Diamont: Dynamic Monitoring of Uncertainty for Distributed Asynchronous Programs

Vimuth Fernando, Keyur Joshi, Jacob Laurel and Sasa Misailovic

University of Illinois Urbana-Champaign.

*Corresponding author(s). E-mail(s): wvf2@illinois.edu;
Contributing authors: kpjoshi2@illinois.edu; jlaurel2@illinois.edu; misailo@illinois.edu;

Many application domains including graph analytics, the Internet-of-Things, precision agriculture, and media processing operate on noisy data and/or produce approximate results. These applications can distribute computation across multiple (often resource-constrained) processing units. Analyzing the reliability and accuracy of such applications is challenging, since most existing techniques operate on specific fixed error models, check for individual properties, or can only be applied to sequential programs.

We present Diamont, a system for dynamic monitoring of uncertainty properties in distributed programs. Diamont programs consist of distributed processes that communicate via asynchronous message passing. Diamont includes datatypes that dynamically monitor uncertainty in data and provides support for checking predicates over the monitored uncertainty at runtime. We also present a general methodology for verifying the soundness of the runtime system and optimizations using canonical sequentialization.

We implemented Diamont for a subset of the Go language and evaluated eight programs from precision agriculture, graph analytics, and media processing. We show that Diamont can prove important end-to-end properties on the program outputs for significantly larger inputs compared to prior work, with modest execution time overhead: 3% on average (max 16.3%) for our main evaluation input set and 15% on average for 8x larger inputs.

1 Introduction

Many emerging distributed applications operate on inherently noisy data or produce approximate results [55]. 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 [18], TensorFlow's precision reduction [57], or Hogwild's synchronization-eschewing stochastic gradient descent [45]. Also, researchers explored various non-conventional architectures and networks-on-chip [9, 20, 44, 56].

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 9. 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 runtime inefficiencies and/or bugs caused by different interleavings having different effects on uncertainty. 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 Diamont, the first system for *sound, precise and efficient* runtime monitoring of uncertainty in *distributed applications*. Diamont offers a flexible runtime system for specifying and verifying uncertainty bounds in the face of *various sources of uncertainty*. Diamont 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. Diamont 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, Diamont introduces dynamic channels that use specialized communication primitives to reliably transfer the monitoring information.
- **Runtime Monitoring of Uncertainty:** Diamont maintains *uncertain intervals* for dynamically monitored variables – these map variables to a maximum error bound and a probability that the error is within the bound. Diamont propagates this uncertainty through computations. It can precisely do so even for individual array elements and unbounded loops – factors that usually reduce precision of existing analyses like Parallely [21] and DECAF [7].
- **Checkers:** Diamont'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 Diamont'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 Diamont for a distributed fragment of the Go language, extended with the dynamic type and check statements. Diamont 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.

Diamont 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 Diamont a significant advantage over direct extensions of systems like Decaf [7] or AffineFloat [14] to parallel programs.

Verified Runtime and Optimizations. 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. An optimization is sound if all check failures in a program are also guaranteed to occur in its optimized version.

Diamont's runtime system is sound for programs that satisfy the *symmetric nondeterminism* property [4] – 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 [21, 48] satisfy this property. We use *canonical sequentialization* [4, 21], 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, Diamont 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 Diamont 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 Diamont 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 [12, 21, 39].

Our optimizations reduced the runtime overhead of Diamont 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.

Case Studies. We present two case studies expanding the base Diamont system. The first case study looks at supporting distributed recovery mechanisms in Diamont. When checks on uncertainty in the program fail, developers use recovery mechanisms to either re-run computations, or to run more involved checks on the program data to verify their safety. Such distributed recovery mechanisms are difficult to manually implement as the decision to trigger recovery must be consistent across all involved processes. We show such computation patterns can be implemented with Diamont runtime in a sound and safe manner. The second case study looks at verifying algorithmic fairness properties using Diamont. In some decision making programs, fairness can be expressed as arithmetic expressions over expectations of random variables. In our case study, we show that Diamont constructs can be used for runtime fairness analysis.

Contributions. The paper makes several contributions:

- **Diamont.** Diamont is a system for dynamically monitoring uncertainty properties in strongly-typed, message-passing, asynchronous programs. We show that Diamont can soundly monitor uncertainty (error probability and magnitude).
- **Optimizations for reducing overhead.** We present several optimizations to reduce the overhead of runtime monitoring across processes.
- **Implementation.** We implement Diamont's analysis and runtime system with optimizations for a subset of Go.
- **Evaluation.** We evaluate Diamont on 8 benchmarks. We show that Diamont can verify important correctness properties with small runtime overheads.
- **Case Studies.** We present two case studies 1) implementing mechanisms to check for, and recover from excessive uncertainty, 2) extending Diamont to the domain of fairness analysis.

2 Example

We consider a scenario from precision agriculture [23]. 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 1 shows an implementation of the application in Go. The program has multiple parallel processes that communicate over typed channels using the Diamont 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 running on embedded systems $\texttt{IoTDevice}_{1,\ldots,\text{NUMSENSORS}}$ that 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. Diamont 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 35). 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 22-25) for that subset. The `Manager` combines the partial results from each `Worker` and redistributes them (Line 46).

```
1  var Q = [NUMSENSORS] process
2  var R = [NUMWORKERS] process
3
4  type point struct {
5      /*@dynamic*/ temperature, humidity float64
6  }
7
8  func IoTDevice {
9    /*@dynamic*/ var temperature, humidity float64
10   tempVal, tempErr, tempConf := readTemperature()
11   humidVal, humidErr, humidConf := readHumidity()
12   temperature = track(tempVal, tempErr, tempConf)
13   humidity = track(humidVal, humidErr, humidConf)
14   send(Manager, point{temperature, humidity})
15 }
16
17 func Worker {
18   var data [NUMSENSORS] point
19   var centers, newcenters [NUMCENTERS] point
20   /*@dynamic*/ var assign [PERTHREAD] int
21   data = receive(Manager)
22   for iter:=0; iter<ITERATIONS; iter++ {
23     centers = receive(Manager)
24     newcenters = kmeansKernel(data, centers, assign)
25     send(Manager, newcenters)
26 } }
```

```
30 func Manager {
31   // declarations & setup skiped to preserve space
32   for i, IoTDevice := range(Q) {
33     data[i] = receive(IoTDevice)
34   }
35   centers = // randomly pick some nodes
36   for i, Worker := range(R) {
37     send(Worker, data)
38   }
39   for j:=0; j<ITERATIONS; j++ {
40     for _, Worker := range(R) {
41       send(Worker, centers)
42     }
43     for i, Worker := range(R) {
44       newcenters[i] = receive(Worker)
45     }
46     centers = AverageOverThreads(newcenters)
47   }
48   checkArr(centers, 1, 0.99, 4, 0.99)
49 }
```

Figure 1: GoLang: Smart Agriculture Setup

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 [35]). 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. [35]). 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., [60]). Messages in such channels can be corrupted with a small probability [43]. 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, ψ(`Manager`,`Worker`,`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 [8, 12, 49].

