is defined as the difference in x in any run compared to its value in the fully deterministic execution (1_ψ). The *true probability* of the program satisfying an accuracy predicate Q_A is defined using the *environment distributions*. $[\![\mathcal{R}^*(Q_A)]\!]$ is the probability that an environment satisfies Q_A: $[\![\mathcal{R}^*(Q_A)]\!](\epsilon, \varphi) = \sum_{\epsilon_u \in \mathcal{E}(Q_A, \epsilon)} \varphi(\epsilon_u)$. where $\mathcal{E}(Q_a, \epsilon)$ represents the set of all environments in which the predicate Q_A is satisfied

$$\mathcal{E}(Q_A, \epsilon) = \{\epsilon' \ \mid\ \epsilon' \in Env\ \wedge\ \epsilon' \in [\![Q_A]\!]\} \tag{4.3}$$

4.3.5 Proof of Soundness

We can now use these definitions to prove our soundness theorem. We need to show that if the program s type checks, and evaluates in the global environment σ and uncertainty map D to s', resulting in the environment σ' and uncertainty map D', then, for all dynamic variables x, the *true error* of x ($\Delta(x)$) is at most $D'[x].\epsilon$ with probability at least $D'[x].\delta$. This indicates that we soundly over-approximate the uncertainty of x.

Similar to Parallely, programs in Diamond satisfy a *non-interference* property enforced using the type system. This ensures that dynamic typed variables do not affect the control flow of the program (except through the conditional choice statements). Therefore control flow remains unaffected by uncertainty in the data.

First, we use induction over the sequential subset of Diamond to show that the theorem holds. We prove this theorem using induction on the length of the trace from s to s'. If it is 0, theorem holds as ω is initialized to be $\langle 0,\ 1\rangle$ for all dynamically monitored variables.

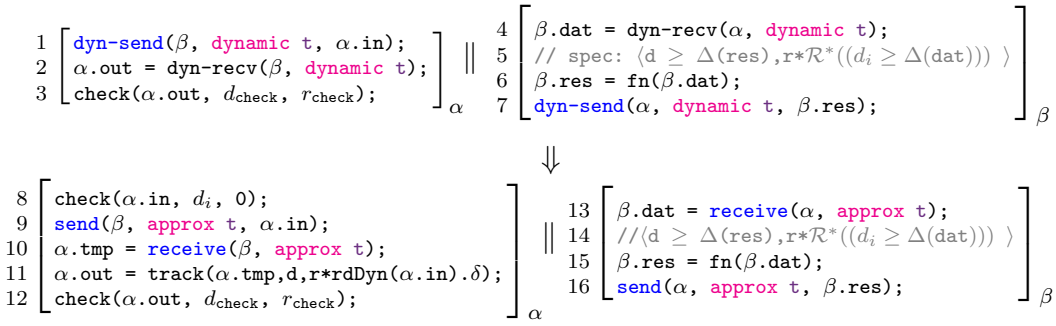


Figure 4.12: Optimizations Using Static Analysis in Diamond.

We will assume that the statement true for any trace of length n. Therefore we see that, $\langle s,\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s^n,\ \langle \epsilon^n, \omega^n,\ \varphi^n\rangle\rangle$ and $\forall x,\ [\![\mathcal{R}^*(\omega^n[x].\epsilon\ \geq\ \Delta(x))]\!](\epsilon^n, \varphi^n) \geq \omega[x]$.
Next, We will reason over all possible ways of taking the next step.
$\langle s^n,\ \langle \epsilon^n, \omega^n,\ \varphi^n\rangle\rangle \Downarrow \langle s^{n+1},\ \langle \epsilon^{n+1}, \omega^{n+1},\ \varphi^{n+1}\rangle\rangle$ from an individual process taking the following step: $\langle \epsilon^n, \mu^n, D^n,\ s^n\rangle \xrightarrow{\alpha,p}_{\psi} \langle \epsilon^{n+1}, \mu^{n+1}, D^{n+1},\ s^{n+1}\rangle$
Case S-Assign-Dyn (y = e): From the semantics of assignment (Figure 4.4) we can see that only the assigned variable in the statement changes in the environment. The maximum error and error confidence of all the other variables remain the same and the property follows from the inductive hypothesis. We need to show that the theorem holds if the assigned variable is of dynamic type.

We start with the observation that by definition,

$$[\![\mathcal{R}^*(\omega'[y].\epsilon \geq \Delta(y))]\!](\epsilon^{n+1}, \varphi^{n+1}) = \sum_{\epsilon_u \in \mathcal{E}(\omega'[y].\epsilon \geq \Delta(y), \epsilon^{n+1})} \varphi^{n+1}(\epsilon_u) \tag{4.4}$$

The subset of correct executions is a subset of all executions that end up at states equivalent to ϵ^{n+1} ($\mathcal{E}(\omega'[y].\epsilon \geq \Delta(y), \epsilon^{n+1})$).

Assignment is deterministic and does not introduce any uncertainty. Therefore, the maximum error that y can accumulate is determined through the errors in the variables used in e. We calculate this based on the `calc-eps(e,D)` function defined in Figure 4.5 using interval arithmetic. The soundness of the calculations has been shown in prior work [160]. Next we need to calculate the probability of the execution ending up at a state where the error is within the calculated bound.

As the system type checked, we know that only dynamic typed variables or `precise` typed variables are used in e. Precise typed variables do not contribute any additional uncertainty. From the inductive hypothesis, we can assume that the maximum error of the dynamic typed variables are calculated correctly.

The probability that the error exceeding the bound is calculated as the probability that

any variable in e is outside the intervals used in the previous calculation. We use the union bound to calculate the probability and show that it is sound in lemma 3 [159, (D)].

Case S-Prob-True ($y = e_1[r]e_2$): Similarly, from the definition of semantics we know that only the variable assigned to in the statement changes in the environment.

If the assigned variable is typed `dynamic`, The 1ψ execution results in the variable y having the value of e_1. Therefore we know that the maximum error y can have in a correct execution is the error from e_1 which we calculate similar to the above case.

But in this statement the assignment is not deterministic. Therefore the error confidence of y is the probability that e_1 was executed *and* that the error in e_1 is within bounds. As these two events are independent we can multiply the relevant probabilities to calculate the error confidence.

Case S-If-True, S-IF-False: As the program type checked only precise values are allowed in the conditionals. Therefore they do not introduce any relative error from a precise execution. In addition, the environment does not change, therefore the runtime does not need to update the Dynamic map.

Case S-Array-Store, S-Array-Load: as $\Theta \vdash s : \Theta'$, the array indexes are calculated using precise values. Therefore the reliability only depend on the data on the array location being accessed. Therefore the dynamic map is set according to the value mapping to that location. The property follows from the inductive hypothesis.

Case S-Dbl-To-Float: We calculate the maximum error based on the error of the value being cast-ed as shown in `calc-casting-error (x ,v ,D)` function. As casting is assumed to be done deterministic, the error confidence is the same as that of the cast-ed value.

We can use similar reasoning for the remaining sequential statements in Diamond.

QED.

Next, we utilize canonical sequentialization to prove that the theorem holds for the parallel subset of the language as well. First, we will extend the results from [152] to prove that if we can rewrite a parallel program P into a sequential program Δ, then P and Δ have equivalent behavior. We will use this fact to reason that our proof of soundness for the sequential subset of Diamond is also applicable to parallel programs that can be canonically sequentialized.

4.3.6 Rewrite rules

Following are the new rewrite rules for the statements added by the Diamond language. The notation uses the definitions from Parallely (Section 3.5.2). The remaining rewriting

rules from Parallely can be trivially extended.

$$\text{R-CONDSEND}\quad \frac{\Delta \models x = \beta \qquad \Gamma[\alpha,\beta,t] = m \qquad \Gamma' = \Gamma[\alpha,\beta,t \mapsto m + +y]}{\Gamma, \Delta, [\text{cond-send}(x,t,y,)]_\alpha \rightsquigarrow \psi\Gamma', \Delta, \texttt{skip}}$$

$$\text{R-CONDRECEIVE}\quad \frac{\Delta \models x = \alpha \qquad \Gamma[\alpha,\beta,t] = y :: m \qquad \Gamma' = \Gamma[\alpha,\beta,t \mapsto m] \qquad \Delta' = [\beta.\text{y} = \alpha.y\ [\psi(\alpha,\beta,t)]\ \beta.y]_\beta}{\Gamma, \Delta, [\text{y} = \text{cond-receive}(x,t)]_\beta \rightsquigarrow \psi\Gamma', \Delta; \Delta', \texttt{skip}}$$

$$\text{R-DYNSEND}\quad \frac{\Delta \models x = \beta \qquad \Gamma[\alpha,\beta,t] = m \qquad \Gamma' = \Gamma[\alpha,\beta,t \mapsto m + +y]}{\Gamma, \Delta, [\text{dyn-send}(x,t,y)]_\alpha \rightsquigarrow \psi\Gamma', \Delta, \texttt{skip}}$$

$$\text{R-DYNRECEIVE}\quad \frac{\Delta \models x = \alpha \qquad \Gamma[\alpha,\beta,t] = y :: m \qquad \Gamma' = \Gamma[\alpha,\beta,t \mapsto m] \qquad \Delta' = [\beta.\text{y} = \alpha.y\ [\psi(\alpha,\beta,t)]\ \beta.y]_\beta}{\Gamma, \Delta, [\text{y} = \text{dyn-receive}(x,t)]_\beta \rightsquigarrow \psi\Gamma', \Delta; \Delta', \texttt{skip}}$$

$$\text{R-COMMONREPEAT}\quad \frac{\Delta \models N = M \qquad \Gamma, \Delta, [S_0]_\alpha \parallel [S_1]_\beta \rightsquigarrow \psi\Gamma, \Delta; \Delta', \texttt{skip}}{\Gamma, \Delta, [\text{repeat N } S_0]_\alpha \parallel [\text{repeat M } S_1]_\beta \rightsquigarrow \psi\Gamma, \Delta; \text{repeat N } \{\Delta'\}, \texttt{skip}}$$

Figure 4.13: Rewrite Rules

Lemma 4.1. If $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$ then $\Gamma, \Delta, P \sqsubseteq \Gamma', \Delta; \Delta', P'$

Proof: The proof is by induction on the derivation of $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$. Each rewrite rule has a seperate case. Below are the cases for the new rewrite rules:

Case R-DynSend: Let $(\epsilon, \mu, \omega) \in [\![\Delta, \Gamma]\!]_\varnothing$ and assume the following steps take place $(\epsilon, \mu, \omega, [\text{dyn-send}(x,t,y)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$. `dyn-send` is a left mover (proof is same as the proof that cond-send is a left mover), therefore we can move it to the sequential prefix as follows $(\epsilon, \mu, \omega, [\text{dyn-send}(x,t,y)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$.
Suppose $\epsilon(x) = \beta$. By the R-DynSend rewrite step, $\Gamma' = \Gamma[(\alpha,\beta,t) \mapsto \Gamma(\alpha,\beta,t) + +y]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon', \mu', \omega') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$.
Then $\epsilon' = \epsilon$, $\omega' = \omega$, and $\mu' = \mu[(\alpha,\beta,t) \mapsto \mu(\alpha,\beta,t) + +m][(\alpha,\beta,D_t) \mapsto \mu(\alpha,\beta,D_t) + +d]$, where $d = \omega(y)$ and m is either $\epsilon(y)$ or $\varnothing$.
Suppose the send succeeds. Then by semantic rule E-DynSend-True and E-CondSend-True, $(\epsilon, \mu[(\alpha,\beta,t) \mapsto \mu(\alpha,\beta,t) + +\epsilon(y)][(\alpha,\beta,D_t) \mapsto \mu(\alpha,\beta,D_t) + +d], \omega, P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$.
Suppose the send fails. Then by semantic rule E-DynSend-False and E-CondSend-False, $(\epsilon, \mu[(\alpha,\beta,t) \mapsto \mu(\alpha,\beta,t) + +\varnothing][(\alpha,\beta,D_t) \mapsto \mu(\alpha,\beta,D_t) + +d], \omega, P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$ that is, $(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$.
Therefore, $(\epsilon, \mu, \omega, [\text{dyn-send}(x,t,n)]_\alpha \ltimes P_x) \sqsubseteq (\epsilon', \mu', \omega', P_x)$.

Case R-CondSend: This proof is similar to the R-DynSend proof. The main differences are that the dyn-send is replaced with a cond-send, the dynamic channel is untouched, and the semantics do not step through the E-DynSend-True or E-DynSend-False rules.

Case R-DynReceive: Assume $(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x, t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$ where $(\epsilon, \mu, \omega) \in [\![\Delta, \Gamma]\!]_\varnothing$. dyn-receive is a left mover (proof is same as the proof that cond-receive is a left mover), therefore $(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x, t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$
Suppose $\epsilon(x) = \alpha$. By the R-DynReceive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.y = \alpha.y\ [\psi(\alpha, \beta, t)]\ \beta.y]$ when $\text{head}(\Gamma(\alpha, \beta, t)) = y$. Suppose $(\epsilon', \mu', \omega') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$, Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))]$ and $\omega' = \omega[\beta.y \mapsto d]$ where $d = \texttt{get-dyn-rec}(\text{head}(\mu(\alpha, \beta, D_t)), \psi(\alpha, \beta, t))$.
Further, either $\epsilon' = \epsilon[\beta.y \mapsto \alpha.y]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.y$ or $\epsilon' = \epsilon$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$. Suppose the send succeeded. Then by the definition of semantic rules E-DynReceive-True and E-CondReceive-True,

$$\begin{aligned}(\epsilon[\beta.y \mapsto \alpha.y], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))], \omega[\beta.y \mapsto d], P_x) \\ \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.5}$$

If the send failed. Then by semantic rule E-DynReceive-False and E-CondReceive-False,

$$\begin{aligned}(\epsilon, \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))], \omega[\beta.y \mapsto d], P_x) \\ \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.6}$$

that is, $(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$
Therefore, $(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x, t)]_\beta \ltimes P_x) \sqsubseteq (\epsilon', \mu', \omega', P_x)$.

Case R-CondReceive: This proof is similar to the R-DynReceive proof. The main differences are that the dyn-receive is replaced with a cond-receive, the dynamic channel is untouched, and the semantics do not step through the E-DynReceive-True or E-DynReceive-False rules.

Case R-CommonRepeat: Since the rewrite rule ensures that N and M are the same, this proof refers to both as N. The proof for this case is by induction on N, the number of repetitions.
Suppose $N = 1$. Then, $[\text{repeat N } S_0]_\alpha \parallel [\text{repeat N } S_1]_\beta \equiv [S_0]_\alpha \parallel [S_1]_\beta$. This can be rewritten to Δ', which is equivalent to repeat N $\{\Delta'\}$.
Suppose $N > 1$. $[\text{repeat N } S_0]_\alpha \parallel [\text{repeat N } S_1]_\beta \equiv [S_0; \text{repeat N-1 } S_0]_\alpha \parallel [S_1; \text{repeat N-1 } S_1]_\beta$ by inductive hypothesis, this can be rewritten to Δ'; repeat N-1 $\{\Delta'\}$, which is equivalent to repeat N $\{\Delta'\}$.

Transitions to Error States: Some statements, such as function calls and check statements, can make the parallel program transition to an error state. This happens if 1) the input to a function does not satisfy its function specification, 2) a check fails for a variable or a checkarray fails for an element of an array. In such cases, using the inductive hypothesis, we can say that if such a failure occurs in the parallel program, it will also occur in the sequential program.

Lemma 4.2. If $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$ then $\Gamma, \Delta, P \sqsupseteq \Gamma', \Delta; \Delta', P'$

Proof: The proof is by induction on the derivation of $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$. Each rewrite rule has a seperate case. Below are the cases for the new rewrite rules:

Case R-DynSend: Let $(\epsilon', \mu', \omega') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$. By the R-DynSend rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +y]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon, \mu, \omega) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$, $\omega' = \omega$, and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d]$, where $d = \omega(y)$ and m is either $\epsilon(y)$ or $\varnothing$. Therefore,

$$\begin{aligned}(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d], \omega, P_x) \\ \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.7}$$

by definition of the semantic rules E-DynSend-True and E-CondSend-True, or E-DynSend-False and E-CondSend-False (depending on m),

$$\begin{aligned}(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha; P_x) \xrightarrow{\alpha} (\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D) \\ \mapsto \mu(\alpha, \beta, D) + +d], \omega, P_x)\end{aligned} \tag{4.8}$$

Therefore, $(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$. Since dynsend is a left mover, $(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$.

