Appendix: Diamont: Dynamic Monitoring of Uncertainty for Distributed Asynchronous Programs

In this appendix we present the following:

- Appendix A - Syntax and Semantics of Diamont.
- Appendix B - Type system for Diamont and additional semantic rules.
- Appendix C - Rewrite rules and proof of soundness for the rewrite system.
- Appendix D - Precise definition of reliability. Proofs for the soundness properties stated in the paper.
- Appendix E - Additional details of our evaluation.
- Appendix F - Simplified sequentialized version of the code in the example section.

Appendix A

A.1 SYNTAX

n	$\in$	$\mathbb{N}$	*quantities*
m, v	$\in$	$\mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$	*values*
d	$\in$	$\mathbb{R}$	*reals*
r	$\in$	$[0, 1.0]$	*probability*
x, b, X	$\in$	Var	*variables*
a	$\in$	ArrVar	*array variables*
α, β	$\in$	Pid	*process ids*
Exp	$\rightarrow$	$m \mid \langle\, m, v \,\rangle \mid x$	*expressions*
		$\mid (Exp) \mid Exp\ op\ Exp$	
AExp	$\rightarrow$	$d \mid d \cdot x \mid d \cdot a[Exp^+]$	*affine*
		$\mid AExp + AExp$	*expressions*
q	$\rightarrow$	`precise` | `approx` | `dynamic`	*type qualifiers*
t	$\rightarrow$	`int<n>` | `float<n>`	*basic types*
T	$\rightarrow$	$q\ t \mid q\ t\ [] \mid$ struct T^+	*types*
Dec	$\rightarrow$	T x | T a$[n^+]$ | *Dec*; *Dec*	*declarations*
P	$\rightarrow$	$[Dec\,;\ S]_\alpha$	*process*
		$\mid \Pi.\alpha : X\ [Dec\,;\ S]_\alpha$	*process group*
		$\mid P \parallel P$	*parallel comp*

S	$\rightarrow$	`skip`	*empty program*
		$\mid x$ = *Exp*	*assignment*
		$\mid$ send(α, T, x)	*send message*
		$\mid x$ = receive(α, T)	*receive a message*
		$\mid x = a[Exp^+]$	*array load*
		$\mid a[Exp^+] = Exp$	*array store*
		$\mid$ `for` x : [Pid$^+$]{S}	*iterate over processes*
		$\mid$ `if` $x\ S\ S$	*branching*
		$\mid x = Exp?\ Exp : Exp$	*conditional choice*
		$\mid$ while b {S}	*while loop*
		$\mid x$ = (T)*Exp*	*cast*
		$\mid S\,;S$	*sequence*
		$\mid$ cond-send(x, α, T)	*send that can fail*
		$\mid b, x$ = cond-receive(α, T)	*receive a cond-send*
		$\mid x = Exp\ [r]\ Exp$	*probabilistic choice*
		$\mid$ dyn-send(α, T, x)	*send dynamic*
		$\mid x$ = dyn-recv(α, T)	*receive a dyn-send*
		$\mid x$ = rdDyn(y)	*read dynamic property*
		$\mid x$ = endorse(y)	*cast to precise*
		$\mid x$ = track(y, $\langle$d, r$\rangle^+$)	*initiate monitoring*
		$\mid$ Check	*check specification*
Check	$\rightarrow$	check(*AExp*, $\langle$d, r$\rangle^+$)	*check dynamic property*
		$\mid$ checkArr(a, $\langle$d, r$\rangle^+$)	*check array*

Fig. 1. Diamont Syntax

A.2 SEMANTICS OF THE DIAMONT INTERMEDIATE REPRESENTATION

A.2.1 Basic Definitions and Semantics

References, Frames, Stacks, and Heaps. A *reference* is a pair $\langle n_b, \langle n_1, ..., n_k \rangle \rangle \in \texttt{Ref}$ that contains a base address $n_b \in Loc$ and dimension descriptor $\langle n_1, ..., n_k \rangle$ denoting the location and dimension of variables in the heap. A *frame* $\sigma \in \mathrm{E} = \mathrm{Var} \rightarrow \mathrm{Ref}$ maps program variables to references. A *heap* $h \in H = \mathbb{N} \rightarrow \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ is a finite map from addresses to values. Values can be an Integer, Float or the special *empty message* ($\varnothing$).

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

Dynamic Property Map. A *dynamic property map* D is an element of $Dyn = \mathrm{Var} \times \mathbb{N} \rightarrow \mathbb{R} \times [0, 1.0]$. $D[\langle x,\ i \rangle]$ refers to the i^{th} element of an array `x`. For regular variables we use $D[x]$ instead of $D[\langle x,\ 1 \rangle]$. This map maintains a maximum absolute error (ϵ), and a probability/confidence (δ) that the true error is below ϵ for each dynamic typed variable (or element of an array). We use $D[x].\epsilon$ to refer to the first element in the pair (maximum error) and $D[x].\delta$ for the second element (probability).

Typed Channels and Message Orders. The global channel $\mu \in Channel = Pid \times Pid \times Type \rightarrow Val^*$ consists of a set of sub-channels for each pair of processes, further split by the type of the message. Processes communicate by sending and receiving messages over these typed channels. Messages on the same sub-channel are delivered in order but there are no guarantees for messages sent on separate (sub)channels. For each type t, we define a special sub-channel ($\mu[\langle \alpha, \beta, D_t \rangle]$) for transmitting dynamically monitored information for variables of type t between a pair of processes. Messages sent in this channel are sent as `precise` and should not be corrupted or dropped.

Global and Local Environments. Each process maintains its own private environment consisting of a frame, and a heap $\langle \sigma^i, h^i \rangle \in \Lambda = \{H \times \mathrm{E}\} \cup \perp$, where $\perp$ is considered to be an error state. Individual processes may only access their own private environments. We define a global configuration as $\langle \epsilon, \mu, \omega, P \rangle$, consisting of a global environment, channel, global dynamic property map, and global program. The global environment is a map from Pids to the local environment $\epsilon \in Env = Pid \mapsto \Lambda$. The global dynamic map ω is a map from the Pids to the local dynamic map $\omega \in Pid \mapsto Dyn$.

Uncertainty in programs. To define randomness in programs executions we define an *uncertainty model* $\psi \in (\mathrm{Pid} \times \mathrm{Pid} \times T \rightarrow \mathbb{R}) \times (x \rightarrow \mathbb{R})$. ψ maps each probabilistic choice statement and each channel to a probability of error. We also use this model to define functional specifications. We define a special uncertainty model 1_ψ, which models the evaluation of the program in a deterministic setting without any randomness through errors, etc. Under exact execution, probabilistic choice statements always evaluate to the first option and casting performs no change.