2.2 Verification

Properties. We wish to verify that the final values of `centers` are close to the values that would be calculated by a fully precise computation 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. However, we are unable to verify this property using static verification tools such as Parallely [21] due to the following features of the program:

- 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 invalidates 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. Diamont combines static and dynamic analyses to verify safety and accuracy properties at *runtime*. Figure 2 shows the workflow for generating

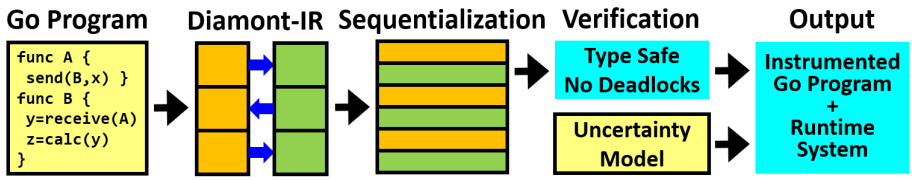


Figure 2: Diamont Workflow

an instrumented program in Diamont. Given a Go program, Diamont 1) translates it to Diamont-IR, whose semantics allows Diamont to easily 2) sequentialize the program and 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 1 is in [19, A.5].

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 Diamont. When a dynamic variable is updated, Diamont also updates the uncertain interval. Diamont'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. Diamont's instrumentation adds runtime overhead. To reduce overhead, Diamont applies optimizations such as constant propagation, dead code elimination, and simplification of monitoring uncertainty in arrays. To reduce overhead when transmitting arrays, Diamont transmits the maximum uncertainty among the elements of the array as the uncertainty of every element of the array. This allows Diamont to transmit only one uncertain interval across processes, while maintaining high analysis precision in other parts of the program. These optimizations reduce Diamont's overhead from 42% to 3.2%. Increasing the number of sensors does not significantly increase overhead (Section 6.3). Even for 2-8x larger data, the overhead remains below 5%.

3 Diamont System

Diamont takes as input a Go program and an uncertainty model. Diamont first converts the program to the Diamont-IR and verifies important safety properties necessary to ensure that the runtime system will be sound. Finally, Diamont generates instrumented Go code. The full syntax and semantics of Diamont are available in [19, 4.3].

3.1 Syntax

Go Language. Diamont supports a subset of the Go Programming Language (matching the features of Diamont-IR along with external functions that do not perform communication) extended with an API for distributed communication and annotations in comments for type qualifiers.

Diamont-IR. Diamont's intermediate representation supports a strongly typed imperative language with primitives for asynchronous communication. Diamont extends the syntax of Parallely [21] with support for the additional `dynamic` type. Figure 3 defines the subset of Diamont syntax dealing with `dynamic` data. 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. Diamont's type qualifiers explicitly split data into either `precise` (no uncertainty), `dynamic` (uncertainty monitored at runtime), or `approx` (uncertain but unmonitored). Diamont'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

$m, v \in \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ *values*

$Exp \to m \mid \langle m,v \rangle \mid x \mid Exp\ op\ Exp$ *expressions*

$AEx \to d \mid d \cdot x \mid d \cdot a[Exp^+]$ *affine*
$\mid AEx \pm AEx$ *expressions*

$q \to$ `precise` $\mid$ `approx` $\mid$ `dynamic` *type qualifiers*

$t \to$ `int<n>` $\mid$ `float<n>` *basic types*

$T \to q\ t \mid q\ t\ [] \mid$ struct T^+ *types*

$P \to [S]_\alpha$ *process*
$\mid \Pi.\alpha : X\ [S]_\alpha$ *process group*
$\mid P \| P$ *parallel comp*

$S \to$ T x $\mid$ T a$[n^+]$ *declarations*
$\mid x$ = *Exp* *assignment*
$\mid x$ = *Exp* $[r]$ *Exp* *probabilistic choice*
$\mid$ dyn-send(α, T, x) *send dynamic*
$\mid x$ = dyn-recv(α, T) *receive dynamic*
$\mid x$ = rdDyn(y) *read dynamic map*
$\mid x$ = endorse(y) *cast to precise*
$\mid x$ = track(y, $\langle$d,r$\rangle^+$) *initiate monitoring*
$\mid x$ = (`dynamic` t) y *cast dynamic to another*
$\mid x$ = *Exp*? *Exp* : *Exp* *conditional choice*
$\mid$ check(AEx, $\langle$d,r$\rangle^+$) *check error*
$\mid$ checkArr(a, $\langle$d,r$\rangle^+$) *check array error*

Figure 3: Diamont-IR Syntax Extensions (full language contains conditionals, loops and function calls)

inference, Diamont 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, Diamont 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. Diamont supports communication of `dynamic` type data through `dyn-send` and `dyn-recv` statements, which also send the monitored uncertainty using reliable channels.

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 Diamont. 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 [49]). 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. Diamont exposes the following statements to check specifications of dynamically monitored variables.

- check(AEx, $\langle$d,r$\rangle$): It checks if an affine expression AEx has a maximum error $\leq d$ with probability $\geq r$. If not, the check fails and creates an error. For example, check($x+y$, $\langle 0.01, 0.99 \rangle$) checks that the error of $x+y$ is ≤ 0.01 with probability ≥ 0.99.
- checkArr(a, $\langle$d,r$\rangle$): It checks if the dynamically monitored uncertainty for *each* element in array a satisfies the specification. Diamont does not support checking of affine expressions over arrays.