Case R-CondSend: This proof is similar to the R-DynSend proof. The main differences are that the dyn-send is replaced with a cond-send, the dynamic channel is untouched, and the semantics do not step through the E-DynSend-True or E-DynSend-False rules.

Case R-DynReceive: Let $(\epsilon', \mu', \omega') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume $(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$. By the R-DynReceive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and the sequential prefix, $\Delta' = [\beta.y = \alpha.y\ [\psi(\alpha, \beta, t)]\ \beta.y]$ when $\text{head}(\Gamma(\alpha, \beta, t)) = y$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))]$ and $\omega' = \omega[\beta.y \mapsto d]$ where $d = \texttt{get-dyn-rec}(\text{head}(\mu(\alpha, \beta, D_t)), \psi(\alpha, \beta, t))$. Further, either $\epsilon' = \epsilon[\beta.y \mapsto \alpha.y]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.y$ or $\epsilon' = \epsilon$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose the send succeeded,

$$\begin{aligned}(\epsilon[\beta.y \mapsto \alpha.y], \mu[(\alpha,\beta,t) \mapsto \text{pop}(\mu(\alpha,\beta,t))][(\alpha,\beta,D_t) \\ \mapsto \text{pop}(\mu(\alpha,\beta,D_t))], \omega[\beta.y \mapsto d], P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.9}$$

by semantic rule E-DynReceive-True and E-CondReceive-True,

$$\begin{aligned}(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x,t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon[\beta.y \mapsto \alpha.y], \mu[(\alpha,\beta,t) \\ \mapsto \text{pop}(\mu(\alpha,\beta,t))][(\alpha,\beta,D_t) \\ \mapsto \text{pop}(\mu(\alpha,\beta,D_t))], \omega[\beta.y \mapsto d], P_x)\end{aligned} \tag{4.10}$$

If instead, the send failed.

$$\begin{aligned}(\epsilon, \mu[(\alpha,\beta,t) \mapsto \text{pop}(\mu(\alpha,\beta,t))][(\alpha,\beta,D_t) \mapsto \text{pop}(\mu(\alpha,\beta,D_t))], \omega[\beta.y \mapsto d], P_x) \\ \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.11}$$

by semantic rule E-DynReceive-False and E-CondReceive-False,

$$\begin{aligned}(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x,t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon, \mu[(\alpha,\beta,t) \mapsto \text{pop}(\mu(\alpha,\beta,t))][(\alpha,\beta,D_t) \\ \mapsto \text{pop}(\mu(\alpha,\beta,D_t))], \omega[\beta.y \mapsto d], P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)\end{aligned} \tag{4.12}$$

Therefore, $(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x,t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$. Since dynreceive is a left mover, $(\epsilon, \mu, \omega, [y = \text{dyn-receive}(x,t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$.

Case R-CondReceive: This proof is similar to the R-DynReceive proof. The main differences are that the dyn-receive is replaced with a cond-receive, the dynamic channel is untouched, and the semantics do not step through the E-DynReceive-True or E-DynReceive-False rules.

Case R-CommonRepeat: This proof is similar to the R-CommonRepeat proof for Lemma 3.5.

Transitions to Error States: Some statements, such as function calls and check statements, can make the parallel program transition to an error state. In such cases, using the inductive hypothesis, we can say that if such a failure occurs in the sequential program, it will also occur in the parallel program.

The above two lemmas allows us to define the following theorem, which similar to Parallely states that the sequentialization preserves the halting program states.

Theorem 4.2 (equivalence of sequentialized program for halted states)**.** If $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$ then $(\varnothing, \varnothing, P) \rightarrow^* (\epsilon_H, \varnothing, H)$ if and only if $(\varnothing, \varnothing, \Delta) \rightarrow^* (\epsilon'_H, \varnothing, H')$ such that $\epsilon_H = \epsilon'_H$ where all processes are permanently halted in $(\epsilon_H, \varnothing, H)$.

Proof: Similar to Parallely, Theorem 4.2 directly follows form Lemma 4.2 and Lemma 4.1, which state that the sequentialized program is an over-approximation of the parallel program and that the parallel program is an over-approximation of the sequential program respectively (with respect to halted processes). Thus, the sequentialized program is equivalent to the parallel program with respect to halted processes.

Corollary 4.1 (Deadlock-Freedom of Sequentializable Programs)**.** If a parallel program P can be sequentialized to Δ, then P is deadlock free.

Corollary 4.1 is proved in [87].

Limitations: Our analysis only applies to programs with `track` and `endorse` statements if developers use them in a sound manner. For `track` statements, developers must ensure that the bounds they provide are a sound over-approximation of the true uncertainty at that program point. As in prior work [41], by inserting `endorse` statements, developers certify that treating the relevant `approx` or `dynamic` value as `precise` is always safe and will not result in undesirable behavior.

4.4 OPTIMIZATIONS FOR REDUCING OVERHEAD

We implemented several optimizations that transform the programs to reduce the overhead of dynamic monitoring and proved them to be sound.

4.4.1 Soundness of Optimizations

For each optimization we show that both the original program (s) and the optimized version (s_{opt}) produce the same behavior, i.e., if the original program fails a check, the optimized version is also guaranteed to fail. Canonical sequentialization makes such proofs easier. Formally, we define the soundness of an optimization as follows:

Definition 4.4 (Optimization soundness)**.** For a program s and its optimized version s_{opt}, $\langle s,\, \langle \sigma, h\rangle,\, \mu,\, D\rangle \rightarrow^*_\psi \langle s', \bot, \_, \_\rangle \implies \langle s_{opt},\, \langle \sigma, h\rangle,\, \mu,\, D\rangle \rightarrow^*_\psi \langle s'', \bot, \_, \_\rangle$

This definition states that if there is an execution where the original program s starting from an environment σ, heap h, uncertainty map D, and the global channel set μ evaluates to s' and enters into the error state ($\perp$), the optimized version s_{opt} starting from the same state σ, heap h, and D must also enter the error state (even if the final channel or uncertainty map states differ). For each optimization, we show that the pairs s and s_{opt} are sound according to this definition.

Proving the soundness of optimizations in this domain requires us to show that the two parallel programs produce the same result with regards to the dynamic monitoring. We can simplify this process significantly by using sequentialization. We first show that the two versions of the program can be sequentialized to some s^{seq} and s^{seq}_{opt}. We know that these sequentializations produce final environments that are equivalent to the original versions as proven in Theorem 4.2. We can now simplify the proof to reasoning over the two sequential programs s^{seq} and s^{seq}_{opt}. We can next argue over all executions resulting in a check failure in s^{seq} and show that they result in a check failure in s^{seq}_{opt}.

4.4.2 Communication.

When communicating large `dynamic` type arrays, Diamond must also communicate the uncertain interval for each array element, resulting in a large communication overhead. One way to reduce this overhead is to calculate a single conservative approximation of the set of uncertain intervals for the array elements. For example, the maximum error of any element of an array can be soundly over-approximated by the largest maximum error among all of its elements (similarly, the smallest error confidence). The process sending the data calculates the conservative approximation while using the regular communication primitives for the data. At the end it sends the conservatively approximate uncertain interval. At the receiver, this uncertain interval is taken as the uncertain interval of *each* element in the received array and the compiler adds track statements to restart dynamic monitoring.

This optimization does not approximate the uncertain interval of the array at all program points, rather it affects only communication statements. Even with the resulting loss in precision of the analysis, Diamond still achieves better results than existing static analyses which use a single uncertain interval for arrays through the *entire* program.

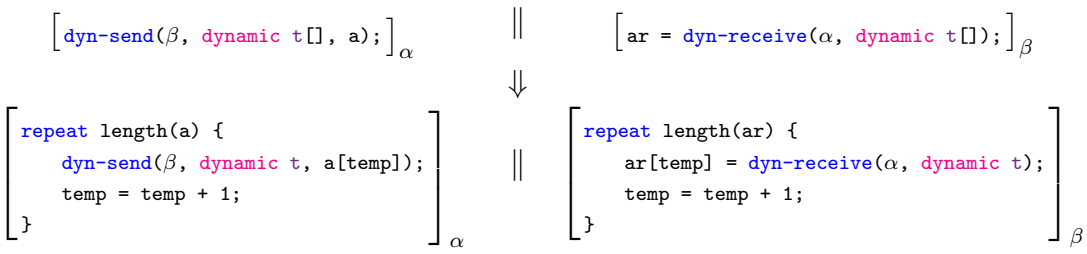


Figure 4.14: Array Communication in Diamond.

Soundness. In Diamond, communication of arrays is handled by converting them to a set of send and receive statements for each array element. The following figure shows how Diamond de-sugars array communication into a set of `send`s and `receive`s. Based on the semantics of communicating dynamic values, they are sent on the dynamic channel.

the sequentialized version of this communication pattern converts it into a traversal of the array copying each element (the lengths of the two arrays need to be the same and this needs to be verifiable at sequentialization). Based on the semantics of assignment the dynamically monitored interval of `ar` elements is updated at each assignment.

```
[precise int β.temp = 0;
precise int α.temp = 0;

repeat length(ar) {
    ar[β.temp] = a[α.temp] [ψ(α,β, dynamic t)] ar[β.temp];
    β.temp = β.temp + 1;
    α.temp = α.temp + 1;
}]seq
```

Figure 4.15: Sequentialized Form of Array Communication in Diamond.

Our optimization changes the de-sugaring step of the `send` statement to do the following steps: 1) convert all `dynamic` typed data to `approx` type using endorse statements, 2) calculate the maximum error and minimum reliability of the array elements by looking up the relevant entries in the dynamic map, 3) send the values of the array using the `approx` channel, and 4) send the calculated interval value on the `precise` channel.

Similarly, the optimization changes the de-sugaring step of the `receive` statement to do the following steps: 1) receive each element of the array, and convert them to `dynamic` type using the `track` statements, 2) receive the dynamic reliability, 3) Update the dynamic map to the received dynamic property.

The correctness of this communication related optimization can be easily proved on the sequentialized version of the program. If we consider s_{array} to be the de-sugared optimized

code, we wish to prove the proposition 4.1 that extends the soundness proof from before to work for array assignment. The proposition notes that as each array element is copied over, the resultant array elements have their interval correctly updated in the runtime. The code presented in Figure 4.16 shows the optimized version of the program.

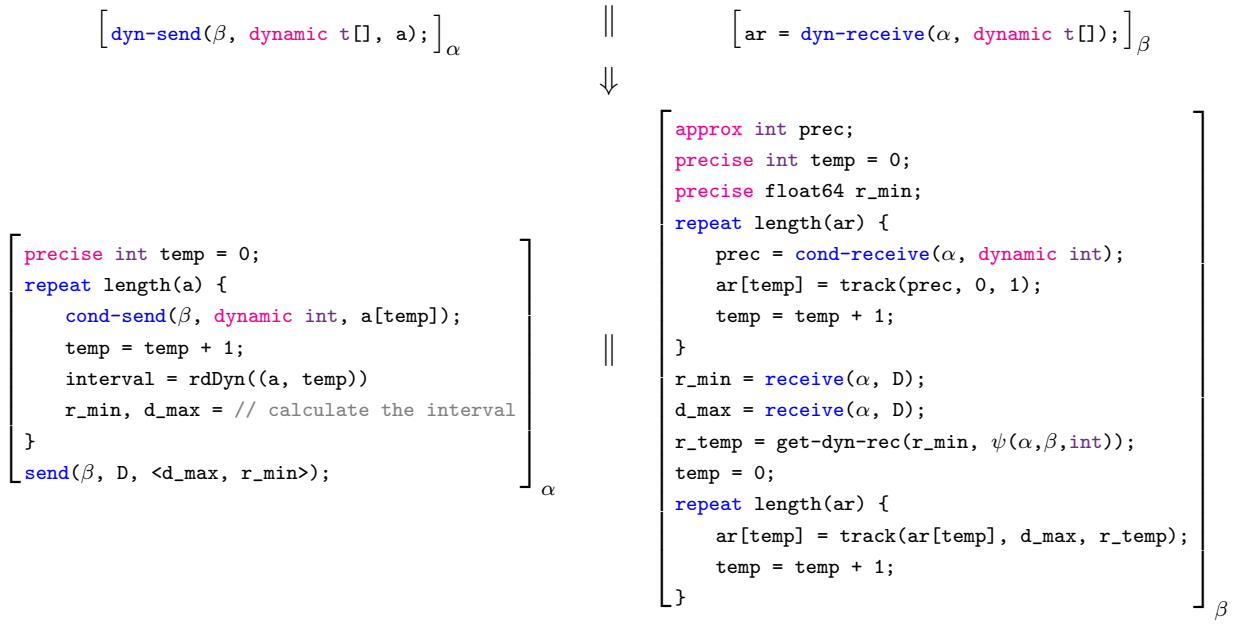


Figure 4.16: Optimized Array Communication in Diamond.

The communication in Figure 4.16 generates the sequentialization in Figure 4.17.

Proposition 4.1. If $\langle s_{array},\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s',\ \langle \epsilon', \omega',\ \varphi'\rangle\rangle$ then, $\forall i \in [0..\text{length}(a)).$ $[\![\mathcal{R}^*(\omega'[ar,i].\epsilon \geq \Delta(ar[i]))]\!](\epsilon', \varphi') \geq \omega'[ar,i].\delta$

Proof. (sketch) Note that $[\![\omega'[ar,i].\epsilon \geq \Delta(ar[i])]\!]$ denotes the subset of approximate environments where the error in `ar[i]` is less than $\omega'[ar,i].\epsilon$. And $[\![\mathcal{R}^*(\omega'[ar,i].\epsilon \geq \Delta(ar[i]))]\!](\epsilon', \varphi')$ denotes the probability of being in such an environment.
Therefore, for any y s.t $y \geq \omega'[ar,i].\epsilon$, $[\![\omega'[ar,i].\epsilon \geq \Delta(ar[i])]\!] \subseteq [\![y \geq \Delta(ar[i])]\!]$
Therefore,

$$\begin{aligned}[\![\mathcal{R}^*(\omega'[ar,i].\epsilon \geq \Delta(ar[i]))]\!](\epsilon', \varphi') \geq \omega'[ar,i].\delta \\ \Rightarrow [\![\mathcal{R}^*(y \geq \Delta(ar[i]))]\!](\epsilon', \varphi') \geq \omega'[ar,i].\delta\end{aligned} \tag{4.13}$$

Based on Theorem 4.3, we will assume that the runtime is sound before reaching this point of the program. Therefore, $\forall i \in [0..\text{length}(a)].\ [\![\mathcal{R}^*(\omega[(a,i)].\epsilon \geq \Delta(a[i]))]\!](\epsilon, \varphi) \geq \omega[(a,i)].\delta$

```
approx int β.prec;
precise int β.temp = 0;
precise int α.temp = 0;
precise float64 β.r_min = 1;
precise float64 β.d_max = 0;
precise float64 α.r_min;

repeat length(a) {
    β.prec = α.a[temp] [ψ(α, β, dynamic int)] β.prec
    ar[β.temp] = track(β.prec, 0, 1);

    α.interval = rdDyn((α.a, α.temp))
    if α.r_min > α.interval[0] then {
        α.r_min = α.interval[0];
    };
    if α.d_max < α.interval[1] then {
        α.r_min = α.interval[1];
    };

    β.temp = β.temp + 1;
    α.temp = α.temp + 1;
}
β.r_min = α.r_min;
β.d_max = α.d_max;
β.r_temp = get-dyn-rec(β.r_min, ψ(α,β,int));
β.temp = 0;
repeat length(ar) {
    ar[β.temp] = track(ar[β.temp], β.d_temp, β.r_temp);
    β.temp = β.temp + 1;
}
```
seq

Figure 4.17: Sequentialized Form of Optimized Array Communication in Diamond.

Based on the calculation of α.d_max, we see that $\forall i \in [0..\text{length}(a)].\ \alpha.\text{d\_max} \geq \omega[(a,i)].\epsilon$ Therefore as discussed above, $\forall i \in [0..\text{length}(a)]$,

$$\begin{aligned} &[\![\mathcal{R}^*(\omega'[ar,i].\epsilon \ \geq\ \Delta(ar[i]))]\!](\epsilon',\varphi') \geq \omega'[ar,i].\delta \\ \Rightarrow\ &[\![\mathcal{R}^*(\alpha.\texttt{d\_max} \ \geq\ \Delta(ar[i]))]\!](\epsilon',\varphi') \geq \omega'[ar,i].\delta \end{aligned} \tag{4.14}$$

We can see that at the end of s_{array}, $\forall i \in [0..\text{length}(a)].\ \ \omega'[ar,i] = \alpha.\texttt{d\_max}$ due to the `track` statements. By the definition of `get-dyn-rec`, β.d_temp $= \omega[(a,i)] \times \psi(\alpha,\beta,\text{dynamic int})$. We can also prove the correctness of the probability $\omega'[ar,i].\delta$ similarly.
Note that the proof is on the sequentialized version of the code. Due to the equivalence we proved earlier the optimization does not have to be proved correct for parallel programs, which is difficult. QED.