Sensors. We consider that a precise execution of the program reads from an oracle sensor. We model successive readings from this sensor as a tape of perfect sensor measurements predetermined at the start of the program. Under 1_ψ, the program will use these ground truth measurements. We model noisy sensors as reading perturbed values from this tape using probabilistic choice statements. Several probabilistic choice statements could be combined to define various error models (`temp = readSensor()[r1] (readSensor()+noise [r2] ...);`). This interpretation allows us to consider the relative difference in an execution from a error free run. We assume a finite level of precision in the sensors to enable us to define a finite set of environments that can deviate from the original.

A.2.2 Review of Semantics of Basic Statements

We now review the semantics of statements (extends those of Parallely [3])

Expressions. We use a big-step evaluation relation of the form $\boxed{\langle e, \sigma, h\rangle \Downarrow v}$. In Diamont, an expression e evaluates deterministically in a single step in a frame σ, a heap h to a value v without any changes to the environment. Diamont supports typical integer and floating point operations. We allow calls to functions that do not perform communication and inline them as a preliminary step. $\rho(e)$ returns the list of variables used in an expression e. The semantics are available in Figure 2.

$$\text{E-Const}\ \frac{}{\langle m, \sigma, h\rangle \Downarrow m} \qquad \text{E-Var}\ \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x)}{\langle x, \sigma, h\rangle \Downarrow h(n_b)} \qquad \text{E-Iop}\ \frac{\langle e_1, \sigma, h\rangle \Downarrow v_1 \qquad \langle e_2, \sigma, h\rangle \Downarrow v_2}{\langle e_1\ op\ e_2, \sigma, h\rangle \Downarrow v_1\ op\ v_2}$$

Fig. 2. Big-step Semantics of Expressions

Statements. The small-step relation of the form $\boxed{\langle s,\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle s',\ \langle \sigma', h'\rangle,\ \mu',\ D'\rangle}$ defines a process in the program evaluating in its local frame σ, heap h, dynamic property map D, and the global channel μ. Figure 3 defines the semantics for basic statements. Expression assignment updates the heap with the result of evaluating the expression. The send rules asynchronously adds a new message to the end of the communication queue between the processes. The receive rules blocks until it can remove a message from the head of the communication queue (`v::q` denote accessing a queue's head `v`, and `q++v` denotes appending `v` to the queue).

A.2.3 Global Semantics

Small step transitions of the following form define the parallel program taking a step due a single process α taking a local step with probability p.

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

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

A.2.4 Semantics of Dynamic monitoring

Figure 4 presents the semantics of statements that perform dynamic monitoring. These statements update the environment for `dynamic` typed variables and also update their corresponding value in the Dynamic property map. Statements that update dynamically monitored variables can only use variables of type `dynamic` and constants. This property is enforced by the type checker.

Dynamic monitoring variants of each of the regular statements have the same functionality along with a function that can calculate the uncertainty interval for the updated variables. Based

$$\textsc{S-Assign-Static}\quad \frac{\langle e,\sigma,h\rangle \Downarrow v \qquad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h' = h[n_b \mapsto v]}{\langle [x = \mathrm{e}]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h'\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-Send}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \quad \mu[\langle\alpha,\beta,t\rangle] = m \quad h[n_b] = v \quad \mu' = \mu[\langle\alpha,\beta,t\rangle \mapsto m + +v]}{\langle [\mathrm{send}(\beta,t,y)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-Receive}\quad \frac{\mu[\langle\beta,\alpha,t\rangle] = v :: m \quad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \quad h' = h[n_b \mapsto v] \quad \mu' = \mu[\langle\beta,\alpha,t\rangle \mapsto m]}{\langle [\mathrm{x} = \mathrm{receive}(\beta,t)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h'\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-CondSend-True}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = v \qquad \mu[\langle\alpha,\beta,t\rangle] = m \qquad \mu' = \mu[\langle\alpha,\beta,t\rangle \mapsto m + +v]}{\langle [\text{cond-send}(\beta,t,y)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{\psi(\alpha,\beta,t)}_\psi \langle \mathtt{skip},\ \langle\sigma,h\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-CondSend-False}\quad \frac{\mu[\langle\alpha,\beta,t\rangle] = m \qquad \mu' = \mu[\langle\alpha,\beta,t\rangle \mapsto m + +\varnothing]}{\langle [\text{cond-send}(\beta,t,y)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1-\psi(\alpha,\beta,t)}_\psi \langle \mathtt{skip},\ \langle\sigma,h\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-CondReceive-True}\quad \frac{\mu[\langle\beta,\alpha,t\rangle] = v :: m \qquad v \neq \varnothing \qquad \langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h' = h[n_b \mapsto v] \qquad \mu' = \mu[\langle\beta,\alpha,t\rangle \mapsto m]}{\langle [x = \text{cond-receive}(\beta,t)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h'\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-CondReceive-False}\quad \frac{\mu[\langle\beta,\alpha,t\rangle] = \varnothing :: m \qquad \mu' = \mu[\langle\beta,\alpha,t\rangle \mapsto m]}{\langle [x = \text{cond-receive}(\beta,t)]_\alpha,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h\rangle,\ \mu',\ D\rangle}$$

$$\textsc{S-Seq-R1}\quad \frac{\langle s_1,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{p}_\psi \langle s_1',\ \langle\sigma',h'\rangle,\ \mu',\ D'\rangle}{\langle s_1\,;s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{p}_\psi \langle s_1';s_2,\ \langle\sigma',h'\rangle,\ \mu',\ D'\rangle}$$

$$\textsc{S-Seq-R2}\quad \frac{}{\langle \mathtt{skip};s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle}$$

$$\textsc{S-repeat-C}\quad \frac{N \neq 0}{\langle \text{repeat N } \{S\},\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \text{S;repeat N-1 } \{S\},\ \langle\sigma,h\rangle,\ \mu,\ D\rangle}$$

$$\textsc{S-repeat-S}\quad \frac{N = 0}{\langle \text{repeat N } \{S\},\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle \mathtt{skip},\ \langle\sigma,h'\rangle,\ \mu,\ D\rangle}$$

$$\textsc{S-Par-Iter}\quad \frac{Q = \{\alpha_1,\ldots,\alpha_k\}}{\langle \mathtt{for}\ x : Q\ \{S\},\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle [S[\alpha_1/x]]_{\alpha_1} \parallel \ldots \parallel [S[\alpha_k/x]]_{\alpha_k},\ \langle\sigma,h\rangle,\ \mu,\ D\rangle}$$

$$\textsc{S-If-True}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] \neq 0}{\langle \mathtt{if}\ x\ s_1\ s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle s_1,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle}$$

$$\textsc{S-If-False}\quad \frac{\langle n_b,\ \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] = 0}{\langle \mathtt{if}\ x\ s_1\ s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_\psi \langle s_2,\ \langle\sigma,h\rangle,\ \mu,\ D\rangle}$$

Fig. 3. Operational Semantics of Statements

on the definition of these functions, dynamic monitoring instrumentation code is generated by the compiler.