While this version of Diamont stops the execution if a check fails, it can be extended to trigger a recovery mechanism instead [1, 17, 26]. Aloe [26] represents recoverable computations with blocks of the form `try {...} check (...) recover {...}`. Using this construct, Diamont can recover the execution if a check fails, and calculate the effect of (possibly imperfect) checks and recovery mechanisms on uncertainty. In Section 7, we provide a case study that looks at implementing recovery mechanisms for distributed programs.

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

3.2 Diamont Semantics

Semantics for `precise` and `approx` data in Diamont are the same as those from Parallely[21]. 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 $\langle n_b, \langle n_1, ..., n_k \rangle \rangle \in$ `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*

S-Assign-Dyn

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

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

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

S-Cast

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

S-Prob-True

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

S-Prob-False

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

S-check-pass

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

S-Check-Fail

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

Figure 4: Semantics of Dynamic Monitoring (Selection)

$$\texttt{calc-eps}(e,D)=\begin{cases} 0 & e \text{ is a constant} \\ D[x].\varepsilon & e \text{ is a variable } x \\ D[x].\varepsilon+D[y].\varepsilon & e \text{ is } x \pm y \\ |x|\times D[y].\varepsilon+|y|\times D[x].\varepsilon+D[x].\varepsilon\times D[y].\varepsilon & e \text{ is } x \times y \\ \infty & e \text{ is } x \div y \ \wedge\ 0\in[y\pm D[y].\varepsilon] \\ \dfrac{(|x|\times D[y].\varepsilon+|y|\times D[x].\varepsilon)}{(|y|\times(|y|-D[y].\varepsilon))} & e \text{ is } x \div y \ \wedge\ 0\notin[y\pm D[y].\varepsilon] \end{cases}$$

$$\begin{aligned} \texttt{calc-del}(e,D) &= \max(0,(\Sigma_{x\in\rho(e)}D[x].\delta)-(|\rho(e)|-1)) \\ \texttt{cast-eps}(x,v,D) &= \max(\max(x+D[x].\varepsilon,v+D[x].\varepsilon)-v,v-\min(x-D[x].\varepsilon,v-D[x].\varepsilon))) \end{aligned}$$

Figure 5: Runtime for Dynamic Monitoring of Uncertainty

$\sigma\in\mathrm{E}=\mathrm{Var}\to\mathrm{Ref}$ maps 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 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 $\langle\sigma^i,h^i\rangle\in\Lambda=\{H\times\mathrm{E}\}\cup\bot$, where $\bot$ is considered to be an error state.

Programs. Diamont defines a program as a parallel composition of processes. We denote a program as $P=[P]_1\,\|\cdots\|\,[P]_n$, where $1...n$ are process identifiers. Individual processes execute their statements sequentially. Each process has a unique process identifier (Pid). Processes can refer to each other using Pids. We write $\langle\text{pid}\rangle.\langle\text{var}\rangle$ to refer to variable $\langle\text{var}\rangle$ of process $\langle\text{pid}\rangle$. When unambiguous, we will omit $\langle\text{pid}\rangle$ and just write $\langle\text{var}\rangle$.

Uncertainty Map. For each process, Diamont defines an *uncertainty map* (D) to attach each variable with an 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 $\boxed{\langle s,\langle\sigma,h\rangle,\mu,D\rangle \xrightarrow{p}_{\psi} \langle s',\langle\sigma',h'\rangle,\mu',D'\rangle}$ 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 presents a selection of the semantics.

- **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 is applied when a `dynamic` variable is updated by assigning it an expression `e`. We use a big-step evaluation relation of the form $\boxed{\langle e,\sigma,h\rangle \Downarrow v}$ to compute the result of the expression. Diamont supports typical integer and floating point operations.

  For `dynamic` variables, in addition to the assigned variable, Diamont updates its interval using the uncertain interval arithmetic defined in Figure 5. The `calc-eps` function is used to calculate an expression's maximum error by propagating the accompanying error ε through sub-expressions, similarly to how automatic differentiation propagates dual numbers through arithmetic expressions [32, 33]. The confidence in this maximum error is then 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 (unlike the strict independence assumptions of [7]) Diamont uses the conservative union bound.
- **Communication:** When sending `dynamic` variables of type T to another process (rule `S-DynSend`), Diamont uses special channels (D_T) that are assumed to be fully reliable to communicate the relevant uncertain intervals before sending the data. If reliable channels are not readily available, Diamont uses transmission protocols to achieve reliability over unreliable channels. ++ denotes adding an element to the end of the message queue. At the receiver (rule `S-DynReceive`), Diamont updates the local uncertainty map. Diamont 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:** Diamont 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.
- **Conditionals:** For branching on `dynamic` values, Diamont supports an operator x `= cond?` $e_1 : e_2$ (conditional choice) where `cond` compares a `dynamic` value against a threshold. We check if the *entire* interval associated with the value is greater or less than the threshold. If neither case is true, we compute both expressions and the interval of x becomes the smallest closed interval that contains all possible intervals.
- **Checks:** If a check fails, the Diamont program transitions into an error state (Figure 4 rule `S-Check-Fail`). To prevent such check failures, the user can implement error recovery mechanisms.

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

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

Figure 6: Diamont Global Semantics

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 6, small step transitions of the form $\boxed{(\epsilon,\omega,\mu,P)\xrightarrow{\alpha,r}_{\psi}(\epsilon',\omega',\mu',P')}$ define a process α taking a step and thus changing the global configuration. Inter-process communication happens using the typed channels – though processes adding to and reading from the relevant queue. Complete semantics are available in [19, 4.3].

3.3 Canonical Sequentialization

Diamont's runtime system works across distributed processes. We use *Canonical Sequentialization* [4] to simplify our reasoning about the soundness of the

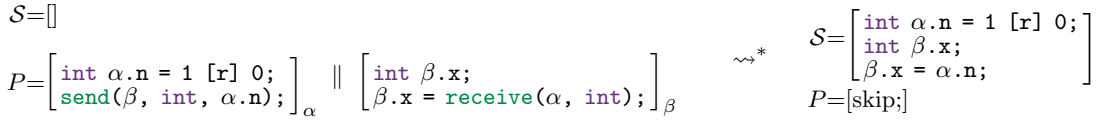


Figure 7: Canonical Sequentialization: An Example of the Rewriting Process.

runtime system. Canonical sequentialization generates 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 $\boxed{\Gamma,\mathcal{S},P \rightsquigarrow \Gamma',\mathcal{S}',P'}$ to rewrite a parallel program P to a sequential program $\mathcal{S}'$ step by step (the rules are available in [19, 4.3]). 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 $\mathcal{S}'$. We extend the results from prior work [4, 21] 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 7 shows a small program with inter-process communication (P) and its canonical sequentialization ($\mathcal{S}$) 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:

```
[α.res = receive(*);]α ‖ [β.out = func1();
                          send(α, β.out);]β ‖
                         [γ.out = func2();
                          send(α, γ.out);]γ
```

[1] Many popular parallel application patterns (e.g. Map, Reduce, Scatter-Gather, Stencil) exhibit symmetric non-determinism [4, 21] and programs satisfying this property are less error-prone [4].