4.4.3 Utilizing static analysis.

We can further reduce overheads by exploiting common communication patterns. For example, the program at the top of Figure 4.18 contains a remote procedure call. Process α sends an input to process β, which applies the function `fn` to the input and returns the

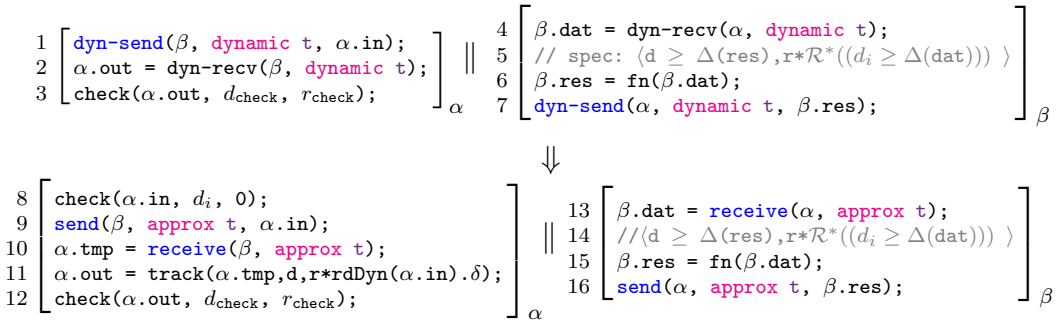


Figure 4.18: Optimizations Using Static Analysis in Diamond.

$$S^{seq} = \begin{bmatrix} \beta.\texttt{dat} = \alpha.\texttt{in}; \\ \beta.\texttt{res} = \texttt{fn}(\beta.\texttt{dat}); \\ \alpha.\texttt{out} = \beta.\texttt{res}; \\ \texttt{check}(\alpha.\texttt{out}, d_{\texttt{check}}, r_{\texttt{check}}); \end{bmatrix} \qquad S^{seq}_{opt} = \begin{bmatrix} \texttt{check}(\alpha.\texttt{in}, d_i, 0); \\ \beta.\texttt{dat} = \alpha.\texttt{in}; \\ \beta.\texttt{res} = \texttt{fn}(\beta.\texttt{dat}); \\ \alpha.\texttt{tmp} = \beta.\texttt{res}; \\ \alpha.\texttt{out} = \texttt{track}(\alpha.\texttt{tmp},\texttt{d},\texttt{r*rdDyn}(\alpha.\texttt{in}).\delta); \\ \texttt{check}(\alpha.\texttt{out}, d_{\texttt{check}}, r_{\texttt{check}}); \end{bmatrix}$$

Figure 4.19: Example Sequentializations Used in the Proofs

value. Transferring uncertain intervals along with the data can become expensive if many such calls are made.

We use existing static analysis techniques [42, 43, 152] to analyze only the remote function call and generate function specifications (as defined in Section 4.3), even if they are unable to analyze the entire program. Consider the transformed program at the bottom of Figure 4.18. Using the specification, Diamond produces the same behavior as the original program by generating code to 1) check if the specification requirements are satisfied (Line 8), 2) transfer the data as `approx` type (Line 9), 3) compute without dynamic monitoring, and 4) re-initialize dynamic monitoring using the error guarantees from the specification (Line 11).

This optimization can be safely used when the function performs no communication and has no other side effects. However, it may not be possible to verify some static specifications at runtime. For example: the runtime will not be able to calculate $\mathcal{R}^*(d_i \geq \Delta(\texttt{dat}))$ for some values for d_i. Therefore, this optimization may introduce some imprecision to the dynamic monitoring.

Soundness. Figure 4.19 shows the sequentialized versions of the programs from Figure 4.18. We will reason that S and S^{opt} are equivalent because S_{seq} and S^{opt}_{seq} are equivalent:

Proposition 4.2 (Soundness)**.** The optimization is sound.
$\langle S, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \bot, \mu', D'\rangle \implies \langle S^{opt}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s'', \bot, \mu'', D''\rangle$

Proof. As $\varnothing, \varnothing, S \rightsquigarrow^* \varnothing, S_{seq}, \texttt{skip}$, From Theorem 4.3 on the equivalence of sequentialized

programs we claim that, $\langle S, \langle \sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \perp, \mu', D'\rangle \implies \langle S_{\text{seq}}, \langle \sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \perp, \mu', D'\rangle$,

Based on the definition of Diamond's runtime, there are two statements in S_{seq} that can fail and lead to an error state.

Case 1: The function specifications requirements are not satisfied. Based on the definition of `S-FUNCTION-FAIL` rule, the failure results from an execution where,
$\langle S_{\text{seq}}, \langle \sigma, h\rangle, \mu, D\rangle \longrightarrow_{\psi} \langle \beta.\text{output} = \text{func}(\beta.\text{data}); s', \langle \sigma', h'\rangle, \mu', D'\rangle \longrightarrow_{\psi} \langle \texttt{skip}, \perp, \mu', D'\rangle$
Based on the definition of the rule, this implies that $\text{D}'[\beta.\text{data}].\epsilon \geq \text{d1}$
we can also see that, $\text{D}'[\beta.\text{data}] = \text{D}[\alpha.\text{input}]$.
Therefore, in a this program, $\text{D}[\alpha.\text{input}].\epsilon \geq \text{d1}$.
Now, lets consider $\text{S}^{\text{opt}}_{\text{seq}}$. Again, based the definition of the runtime, if $\text{D}[\alpha.\text{input}].\epsilon \geq \text{d1}$, the check at the start of the program will fail. This would result in a error state (`S-check-fail` rule in the semantics along with the definition of `dyn-check(`α`.input, d1 ,0 ,D)`).

Case 2: The check function (`check(`α`.output,` d_{check}`,` r_{check}`)`) at the end of S_{seq} fails.
Based on the definition of `S-check-fail`, $(\text{D}[\alpha.\text{output}].\epsilon > d_{\text{check}} \vee D[\alpha.\text{output}].\delta < r_{\text{check}})$
In addition, we can see that, $\text{D}'[\alpha.\text{output}] = \text{D}'[\beta.\text{output}]$
Therefore,

$$(D[\beta.\text{output}].\epsilon > d_{\text{check}}) \vee (D[\beta.\text{output}].\delta < r_{\text{check}}) \tag{4.15}$$

As the function `func` has been verified to guarantee the specification for all inputs, we can use the static analysis to identify maximum error that β.output can take. The rule that apply to functions calls perform the following updates defined in `S-FUNCTION` in Figure 4.4:
$\text{D}'[\beta.\text{output}] = \langle d, r \times \texttt{calc-del}(f(\beta.\text{data}), D)\rangle = \langle d, r \times D'[\beta.\text{data}].\delta\rangle$
and $D'[\beta.\text{data}] = D'[\alpha.\text{input}]$
Therefore, $\text{D}'[\beta.\text{output}] = \langle d, r \times D'[\beta.\text{data}].\delta\rangle = \langle d, r \times D'[\alpha.\text{input}].\delta\rangle$
So, based on 4.15,

$$(d > d_{\text{check}}) \vee (r \times D'[\alpha.\text{input}].\delta < r_{\text{check}}) \tag{4.16}$$

Let us again consider the execution of $\text{S}^{\text{opt}}_{\text{seq}}$, based the definition of the runtime, in Figure 4.5, we can see that, due to the `track` statement, $\text{D}[\alpha.\text{output}] = \langle d_{\text{check}}, r \times D[\alpha.\text{input}].\delta\rangle$
As, $(d > d_{\text{check}} \vee r \times D[\alpha.\text{output}].\delta \text{ ¡ } r_{\text{check}})$, this check will fail, resulting in an error state.

We can see that $\varnothing, \varnothing, S^{\text{opt}} \rightsquigarrow^* \varnothing, S^{\text{opt}}_{\text{seq}}, \texttt{skip}$. Therefore from Theorem 4.3,

$\langle S^{\text{opt}}_{\text{seq}}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \bot, \mu', D'\rangle \implies \langle S^{\text{opt}}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \bot, \mu', D'\rangle$,

QED.

4.4.4 Early checking.

For a subset of instructions we can perform static analysis to stop runtime monitoring earlier. We perform this task by *moving up* the check to the earliest possible location using a set of rewrites. The rewrite rule in Figure 4.20 are some examples.

$$\begin{bmatrix} \alpha\texttt{.x = } \alpha\texttt{.a + } \alpha\texttt{.b;} \\ \texttt{check(AExp, d, r);} \end{bmatrix} \Rightarrow \begin{bmatrix} \texttt{check(AExp[(}\alpha\texttt{.a+}\alpha\texttt{.b)/}\alpha\texttt{.x], d, r);} \\ \alpha\texttt{.x = } \alpha\texttt{.a + } \alpha\texttt{.b;} \end{bmatrix}$$

$$\begin{bmatrix} \alpha\texttt{.x = } \alpha\texttt{.a - } \alpha\texttt{.b} \\ \texttt{check(ae, d, r)} \end{bmatrix} \Rightarrow \begin{bmatrix} \texttt{check(ae[(}\alpha\texttt{.a-}\alpha\texttt{.b)/}\alpha\texttt{.x], d, r)} \\ \alpha\texttt{.x = } \alpha\texttt{.a - } \alpha\texttt{.b} \end{bmatrix}$$

$$\begin{bmatrix} \texttt{x = } e_1 \texttt{ [r\_exp] } e_2 \\ \texttt{check(ae, d, r)} \end{bmatrix} \Rightarrow \begin{bmatrix} \texttt{check(ae[AE(e\_1)/x], d, r/r\_exp)} \\ \texttt{x = } e_1 \texttt{ [r\_epx] } e_2 \end{bmatrix}$$

Figure 4.20: Moving checks earlier in Diamond.

In the first rule, Diamond looks for a check immediately following an addition. Since the error magnitude of the result of the addition is the sum of the error magnitudes of the variables that are being added, we can substitute the result variable $\alpha.x$ in the check with $\alpha.a + \alpha.b$. As the `calc-del` function of the runtime looks for the set of variables in the specification (AExp), the error probability is calculated correctly as well. Diamond can now safely move the check before the addition.

These re-write rules closely follow the static analysis as defined and proven sound in [152] for the sequential subset of the language. To extend the optimization for multiplication and division we cannot easily perform such optimizations because the maximum error depends on the actual value variables can take. We can use an interval analysis and use the static bounds on variables as an potential alternative.

This optimization reduces updates to the uncertainty map as monitoring can be stopped after the check is performed. However, it can only be applied when the check refers to variables from a single process. Further, the check cannot be moved up if error calculations depend on the value of variables (as in multiplication/division).

Soundness.

Proposition 4.3 (Soundness of optimization)**.** For all statements s their and optimized version s_{opt}, $\langle s, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{1}_{\psi} \langle s', \bot, \mu', D'\rangle \implies \langle s_{opt}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{1}_{\psi} \langle s'', \bot, \mu'', D''\rangle$

Proof. (sketch) As we apply the optimizations only on sequentializable programs, for each rule we can show the soundness considering the sequential program alone without needing to consider parallel interleavings, potential of deadlocks, etc.

For the first rule, the check looks up the dynamic property map for the variable x. Instead, we can move the check before the assignment as we can calculate the relevant error expression without having to evaluate the expression. These rules closely follow the static analysis as defined and proven sound in [152, 160] for the sequential subset of the language.

Based on the definitions in Figure 4.4, and Figure 4.5, we can see that s in the first rule goes into an error state due to $\texttt{calc-eps}(ae, D')$ evaluating to false. The only difference from D to D' is the execution of the assignment statement.

Based on the definition of the runtime we can see that, $D'[\alpha.x].\epsilon = D[\alpha.a].\epsilon + D[\alpha.b].\epsilon$ and $D'[\alpha.x].\delta = D[\alpha.a].\delta + D[\alpha.b].\delta - 1$. Since we replace x with $a + b$ in ae we can see that

$$\texttt{calc-eps}(ae, D') = \texttt{calc-eps}(ae[(\alpha.a+\alpha.b)/\alpha.x], D) \tag{4.17}$$

Similarly we can see that the error confidence is calculated correctly as,

$$\texttt{calc-del}(ae, D') \leq r \implies \left(\sum_{v \in \rho(ae)} D'[v].\delta\right) - (|\rho(ae)| - 1) \leq r \tag{4.18}$$

since $\rho(ae[(\alpha.a + \alpha.b)/\alpha.x]) - \rho(ae) = \{\alpha.a, \alpha.b\}$,

$$\begin{aligned} &\left(\sum_{v \in \rho(ae[(\alpha.a+\alpha.b)/\alpha.x])} D[v].\delta\right) - (|\rho(ae[(\alpha.a+\alpha.b)/\alpha.x])| - 1) \\ &= \left(\sum_{v \in \rho(ae)} D[v].\delta\right) - (|\rho(ae[(\alpha.a+\alpha.b)/\alpha.x])| - 1) \end{aligned} \tag{4.19}$$

To extend the optimization for multiplication and division we cannot easily perform such optimizations because the maximum error depends on the actual value variables can take. We can use an interval analysis and use the static bounds on variables as an potential alternative.

The last rule shows how communication in the parallel program represented as assignment in the sequentialized program gets optimized. We perform such optimizations until all the variables referred to in the `check` function belong to a single process. Optimizations that result in a check referring to variables of more than one process are abandoned. QED.

4.4.5 Debloating and compiler optimizations.

Diamond further reduces overhead by using constant propagation and dead code elimination to remove unnecessary updates to the uncertainty map. In addition, Diamond eliminates either error magnitude monitoring or confidence monitoring based on the checks in the program. For example, if all checks require the error magnitude to be zero (reliability in [42]) Diamond will only calculate confidence at runtime.

4.5 METHODOLOGY

Implementation and Testing Setup We parsed and translated Go programs written using a library of Diamond primitives to Diamond-IR using ANTLR. We used Python to sequentialize Diamond programs for checking properties such as type safety and deadlock-freedom, and then for generating instrumented Go code. We implemented distributed communication using RabbitMQ 3.8.7. We ran our experiments on a machine with a Xeon E5-1650 v4 CPU, 32 GB RAM, and Ubuntu 18.04. Each benchmark consisted of 8-10 worker processes.

Benchmarks We implemented a set of popular parallel benchmarks from prior literature that exhibit diverse parallel patterns and verified properties that quantify uncertainty in their executions (Table 4.2). We looked at the following benchmarks:

- *PageRank, SSSP, BFS:* Graph benchmarks commonly used in distributed Big Data applications. PageRank is used for search result optimization [95]. Single Source Shortest Path is used to make data routing decisions. Breadth First Search is used to find connected components in graphs. From CRONO [96].
- *SOR:* A kernel for successive over-relaxation. Used to extrapolate the state of a system over time. From Chisel [43].
- *Sobel:* Edge-detection filter. From AxBench [98].
- *Matrix Mult.:* Multiplies two square matrices. Each worker process computes a subset of rows of the product.
- *Kmeans-Agri:* Partitions n-dimensional input points into k clusters (Section 4.2).
- *Regression:* Performs distributed linear regression on 2-D data. Each worker performs regression on a subset of data. The master thread averages the results.

Table 4.1: Input Size and Number of Threads Used for the Evaluation

Benchmark	Parallel Pattern	Workers	Input Size
PageRank	Scatter-Gather	8	8 iterations on roadNet-PA graph from SNAP
SSSP	Scatter-Gather	10	62K nodes (p2p-Gnutella31 graph from SNAP)
BFS	Scatter-Gather	10	62K nodes (p2p-Gnutella31 graph from SNAP)
Kmeans-Agri	Stencil	8	248-2048 points of 2D data
SOR	Stencil	10	10 iterations on 100×100 random array
Sobel	Map	10	100×100 upto randomly generated array
Matrix Mult.	Map	10	two 100×100 randomly generated matrices
Regression	Map-Reduce	10	1000 randomly generated floats

Inputs. Table 4.1 gives the size of the primary inputs we used to evaluate each benchmark and the number of worker threads. Apart from the worker threads, each benchmark also contained one master thread. We used additional input sizes solely to evaluate the effect of optimization on runtime and communication volume. For Section 4.6.3, we increased the input sizes to 400×400 for Sor, 180×180 Sobel, the two matrices were increased in size to 200×200 for Matrix Multiplication, For graph algorithms we used 4 graphs from SNAP [161] (p2p-Gnutella - 09, 25, 30, and 31).