Diamont associates each `dynamic` typed variable with a maximum error ϵ and a confidence δ. If a variable's stored value is v, then the true value should lie in the range $v \pm \epsilon$ with probability δ.

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

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

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

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

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

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

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

Fig. 4. Semantics of Dynamic Monitoring

Communication. During communication that uses the `dyn-send` statements, the dynamically monitored property value is communicated using the relevant dynamic channel ($\mu[\langle\alpha,\beta,D_t\rangle]$) in addition to the message itself through code generated by the compiler.

Conditionals. Diamont has the conditional choice statement x = *bExp*? $Exp_1 : Exp_2$ that can use dynamically monitored variables inside the *bExp*. In these situations the maximum error and error confidence of the assigned variable x need to be calculated with care based on the uncertainty interval. If the entire interval satisfy a condition we can confidently calculate the maximum error form the resultant expression. If not we must assume the worst case whereby we could have chosen the wrong branch, thus the maximum error depends on the difference between Exp_1 and Exp_2. Figure 5 defines the precise semantics for conditionals.

S-Assign-Cond-Dyn-True

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

S-Assign-Cond-Dyn-False

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

S-Assign-Cond-Dyn-unknown-True

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

S-Assign-Cond-Dyn-unknown-False

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

Fig. 5. Semantics of Dynamic Conditionals

Arrays. To increase the precision of array analysis, when `dynamic` type arrays are declared, we allocate an entry in D for each array element. When array elements are updated, we also update their corresponding dynamically monitored value. The array semantics are available in Figure 6.

Dec-Array

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

S-Array-Load

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

S-Array-Store

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

Fig. 6. Semantics of Arrays

Checks and Type conversions. Figure 7 shows the semantics of dynamic checkers (`check`, and `check-array`) from Section 3. Semantics of type conversion (`track`, `endorse`) are in Figure 8.

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

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

$$\textsc{S-array-check-true}\quad \frac{\forall i.\ D[y, i].\epsilon \le d \wedge D[y, i].\delta \ge r}{\langle \text{checkArr(y, p)},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{1}_{\psi} \langle \texttt{skip},\ \langle \sigma, h\rangle,\ \mu,\ D\rangle}$$

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

Fig. 7. Semantics of Checks

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

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

Fig. 8. Semantics of type conversion

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

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

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

Fig. 9. Semantics of statically verified functions

Appendix B

B.1 TYPE SYSTEM

Diamont's type system uses similar type annotations as in Parallely. The type qualifiers explicitly specify data that may be subject to approximations and data that needs dynamic monitoring. We use a static type environment $\Theta : \text{Var} \mapsto T$ that maps variables to their type to check type-safety of statements. We use two type judgments, 1) expressions are assigned a type: $\boxed{\Theta \vdash e : T}$, 2) statements update the type environment: $\boxed{\Theta \vdash S : \Theta'}$.
The type system enforces the following properties.

- approx data is not allowed to flow into `precise` data.
- dynamic expression can only use `dynamic` data.
- Data can only be converted to `dynamic` through the relevant type conversion statements.
- Only `precise` data can influence the calculation of conditionals, or array indexes
- `approx` variables can appear in conditional choice statements

$$\text{TR-Val}\ \frac{\text{type(n)} = t}{\Theta \vdash n : \text{precise } t} \qquad \text{TR-Var}\ \frac{\Theta(x) = T}{\Theta \vdash x : T} \qquad \text{TR-Iop}\ \frac{\Theta \vdash e_1 : T \qquad \Theta \vdash e_2 : T}{\Theta \vdash e_1\ op\ e_2 : T}$$

$$\text{TR-Iop-approx1}\ \frac{\Theta \vdash e_1 : \text{approx } t \qquad \Theta \vdash e_2 : \text{precise } t}{\Theta \vdash e_1\ op\ e_2 : \text{approx } t} \qquad \text{TR-Iop-approx2}\ \frac{\Theta \vdash e_1 : \text{precise } t \qquad \Theta \vdash e_2 : \text{approx } t}{\Theta \vdash e_1\ op\ e_2 : \text{approx } t}$$

Fig. 10. Types for Integer Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 11. Types for Statements

$$\text{TR-Track}\ \frac{\Theta \vdash x : \text{dynamic}\ t \qquad \Theta \vdash y : \text{approx}\ t \vee \Theta \vdash y : \text{precise}\ t}{\Theta \vdash \text{x = track(y, p)} : \Theta}$$

$$\text{TR-Check}\ \frac{\Theta \vdash y : \text{dynamic}\ t}{\Theta \vdash \text{check(y, p)} : \Theta}$$

$$\text{TR-CheckArray}\ \frac{\Theta \vdash y : \text{dynamic}\ t[\ ]}{\Theta \vdash \text{checkArr(y, p)} : \Theta, \text{dynamic}}$$

$$\text{TR-Prob-D}\ \frac{\Theta \vdash e_1 : \text{dynamic}\ t \qquad \Theta \vdash e_2 : q\ t \qquad \Theta \vdash x : \text{dynamic}\ t}{\Theta \vdash \text{x} = e_1\ [p]\ e_2 : \Theta}$$

$$\text{TR-COND-D}\ \frac{\Theta \vdash e_1 : \text{dynamic}\ t \qquad \Theta \vdash e_2 : \text{dynamic}\ t \qquad \Theta \vdash x : \text{dynamic}\ t \qquad \Theta \vdash x : \text{dynamic}\ t}{\Theta \vdash \text{x} = b?\ e_1 : e_2 : \Theta}$$

$$\text{TR-DynSend}\ \frac{\Theta \vdash y : \text{dynamic}\ t \qquad T = \text{dynamic}\ t}{\Theta \vdash \text{dyn-send}(q, T, y) : \Theta}$$

$$\text{TR-DynReceive}\ \frac{\Theta \vdash x : \text{dynamic}\ t \qquad T = \text{dynamic}\ t \qquad \Theta \vdash b : \text{dynamic int}}{\Theta \vdash b, x = \text{dyn-recv}(q, T) : \Theta}$$

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

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

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

Fig. 12. Types for Statements (dynamic)

Appendix C

C.1 REWRITE RULES

We present the new rewrite rules for the Diamont language. The remaining rewrite rules are from Parallely [3].