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

After providing some necessary background and definitions in Section 3.4, we prove in Section 3.5 that Diamont's runtime monitoring system is sound for programs with a canonical sequentialization. Specifically, we show that, *if a program is canonically sequentializable, then Diamont's dynamic uncertainty monitoring may over-estimate, but never under-estimates the true uncertainty of a program variable.*

3.4 Background and Definitions

In this section, we define the terms used in our proof for the soundness of Diamont's dynamic monitoring, using the notation developed in Chisel [39]. We define how to quantify *true error* and *true reliability* (probability of being within the error bound) in programs using *paired* execution semantics.

Def 1 (Partial Trace Semantics for Parallel Programs)
$\langle s, \epsilon,\omega\rangle \overset{\tau,p}{\Longrightarrow}_\psi \langle s', \epsilon',\omega'\rangle \equiv \langle \epsilon,\omega,., s\rangle \xrightarrow{\lambda_1,p_1}_\psi \ldots \xrightarrow{\lambda_n,p_n}_\psi \langle \epsilon',\omega',., s'\rangle$

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 τ 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.

Def 2 (Aggregate Semantics for Parallel Programs)

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

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.

Def 3 (Paired Execution Semantics)

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

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:

$$[\![\mathcal{R}^*[Q_A]]\!](\epsilon,\varphi)=\textstyle\sum_{\epsilon_u\in\mathcal{E}(Q_A,\epsilon)}\varphi(\epsilon_u)$$

where $\mathcal{E}(Q_a,\epsilon)$ represents the set of all environments in which the predicate Q_A is satisfied ($\mathcal{E}(Q_A,\epsilon)=\{\epsilon' \mid \epsilon'\in Env \wedge \epsilon'\in[\![Q_A]\!]\}$).

3.5 Proof of Soundness

We prove the following soundness theorem for Diamont programs with a canonical sequentialization:

Theorem 1 (Soundness of dynamic monitoring) *For programs not containing* `track` *and* `endorse` *statements, for all statements s, and for all x s.t. $\Theta\vdash x$: dynamic t, $\Theta \vdash s : \Theta'$ and $\langle s,\langle\sigma,D,\varphi\rangle\rangle \Downarrow \langle s',\langle\sigma',D',\varphi'\rangle\rangle \implies [\![\mathcal{R}^*[D'[x].\varepsilon \geq \Delta(x)]]\!](\sigma',\varphi') \geq D'[x].\delta$*

Recall that Diamont's runtime monitors two properties for each `dynamic` variable x: (1) the maximum possible error magnitude ($D[x].\varepsilon$) and (2) a probability ($D[x].\delta$) that the *precise* value of x is within $x\pm D[x].\varepsilon$. 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.

Similar to Parallely, programs in Diamont 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 Diamont 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.

We start by assuming that the statement is true for all traces of length n. Then, $\langle s,\langle\epsilon,\omega,\varphi\rangle\rangle \Downarrow \langle s^n,\langle\epsilon^n,\omega^n,\varphi^n\rangle\rangle$ and $\forall x, [\![\mathcal{R}^*[\omega^n[x].\varepsilon \geq \Delta(x)]]\!](\epsilon^n,\varphi^n) \geq \omega^n[x].\delta$[2]. Next, we 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 a process taking the step $\langle\epsilon^n,\mu^n,D^n, s^n\rangle \xrightarrow{\alpha,p}_{\psi} \langle\epsilon^{n+1},\mu^{n+1},D^{n+1}, s^{n+1}\rangle$ We show here intuition behind the proof for the case of expression assignment.

Case S-Assign-Dyn $s : y = e$;

From the semantics of assignment (Figure 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.

[2]We assume that the variable x is only present in a process α and abbreviate $\omega[\alpha][x]$ as $\omega[x]$

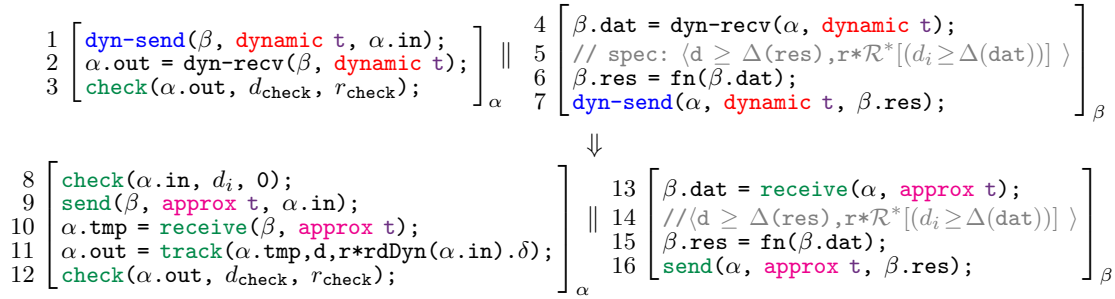


Figure 8: Optimizations Using Static Analysis in Diamont.

We start with the observation that by definition,

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

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

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 5 using interval arithmetic. The soundness of the calculations has been shown in prior work [38]. 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 [19, 4.3].

Case S-Prob-True s : $y = e_1[r]e_2$

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.

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

Next, we utilize canonical sequentialization to prove that the theorem holds for the parallel subset of the language as well. First, we extend the results from [21] to prove that if we can rewrite a parallel program P into a sequential program $\mathcal{S}$, then P and $\mathcal{S}$ have equivalent behavior. We use this fact to reason that our proof of soundness for the sequential subset of Diamont is also applicable to parallel programs that can be canonically sequentialized. Therefore, Theorem 1 holds and our overall analysis is sound (full proof is available in [19, 4.3]).