Sources of uncertainty. *Noisy channels* occasionally corrupt data sent over them (used for PageRank, SSSP, BFS, and Kmeans-Agri). We use a corruption rate of 10^{-7}. *Precision reduction* reduces floating point precision from 64-bit to 32-bit during communication only to save bandwidth (used in SOR, Sobel, Matrix Mult.). The *input* provided to the program itself can have inherent uncertainty. For Kmeans-Agri, we assume a 50:50 mixture of two different temperature-humidity sensors with different error specifications. *Timing errors* can cause the program to use stale or incomplete values (used for Regression).

Baselines. We compare the runtime of Diamond with optimizations to a baseline which is a straightforward parallel implementation of an existing static analysis via Diamond (either Decaf [120] or AffineFloat [153] without roundoff errors).

4.6 EVALUATION

4.6.1 Can we verify important uncertainty properties using Diamond?

For each benchmark, we used Diamond to verify the properties shown in Column 3 of Table 4.2. Diamond successfully verified these properties on the final output of the program.

Table 4.2: Benchmarks, Verified Properties, and Runtime Monitoring Overhead for Diamond. Baselines: ⋆:Decaf, †:AffineFloat

Benchmark	Uncertainty Source	Verified Property	Overhead	
			Baseline	Diamond
PageRank	Noisy Channel	checkArray(pagerank, 0, 0.9912)	30%*	3.63%
SSSP	Noisy Channel	checkArray(distance, 0, 0.9251)	33%*	2.31%
BFS	Noisy Channel	checkArray(visited, 0, 0.9925)	30%*	4.06%
SOR	Precision Reduction	checkArray(output, 1.19×10^{-7}, 1)	60%†	3.49%
Sobel	Precision Reduction	checkArray(output, 2.38×10^{-7}, 1)	71%†	9.71%
Matrix Mult.	Precision Reduction	checkArray(product, 6.6×10^{-6}, 1)	80%†	16.27%
Kmeans-Agri	Noisy Channel, Input	checkArray(centers, $\langle$1.5, 0.9948$\rangle$, $\langle$2,0.9948$\rangle$)	42%*†	3.32%
Regression	Timing Error	check(alpha, 0, 0.99)$\wedge$check(beta, 0, 0.99)	37%*	0.45%

Each check places an error magnitude and confidence bound on a single variable. For arrays each element must satisfy these bounds. For PageRank, SSSP, and BFS, the bounds ensure that key graph properties are calculated exactly $\geq 99\%$ of the time per node. For SOR, Sobel and Matrix Mult., the bounds limit the maximum error of the output due precision reduction. Kmeans-Agri was discussed in the example. For Regression, the bounds ensure that the output line parameters are correct $\geq 99\%$ of the time (high confidence is desirable for predictive models).

Parallely [152] cannot verify these properties. Diamond's dynamic analysis of arrays and unbounded loops more effectively handles irregular input structure (e.g., graphs), which had to be conservatively bounded for static analysis. This allowed us to verify stronger properties for significantly bigger inputs than previously possible for existing reliability and accuracy static analyses. We observed that, even in the presence of errors, the error magnitude of the final outputs of our programs was acceptable.

Optimizations can affect the precision of the analysis. This effect is prominent in benchmarks with irregular computations (graph benchmarks). However, in our benchmarks, we found that baseline and optimized Diamond could verify nearly the same uncertainty bounds. For example, for BFS, Diamond could verify a confidence of 0.999 when using the baseline version. For benchmarks with regular computation patterns, such as SOR and Regression, there was no significant change.

In summary, *Diamond verifies important end-to-end uncertainty properties that cannot be verified using existing static analyses.*

4.6.2 What are the overheads associated with Diamond?

Columns 4 and 5 of Table 4.2 present the overhead of the baseline and optimized Diamond benchmarks respectively. Time for I/O and setup is excluded. Overhead is calculated as the percentage increase in runtime w.r.t. an unmonitored benchmark.

In our benchmarks, the runtime is dominated by communication, as is common in many distributed settings. In most cases, the runtime overhead for computing the uncertain intervals is a small fraction of the total runtime. Error magnitude calculation requires more computation than error confidence (see Figure 4.5). As a result, overhead for error magnitude benchmarks (SOR, Sobel, Matrix Mult.), is higher. This was especially true for the computationally intensive Matrix Mult.

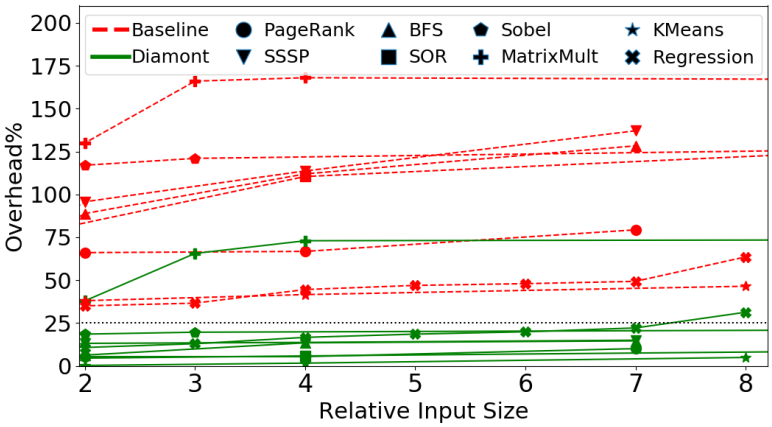
Optimization impact. The Regression benchmark used a statically verified kernel error specification to eliminate monitoring. The communication optimization contributes around 98% of savings in all other benchmarks. Debloating also provided significant speedups. For example, without debloating PageRank is 3.9x slower and Sobel is 3.3x slower (our baseline is comparable to Diamond with debloating).

Are the overheads justified? Approximations have led to significant savings in prior work: 1) Communication: up to 62% performance improvement in approximate NoCs [40, 162], and 2) Computation: 2x speedup in loop perforation [101], 2.7x speedup in Paraprox [30], and up to 1.3x speedup from reduced precision in Precimonious [102]. As Diamond's post-optimization overhead is lower than the speedups from these approximations, it can be used in conjunction with them to provide guarantees on the quality of results while still getting speedups.

In summary, *With optimization, overhead of Diamond analysis is at most 16.3% for our benchmarks, with a geomean of 3.04%.*

4.6.3 How does Diamond overhead depend on the program inputs?

Figure 4.21 shows the effect of input size on Diamond overhead. The X-Axis shows the relative input size and the Y-Axis shows overhead. The dashed and solid lines show the unoptimized baseline and optimized Diamond versions respectively. Each marker indicates a different benchmark. Overall, the overhead of the optimized versions is significantly lower than the baseline versions. Most optimized versions have an overhead less than 25% for all inputs. The table in Figure 4.21 shows the geomean of the overhead across all benchmarks



Size	Baseline	Diamond
2x	69.1%	8.68%
4x	84.4%	10.2%
6x	93.7%	14.1%
8x	91.8%	12.9%

Figure 4.21: Input Size vs. Overhead. Table shows geomean overheads across programs.

for different relative input sizes. While baseline overhead increases to an average of 94%, optimized overhead only reaches 14%.

For Matrix Mult., computation increases faster with input size than communication ($O(n^3)$ vs. $O(n^2)$). Thus the major source of overhead becomes the computation of the monitored uncertainty, rather than communication. This benchmark illustrates that Diamond is more useful in cases where the program is communication-bound.

The unoptimized baseline also sends significantly more data (3x to 5x) compared to the optimized version. This is due to the array communication optimization. The communication overhead of the optimized version is negligible.

In summary, *as input size grows, the improvement caused by optimizations on Diamond runtime performance increases over the baseline system.*

4.7 RELATED WORK

Several analyses are related (in part) to Diamond's functionality, as shown in Table 4.3. Columns 2-4 indicate whether the analysis is static, empirical (sampling-based), or runtime based. Columns 5-6 indicate support for error confidence (reliability) and error magnitude (accuracy) analysis. Column 9 indicates if the system can support multiple sources of uncertainty. In contrast to all these analyses, Diamond is the only one flexible enough to simultaneously support *multiple* analyses and approximation sources, and in addition, extending these to *parallel* programs.

Static Analyses for Approximate Programs. Though multiple static analyses target approximate programs (e.g., [41, 43, 44, 88, 111, 117, 163, 164]), most relevant to Diamond is Parallely [152], which retains the limitations of the underlying static analyses requiring developers to provide bounds on loop iterations, array sizes, and number of processes. In contrast, Diamond successfully combines static and dynamic analysis and works on a real language (Go), which jointly allow for verification of much larger benchmarks. Additionally, Diamond also extends sequentialization for dynamic conditions.

Dynamic Analysis and Runtime Monitoring. DECAF [120] performs dynamic reliability verification through type inference. Our work avoids DECAF's strict independence assumptions by adding reliabilities instead of multiplying (both bounds are close in practice). Ringenburg et al. [165] propose offline and online approaches to monitor the quality of programs, using methods such as dataflow techniques and comparison to the precise program. Diamond instead propagates uncertain intervals during both static and dynamic phases, allowing it to monitor uncertainty with greater precision. Maderbacher et al. [166] focus on precisely correcting bitflips with minimal checks. In contrast, Diamond monitors uncertainty from many sources in programs that can tolerate some error.

AffineFloat [153] and Ceres [167] provide dynamic analysis for numerical error. Herbgrind [168] locates possible sources of numerical error. These tools measure floating point roundoff errors, but have high overhead. Diamond focuses on analyzing error from casting and external sources e.g., sensors. Uncertain⟨T⟩ [169] used an early form of uncertain intervals, however they use sampling to determine error. Statistical model checking tools [170] can provide statistical guarantees on program properties expressed in a temporal logic. PAssert [171] and AxProf [107] statistically verify at development time a single probabilistic assertion at the end of the program. In contrast, Diamond supports many checks at different points in the program at runtime.

4.8 CONCLUSION

The past decade brought many techniques for developing new approximations and analyzing uncertainty for specific scenarios (see Section 4.7), but much less work has been done in integrating these various concepts in a unifying, rigorous, and extensible framework. Diamond aims to pave the way toward that goal – it supports multiple uncertainty sources (input noise, variable-precision code, errors in communication, and unreliability in hardware), combines static analysis and dynamic monitoring, supports a significant fragment of the Go language, and operates on several emerging applications (precision agriculture, graph analytics, media

Table 4.3: Comparison of Related Work - Stat (static), Em (Empirical), RT (Runtime), Rel (Reliability), Acc (Accuracy), Verif (Verified), Par (Parallel), Src+ (Mulitple Sources). (✓* indicate analyses that track confidence intervals, which is another interpretation of Diamond's uncertain intervals)

Method	Stat	Em	RT	Rel.	Acc.	Verif.	Par.	Src+
Diamond	✓	×	✓	✓	✓	✓	✓	✓
Parallely	✓	×	×	✓	✓	✓	✓	✓
Rely	✓	×	×	✓	×	✓	×	×
Chisel	✓	✓	×	✓	✓	✓	×	✓
DECAF	✓	×	✓	✓	×	✓	×	×
EnerJ	✓	✓	×	×	×	✓	×	×
AffineFloat	✓	×	✓	×	✓	✓	×	×
PAssert	×	✓	✓	✓*	✓*	✓	×	×
Uncertain<T>	×	✓	✓	✓*	✓*	×	×	×

processing).

We demonstrated the benefit of our analysis and optimizations by reducing the execution overhead to 16.3% or less. We believe this work can serve as a starting point for safe runtime systems in other domains needing to handle uncertainty, such as robotics or the Internet-of-Things.

Chapter 5: Case Studies

5.1 RESPONDING TO CHECK FAILURES

As Diamond monitoring exposes the uncertainty of variables to the program, making decisions based on the uncertainty is a logical choice. Diamond provides constructs for safely implementing recovery mechanisms distributed across multiple processes. In Diamont, check failures result in errors. Instead, we can provide syntax for developers to define recovery routines that will run when a check fails. Recovery mechanisms allow the distributed program to recover from excessive error by redoing the computation or communicating using more reliable channels. It also allows programs to operate on larger inputs by isolating and reducing error at the source.

Developers manually implementing distributed recovery mechanisms may unwittingly introduce bugs that can cause deadlocks. Figure 5.1 shows one such scenario. The two functions in the code are executed as two processes. In this program, the worker first sends compressed data to the manager. A check is executed after the data the manager process receives the data. The check fails if the error in the received data is too high. The recovery mechanism is then triggered, prompting the worker to resend data in an uncompressed format. This method allows the system to save bandwidth when the error level is reasonable after compression. However, in this version, the manager only sends the check result if it fails. If it passes, the worker gets stuck waiting for the result.

```
func Manager {
  r32 = receive(Worker)
  r = ([]float64)r32
  chk = check(r,1.0,0.1)
  if !chk {
    send(Worker, chk)
    r = receive(Worker)
} }
```

```
func Worker {
  r32 = ([]float32)(r)
  send(Manager, r32)
  chk = receive(Manager)
  if !chk {
    send(Manager, r)
} }
```

Figure 5.1: Manual Recovery Causing Deadlock

Such errors become more challenging to notice and fix when many processes participate in a computation. Distributed recovery poses additional challenges. Consider two processes communicating a variable over an unreliable channel. As the variable is sent unreliably, its monitored uncertainty interval is different at the source and destination. If both processes independently perform checks on the communicated variable, their decisions may differ. This

```
func Manager {
  TryCheckRecover {
    Try: func() {
      r32 = receive(Worker)
      r = ([]float64)r32
    }
    Check: func() bool {
      return check(r,1.0,0.1)
    }
    Recover: func() {
      r = receive(Worker)
    }
  }.Execute()
}
```

```
func Worker {
  TryCheckRecover {
    Try: func() {
      r32 = ([]float32)(r)
      send(Manager, r32)
    }
    Recover: func() {
      send(Manager, r)
    }
  }.Execute()
}
```

Figure 5.2: Distributed Recovery Mechanism in Go+Diamond

```
try {
  r32 = receive(Worker, float32[]);
  r = (float64[])r32;
}
check(r,1.0,0.1)
recover-with [Worker] {
  r = receive(Worker, float64[]);
}
```

```
try {
  r32 = (float32[]) r;
  send(Manager, float32[], r32);
}
recover-from [Manager] {
  send(Manager, float64[], r);
}
```

Figure 5.3: Distributed Recovery Mechanism in Diamond-IR. (All variables are dynamic)

can lead to a deadlock if only one process decides that resending the variable is necessary.

Figure 5.2 shows the same program as above implemented using the recovery mechanisms in Diamond. The two processes first execute the initial code in `try`. One process defines a `check`, which decides if recovery will be necessary. Lastly, depending on the result of the check, all processes may execute the code in `recover`. Diamond automatically handles the transmission of the check result and checks for other safety properties to ensure bug-free recovery.

Translation. Figure 5.3 shows how the mechanism is translated to Diamond-IR. By analyzing this IR, Diamond needs to ensure that 1) the result of the check is sent to all processes participating in the recovery mechanism, 2) the computation remains deadlock free, and 3) dynamic monitoring remains sound. We use the `recover-with` and `recover-from` as annotations in the IR to generate code in the runtime to send and receive the results of the check. We use the sequentialization analysis to determine these sets of processes.

```
try {
  Worker.res32 = (float32[])Worker.res;
  Manager.res32 = Worker.res32;
  Manager.res = (float64[])Manager.res32;
}
check(Manager.res,1.0,0.1)
recover {
  Manager.res = Worker.res;
}
```

Figure 5.4: Recovery Code After Sequentialization

```
S_try;
if (check(Manager.res,1.0,0.1)) {
   skip;
} else {
  S_rec;
}
```

Figure 5.5: Encoding recovery using conditionals

Sequentialization. As multiple processes may be involved in recovery computations, Diamond must ensure that the program remains deadlock-free. We can verify deadlock freedom, regardless of the check result, using sequentialization.

To sequentialize the recovery mechanism, Diamond must show that a set of other processes exist such that all of their `try` and `recover` sections can be sequentialized separately and that only one of the processes performs a check. For this, 1) Diamond scans the code in `try` and `recover` to find all communicating processes, 2) it scans the code in these other processes to find matching `try` and `recover` sections, 3) once all processes involved in distributed recovery are found, Diamond ensures that only one process performs a check, 4) Diamond rewrites the distributed recovery mechanism to a sequential recovery, 5) to the process performing the check, Diamond writes the list of other processes to the `recover-with` annotation, 6) lastly, for the other processes, Diamond writes the process performing the check to the `recover-from` annotation. If Diamond successfully generates a sequentialization, the whole computation is deadlock free for both outcomes of the check. Figure 5.4 shows the sequentialized version of the code in Figure 5.3.