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

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

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

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

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

Fig. 13. Rewrite Rules

C.2 REWRITE RULE SOUNDNESS

Lemma 1. *If* $\Gamma, \mathcal{S}, P \rightsquigarrow \Gamma', \mathcal{S}; \mathcal{S}', P'$ *then* $\Gamma, \mathcal{S}, P \sqsubseteq \Gamma', \mathcal{S}; \mathcal{S}', P'$

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

Case R-DynSend:

Let $(\epsilon, \mu, \omega) \in [\![\mathcal{S}, \Gamma]\!]_\varnothing$ and assume

$$(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

dyn-send is a left mover (proof is same as the proof that cond-send is a left mover), therefore

$$(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

Suppose $\epsilon(x) = \beta$. By the R-DynSend rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +y]$ and $\mathcal{S}' = \texttt{skip}$. Suppose $(\epsilon', \mu', \omega') \in [\![\mathcal{S}; \mathcal{S}', \Gamma']\!]_\varnothing$. Then $\epsilon' = \epsilon$, $\omega' = \omega$, and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d]$, where $d = \omega(y)$ and m is either $\epsilon(y)$ or $\varnothing$.

Suppose the send succeeds. Then by semantic rule E-DynSend-True and E-CondSend-True,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\epsilon(y)][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d], \omega, P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

Suppose the send fails. Then by semantic rule E-DynSend-False and E-CondSend-False,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\varnothing][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d], \omega, P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

that is,

$$(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

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

Case R-CondSend:

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

Case R-DynReceive:

Let $(\epsilon, \mu, \omega) \in [\![\mathcal{S}, \Gamma]\!]_\varnothing$ and assume

$$(\epsilon, \mu, \omega, [y = \text{dyn-recv}(x, t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

dyn-receive is a left mover (proof is same as the proof that cond-receive is a left mover), therefore

$$(\epsilon, \mu, \omega, [y = \text{dyn-recv}(x, t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

Suppose $\epsilon(x) = \alpha$. By the R-DynReceive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\mathcal{S}' = [\beta.y = \alpha.y\ [\psi(\alpha, \beta, t)]\ \beta.y]$ when $\text{head}(\Gamma(\alpha, \beta, t)) = y$. Suppose $(\epsilon', \mu', \omega') \in [\![\mathcal{S}; \mathcal{S}', \Gamma']\!]_\varnothing$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))]$ and $\omega' = \omega[\beta.y \mapsto d]$ where $d = \texttt{get-dyn-rec}(\text{head}(\mu(\alpha, \beta, D_t)), \psi(\alpha, \beta, t))$. Further, either $\epsilon' = \epsilon[\beta.y \mapsto \alpha.y]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.y$ or $\epsilon' = \epsilon$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose the send succeeded. Then by semantic rule E-DynReceive-True and E-CondReceive-True,

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

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

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

that is,

$$(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

Therefore, $(\epsilon, \mu, \omega, [y = \text{dyn-recv}(x, t)]_\beta \ltimes P_x) \sqsubseteq (\epsilon', \mu', \omega', P_x)$.

Case R-CondReceive:

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

Case R-CommonRepeat:
Since the rewrite rule ensures that N and M are the same, this proof refers to both as N. The proof for this case is by induction on N, the number of repetitions.
Suppose $N = 1$. Then,

$$[\text{repeat N } S_0]_\alpha \parallel [\text{repeat N } S_1]_\beta \equiv [S_0]_\alpha \parallel [S_1]_\beta$$

which can be rewritten to $\mathcal{S}'$, which is equivalent to repeat N $\{\mathcal{S}'\}$.
Suppose $N > 1$. Then,

$$[\text{repeat N } S_0]_\alpha \parallel [\text{repeat N } S_1]_\beta \equiv [S_0; \text{repeat N-1 } S_0]_\alpha \parallel [S_1; \text{repeat N-1 } S_1]_\beta$$

by inductive hypothesis, this can be rewritten to $\mathcal{S}'$; repeat N-1 $\{\mathcal{S}'\}$, which is equivalent to repeat N $\{\mathcal{S}'\}$.
Transitions to Error States:
Some statements, such as function calls and check statements, can make the parallel program transition to an error state. This happens if 1) the input to a function does not satisfy its function specification, 2) a check fails for a variable or a checkarray fails for an element of an array. In such cases, using the inductive hypothesis, we can say that if such a failure occurs in the parallel program, it will also occur in the sequential program.

Lemma 2. *If* $\Gamma, \mathcal{S}, P \rightsquigarrow \Gamma', \mathcal{S}; \mathcal{S}', P'$ *then* $\Gamma, \mathcal{S}, P \sqsupseteq \Gamma', \mathcal{S}; \mathcal{S}', P'$

Proof: The proof is by induction on the derivation of $\Gamma, \mathcal{S}, P \rightsquigarrow \Gamma', \mathcal{S}; \mathcal{S}', P'$. Each rewrite rule has a seperate case. Below are the cases for the new rewrite rules:
Case R-DynSend:
Let $(\epsilon', \mu', \omega') \in [\![\mathcal{S}; \mathcal{S}', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

By the R-DynSend rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +y]$ and $\mathcal{S}' = \texttt{skip}$. Suppose $(\epsilon, \mu, \omega) \in [\![\mathcal{S}, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$, $\omega' = \omega$, and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d]$, where $d = \omega(y)$ and m is either $\epsilon(y)$ or $\varnothing$. Therefore,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m][(\alpha, \beta, D_t) \mapsto \mu(\alpha, \beta, D_t) + +d], \omega, P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

by semantic rule E-DynSend-True and E-CondSend-True or E-DynSend-False and E-CondSend-False (depending on m),

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

Therefore,

$$(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

since dynsend is a left mover,

$$(\epsilon, \mu, \omega, [\text{dyn-send}(x, t, y)]_\alpha \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

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

Case R-DynReceive:
Let $(\epsilon', \mu', \omega') \in [\![\mathcal{S}; \mathcal{S}', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', \omega', P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

By the R-DynReceive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\mathcal{S}' = [\beta.y = \alpha.y\ [\psi(\alpha, \beta, t)]\ \beta.y]$ when $\text{head}(\Gamma(\alpha, \beta, t)) = y$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))][(\alpha, \beta, D_t) \mapsto \text{pop}(\mu(\alpha, \beta, D_t))]$ and $\omega' = \omega[\beta.y \mapsto d]$ where $d = \texttt{get-dyn-rec}(\text{head}(\mu(\alpha, \beta, D_t)), \psi(\alpha, \beta, t))$. Further, either $\epsilon' = \epsilon[\beta.y \mapsto \alpha.y]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.y$ or $\epsilon' = \epsilon$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose the send succeeded.

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

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

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

Suppose the send failed.

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

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

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

therefore,

$$(\epsilon, \mu, \omega, [y = \text{dyn-recv}(x, t)]_\beta; P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

since dynreceive is a left mover,

$$(\epsilon, \mu, \omega, [y = \text{dyn-recv}(x, t)]_\beta \ltimes P_x) \rightarrow^* (\epsilon_f, \mu_f, \omega_f, H)$$

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

Case R-CommonRepeat:
This proof is similar to the R-CommonRepeat proof for Lemma 1.