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 [49], 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 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.

Communication. When communicating large `dynamic` type arrays, Diamont must also communicate the uncertain interval for each array

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

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

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

Figure 9: Moving checks earlier in Diamont.

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

Figure 10: Example Sequentializations Used in the Proofs

element. Because Diamont requires untertain intervals to be communicated over reliable channels (or transmission protocols), this results 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, Diamont still achieves better results than existing static analyses which use a single uncertain interval for arrays through the *entire* program.

Utilizing static analysis. We can further reduce overheads by exploiting common communication patterns. For example, the program at the top of Figure 8 contains a remote procedure call. Process α sends an input to process β, which applies the function `fn` to the input and returns the value. Transferring uncertain intervals along with the data can become expensive if many such calls are made.

We use existing static analysis techniques [12, 21, 39] to analyze only the remote function call and generate function specifications (precise semantics are in [19, 4.3]), even if they are unable to analyze the entire program. Consider the transformed program at the bottom of Figure 8. Using the specification, Diamont 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.

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 9 are some examples.

In the first rule, Diamont 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. Diamont can now safely move the check before the addition.

These re-write rules closely follow the static analysis as defined and proven sound in [21] for the

sequential subset of the language. 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).

Debloating and compiler optimizations. Diamont reduces overhead by using constant propagation and dead code elimination to remove unnecessary updates to the uncertainty map. In addition, Diamont 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 [12]) Diamont will only calculate confidence at runtime.

4.1 Soundness

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:

Def 4 (Optimization soundness) *For a program s and its optimized version s_{opt}, $\langle s,\langle\sigma,h\rangle,\mu,D\rangle \xrightarrow{*}_{\psi} \langle s',\perp,\_,\_\rangle \implies \langle s_{opt},\langle\sigma,h\rangle,\mu,D\rangle \xrightarrow{*}_{\psi} \langle s'',\perp,\_,\_\rangle$*

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. Consider the static analysis based optimization in Figure 8. Proving the soundness of this optimization 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 s^{seq} and s^{seq}_{opt} in Figure 10. These sequentializations produce final environments that are equivalent to the original versions. 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} (The full proofs are in [19, 4.4]).

5 Methodology

Implementation and Testing Setup We parsed and translated Go programs written using a library of Diamont primitives to Diamont-IR using ANTLR. We used Python to sequentialize Diamont 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 1). 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 [41]. Single Source Shortest Path is used to make data routing decisions. Breadth First Search is used to find connected components in graphs. From CRONO [2].
- *SOR:* A kernel for successive over-relaxation. Used to extrapolate the state of a system over time. From Chisel [39].
- *Sobel:* Edge-detection filter. From AxBench [59].
- *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 (as discussed in Section 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.

Inputs. Table 2 gives the size of the primary inputs we used to evaluate each benchmark (Column 3) and the number of worker threads (Column 2). Apart from the worker threads, each benchmark also contained one master thread. We used additional input sizes solely to evaluate the effect of

Table 1: Benchmarks, Verified Properties, and Overhead for Diamont. Baselines: ⋆:Decaf, †:AffineFloat

Benchmark	Pattern	Verified Property	Overhead	
			Baseline	Diamont
PageRank	Scatter-Gather	checkArr(pagerank, 0, 0.9912)	30%*	3.63%
SSSP	Scatter-Gather	checkArr(distance, 0, 0.9925)	33%*	2.31%
BFS	Scatter-Gather	checkArr(visited, 0, 0.9925)	30%*	4.06%
SOR	Stencil	checkArr(output, 1.19×10^{-7}, 1)	60%†	3.49%
Sobel	Stencil	checkArr(output, 2.38×10^{-7}, 1)	71%†	9.71%
Matrix Mult.	Map	checkArr(product, 6.6×10^{-6}, 1)	80%†	16.27%
Kmeans-Agri	Map	checkArr(centers, ⟨1.5, 0.9948⟩, ⟨2,0.9948⟩)	42%*†	3.32%
Regression	Map-Reduce	check(alpha, 0, 0.99)∧check(beta, 0, 0.99)	37%*	0.45%

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

Table 2: Input Size and Number of Threads Used for Evaluation of Benchmarks

optimization on runtime and communication volume. For Section 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 [34] (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 Diamont with optimizations to a baseline which is a straightforward parallel implementation of an existing static analysis via Diamont (either Decaf [7] or AffineFloat [14] without roundoff errors).

6 Evaluation

6.1 Can we verify important uncertainty properties using Diamont?

For each benchmark, we used Diamont to verify the properties shown in Column 3 of Table 1. Diamont successfully verified these properties on the final output of the program. 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. We discussed Kmeans-Agri in detail in Section 2. 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 [21] cannot verify these properties. Diamont'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 Diamont could verify nearly the same uncertainty bounds. For example, for BFS, Diamont could verify a confidence of 0.999 when using the baseline version. Dividing an array into a small number of chunks and maintaining an uncertain interval for each chunk is a middle ground that would reduce the impact on precision, and is an interesting topic for future work. For benchmarks with regular computation patterns, such as SOR and Regression, there was no significant change.

In summary, *Diamont verifies important end-to-end uncertainty properties that cannot be verified using existing static analyses.*

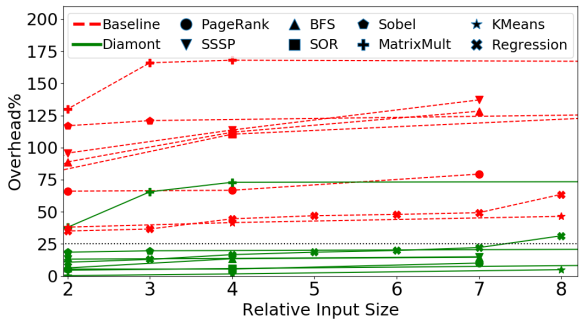
6.2 What are the overheads associated with Diamont?

Columns 4 and 5 of Table 1 present the overhead of the baseline and optimized Diamont 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 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 and vastly reduces the number of uncertain intervals that Diamont must send reliably. Debloating also provided significant speedups. For example, without debloating, Diamont overhead is 3.9x higher for PageRank and 3.3x higher for Sobel.