Encoding detection and recovery in Diamont. As discussed in Aloe [158], existing quantitative reliability analyses from Rely and Parallely cannot be used to accurately calculate the reliability of a try-check-recover block. To represent this computation in the sequentialized Parallely code one can convert the try-check-recover statement to a conditional statement, as shown in Figure 5.5, where S_{try} represents the instructions in the try block of our try-check-recover statement, and S_{rec} represents the instructions in the recover block.

The semantics of this computation closely mirror those of our try-check-recover block. However, Parallely's analysis cannot infer that S_{rec} only executes when S_{try} fails and that it eliminates the error produced by S_{try}. It instead conservatively assumes that errors in S_{try} can remain uncorrected.

Table 5.1: Recovery Rates for Benchmarks

Benchmark	Recovery Rate
Pagerank	1.4%
SSSP	4.5%
BFS	4.6%
Matrix Mult	28%
Sobel	8.1%

Soundness. To maintain soundness of the dynamic monitoring, Diamond needs to perform additional checks. Diamond's notion of accuracy compares approximate executions of a program to a perfectly precise execution. A precise execution will always execute `try` and pass the check. To ensure that the result of executing `recover` is comparable without costly checkpointing, Diamond restricts the code in the `try` sections to be idempotent and requires that `try` and `recover` perform the same computation, starting from the same (read-only) input variables, and storing the results in the same (write-only) output variables. We assume that the code blocks contain no I/O and that any external functions called are idempotent. This restriction is the same as that in [158], which uses the same notion of accuracy. Sequentialization simplifies this analysis as it can be performed on a single representative sequential version of the parallel program.

Evaluation. To evaluate recovery mechanisms in Diamond, we ran Pagerank, SSSP, BFS, Matrix Mult, and Sobel using multiple randomly generated inputs. For the graph benchmarks, the `try` section ran the algorithm using a noisy channel for communication, while `recover` reran the algorithm using a reliable channel. For Matrix Mult and Sobel, `try` used compression and 32-bit channels for communicating floats, while `recover` used an uncompressed 64-bit channel.

Table 5.1 shows the recovery rate, i.e. the fraction of dynamic instances of the recovery mechanisms that had to be invoked due to high uncertainty. For the graph benchmarks, we observed that recovery was triggered only for graphs with very high connectivity. The uneven nature of the input graphs led to cases where some workers' calculations had higher errors compared to others, thus triggering recovery. For Matrix Mult and Sobel using this technique led to communication bandwidth savings of 22% and 42% respectively, compared to using uncompressed communication at all times while ensuring excessive errors are not produced. In all cases the recovery mechanisms executed safely without deadlocks or program crashes.

5.2 ALGORITHMIC FAIRNESS

In addition to checking accuracy and reliability, Diamond is expressive enough to monitor *algorithmic fairness* properties, such as those in [172, 173].

Fairness specifications are given in [172, 173] as arithmetic expressions over expectations of random variables, such as $\varphi \triangleq \frac{\mathbb{E}[X]}{\mathbb{E}[Y]} > c$. However the true values of these expectations are not known a priori, and thus have to be over-approximated with an uncertainty interval. In the fairness setting, the uncertainty interval simply reduces to a confidence interval around the true mean, obtainable via Hoeffding's inequality.

We now describe this encoding on a semantic level. For each expectation in φ (e.g. $\mathbb{E}[X]$ and $\mathbb{E}[Y]$), we have a distinct dynamically tracked variable (e.g. `x` and `y`). Semantically, this allows Diamond to associate to each expectation an uncertainty interval which will serve as a statistical confidence interval. However as the tightness of a confidence interval is solely a function of the number of samples taken, this is the *only* source of uncertainty. Therefore, unlike in the case of system-level approximation (e.g. approximate sends and receives) where the approximate statement will cause the runtime to automatically update a variable's uncertainty interval, for encoding fairness properties, we must explicitly force the runtime to update these expectation variables' uncertainty intervals whenever receiving a new sample. In order to make the runtime update the uncertainty interval, we must explicitly recompute the new uncertainty bound on a variable via Hoeffding's inequality whenever we receive a new empirical sample of that variable (meaning we must also know the number of samples seen). Syntactically, we encode this by using an explicit `track` statement where the arguments come from the computation of Hoeffding's inequality, thus ensuring the runtime sets a variable's uncertain interval to the correct confidence interval. A GoLang source-level encoding of this (which will compile down to the Diamond IR) can be seen in Fig. 5.6. The distinct dynamically tracked variables for each expectation (e.g. `x` and `y`) are represented by the class's `mean` variable. Additionally, the class's `AddSample` performs the recomputation of the updated uncertainty bound (using Hoeffding's inequality) on the dynamic `mean` variable whenever a new sample is added.

Having defined how to encode a confidence interval for an empirical estimate of an expectation as an uncertainty interval in Diamond, we can now describe how to encode the full property φ which is a logical predicate over arithmetic operations of multiple such expectations. To encode φ, we leverage the fact that Diamond can propagate uncertainty intervals through arithmetic expressions as shown in Fig. 4.5. Hence if we already have a dynamic variable `x` tracking the confidence interval of the empirical estimate of $\mathbb{E}[X]$ and another dynamic variable `y` tracking the confidence interval of the empirical estimate of

```
type MeanTracker struct {
    successes int
    totalSamples int
    /*@dynamic*/ mean float64
}
func (b *MeanTracker) AddSample(sample bool) {
    b.successes += bool2int(sample)
    b.totalSamples += 1
    tmp := float64(b.successes)/float64(b.totalSamples)
    //sets confidence interval via Hoeffding's inequality
    b.mean = track(tmp,Hoeffding(b.totalSamples,δ),δ)
 }
```

Figure 5.6: Source-level Fairness Encoding

$\mathbb{E}[Y]$, we only need to write `z = x/y` to then have a dynamic variable for the ratio $\frac{\mathbb{E}[X]}{\mathbb{E}[Y]}$. The Diamond runtime will compute a valid uncertainty interval for this entire ratio, without any further progammer intervention. To perform this using the high-level class interface, we only need to divide their `mean` variables.

Upon computing uncertainty bounds for the expressions in the inequality, the final step is to then certify whether the full inequality φ holds. However because of the inherent uncertainty in the variables our certification is probabilistic, meaning we only certify that that the predicate φ holds, *with high probability*. However for algorithmic fairness, this is standard practice, as the predicates in [172, 173] are also certified probabilistically. If we have dynamically tracked uncertainty intervals for all variables, then checking that φ holds with high probability can be performed by use of the `check` statement. However the semantics of the `check` function only checks if the error and probability associated to a dynamically tracked variable are within some threshold. To certify inequalities with ratios of the form $\varphi \triangleq \frac{\mathbb{E}[X]}{\mathbb{E}[Y]} > c$ hold probabilistically, we need to certify that lower bound of the uncertain interval associated to $\frac{\mathbb{E}[X]}{\mathbb{E}[Y]}$ is greater than c with high probability. Luckily, this can still be semantically encoded using Diamond's `check` statement, albeit with a minor algebraic re-arrangement. If `z` is the dynamic variable corresponding to $\frac{\mathbb{E}[X]}{\mathbb{E}[Y]}$ and we want to check if φ holds with probability at least Δ, then we can encode this as `check(z,z-c,`Δ`)`.

Evaluation. Empirically we evaluate this approach on benchmarks taken from [172, 173] which are `Hiring`, `Income SVM`, `Income Decision Tree` and `Income Neural Network` which represent classifiers. In all cases the fairness property we want to certify is $\varphi \triangleq \frac{\mathbb{E}[X]}{\mathbb{E}[Y]} > 0.8$ with probability at least 0.9. where $\mathbb{E}[X]$ is the expectation of the indicator $X = \mathbf{1}_{Hire|Male}$ and $\mathbb{E}[Y]$ is the expectation of the indicator $Y = \mathbf{1}_{Hire|Female}$. The fairness certification check compiles down to Diamond IR as `check(z,z-0.8,0.9)` where as before, `z = x/y` where `x`

Table 5.2: Overheads for Fairness Benchmarks

Benchmark	**Overhead**
Hiring	3.9%
Income SVM	3.0%
Income Decision Tree	6.1%
Income Neural Network	2.5%

and `y` are dynamically tracked variables for $\mathbb{E}[X]$ and $\mathbb{E}[Y]$.

Table 5.2 shows the overheads for verifying φ using Diamond. In all cases the overheads were low, highlighting the fact that Diamond is expressive and flexible enough to be adapted to efficiently certify properties beyond those found in standard approximate computing.

5.3 UNCERTAINTY MONITORING ON THE *WIPACKAGE* ARCHITECTURE

In this section we will discuss how to extend the runtime verification from Diamont to a novel architecture containing a network-on-chip that can add uncertainty to shared data. While this thesis not cover the hardware design of *WiPackage*, this chapter will present how to extend Diamont to programs running on the hardware.

As the semiconductor industry moves to smaller devices, the costs of producing large dies continues to increase, and fabricating large monolithic dies becomes increasingly less economical. Therefore, modern high-performance computing (HPC) processors partition large multi-core designs into smaller *chiplets* that deliver better yield and lower the production costs [174, 175]. In this environment, the design of interconnection networks that provide fast and efficient chiplet-to-chiplet communication is a major challenge. In current interconnect systems, the high latency associated with inter-chiplet communication renders multi-hop transactions across far-off chiplets extremely costly, decreasing performance and jeopardizing the scaling of the system. In this context, wireless communication technology represents an opportunity to greatly alleviate the issues of existing chiplet interconnects.

Furthermore, in a system with many chiplets, there is an opportunity to combine wired with wireless communication. Communication within chiplets or between neighboring chiplets should use wired communication. However, communication across non-neighbor chiplets can benefit from wireless communication. In particular, wireless communication can efficiently support message-passing collective primitives, such as MPI's Barrier, Bcast, Scatter, Gather, Allgather, Reduce, and Allreduce.

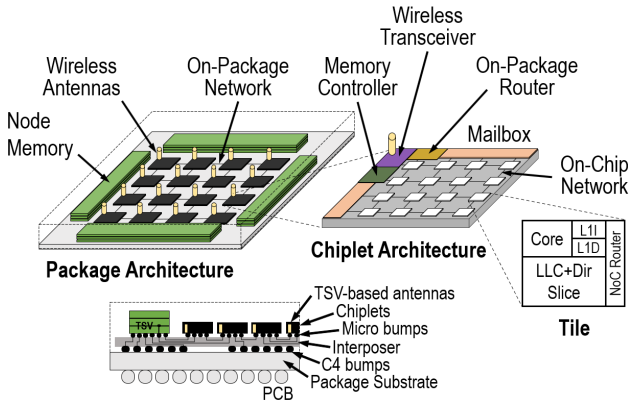


Figure 5.7: Overview of the *WiPackage* architecture.

Architecture Overview. *WiPackage* is an architecture that combines two such networks for inter-chiplet communication [63]. *WiPackage* envisions a supercomputer in a package as shown in Figure 5.7. *WiPackage* is comprised of a set of chiplets, each operating as a separate shared-memory domain, with its own separate wired on-chip network. *WiPackage* provides a message-passing wired/wireless-based interconnection network, aimed at sending data across chiplets. It augments each chiplet with a multi-channel wireless transceiver, and extends an MPI implementation with some wireless transactions. This enables the use of wireless channels to send and receive data across chiplets, bypassing the high-latency multi-hop transactions of the wired network-on-package.

Figure 5.7 illustrates the *WiPackage* architecture. *WiPackage* consists of an array of chiplets surrounded by High Bandwidth Memory (HBM) modules, acting as node memory. At the same time, each chiplet contains an array of cores (with their corresponding private L1 caches), a shared last-level cache (LLC) with its corresponding directory, a local memory that we use as a mailbox for incoming and outgoing inter-chiplet messages, and a DRAM controller. Additionally, each chiplet contains a network-on-package router and wireless transceiver, used for chiplet-to-chiplet communication. Compared to Replica, *WiPackage* contains less wireless routers (one per chiplet) allowing us to have bigger broadcast memories.

Programming Model. Furthermore, compared to Replica, which used a shared memory abstraction, *WiPackage* uses MPI with synchronous message passing. This makes communication more explicit in the program and allows for Diamont's runtime verification to be extended straightforwardly. All data sharing among the distributed processes happens through the MPI interface.

In *WiPackage*, communication primitives are implemented using a specialized library. The library support MPI's `Barrier`, `Bcast`, `Scatter`, `Gather`, and `Reduce` primitives. The

collective communication primitives are implemented by combining a set of point-to-point messages among the chiplets. Some collective primitives can be optimized to reduce the number of messages by assuming a logical tree based structure to the cores. For verification purposes in this case study we assume a simplified implementation where no such optimizations are used. Programs can have arbitrary sequential computations.

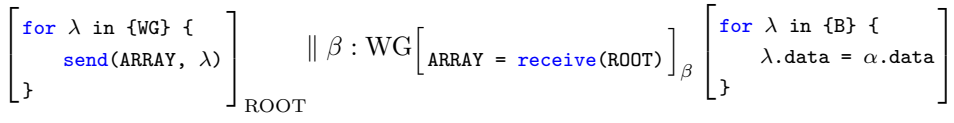
Broadcast. We can consider *broadcast* messages to be a set of point-to-point messages sent to each of the participants. The MPI syntax uses the same statement is all processes participating in the communication operation. We can see the minimum required C code for a simple broadcast operation using MPI in Figure 5.8. The MPI code contains a single `MPI_Bcast` function call with the root (sender of the broadcast message) indicated in the function. Under the hood the runtime library provided with *WiPackage* implements this function call using multiple steps which sends point-to-point messages.

```
int main(int argc, char** argv) {
  int rank, size;
  // setup code is skipped to preserve space

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int* ARRAY = (int*) malloc(sizeof(int) * N); /*
     Dynamic*/

  // the function call is same for root and
     others
  MPI_Bcast(ARRAY, NELEMS, MPI_INT, ROOT,
     MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

Figure 5.8: A MPI program implementing a broadcast operation

Diamont can be extended to support the broadcast communication pattern by representing the `MPI_BCast` operations as a set of ordered messages to individual processes as shown below. Using the function call in the original C program we can identify the primitives, and the participating threads. For the `ROOT` process we add code to send the message to the other processes in the communications group. The non `ROOT` processes are converted to a simple receive statement.

Following Diamond-IR code shows how to represent this communication pattern (the call to the `MPI_Bcast` function) in a desugared form that can be checked for the existence of a sequentialization. Further, the figure shows the desired sequentialized program where the data in all the participating threads are updated in the same order.



(a) De-sugared form of broadcast messages (b) sequentialized

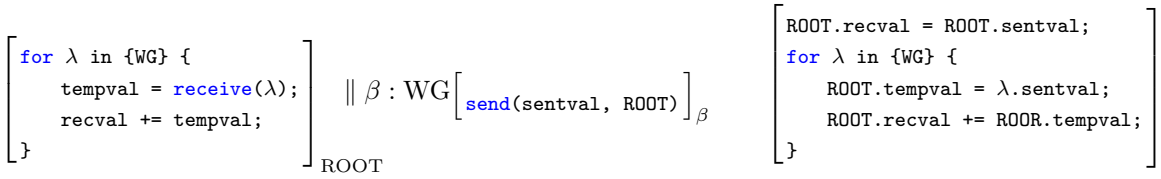
Figure 5.9: Supporting broadcast messages in Parallely

To generate this code we need to identify the sender (`ROOT`) of the broadcast message and the other members. To simplify our implementation we assume that process 0 always remains the root. Based on this assumption, we generate the intermediate representation and perform the sequentialization based analysis.

Reduce. Similarly reduce operations in the MPI program is expressed using the following syntax.