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

Corollary 1 (Deadlock-Freedom of Sequentializable Programs). *If a parallel program P can be sequentialized to $\mathcal{S}$, then P is deadlock free.*

Corollary 1 is proved in [1].

Appendix D

D.1 SOUNDNESS OF THE RUNTIME

We first define basic definitions that give a semantic meaning to the uncertainty intervals. We extend the notion from Rely [2] in this semantics for intermediate states of execution of statements.

D.1.1 Big Step Notation.

Definition 1 (Partial Trace Semantics for Parallel Programs).

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

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

Definition 2 (Aggregate Semantics for Parallel Programs).

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

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. For reliability and accuracy analysis we define a *paired execution semantics* that pairs an original (exact) execution of a program with an approximate execution, expanding the definition from Rely.

Definition 3 (Paired Execution Semantics).

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

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

We use the syntax from [5] for accuracy predicates. **Extended Reliability Predicates.** A predi-

cate Q has the following form:

$$Q_A := d \geq A_e \mid Q_A \wedge Q_A \mid \text{True} \mid \text{False}$$
$$A_e := d \mid d \cdot \Delta(o) \mid A_e + A_e$$

An accuracy predicate Q_A is a conjunct of accuracy predicates or a comparison between a numerical constant and a Accuracy expression A_e. An accuracy expression can be a constant term d $\in \mathbb{R}$, or a product of a constant an a *distance*. A distance operator $\Delta(o)$ relates the values of operand o in an exact and approximate run.

Accuracy of variables. We now define the semantics of accuracy by following the exposition by [5]. The accuracy of a variables is defined using the *environment distributions*. The denotation of a accuracy predicate $\boxed{[\![Q_A]\!] \in \mathcal{P}(Env \times \Phi)}$ is the set of environment and environment distribution pairs that satisfy the predicate.

The denotation of $\mathcal{R}^*[\, Q_A \,]$ is the probability that an environment ϵ_a sampled from Φ satisfied Q_A:

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

where, $\mathcal{E}(O, \epsilon)$ is the set of all environments in which Q_A is satisfied.

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

Precondition Generator for static analysis.

$$C(\, x = e,\, Q\,) = \text{let } Q'_A = Q_A[AE(e)/\Delta(x)] \text{ in } Q\ [\mathcal{R}^*[\, Q'_a \,]/\mathcal{R}^*[\, Q_a \,]]$$
$$C(\, x = e_1\ [r]\ e_2,\, Q\,) = \text{let } Q'_A = Q_A[AE(e_1)/\Delta(y)] \text{ in } Q\ [r * \mathcal{R}^*[\, Q'_a \,]/\mathcal{R}^*[\, Q_a \,]]$$

D.2 SOUNDNESS.

Lemma 3 (Probability calculation for expressions). *For any operation $\oplus \in \{+, -, *, /\}$ assuming maximum error of any expression AE(x $\oplus$ y) is correctly calculated using maximum error of x and y x.ϵ and y.ϵ,*

$[\![\mathcal{R}^*[\, \Delta(x \oplus y) \leq \mathtt{AE}(x \oplus y)\,]]\!](\epsilon, \varphi) \;\geq\; [\![\mathcal{R}^*[\, \Delta(x) > x.\epsilon\,]]\!](\epsilon, \varphi) + [\![\mathcal{R}^*[\, \Delta(y) > y.\epsilon\,]]\!](\epsilon, \varphi) - 1$

Proof. By definition, $[\![\mathcal{R}^*[\, \Delta(x \oplus y) \leq \mathtt{AE}(x \oplus y)\,]]\!](\epsilon, \varphi) = \sum_{\epsilon_u \in \mathcal{E}(AE, \epsilon)} \varphi(\epsilon_u)$

Since we assume the maximum error $\mathtt{AE}(x \oplus y)$ is calculated correctly based on the maximum error of x and y, For the error of the result to exceed $\mathtt{AE}(x \oplus y)$, either x or y must have had an error bigger than x.ϵ or y.ϵ. The probability of that occurring is

$P(x.\epsilon > \Delta(x)) = 1 - [\![\mathcal{R}^*[\, \Delta(x) > x.\epsilon\,]]\!](\epsilon, \varphi)$ and $P(y.\epsilon > \Delta(y)) = 1 - [\![\mathcal{R}^*[\, \Delta(y) > y.\epsilon\,]]\!](\epsilon, \varphi)$.

Therefore, $1 - [\![\mathcal{R}^*[\, \Delta(x \oplus y)\,] \leq \mathtt{AE}(x \oplus y)]\!] = P(x.\epsilon > \Delta(x) \cup y.\epsilon > \Delta(y))$

From the union bound, $1 - [\![\mathcal{R}^*[\, \Delta(x \oplus y)\,] \leq \mathtt{AE}(x \oplus y)]\!] \;\leq\; P(x.\epsilon > \Delta(x)) \;+\; P(y.\epsilon > \Delta(y)))$

$$1 - \mathcal{R}^*[\, \Delta(x \oplus y) \leq \mathtt{AE}(x \oplus y)\,] \;\leq\; 1 - [\![\mathcal{R}^*[\, \Delta(x) > x.\epsilon\,]]\!](\epsilon, \varphi) \;+\; 1 - [\![\mathcal{R}^*[\, \Delta(y) > y.\epsilon\,]]\!](\epsilon, \varphi)$$

Therefore,
$[\![\mathcal{R}^*[\, \Delta(x \oplus y) \leq \mathtt{AE}(x \oplus y)\,]]\!](\epsilon, \varphi) \geq [\![\mathcal{R}^*[\, \Delta(x) > x.\epsilon\,]]\!](\epsilon, \varphi) + [\![\mathcal{R}^*[\, \Delta(y) > y.\epsilon\,]]\!](\epsilon, \varphi) - 1$ □

Theorem 1 (Soundness of dynamic monitoring for sequential programs).
For all statements s, and for all variables x s.t. $\Theta \vdash x : \textit{dynamic}\ t$,
$\Theta \vdash s : \Theta'$ *and* $\langle s,\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s',\ \langle \epsilon', \omega',\ \varphi'\rangle\rangle \implies [\![\mathcal{R}^*[\ \omega'[x].\epsilon\ \geq\ \Delta(x)\ ]\ ]\!](\epsilon', \varphi') \geq \omega'[x].\delta$

We will now show here intuition behind the proof of Theorem 2 for the case of expression assignment.

Proof. Induction on k, the length of the trace from s to s'. If k = 0, theorem holds as ω is initialized to be $\langle 0,\ 1\rangle$ for all dynamically monitored variables.