Are the overheads justified? Approximations have led to significant savings in prior work: 1) Communication: up to 62% performance improvement in approximate NoCs [13, 20], and 2) Computation: 2x speedup in loop perforation [54], 2.7x speedup in Paraprox [48], and up to 1.3x speedup from reduced precision in Precimonious [47]. As Diamont'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.



Size	Baseline	Diamont
2x	69.1%	8.68%
4x	84.4%	10.2%
6x	93.7%	14.1%
8x	91.8%	12.9%

Figure 11: Input Size vs. Overhead. Table shows geomean overheads across programs.

In summary, *With optimization, overhead of Diamont analysis is at most 16.3% for our benchmarks, with a geomean of 3.04%.*

6.3 How does Diamont overhead depend on the program inputs?

Figure 11 shows the effect of input size on Diamont 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 Diamont 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 11 shows the geomean of the overhead across all benchmarks 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 Diamont 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. Figure 12 shows the communicated data volume for each benchmark as the input size is increased. The results show that the amount of communicated data scales better with the Diamont optimizations. The benefits are 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 Diamont runtime performance increases over the baseline system.*

7 Case Study: Responding to Check Failures

As Diamont monitoring exposes the uncertainty of variables to the program, making decisions based on the uncertainty is a logical choice. Diamont can provide mechanisms for easily specifying recovery mechanisms distributed across multiple processes. This allows the distributed program to recover from excessive error by redoing computation or communication in a more precise manner. 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 13 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. If the manager decides that the error in the received data is too high, the check fails, and it prompts the worker to resend uncompressed data. This allows the system to save bandwidth when error is reasonable after compression. However, in this version the manager is only sending the result of the check if it fails. If it passes, the worker gets stuck waiting for the result.

Such errors become more difficult to notice and fix when many processes are participating 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 performs checks on the communicated variable, their decisions may differ. This can lead to a deadlock if only one process decides that resending the variable is necessary.

Figure 14 shows the same program as above implemented using the recovery mechanisms in Diamont. 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`. Diamont automatically handles the transmission of the check result and checks for other safety properties to ensure bug-free recovery.

Translation. Figure 15 shows how the mechanism is translated to Diamont-IR. By analyzing this IR, Diamont 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.

Sequentialization. As multiple processes may be involved in recovery computations, Diamont 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, Diamont 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) Diamont 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, Diamont ensures that only one process performs a check, 4) Diamont rewrites the distributed recovery mechanism to a sequential recovery, 5) to the process performing the check, Diamont writes the list of other processes to the `recover-with` annotation, 6) lastly, for the other processes, Diamont writes the process performing the check to the `recover-from` annotation. If Diamont successfully generates a sequentialization, the whole computation is deadlock free for both outcomes of the check. Figure 16 shows the sequentialized version of the code in Figure 15.

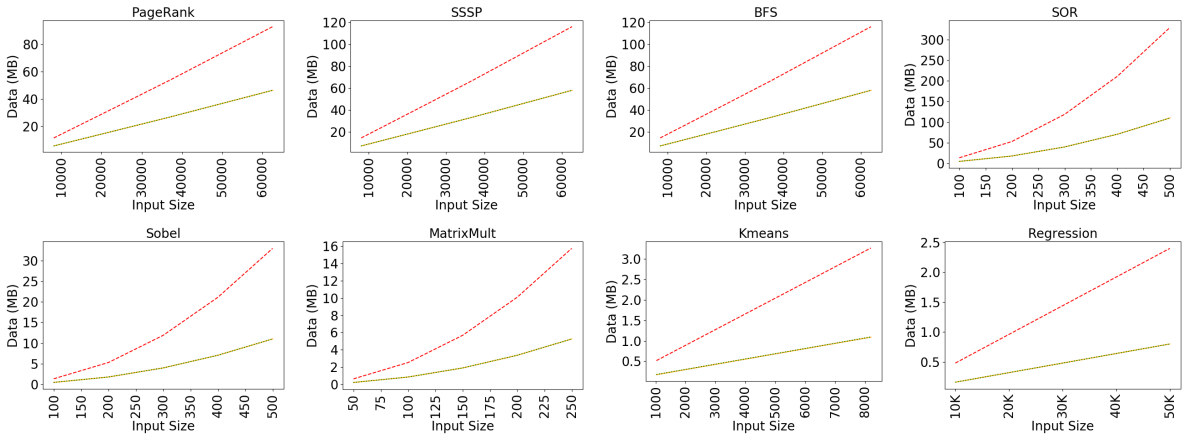


Figure 12: Input size vs. communication volume. Dashed red line is the baseline, orange line is Diamont.

```
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 13: Manual Recovery Causing Deadlock

Soundness. To maintain soundness of the dynamic monitoring, Diamont needs to perform additional checks. Diamont'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, Diamont 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 [26], 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 Diamont, 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` re-ran 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 3 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 lead 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.

8 Case Study: Algorithmic Fairness

In addition to checking accuracy and reliability, Diamont is expressive enough to monitor *algorithmic fairness* properties, such as those in [3, 5].

Fairness specifications are given in [3, 5] 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

```
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 14: Distributed Recovery Mechanism in Go+Diamont

```
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 15: Distributed Recovery Mechanism in Diamont-IR. (All variables are dynamic)

```
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 16: Recovery Code After Sequentialization