```
MPI_Reduce ( &sentval, &recval, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD )
```

We can generate the following intermediate representation by analyzing the function call. Similar to the broadcast operation, reductions also has a main process (`ROOT`) that collects the data together. Using the function call we can convert the program to the following where the reduction is replaced by a `send` statement in all processes other than `ROOT`.



(a) De-sugared form of reduce messages (b) sequentialized

Figure 5.10: Supporting reduce operations in Parallely

Implementation of Runtime System. We allow developers to mark datastructures that require runtime monitoring using annotations in the comments. Our simplified implementation assumes that the whole array is always sent in broadcast operations.

Table 5.3: Benchmark details

Name	Description	MPI Collectives
bfs	Breadth First Search on a random graph	`MPI_Bcast`, `MPI_Gather`
sssp	Single Source Shortest Path on a random graph	`MPI_Bcast`, `MPI_Gather`
sobel	Sobel edge detection on a random bitmap [179]	`MPI_Scatter`, `MPI_Gather`
quad	Approximate an integral using a quadrature rule [180]	`MPI_Bcast`, `MPI_Reduce`
satisfy	Exhaustive search for solutions of the circuit satisfy problem [181]	`MPI_Bcast`, `MPI_Reduce`

Unlike the Go programs we looked at in Chapter 4, many MPI programs are based on the Single-Program-Multiple-Data (SPMD) paradigm. Therefore, we need to identify the programs that are executed by each process by analyzing the code. In our benchmarks, the root process code is different to the members only through the conditional that check if the process id is the root's. We exploit this fact to generate the intermediate representation in Diamont. Our implementation looks at the MPI initialization code to identify process groups used in the program. We use this data to generate the intermediate representations for the communication primitives.

After generating the intermediate representation we perform the safety analyses. After the analysis we generate code to perform the runtime monitoring. We implement the additional communication using a Diamont runtime library specialized to *WiPackage*. The MPI primitives in the applications are wrapped around the functions in our library. Each call to a MPI communication primitive is replaced with its Diamont version that handles the additional communication in the background by duplicating the point-to-point messages for the additional tracked data.

Error Model. Messages in the wireless channel can fail with some probability. Bit Error Rate defines the error rate for the network as the probability of a bit of a message being flipped while in flight. With advances in wireless technology the BER can be as low as 10^{-12} [176]. But, without any error prevention techniques the BER can go down to 10^{-4} [177, 178]. For the experiments in this case study we used multiple BER values in this range.

Evaluation. We looked at five benchmarks from the domains considered in the work presented in this dissertation. Table 5.3 shows the details for the benchmarks used in the evaluation and the MPI communication primitives that are contained in the applications. For each of the benchmarks we assumed that all inter-chiplet communication is handled through the unreliable wireless channel. We evaluated on a 64 chiplet system where each chiplet contains four cores.

For these benchmarks we attempted to verify the *reliability* of the results. Table 5.5

Table 5.4: Benchmark Results

Benchmark	Input	Verified Error Bound (error = 1 - reliability)			
		1×10^{-4}	1×10^{-7}	1×10^{-9}	1×10^{-10}
bfs	p2p-Gnutella31	7×10^{-2}	7.5×10^{-3}	3×10^{-5}	3×10^{-6}
sssp	p2p-Gnutella31	7×10^{-2}	7.5×10^{-3}	3×10^{-5}	3×10^{-6}
sobel	100×100 array	7×10^{-4}	6.99×10^{-7}	6.99×10^{-9}	6.99×10^{-10}
quad	2^{14} elements	6.4×10^{-3}	6.4×10^{-6}	6.4×10^{-8}	6.4×10^{-9}
satisfy	2^{8} elements	6.4×10^{-3}	6.4×10^{-6}	6.4×10^{-8}	6.4×10^{-9}

shows the results of our evaluation. Column 2 of the table shows the inputs used in the evaluation. The remaining columns show the lowest error bounds we were able to verify with the extension of Diamont, for multiple possible error rates in the *WiPackage* wireless network. We looked at four levels of noise for the communication channel. We implemented the programs such that only `dynamic` typed data uses the wireless communication link. All critical data uses the wired network.

The results show that the program outputs maintain high reliability even in the presence of network errors and that we can verify important properties for our benchmarks. As the network error rates decrease, we are able to prove tighter error bounds for the program data as expected. For extremely small error rates the tightest bound Diamont can verify becomes a function of the precision of the floating point representation. Floating point soundness can be added to the implementation using static analyses [182, 183] that can soundly capture all the possible different outputs from varying rounding modes, and orders of computation. Prior work (including the results shown in Chapter 3 and Chapter 4) demonstrate that these reliability bounds produce results with acceptable levels of accuracy.

Table 5.5: Overhead of Runtime Monitoring

Benchmark	Baseline	Diamont
bfs	1.63x	1.08x
sssp	1.69x	1.08x
sobel	2.11x	1.04x
quad	1.30x	1.16x
satisfy	1.05x	1.05x

Table 5.5 shows the overheads associated with runtime monitoring for our benchmarks for the 1×10^{-4} noise rate. Similar to Chapter 4, we compare the overhead of Diamont with optimizations to a *Baseline* implementation, which is a straightforward parallel implementation of runtime monitoring. Our results show that Diamont's optimizations significantly

reduce the overhead in all benchmarks except `satisfy` (This benchmark only uses runtime monitoring for a small number of variables, therefore the optimizations do not affect the already low overhead). Our results show that Diamont can help programs benefit from the high performance network while maintaining their safety and a high level of reliability.

Chapter 6: Conclusions and Future Work

6.1 CONCLUSION

Programs today have to deal with increasing levels of uncertainty in their execution. Uncertainty is inherent in many application domains. Increased volumes of data and the emergence of novel heterogeneous processing systems push us towards parallel programming. The sources of uncertainty in sequential programs are present in parallel programs but are combined with new sources of noise and uncertainty. Therefore, providing foundations of safety and accuracy analyses for parallel programs that deal with uncertainty remains an intriguing and challenging research problem.

This dissertation presents our work on building parallel programming systems that deal with uncertainty in a principled way. We look at the hardware/software co-design of a network-on-chip that can introduce uncertainty to programs and show how to use this new capability optimally. We show how to use empirical methods to optimize performance while maintaining an acceptable level of accuracy. The dissertation also presents two general verification techniques for a subset of parallel approximate programs. Parallely shows how to use static analysis techniques to verify important safety and accuracy properties in parallel programs that deal with uncertainty. The dissertation also presents how to use runtime monitoring to overcome challenges in the static analysis and extend verification to bigger programs.

The past decade brought many techniques for developing new approximations and analyzing uncertainty for specific scenarios. My work integrates these various concepts in a unifying, rigorous, and extensible framework. The work presented in this dissertation can serve as a starting point for safe parallel programming systems in other domains that need to handle uncertainty, such as robotics, precision agriculture, or the Internet-of-Things.

Many emerging applications will be distributed, adaptive, and autonomous. Ensuring the safety and correctness of these programs containing uncertain interactions among many parallel components using varying communication models will remain challenging. In this section, I will discuss potential future directions to extend the insights from this work to handle these challenges.

Extending Analysis to other memory models In this dissertation, we primarily focused on the asynchronous message passing model for parallel programs. Extending our analyses for other memory models and communication patterns presents a challenging research problem.

Many techniques exist that can reduce complex parallel programs into simpler versions, or use representative program traces that are sufficient for reasoning about the properties of the original parallel program [135, 136, 137, 138, 139]. Sequentialization approaches such as [140, 141] reduce parallel programs to sequential versions to provide bounded guarantees. Other approaches [142, 143] allow developers to annotate a sequential program and verify that automated parallelizations are equivalent. We can use these techniques to reduce the complexity of analyzing programs using various memory models.

For example, in the *publisher-subscriber* model a set of processes (publishers) publish messages to a *topic* that can be seen by potentially many other processes (subscribers). The ability of multiple processes to publish and subscribe to the same *topic* can result in many potential interleavings in programs making sequentialization based analyses difficult. We need to identify a potential subset of the model with restrictions on communication patterns. For example, a round based system, where messages published in a single round are only allowed to be read by the subscribers on the next round. While such restrictions may limit the programs expressible in a system, they can help developers write safe programs while enabling efficient verification.

Extending Analysis to other domains Domains such as autonomous systems – systems that are designed to react independently without human intervention to environmental stimuli – are growing in popularity today. Autonomous systems often operate on inherently uncertain data, and in many settings, multiple autonomous agents need to coordinate by sharing data in challenging environments, thus adding uncertainty to their execution. A lot of effort has been spent on developing autonomous systems but testing and verifying the correctness of the applications remain a difficult task. As these systems get deployed in safety-critical situations in the presence of humans it is important to ensure that they behave in a predictable and safe manner. While many theoretical results have been developed in research to increase reliability, exposing them to developers in intuitive ways requires careful programming language design.

Improving Dynamic Monitoring of Uncertainty. In addition, As we discussed in Chapter 4, runtime monitoring can add expensive overheads to programs. Diamont injects

monitoring code into programs to update uncertainty intervals for data after every instruction in the program. This adds significant overheads especially for computation intensive programs. This overhead can be reduced by combining statically generated *summaries* of the impact of errors along with data that can be collected at runtime (ex: number of loop iterations or input properties).

Connecting our tools to HPVM [184], a compiler infrastructure for heterogeneous parallel systems, is also a potential goal for the future. HPVM provides a parallel program representation for runtime scheduling used for load balancing and mapping programs to hardware components. Connecting our tools to this system would allow us to evaluate runtime monitoring in an end-to-end programming system.

Programming Abstractions for Wireless Network On Chip (WiNoc). In wireless architectures like Replica, where resources are highly constrained, efficient handling errors become an integral aspect. Hardware defects, implementation complexities, and high utilization can result in permanent or transient faults in wireless networks on chip [185, 186, 187]. Researchers have been working on developing fault tolerant communication protocols [188, 189, 190] to detect and fix errors, but these can be expensive.

In our Replica work, identifying error tolerant data and restructuring programs to efficiently use the wireless architecture required a lot of manual effort. Identify applications, communication protocols, and programming abstractions that are robust and efficient in the presence of unreliable communication is an interesting problem.

References

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

[2] S. Chakradhar, A. Raghunathan, and J. Meng, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications," in *IPDPS*, 2009.

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

[4] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," *ASPLOS*, 2011.

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

[6] P. Stanley-Marbell and M. Rinard, "Perceived-color approximation transforms for programs that draw," *IEEE Micro*, vol. 38, no. 4, 2018.

[7] K. Shafique, B. A. Khawaja, F. Sabir, S. Qazi, and M. Mustaqim, "Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios," *Ieee Access*, 2020.

[8] J. M. Barcelo-Ordinas, J.-P. Chanet, K.-M. Hou, and J. García-Vidal, "A survey of wireless sensor technologies applied to precision agriculture," in *Precision agriculture'13*. Springer, 2013.

[9] U. Shafi, R. Mumtaz, J. García-Nieto, S. A. Hassan, S. A. R. Zaidi, and N. Iqbal, "Precision agriculture techniques and practices: From considerations to applications," *Sensors*, 2019.

[10] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy characterization of a tiled architecture processor with on-chip networks," in *ISLPED*, 2003.

[11] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *IEEE Journal of Solid-State Circuits*, 2008.

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller et al., "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[13] R. Hegde and N. Shanbhag, "Toward achieving energy efficiency in presence of deep submicron noise," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2000.

[14] S. Rajagopal, M. Vinodhini, and N. Murty, "Multi-bit error correction coding with crosstalk avoidance using parity sharing technique for noc," in *2018 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, 2018.

[15] S. Kose, E. Salman, and E. G. Friedman, "Shielding methodologies in the presence of power/ground noise," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2011.

[16] Y. Xu, J. Yang, and R. Melhem, "A process-variation-tolerant method for nanophotonic on-chip network," *J. Emerg. Technol. Comput. Syst.*, 2018.

[17] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, 2008.

[18] J. R. Stevens, A. Ranjan, and A. Raghunathan, "Axba: an approximate bus architecture framework," in *ICCAD*, 2018.

[19] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures," in *ISCA*, 2017.

[20] K. Duraisamy, H. Lu, P. P. Pande, and A. Kalyanaraman, "Accelerating graph community detection with approximate updates via an energy-efficient noc," in *DAC*, 2017.

[21] D. Bertozzi, L. Benini, and G. De Micheli, "Error control schemes for on-chip communication links: the energy-reliability tradeoff," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[22] A. Ejlali, B. M. Al-Hashimi, P. Rosinger, S. G. Miremadi, and L. Benini, "Performability/energy tradeoff in error-control schemes for on-chip networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2010.

[23] D. Kini, U. Mathur, and M. Viswanathan, "Data race detection on compressed traces," in *ESEC/FSE*. Association for Computing Machinery, 2018.

[24] J. C. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[25] A. Choudhary, S. Lu, and M. Pradel, "Efficient detection of thread safety violations via coverage-guided generation of concurrent tests," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[26] W. Zhuang, X. Chen, J. Tan, and A. Song, "An empirical analysis for evaluating the link quality of robotic sensor networks," in *2009 International Conference on Wireless Communications and Signal Processing*, 2009.

[27] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[29] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI*, 2004.

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

[31] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *ASPLOS*, 2018.

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

[33] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Relax*, 2012.

[34] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM TECS Special Issue on Probabilistic Embedded Computing*, 2013.

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

[36] A. Udupa, K. Rajan, and W. Thies, "Alter: Exploiting breakable dependences for parallelization," in *PLDI*, 2011.

[37] R. Akram, M. M. U. Alam, and A. Muzahid, "Approximate lock: Trading off accuracy for performance by skipping critical sections," in *ISSRE*, 2016.

[38] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu, "On approximate speculative lock elision," *IEEE Transactions on Multi-Scale Computing Systems*, no. 2, 2018.

[39] F. Betzel, K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. Karpuzcu, "Approximate communication: Techniques for reducing communication bottlenecks in large-scale parallel systems," *ACM Computing Surveys (CSUR)*, vol. 51, 2018.

[40] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas, "Replica: A wireless manycore for communication-intensive and approximate data," in *ASPLOS*, 2019.

[41] 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.

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

[43] 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.

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

[45] Cray Research Inc., "CRAY T3D System Architecture Overview," 1993.

[46] S. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *ASPLOS*, 1996.

[47] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the Blue Gene/L System Architecture," in *IBM Journal of Research and Development*, March/May 2005.

[48] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *ISCA*, June 1997.

[49] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence Domain Restriction on Large Scale Systems," in *MICRO*, 2015.