Assume true for n, $\langle s,\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s^n,\ \langle \epsilon^n, \omega^n,\ \varphi^n\rangle\rangle$
and $\forall x, \Theta \vdash x : \text{dynamic}\ t \implies [\![\mathcal{R}^*[\ \omega^n[x].\epsilon \geq \Delta(x)\ ]\!](\epsilon^n, \varphi^n) \geq \omega[x]$

We will now argue over all possible ways of taking the n+1 th step.
$\langle s^n,\ \langle \epsilon^n, \omega^n,\ \varphi^n\rangle\rangle \Downarrow \langle s^{n+1},\ \langle \epsilon^{n+1}, \omega^{n+1},\ \varphi^{n+1}\rangle\rangle$ due to a process taking the step $\langle \epsilon^n, \mu^n, D^n,\ s^n\rangle \xrightarrow{\alpha,p}_{1\psi} \langle \epsilon^{n+1}, \mu^{n+1}, D^{n+1},\ s^{n+1}\rangle$

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

From the definition of this rule we can see that only the variable assigned to 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 typed dynamic,

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

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

In Diamont, assignment is deterministic. Therefore the maximum error that y can accumulate is determined through the errors in the variables used in the expression e. We calculate this based on the `calc-acc(e,D)` function defined in Figure 5. The soundness of the calculation has been proven in prior work [4].

Therefore we assume the error is calculated correctly and look at the probability of the execution ending up at states 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 error. From the inductive hypothesis, we 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 the error of any variable in e exceeding their maximum error. We show the correctness of our calculation in Lemma 3.

Case S-Assign-Prob-Dyn-True, S-Assign-Prob-Dyn-False $\mathrm{S} : y = e_1[r]e_2$

Similar to above, from the definition of semantics we know that only the variable assigned to in the statement changes in the environment. The reliability of all the others remain the same and the property follows from the inductive hypothesis.

If the assigned variable is typed `dynamic`, A `precise` execution result in the variable y having the value of $e1$. Therefore we know that the maximum error y can have in a correct execution is the error from $e1$ 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 $e1$ was executed *and* that the error in $e1$ is within bounds. As these two events

are independent we can multiply the relevant probabilities to calculate the error confidence.

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

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

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

We have shown the soundness of the approach for all sequential programs that can be completely rewritten to a sequential program. We can then extend this proof to hold on parallel programs using Theorem 2, because: 1) checks in Diamont are process local properties (depend only on the local state of a process). 2) If a parallel program can be sequentialized, the analysis of the sequentialized program will also be valid in the parallel program as shown in [3]. Therefore, for parallel programs that have a canonical sequentialization in Diamont, our analysis is sound.

□

Extending to Dynamic Conditionals. Our proof can be extended to the dynamic conditionals as well. We present the intuition here in service of saving space.

Case S-Assign-Cond-Dyn-True, S-Assign-Cond-Dyn-False: Note that the `T-COND-D` rule ensures that every part of the statement has the dynamic type. In addition, this statement cannot lead to any control flow divergence as the updated value id dynamic typed. In both cases we can guarantee that the entire interval satisfies (or fails) the condition. In this situation, the maximum error is guaranteed to be the error from the relevant expression. But our confidence in the relevant errors depend on the confidence in the Boolean guard expression error. As the statement it self cannot add any error we can consider this to be a probabilistic choice bound by the error in the guard.

Case S-Assign-Cond-Dyn-Unknown-True, S-Assign-Cond-Dyn-Unknown-False: This case is similar to the above, but the entirety of the interval does not satisfy the guard. Therefore we over-approximate the error as the maximum possible deviation similar to many static analyses

D.2.1 Array Optimization

Keeping track of the dynamic uncertainty intervals and communicating them across process boundaries can incur significant overheads, especially when communicating large arrays as the interval of each array element needs to be sent along with the array values. One way to reduce this overhead is to communicate one single conservative approximation of the interval of each individual array member. Therefore, we calculate the maximum error interval of the array elements and communicate that single value along with the array members.

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

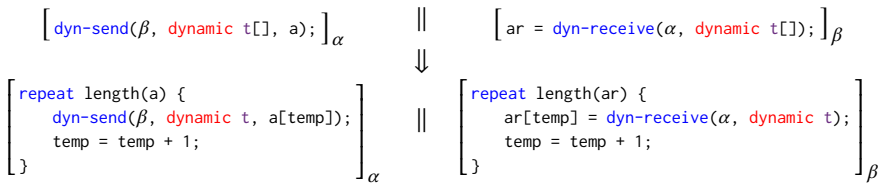

Fig. 14. Array Communication in Diamont.

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

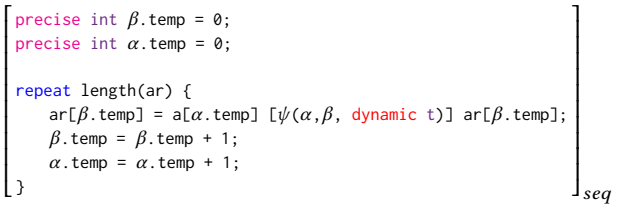

Fig. 15. Sequentialized Form of Array Communication in Diamont.

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

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

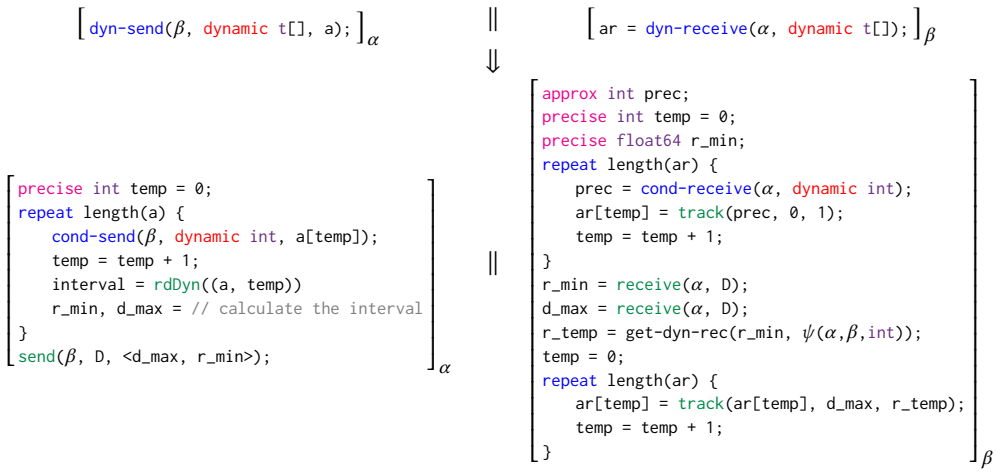

Fig. 16. Optimized Array Communication in Diamont.

This form of communication generates the following sequentialization.

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

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

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

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

seq

Fig. 17. Sequentialized Form of Optimized Array Communication in Diamont.

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

to prove the following proposition that extends the soundness proof from before to work for array assignment. The proposition notes that as each array element is copied over, the resultant array elements have their interval correctly updated in the runtime.