have a distinct dynamically tracked variable (e.g. `x` and `y`). Semantically, this allows Diamont 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 Diamont IR) can be seen in Fig. 17. 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 Diamont, 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 Diamont can propagate uncertainty intervals through arithmetic expressions as shown in Fig. 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 $\mathbb{E}[Y]$, 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 Diamont runtime will compute a valid uncertainty interval for this

```
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 17: Source-level Fairness Encoding

Benchmark	Recovery Rate
Pagerank	1.4%
SSSP	4.5%
BFS	4.6%
Matrix Mult	28%
Sobel	8.1%

Table 3: Recovery Rates for Benchmarks

Benchmark	Overhead
Hiring	3.9%
Income SVM	3.0%
Income Decision Tree	6.1%
Income Neural Network	2.5%

Table 4: Overheads for Fairness Benchmarks

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, which means that we only certify that that the predicate φ holds *with high probability*. However for algorithmic fairness, this is standard practice, as the predicates in [3, 5] 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 Diamont'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`)`.

Empirically we evaluate this approach on benchmarks taken from [3, 5] 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 $X = \mathbf{1}_{Hire|Female}$ and $\mathbb{E}[Y]$ is the expectation of the indicator $Y = \mathbf{1}_{Hire|Male}$. The fairness certification check compiles down to Diamont IR as `check(z,z-0.8,0.9)` where as before, `z = x/y` where `x` and `y` are dynamically tracked variables for $\mathbb{E}[X]$ and $\mathbb{E}[Y]$.

Table 4 shows the overheads for verifying φ using Diamont. In all cases the overheads were low, highlighting the fact that Diamont is expressive and flexible enough to be adapted to efficiently certify properties beyond those found in standard approximate computing.

9 Related Work

Several analyses are related (in part) to Diamont's functionality, as shown in Table 5. 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, Diamont 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., [10, 11, 16, 29–31, 39, 40, 42, 49, 51]), most relevant to Diamont is Parallely [21], which retains the limitations of the underlying static analyses requiring developers to provide bounds on

Table 5: Comparison of Related Work. (✓* indicate analyses that monitor confidence intervals, which is another interpretation of Diamont's uncertain intervals)

Method	Static	Empirical	Runtime	Reliability	Accuracy	Verified	Parallel	Multi-Source
Diamont	✓	×	✓	✓	✓	✓	✓	✓
Parallely	✓	×	×	✓	✓	✓	✓	✓
Rely	✓	×	×	✓	×	✓	×	×
Chisel	✓	✓	×	✓	✓	✓	×	✓
DECAF	✓	×	✓	✓	×	✓	×	×
EnerJ	✓	✓	×	×	×	✓	×	×
AffineFloat	✓	×	✓	×	✓	✓	×	×
PAssert	×	✓	✓	✓*	✓*	✓	×	×
Uncertain<T>	×	✓	✓	✓*	✓*	×	×	×

loop iterations, array sizes, and number of processes. In contrast, Diamont successfully combines static and dynamic analysis and works on a real language (Go), which jointly allow for verification of much larger benchmarks. Additionally, Diamont also extends sequentialization for dynamic conditions.

Dynamic Analysis and Runtime Monitoring. DECAF [7] 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. [46] propose offline and online approaches to monitor the quality of programs, using methods such as dataflow techniques and comparison to the precise program. Diamont instead propagates uncertain intervals during both static and dynamic phases, allowing it to monitor uncertainty with greater precision. Maderbacher et al. [36] focus on precisely correcting bitflips with minimal checks. In contrast, Diamont monitors uncertainty from many sources in programs that can tolerate some error.

Several software [24, 27, 28, 37] and hardware [58] approaches provide representation and arithmetic operations for data with uncertainty using approximations of distributions, or Monte Carlo simulations. Diamont can make use of such systems for local computations, and extend uncertainty monitoring to distributed programs using multiple such devices when the distributions can be converted to the uncertainty intervals used in Diamont.

AffineFloat [14] and Ceres [15] provide dynamic analysis for numerical error. Herbgrind [52] locates possible sources of numerical error. These tools measure floating point roundoff errors, but have high overhead. Diamont focuses on analyzing error from casting and external sources e.g., sensors. Uncertain⟨T⟩ [6] used an early form of uncertain intervals, however they use sampling to determine error. Statistical model checking tools [53] can provide statistical guarantees on program properties expressed in a temporal logic. PAssert [50] and AxProf [25] statistically verify at development time a single probabilistic assertion at the end of the program. In contrast to these techniques, Diamont supports many checks at different points in the program at runtime.

10 Conclusion

The past decade brought many techniques for developing new approximations and analyzing uncertainty for specific scenarios, but much less work has been done in integrating these diverse concepts in a unifying, rigorous, and extensible framework. Diamont 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, and media processing).

We demonstrated the benefit of our analysis and optimizations by reducing the execution overhead to 3% on average (16.3% maximum). We believe this work can serve as a starting point for sound runtime systems in domains that need to rigorously handle uncertainty, such as robotics or the Internet-of-Things.

Acknowledgements. We thank the anonymous reviewers for their useful suggestions. The research presented in this paper was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2028861, and CCF-2008883, USDA Grants No. NIFA-2024827, NIFA-2021-67021-33449, and a gift from Facebook. A previous version of this work appeared in RV 2021 [22].

References

[1] Achour S, Rinard M (2015) Energy Efficient Approximate Computation with Topaz. OOPSLA

[2] Ahmad M, Hijaz F, Shi Q, et al (2015) CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In: IISWC

[3] Albarghouthi A, Vinitsky S (2019) Fairness-aware programming. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* '19

[4] Bakst A, Gleissenthall Kv, Kici RG, et al (2017) Verifying distributed programs via canonical sequentialization. In: OOPSLA

[5] Bastani O, Zhang X, Solar-Lezama A (2019) Probabilistic verification of fairness properties via concentration. In: OOPSLA

[6] Bornholt J, Mytkowicz T, McKinley KS (2014) Uncertain<T>: A First-order Type for Uncertain Data. In: ASPLOS

[7] Boston B, Sampson A, Grossman D, et al (2015) Probability type inference for flexible approximate programming. In: OOPSLA

[8] Boston B, Gong Z, Carbin M (2018) Leto: verifying application-specific hardware fault tolerance with programmable execution models. In: OOPSLA