[50] P. Stenstrom, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," in *ISCA*, 1993.

[51] J. Oh, A. Zajic, and M. Prvulovic, "Traffic Steering Between a Low-latency Unswitched TL Ring and a High-throughput Switched On-chip Interconnect," in *PACT*, 2013.

[52] C. Batten, A. Joshi, V. Stojanovic, and K. Asanovic, "Designing Chip-Level Nanophotonic Interconnection Networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2012.

[53] N. Kirman, M. Kirman, R. K. Dokania, J. F. Martinez, A. B. Apsel, M. A. Watkins, and D. H. Albonesi, "Leveraging Optical Technology in Future Bus-based Chip Multiprocessors," in *MICRO*, 2006.

[54] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. Beausoleil, and J. Ahn, "Corona: System Implications of Emerging Nanophotonic Technology," in *ISCA*, 2008.

[55] C.-K. Liang and Milos Prvulovic, "MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management," in *ISCA*, 2015.

[56] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures," in *ISCA*, 2007.

[57] S. Abadal, E. Alarcón, A. Cabellos-Aparicio, and J. Torrellas, "WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication," in *ASPLOS*, 2016.

[58] K. Duraisamy, H. Lu, P. P. Pande, and A. Kalyanaraman, "High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics," *ACM Trans. Embed. Comput. Syst*, 2016.

[59] K. Duraisamy, H. Lu, P. P. Pande, and Aananth Kalyanaraman, "Accelerating Graph Community Detection with Approximate Updates via an Energy-Efficient NoC," in *DAC*, 2017.

[60] S. Deb, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo, "Wireless NoC as Interconnection Backbone for Multicore Chips: Promises and Challenges," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, no. 2, 2012.

[61] S. Abadal, A. Mestres, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio, "Medium access control in wireless network-on-chip: A context analysis," *IEEE Communications Magazine*, 2018.

[62] A. Mestres, S. Abadal, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio, "A mac protocol for reliable broadcast communications in wireless network-on-chip," in *Proceedings of the 9th International Workshop on Network on Chip Architectures*, 2016.

[63] A. Franques, "On-chip wireless manycore architectures," Ph.D. dissertation, 2021.

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

[65] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[66] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *IISWC*, 2015.

[67] P. C. Diniz and M. C. Rinard, "Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs," *Journal of Parallel and Distributed Computing*, 1998.

[68] H. H. Nguyen and M. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," in *ISMM*, 2007.

[69] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *PACT*, 2014.

[70] "Stanford Network Analysis Project `snap.stanford.edu`," 2018.

[71] E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo, "A comparison of extrinsic clustering evaluation metrics based on formal constraints," *Information retrieval*, 2009.

[72] R. Ubal, P. Mistry, D. Schaa, H. Ave, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *PACT*, 2012.

[73] "Sci-Kit Learn. `scikit-learn.org`," 2018.

[74] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of SPLASH-2 and PARSEC," in *IISWC*, 2009.

[75] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *IPDPS*, 2009.

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

[77] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *CODES*, 2002.

[78] G. Nychis, C. Fallin, and T. Moscibroda, "On-chip networks from a networking perspective: congestion and scalability in many-core interconnects," in *SIGCOMM*, 2012.

[79] B. K. Daya, L.-s. Peh, and A. P. Chandrakasan, "Quest for High-Performance Bufferless NoCs with Single-Cycle Express Paths and Self-Learning Throttling," in *DAC*, 2016.

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

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

[82] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems (TECS)*, 2013.

[83] M. Rinard, "Parallel synchronization-free approximate data structure construction," in *HotPar*, 2013.

[84] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *RACES*, 2012.

[85] A. Udupa, K. Rajan, and W. Thies, "Alter: Exploiting breakable dependences for parallelization," in *PLDI*, 2011.

[86] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu, "On approximate speculative lock elision," *IEEE Transactions on Multi-Scale Computing Systems*, 2018.

[87] A. Bakst, K. v. Gleissenthall, R. G. Kici, and R. Jhala, "Verifying distributed programs via canonical sequentialization," in *OOPSLA*, 2017.

[88] M. Carbin, D. Kim, S. Misailovic, and M. Rinard, "Verified integrity properties for safe approximate program transformations," in *PEPM*, 2013.

[89] S. Achour and M. Rinard, "Energy efficient approximate computation with topaz," in *OOPSLA*, 2015.

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

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

[92] K. Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala, "Pretend synchrony," in *POPL*, 2019.

[93] A. G. Bakst, "Sequentialization and synchronization for distributed programs," Ph.D. dissertation, UC San Diego, 2017.

[94] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL*, 1998.

[95] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Tech. Rep., 1999.

[96] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A benchmark suite for multi-threaded graph algorithms executing on futuristic multicores," in *IISWC*, 2015.

[97] C. Bienia, *Benchmarking modern multiprocessors*, 2011.

[98] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design Test*, vol. 34, no. 2, April 2017.

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

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

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

[102] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC*, 2013.

[103] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014.

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

[105] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data," *IEEE Transactions on Software Engineering*, vol. 43, 2017.

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

[107] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *ICSE*, 2019.

[108] V. Fernando, K. Joshi, D. Marinov, and S. Misailovic, "Identifying optimal parameters for randomized approximate algorithms," in *Workshop on Approximate Computing Across the Stack*, 2019.

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

[110] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *USENIX ATC*, 2018.

[111] 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," Tech. Rep., 2015.

[112] S. He, S. K. Lahiri, and Z. Rakamarić, "Verifying relative safety, accuracy, and termination for program approximations," *Journal of Automated Reasoning*, vol. 60, no. 1, 2018.

[113] B. Boston, Z. Gong, and M. Carbin, "Leto: verifying application-specific hardware fault tolerance with programmable execution models," in *OOPSLA*, 2018.

[114] A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *FPT*, 2002.

[115] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems," in *FPL*, 2007.

[116] S. Misailovic, D. Roy, and M. Rinard, "Probabilistically accurate program transformations," in *SAS*, 2011.

[117] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy-framework for analysis and optimization of numerical programs (tool paper)," in *TACAS*, 2018.

[118] A. Canino and Y. D. Liu, "Proactive and adaptive energy-aware programming with mixed typechecking," in *PLDI*, 2017.

[119] J. Lidman and S. A. Mckee, "Verifying reliability properties using the hyperball abstract domain," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, no. 1, 2018.

[120] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," 2015.

[121] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *POPL*, 2017.

[122] V. Magron, G. Constantinides, and A. Donaldson, "Certified roundoff error bounds using semidefinite programming," *ACM Transactions on Mathematical Software*, vol. 43, no. 4, Jan. 2017.

[123] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[124] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002.

[125] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, ser. PLDI '12, 2012.

[126] P. Audebaud and C. Paulin-Mohring, "Proofs of randomized algorithms in coq," *Science of Computer Programming*, 2009.

[127] J. Hurd, "Formal verification of probabilistic algorithms," University of Cambridge, Computer Laboratory, Tech. Rep., 2003.

[128] S. F. Siegel and G. S. Avrunin, "Modeling wildcard-free mpi programs for verification," in *PPoPP*, 2005.

[129] S. F. Siegel, "Efficient verification of halting properties for mpi programs with wildcard receives," in *VMCAI*, 2005.

[130] S. F. Siegel and G. Gopalakrishnan, "Formal analysis of message passing," in *VMCAI*, 2011.

[131] E. Michael, D. Woos, T. Anderson, M. D. Ernst, and Z. Tatlock, "Teaching rigorous distributed systems with efficient model checking," in *EuroSys*, 2019.

[132] F. Huch, *Verification of Erlang programs using abstract interpretation and model checking*, 1999.

[133] L.-Å. Fredlund and H. Svensson, "Mcerlang: a model checker for a distributed functional programming language," in *ICFP*, 2007.

[134] E. D'Osualdo, J. Kochems, and C.-H. L. Ong, "Automatic verification of erlang-style concurrency," in *International Static Analysis Symposium*, 2013.

[135] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Springer-Verlag, 1996.

[136] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005.

[137] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *POPL*, 2014.

[138] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Communications of the ACM*, vol. 18, 1975.

[139] A. Desai, P. Garg, and P. Madhusudan, "Natural proofs for asynchronous programs using almost-synchronous reductions," in *OOPSLA*, 2014.

[140] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing context-bounded concurrent reachability to sequential reachability," in *International Conference on Computer Aided Verification*, 2009.

[141] A. Lal and T. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," in *International Conference on Computer Aided Verification*, 2008.

[142] M. Huisman, "A verification technique for deterministic parallel programs," in *PPDP*, 2017.

[143] S. Blom, S. Darabi, and M. Huisman, "Verification of loop parallelisations," in *International Conference on Fundamental Approaches to Software Engineering*, 2015.

[144] M. Charalambides, P. Dinges, and G. Agha, "Parameterized, concurrent session types for asynchronous multi-actor interactions," *Science of Computer Programming*, 2016.

[145] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP*, 1998.

[146] G. Agha and C. Hewitt, "Concurrent programming using actors: Exploiting large-scale parallelism," Cambridge, MA, USA, Tech. Rep., 1985.

[147] G. Agha, "An overview of actor languages," in *OOPWORK*, 1986.

[148] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[149] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, no. 3, 1977.

[150] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. D. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, "Exploiting errors for efficiency: A survey from circuits to algorithms," *CoRR*, vol. abs/1809.05859, 2018.

[151] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15, 2015.

[152] V. Fernando, K. Joshi, and S. Misailovic, "Verifying safety and accuracy of approximate parallel programs via canonical sequentialization," in *OOPSLA*, 2019.

[153] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *OOPSLA*, 2011.

[154] N. Golubovic, C. Krintz, R. Wolski, B. Sethuramasamyraja, and B. Liu, "A scalable system for executing and scoring k-means clustering techniques and its impact on applications in agriculture," *International Journal of Big Data Intelligence*, vol. 6, 2019.

[155] T. Liu, "Datasheet for am2302 sensor," https://cdn-shop.adafruit.com/datasheets/Digital+humidity+and+temperature+sensor+AM2302.pdf, 2020.

[156] L. Paradis and Q. Han, "A survey of fault management in wireless sensor networks," *Journal of Network and systems management*, 2007.

[157] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[158] K. Joshi, V. Fernando, and S. Misailovic, "Aloe: Verifying reliability of approximate programs in the presence of recovery mechanisms," in *CGO*, 2020.

[159] V. Fernando, K. Joshi, and S. Misailovic, "Appendix to Parallely https://vimuth.github.io/parallely/appendix.pdf," 2019.

[160] S. Misailovic, "Accuracy-aware optimization of approximate programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[161] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection (roadnet-pa)," http://snap.stanford.edu/data, June 2014.

[162] Y. Chen and A. Louri, "An approximate communication framework for network-on-chips," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[163] S. Lahiri, A. Haran, S. He, and Z. Rakamaric, "Automated differential program verification for approximate computing," Tech. Rep., May 2015.

[164] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *PLDI*, 2015.

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

[166] B. Maderbacher, A. F. Karl, and R. Bloem, "Placement of Runtime Checks to Counteract Fault Injections," in *RV*, 2020.

[167] E. Darulova and V. Kuncak, "Certifying solutions for numerical constraints," in *RV*, 2012.

[168] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," in *PLDI*, 2018.

[169] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain¡T¿: A first-order type for uncertain data," in *ASPLOS*, 2014.

[170] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *CAV*, 2004.

[171] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *PLDI*, 2014.

[172] A. Albarghouthi and S. Vinitsky, "Fairness-aware programming," in *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT)*, 2019.

[173] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," in *OOPSLA*, 2019.

[174] A. Kannan, N. E. Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors," in *MICRO*, 2015.

[175] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families," in *ISCA*, 2021.

[176] X. Yu, J. Baylon, P. Wettin, D. Heo, P. P. Pande, and S. Mirabbasi, "Architecture and design of multichannel millimeter-wave wireless noc," *IEEE Design & Test*, 2014.

[177] J. O. Sosa, O. Sentieys, and C. Roland, "A diversity scheme to enhance the reliability of wireless noc in multipath channel environment," in *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2018.

[178] M. O. Agyeman, Q.-T. Vien, A. Ahmadinia, A. Yakovlev, K.-F. Tong, and T. Mak, "A resilient 2-d waveguide communication fabric for hybrid wired-wireless noc design," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[179] C. Jianxun, S. Chauvin, and T. El-Ghazawi, "Mpi-based sobel edge detection," 2000, https://upc.lbl.gov/download/dist/upc-tests/benchmarks/gwu_bench/sobel/MPI/.

[180] J. Burkardt, "C code which approximates an integral using a quadrature rule." 2010, https://people.sc.fsu.edu/~jburkardt/c_src/quad_mpi/quad_mpi.html.

[181] J. Burkardt, "Circuit satisfiability using mpi," 2016, https://people.sc.fsu.edu/~jburkardt/c_src/satisfy_mpi/satisfy_mpi.html.

[182] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *Programming Languages and Systems*, 2008.

[183] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *Programming Languages and Systems*, D. Schmidt, Ed., 2004.

[184] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *PPoPP*, 2018.

[185] A. Ganguly, P. Wettin, K. Chang, and P. Pande, "Complex network inspired fault-tolerant noc architectures with wireless links," in *Proceedings of the fifth ACM/IEEE International Symposium on Networks-on-Chip*, 2011.

[186] S. Deb, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo, "Wireless noc as interconnection backbone for multicore chips: Promises and challenges," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2012.

[187] X. Timoneda, S. Abadal, A. Franques, D. Manessis, J. Zhou, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio, "Engineer the channel and adapt to it: Enabling wireless intra-chip communication," *IEEE Transactions on Communications*, 2020.

[188] Y. Ouyang, Q. Wang, Z. Li, H. Liang, and J. Li, "Fault-tolerant design for data efficient retransmission in winoc," *Tsinghua Science and Technology*, 2020.

[189] A. Ganguly, P. Wettin, K. Chang, and P. Pande, "Complex network inspired fault-tolerant noc architectures with wireless links," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, ser. NOCS '11, 2011.

[190] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, "Methods for fault tolerance in networks-on-chip," *ACM Computing Surveys (CSUR)*, 2013.

Appendix A: Full Code Examples

A.1 SCATTER-GATHER

The scatter-gather pattern is similar to the map pattern. However, instead of sending a worker one work item and receiving one result, the worker is sent an entire array. The worker may randomly access parts of the array and returns multiple results. In the code below, for compactness, we also use the task id β as an index variable.

Figure A.1: Scatter Gather Pattern in Parallely

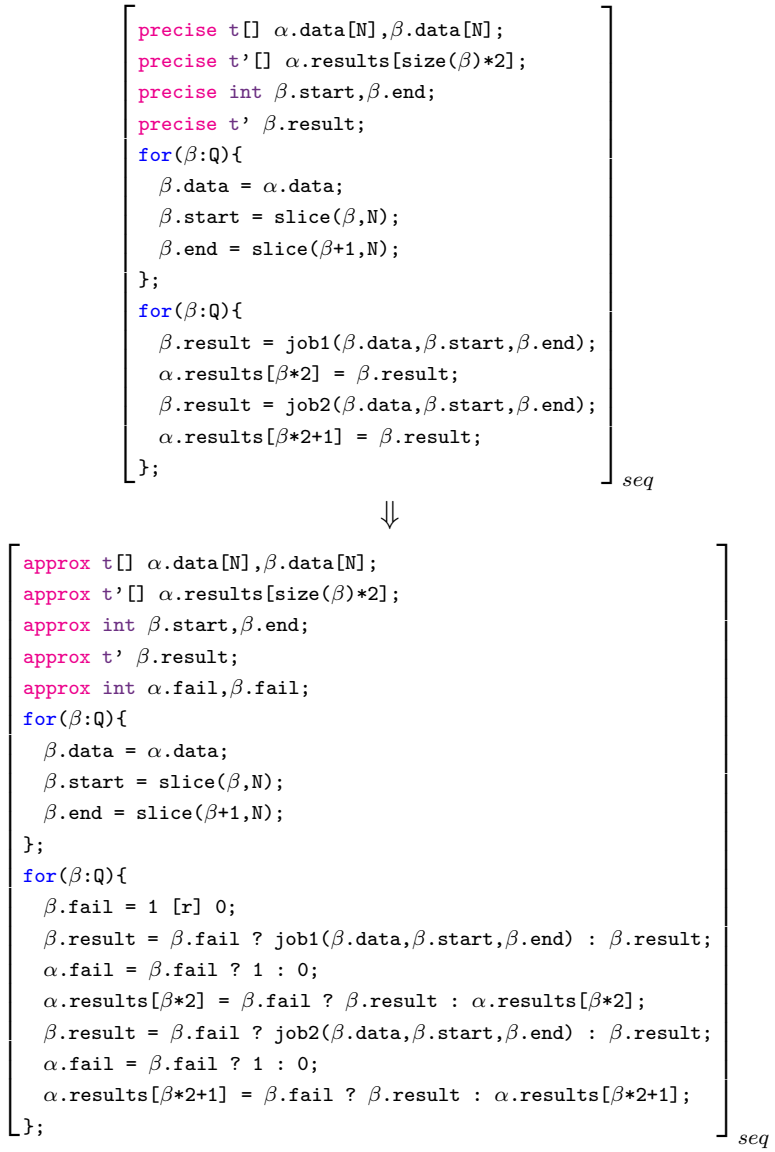


Figure A.2: Sequentialized scatter Gather Pattern in Parallely

Several previous transformations, such as precision reduction, approximate map, failing tasks, etc. can also be applied to this pattern.

A.2 SCAN

The scan pattern takes an input array and generates an output array. The n^{th} element of the output depends on the first n elements of the input and is calculated by an associative function (such as summation, average, etc.) In the code below, for compactness, we also use the task id β as an index variable.

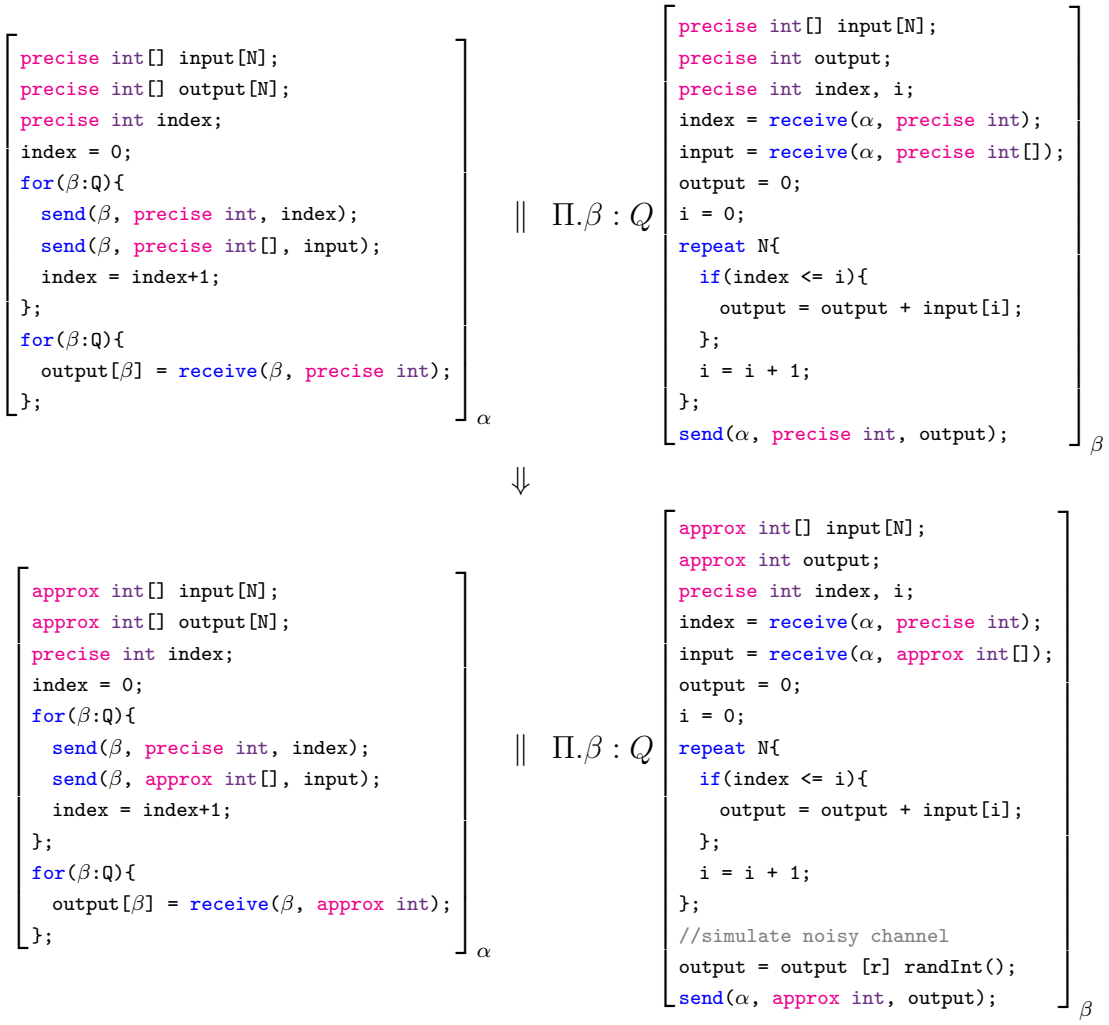


Figure A.3: Scan Pattern in Parallely

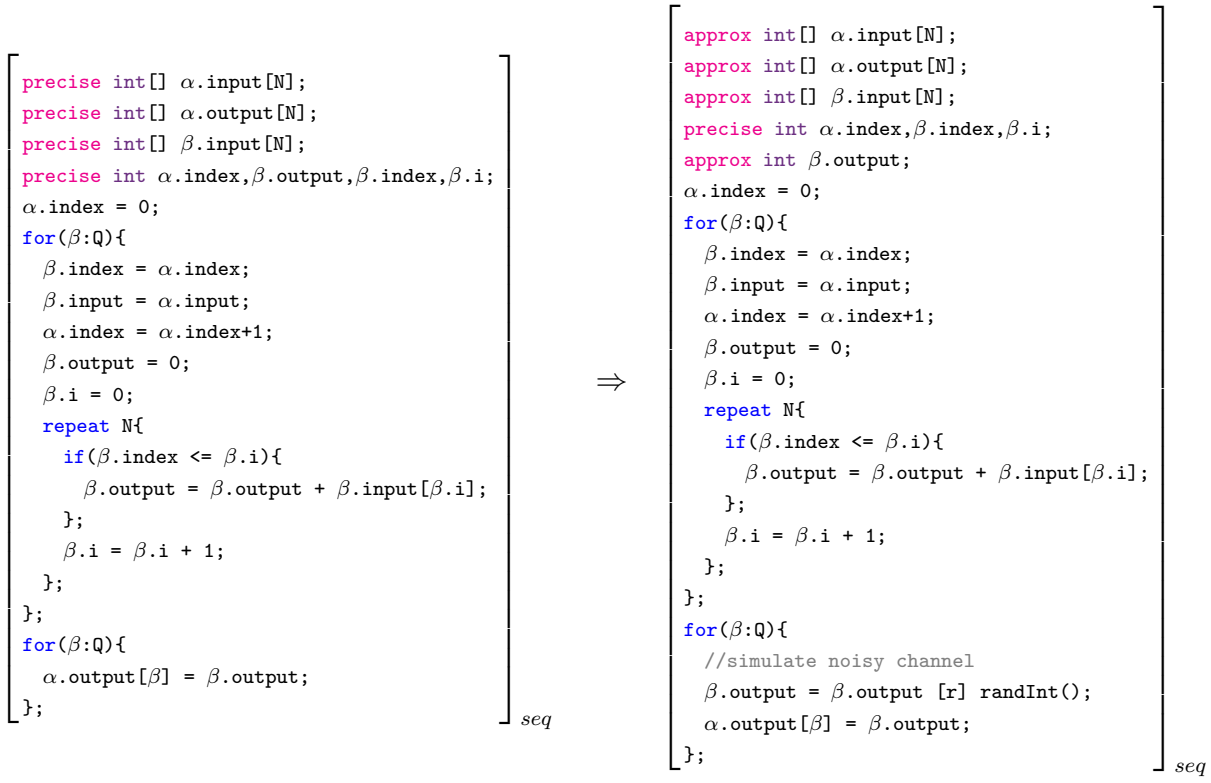


Figure A.4: Sequentialized scan Pattern in Parallely

For this pattern we simulate a noisy channel. Other approximations can also be applied.

A.3 STENCIL

The stencil pattern calculates each element of the output array by applying some function to the corresponding element in the input array along with its neighbors. It is used in many image-processing and scientific applications. In the code below, for compactness, we also use the task id β as an index variable.

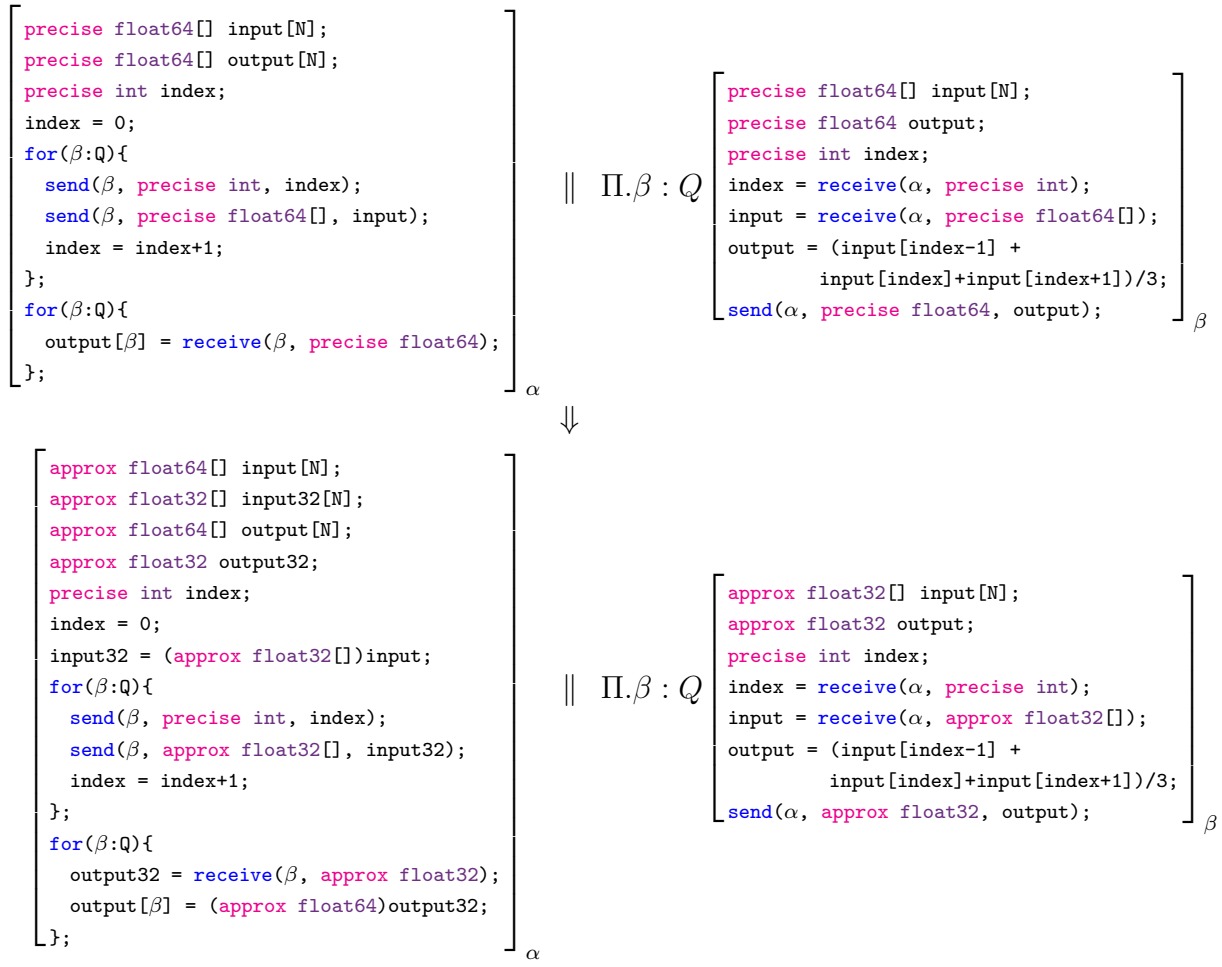


Figure A.5: Stencil Pattern in Parallely