Proposition 1. *If* $\langle s_{array},\ \langle \epsilon, \omega,\ \varphi\rangle\rangle \Downarrow \langle s',\ \langle \epsilon', \omega',\ \varphi'\rangle\rangle$ *then,* $\forall i \in [0..length(a)).$
$[\![\mathcal{R}^*[\ \omega'[ar,i].\epsilon\ \geq\ \Delta(ar[i])\ ]]\!](\epsilon', \varphi') \geq \omega'[ar,i].\delta$

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

$$[\![\omega'[ar,i].\epsilon\ \geq\ \Delta(ar[i])]\!] \subseteq [\![y\ \geq\ \Delta(ar[i])]\!]$$

Therefore,

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

Based on Theorem 2, we will assume that the runtime is sound before reaching this point of the program. Therefore, $\forall i \in [0..\text{length}(a)].\ [\![\mathcal{R}^*[\ \omega[(a,i)].\epsilon \geq \Delta(a[i])\ ]]\!](\epsilon, \varphi) \geq \omega[(a,i)].\delta$
Based on the calculation of α.d_max, we can also show that

$$\forall i \in [0..\text{length}(a)].\ \alpha.\text{d\_max} \geq \omega[(a,i)].\epsilon$$

Therefore as discussed above, $\forall i \in [0..\text{length}(a)]$,

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

We can see that at the end of s_{array}, $\forall i \in [0..\text{length}(a)].\ \ \omega'[ar,i] = \alpha.\texttt{d\_max}$ due to the `track` statements.

By the definition of `get-dyn-rec`, β.d_temp $= \omega[(a,i)] \times \psi(\alpha, \beta, \text{dynamic int})$

We can also prove the correctness of the probability $\omega'[ar,i].\delta$ similarly.

Note that the proof is on the sequentialized version of the code. Due to the equivalence we proved earlier the optimization does not have to be proved correct for parallel programs, which is difficult. □

D.2.2 Early checking Optimization

for a subset of instructions we can perform static analysis to stop runtime monitoring earlier. We perform this by *moving-up* the check to the earliest possible location using a set of rewrites. The following code snippet shows an example of how we perform such rewrites.

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

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

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

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

Such re-writes closely follow the static analysis as defined and proven sound in [3, 4] for the sequential subset of the language.

`error_expression[AE(`e_1`)/x]` expresses substituting `AE(`e_1`)` for every occurrence of x in `error_expression`.

Theorem 2 (Soundness of optimization). *For all statements s their and optimized version* s_{opt},
$\langle s, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{1}_{\psi} \langle s', \bot, \mu', D'\rangle \implies \langle s_{opt}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{1}_{\psi} \langle s'', \bot, \mu'', D''\rangle$

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

For the first rule, the check looks up the dynamic property map for the variable x. Instead, we can move the check before the assignment as we can calculate the relevant error expression without having to evaluate the expression. These rules closely follow the static analysis as defined and proven sound in [3, 4] for the sequential subset of the language.

Based on the definitions in Figure 5, and Figure 7, we can see that s in the first rule goes into an error state due to `calc-eps(`ae, D'`)` evaluating to false. The only difference from D to D' is the execution of the assignment statement.

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

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

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

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

$\left(\sum_{v\in\rho(ae[(\alpha.a+\alpha.b)/\alpha.x])} D[v].\delta\right) - (|\rho(ae[(\alpha.a+\alpha.b)/\alpha.x])| - 1)$

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

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

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

D.2.3 Combining static and runtime verification

Consider the following parallel program S.

$$\left[\begin{array}{l}\texttt{dyn-send}(\beta,\ \texttt{dynamic t},\ \alpha.\texttt{in});\\ \alpha.\texttt{out} = \texttt{dyn-recv}(\beta,\ \texttt{dynamic t});\\ \texttt{check}(\alpha.\texttt{out},\ d_{\text{check}},\ r_{\text{check}});\end{array}\right]_{\alpha} \quad \| \quad \left[\begin{array}{l}\beta.\texttt{dat} = \texttt{dyn-recv}(\alpha,\ \texttt{dynamic t});\\ \texttt{// spec: } \langle \texttt{d} \geq \Delta(\texttt{res}), r \ast \mathcal{R}^{*}[\,(d_i \geq \Delta(\texttt{dat}))\,]\ \rangle\\ \beta.\texttt{res} = \texttt{fn}(\beta.\texttt{dat});\\ \texttt{dyn-send}(\alpha,\ \texttt{dynamic t},\ \beta.\texttt{res});\end{array}\right]_{\beta}$$

Following is the Canonical sequentialization of that program S_{seq}.

$$\left[\begin{array}{l}\beta.\texttt{dat} = \alpha.\texttt{in};\\ \beta.\texttt{res} = \texttt{fn}(\beta.\texttt{dat});\\ \alpha.\texttt{out} = \beta.\texttt{res};\\ \texttt{check}(\alpha.\texttt{out},\ d_{\text{check}},\ r_{\text{check}});\end{array}\right]_{\text{seq}}$$

Let us consider applying the following transformation on the sequentialized program to get the following program S^{opt}_{seq}.

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

Resultant parallel program S^{opt}:

$$\left[\begin{array}{l}\texttt{check}(\alpha.\texttt{in},\ d_i,\ 0);\\ \texttt{send}(\beta,\ \texttt{approx t},\ \alpha.\texttt{in});\\ \alpha.\texttt{tmp} = \texttt{receive}(\beta,\ \texttt{approx t});\\ \alpha.\texttt{out} = \texttt{track}(\alpha.\texttt{tmp},\texttt{d},\texttt{r}\star\texttt{rdDyn}(\alpha.\texttt{in}).\delta);\\ \texttt{check}(\alpha.\texttt{out},\ d_{\text{check}},\ r_{\text{check}});\end{array}\right]_{\alpha} \quad \| \quad \left[\begin{array}{l}\beta.\texttt{dat} = \texttt{receive}(\alpha,\ \texttt{approx t});\\ //\langle \texttt{d} \geq \Delta(\texttt{res}), \texttt{r}\star\mathcal{R}^{*}[\,(d_i \geq \Delta(\texttt{dat}))\,]\ \rangle\\ \beta.\texttt{res} = \texttt{fn}(\beta.\texttt{dat});\\ \texttt{send}(\alpha,\ \texttt{approx t},\ \beta.\texttt{res});\end{array}\right]_{\beta}$$

We will argue that S and S^{opt} are equivalent because S_{seq} and S^{opt}_{seq} are equivalent:

Theorem 3 (Soundness).