[9] Boyapati R, Huang J, Majumder P, et al (2017) APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures. In: ISCA

[10] Carbin M, Kim D, Misailovic S, et al (2012) Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. PLDI

[11] Carbin M, Kim D, Misailovic S, et al (2013) Verified integrity properties for safe approximate program transformations. PEPM

[12] Carbin M, Misailovic S, Rinard M (2013) Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In: OOPSLA

[13] Chen Y, Louri A (2020) An Approximate Communication Framework for Network-on-Chips. IEEE Transactions on Parallel and Distributed Systems

[14] Darulova E, Kuncak V (2011) Trustworthy Numerical Computation in Scala. In: OOPSLA

[15] Darulova E, Kuncak V (2012) Certifying solutions for numerical constraints. In: RV

[16] Darulova E, Izycheva A, Nasir F, et al (2018) Daisy-Framework for Analysis and Optimization of Numerical Programs. In: TACAS

[17] de Kruijf M, Nomura S, Sankaralingam K (2010) Relax: an architectural framework for software recovery of hardware faults. ISCA

[18] Dean J, Ghemawat S (2004) MapReduce: Simplified data processing on large clusters. OSDI

[19] Fernando V (2021) Programming systems for safe and accurate parallel programs in the face of uncertainty. PhD thesis, University of Illinois Urbana-Champaign https://www.ideals.illinois.edu/items/125369

[20] Fernando V, Franques A, Abadal S, et al (2019) Replica: A Wireless Manycore for Communication-Intensive and Approximate Data. In: ASPLOS

[21] Fernando V, Joshi K, Misailovic S (2019) Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. In: OOPSLA

[22] Fernando V, Joshi K, Laurel J, et al (2021) Diamont: Dynamic monitoring of uncertainty for distributed asynchronous programs. In: Runtime Verification

[23] Golubovic N, Krintz C, Wolski R, et al (2019) A scalable system for executing and scoring K-means clustering techniques and its impact on applications in agriculture. International Journal of Big Data Intelligence 6

[24] Jaroszewicz S, Korzeń M (2012) Arithmetic operations on independent random variables: A numerical approach. SIAM Journal on

Scientific Computing 34

[25] Joshi K, Fernando V, Misailovic S (2019) Statistical algorithmic profiling for randomized approximate programs. In: ICSE

[26] Joshi K, Fernando V, Misailovic S (2020) Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms. CGO

[27] Joshi K, Hsieh C, Mitra S, et al (2023) Gas: Generating fast and accurate surrogate models for autonomous vehicle systems. https://arxiv.org/abs/2208.02232

[28] Lafarge T, Possolo A (2015) The nist uncertainty machine. NCSLI Measure 10(3):20–27

[29] Lahiri S, Haran A, He S, et al (2015) Automated Differential Program Verification for Approximate Computing. Tech. rep.

[30] Laurel J, Misailovic S (2020) Continualization of probabilistic programs with correction. In: ESOP

[31] Laurel J, Yang R, Sehgal A, et al (2021) Statheros: Compiler for efficient low-precision probabilistic programming. In: Design Automation Conference (DAC), 2021

[32] Laurel J, Yang R, Singh G, et al (2022) A dual number abstraction for static analysis of clarke jacobians. In: POPL

[33] Laurel J, Yang R, Ugare S, et al (2022) A general construction for abstract interpretation of higher-order automatic differentiation. In: OOPSLA

[34] Leskovec J, Krevl A (2014) SNAP Datasets: Stanford large network dataset collection (roadnet-pa). http://snap.stanford.edu/data

[35] Liu T (2020) Datasheet for AM2302 Sensor. https://cdn-shop.adafruit.com/datasheets/Digital+humidity+and+temperature+sensor+AM2302.pdf

[36] Maderbacher B, Karl AF, Bloem R (2020) Placement of Runtime Checks to Counteract Fault Injections. In: RV

[37] Manousakis I, Goiri Í, Bianchini R, et al (2018) Uncertainty propagation in data processing systems. In: Proceedings of the ACM Symposium on Cloud Computing

[38] Misailovic S (2015) Accuracy-aware optimization of approximate programs. PhD thesis, Massachusetts Institute of Technology

[39] Misailovic S, Carbin M, Achour S, et al (2014) Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In: OOPSLA

[40] Misra A, Laurel J, Misailovic S (2023) Vix: Analysis-driven compiler for efficient low-precision variational inference. In: Design, Automation & Test in Europe Conference & Exhibition (DATE)

[41] Page L, Brin S, Motwani R, et al (1999) The PageRank citation ranking: Bringing order to the web. Tech. rep.

[42] Panchekha P, Sanchez-Stern A, Wilcox JR, et al (2015) Automatically Improving Accuracy for Floating Point Expressions. In: PLDI

[43] Paradis L, Han Q (2007) A survey of fault management in wireless sensor networks. Journal of Network and systems management

[44] Ranjan A, Venkataramani S, Fong X, et al (2015) Approximate Storage for Energy Efficient Spintronic Memories. In: DAC, 2015

[45] Recht B, Re C, Wright S, et al (2011) Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In: Advances in neural information processing systems

[46] Ringenburg M, Sampson A, Ackerman I, et al (2015) Monitoring and debugging the quality of results in approximate programs. ASPLOS

[47] Rubio-González C, Nguyen C, Nguyen H, et al (2013) Precimonious: Tuning assistant for floating-point precision. SC

[48] Samadi M, Jamshidi DA, Lee J, et al (2014) Paraprox: Pattern-based Approximation for Data Parallel Applications. In: ASPLOS

[49] Sampson A, Dietl W, Fortuna E, et al (2011) EnerJ: Approximate data types for safe and general low-power computation. In: PLDI

[50] Sampson A, Panchekha P, Mytkowicz T, et al (2014) Expressing and verifying probabilistic assertions. PLDI

[51] Sampson A, Baixo A, Ransford B, et al (2015) Accept: A programmer-guided compiler framework for practical approximate computing. Tech. rep.

[52] Sanchez-Stern A, Panchekha P, Lerner S, et al (2018) Finding root causes of floating point error. In: PLDI

[53] Sen K, Viswanathan M, Agha G (2004) Statistical model checking of black-box probabilistic systems. In: CAV

[54] Sidiroglou S, Misailovic S, Hoffmann H, et al (2011) Managing Performance vs. Accuracy Trade-offs With Loop Perforation. FSE

[55] Stanley-Marbell P, Alaghi A, Carbin M, et al (2020) Exploiting Errors for Efficiency: A Survey from Circuits to Applications. ACM Computing Surveys Journal,

[56] Stevens JR, Ranjan A, Raghunathan A (2018) AxBA: an approximate bus architecture framework. In: ICCAD

[57] TensorFlow Developers (2021) Tensorflow. https://doi.org/10.5281/zenodo.5159865, URL https://www.tensorflow.org

[58] Tsoutsouras V, Kaparounakis O, Bilgin B, et al (2021) The laplace microarchitecture for tracking data uncertainty and its implementation in a risc-v processor. In: MICRO, MICRO '21

[59] Yazdanbakhsh A, Mahajan D, Esmaeilzadeh H, et al (2017) AxBench: A Multiplatform Benchmark Suite for Approximate Computing. IEEE Design Test 34(2)

[60] Zhuang W, Chen X, Tan J, et al (2009) An empirical analysis for evaluating the link quality of robotic sensor networks. In: WCSP