```
precise float64[] α.input[N];
precise float64[] α.output[N];
precise float64[] β.input[N];
precise float64 β.output;
precise int α.index,β.index;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+1;
  β.output = (β.input[β.index-1]+β.input[β.index]+β.input[β.index+1])/3;
};
for(β:Q){
  α.output[β] = β.output;
};
```

seq

⇓

```
approx float64[] α.input[N];
approx float32[] α.input32[N];
approx float64[] α.output[N];
approx float32[] β.input[N];
approx float32 α.output32,β.output;
precise int α.index,β.index;
α.input32 = (approx float32[])α.input;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input32;
  α.index = α.index+1;
  β.output = (β.input[β.index-1]+β.input[β.index]+β.input[β.index+1])/3;
};
for(β:Q){
  α.output32 = β.output;
  α.output[β] = (approx float64)α.output32;
};
```

seq

Figure A.6: Sequentialized stencil Pattern in Parallely

This code simulates precision reduction.

A.4 PARTITION

This pattern is similar to the stencil pattern, but the calculations are performed on disjoint partitions of the input array to obtain the output array. In the code below, for compactness, we also use the task id β as an index variable.

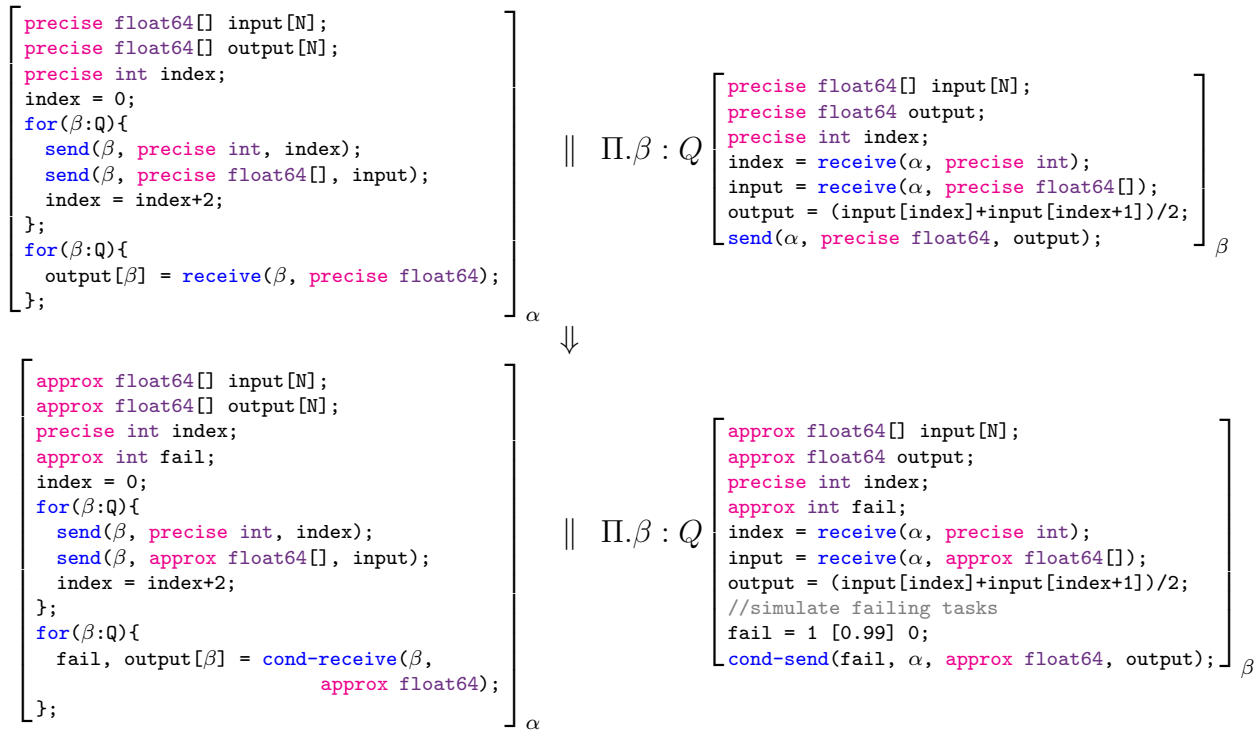


Figure A.7: Partition Pattern in Parallely

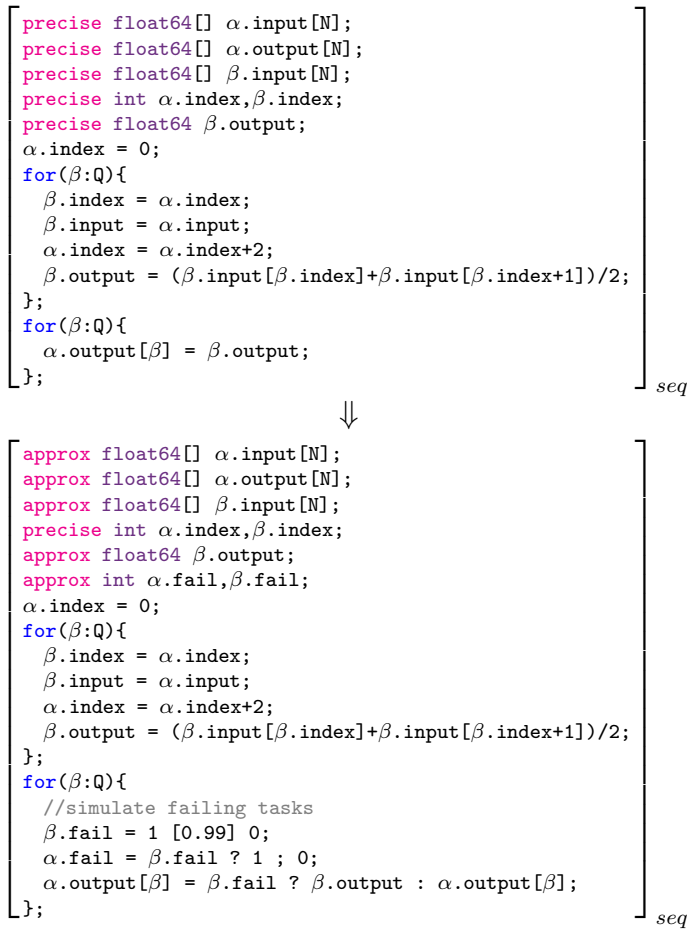


Figure A.8: Sequentialized partition pattern in Parallely

```
for IoTDevice in Q {
  IoTDevice.tempVal, IoTDevice.tempErr, IoTDevice.tempConf := readTemperature()
  IoTDevice.humidVal, IoTDevice.humidErr, IoTDevice.humidConf := readHumidity()
  IoTDevice.temperature = track(IoTDevice.tempVal, IoTDevice.tempErr, IoTDevice.tempConf)
  IoTDevice.humidity = track(IoTDevice.humidVal, IoTDevice.humidErr, IoTDevice.humidConf)
  Manager.data[i] = point{IoTDevice.temperature, IoTDevice.humidity}
}
Manager.centers = // randomly pick some nodes
for Worker in R {
    Worker.data = Manager.data
}
for Manager.j:=0; Manager.j<ITERATIONS; Manager.j++ {
    for Worker in R {
        Worker.centers = Manager.centers
    }
    for Worker in R {
        Worker.newcenters = kmeansKernel(Worker.data, Worker.centers, Worker.assign)
        Manager.newcenters[Worker] = Worker.newcenters [reliability] garbage()
    }
    Manager.centers = AverageOverThreads(Manager.newcenters)
}
```

Figure A.9: Simplified sequentialized program for the Smart Agriculture example