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

Proof. As $\emptyset, \emptyset, S \rightsquigarrow^{*} \emptyset, S_{seq}, \texttt{skip}$, From Theorem 2 on the equivalence of sequentialized programs we claim that, $\langle S,\ \langle\sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{*}_{\psi} \langle s',\ \bot,\ \mu',\ D'\rangle \implies \langle S_{seq},\ \langle\sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{*}_{\psi} \langle s',\ \bot,\ \mu',\ D'\rangle$,

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

Case 1: The function specifications requirements are not satisfied. Based on the definition of `S-FUNCTION-FAIL` rule, the failure results from an execution where,
$\langle S_{seq},\ \langle\sigma, h\rangle,\ \mu,\ D\rangle \xrightarrow{*}_{\psi} \langle \beta.\text{output} = \text{func}(\beta.\text{data}); s',\ \langle\sigma', h'\rangle,\ \mu',\ D'\rangle \xrightarrow{*}_{\psi} \langle \texttt{skip}, \bot, \mu', D'\rangle$
Based on the definition of the rule, this implies that $D'[\beta.\text{data}].\epsilon \geq \text{d1}$
we can also see that, $D'[\beta.\text{data}] = D[\alpha.\text{input}]$. Therefore, in a this program, $D[\alpha.\text{input}].\epsilon \geq \text{d1}$.

Now, lets consider S^{opt}_{seq}. Again, based the definition of the runtime, if $D[\alpha.\text{input}].\epsilon \geq \text{d1}$, the check at the start of the program will fail. This would result in a error state (`S-check-fail` rule in the semantics along with the definition of `dyn-check(`α`.input, d1 ,0 ,D)`).

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

$$(D[\beta.\text{output}].\epsilon > d_{\text{check}}) \vee (D[\beta.\text{output}].\delta < r_{\text{check}}) \tag{1}$$

As the function `func` has been verified to guarantee the specification for all inputs, we can use the static analysis to identify maximum error that β.output can take. The rule that apply to functions calls perform the following updates defined in `S-FUNCTION` in Figure 4:

D′[β.output]= $\langle d, r \times \texttt{calc-del}(f(\beta.\text{data}), D)\rangle$ = $\langle d, r \times D'[\beta.\text{data}].\delta\rangle$
and $D'[\beta.\text{data}] = D'[\alpha.\text{input}]$

Therefore, D′[β.output] = $\langle d, r \times D'[\beta.\text{data}].\delta\rangle$ = $\langle d, r \times D'[\alpha.\text{input}].\delta\rangle$
So, based on 1,

$$(d > d_{\text{check}}) \vee (r \times D'[\alpha.\text{input}].\delta < r_{\text{check}}) \tag{2}$$

Let us again consider the execution of $S^{\text{opt}}_{\text{seq}}$, based the definition of the runtime, in Figure 5, we can see that, due to the `track` statement,
D[α.output] = $\langle d_{\text{check}}, r \times D[\alpha.\text{input}].\delta\rangle$
As, (d > d_{check} $\vee r \times D[\alpha.\text{output}].\delta$ < r_{check}), this check will fail, resulting in an error state.

We can see that $\varnothing, \varnothing, S^{\text{opt}} \rightsquigarrow^* \varnothing, S^{\text{opt}}_{\text{seq}}$, `skip`. Therefore from Theorem 2,
$\langle S^{\text{opt}}_{\text{seq}}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \perp, \mu', D'\rangle \implies \langle S^{\text{opt}}, \langle\sigma, h\rangle, \mu, D\rangle \xrightarrow{*}_{\psi} \langle s', \perp, \mu', D'\rangle$,

□

Appendix E

E.1 INPUT SIZES

Table 1 gives the size of the primary inputs we used to evaluate each benchmark and the number of worker threads. Apart from the worker threads, each benchmark also contained one master thread. We used additional input sizes solely to evaluate the effect of optimization on runtime and communication volume.

Table 1. Input Size and Number of Threads Used for Evaluation of Benchmarks

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

For the sensitivity analysis 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 (p2p-Gnutella[09, 25, 30, and 31])

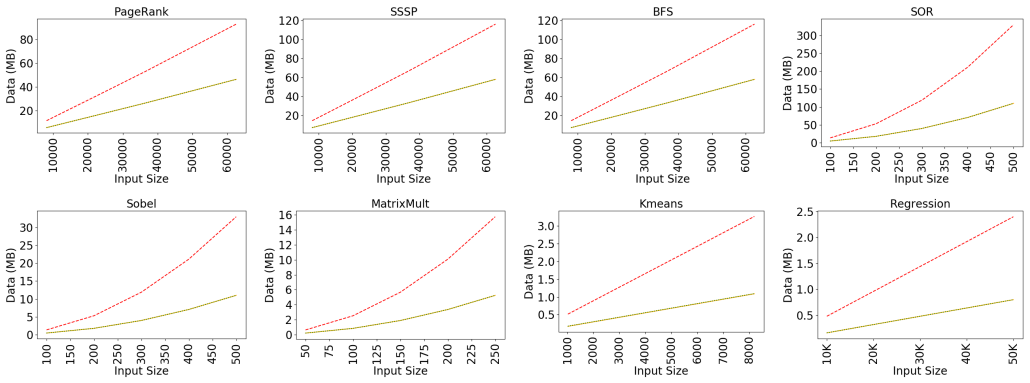

Fig. 18. Effect of input size on communication volume. Dashed red line is the baseline, orange line is Diamont.

Appendix F

```
for IoTDevice in Q {
  IoTDevice.tempVal, IoTDevice.tempErr, IoTDevice.tempConf := readTemperature()
  IoTDevice.humidVal, IoTDevice.humidErr, IoTDevice.humidConf := readHumidity()
  IoTDevice.temperature = track(IoTDevice.tempVal, IoTDevice.tempErr, IoTDevice.tempConf)
  IoTDevice.humidity = track(IoTDevice.humidVal, IoTDevice.humidErr, IoTDevice.humidConf)
  Manager.data[i] = point{IoTDevice.temperature, IoTDevice.humidity}
}
Manager.centers = // randomly pick some nodes
for Worker in R {
    Worker.data = Manager.data
}
for Manager.j:=0; Manager.j<ITERATIONS; Manager.j++ {
    for Worker in R {
        Worker.centers = Manager.centers
    }
    for Worker in R {
        Worker.newcenters = kmeansKernel(Worker.data, Worker.centers, Worker.assign)
        Manager.newcenters[Worker] = Worker.newcenters [reliability] garbage()
    }
    Manager.centers = AverageOverThreads(Manager.newcenters)
}
```

Fig. 19. Simplified sequentialized program for the Smart Agriculture example

REFERENCES

[1] Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *OOPSLA*, 2017.

[2] Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *OOPSLA*, 2013.

[3] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. In *OOPSLA*, 2019.

[4] Saša Misailovic. *Accuracy-Aware Optimization of Approximate Programs.* PhD thesis, Massachusetts Institute of Technology, 2015.

[5] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *OOPSLA*, 2014.