MIT-CSAIL-TR-2014-016 August 19, 2014

Energy-Efficient Approximate Computation in Topaz

Sara Achour and Martin Rinard

Energy-Efficient Approximate Computation in Topaz

Sara Achour Martin Rinard
MIT CSAIL
{sachour,rinard}@csail.mit.edu

Abstract

We present Topaz, a new task-based language for computations that execute on approximate computing platforms that may occasionally produce arbitrarily inaccurate results. The Topaz implementation maps approximate tasks onto the approximate machine and integrates the approximate results into the main computation, deploying a novel outlier detection and reliable reexecution mechanism to prevent unacceptably inaccurate results from corrupting the overall computation. Topaz therefore provides the developers of approximate hardware with substantial freedom in producing designs with little or no precision or accuracy guarantees. Experimental results from our set of benchmark applications demonstrate the effectiveness of Topaz and the Topaz implementation in enabling developers to productively exploit emerging approximate hardware platforms.

1. Introduction

The increasing prominence of energy consumption as a first-order concern in contemporary computing systems has motivated the design of energy-efficient approximate computing platforms [1, 11, 13, 18, 21, 25, 29, 42, 48]. These computing platforms feature energy-efficient computing mechanisms such as components that may occasionally produce incorrect results. For approximate computations that can acceptably tolerate the resulting loss of accuracy, these platforms can provide a compelling alternative to standard precise hardware platforms. Examples of approximate computations that can potentially exploit approximate hardware platforms include many financial, scientific, engineering, multimedia, information retrieval, and data analysis applications [2, 5, 29, 37, 40, 47].

1.1 Topaz

We present a new language, Topaz, for developing approximate software for approximate computing platforms. Topaz supports a task-based model of computation with a main computation (which executes reliably) that generates approximate tasks. Each approximate task is given a set of parameters from the main computation. The approximate tasks execute on approximate computing platforms to produce approximate results, which are then integrated back into the main computation to produce the final result.

1.2 Outlier Detection and Reexecution

Topaz is designed to work with computational platforms that usually produce correct results but occasionally produce arbitrarily inaccurate results. If such inaccurate results are integrated into the main computation, they can easily cause the main computation to produce unacceptably inaccurate results. Topaz therefore deploys an outlier detector that is designed to detect overly inaccurate results that would otherwise unacceptably skew the results that the main computation produces.

When the outlier detector rejects a task, Topaz does not simply discard the task — it instead reexecutes the task on the reliable platform that executes the main computation. It then integrates the correct result from this reliable reexecution into the main computation. This reexecution mechanism has several benefits:

- **Correcting Incorrect Outliers:** When the approximate hardware produces an incorrect outlier, outlier detection plus reexecution ensures that Topaz discards the incorrect result, computes the correct result, and includes that correct result into the main computation.
- **Including Correct Outliers:** When the approximate hardware produces a correct result that the outlier detector rejects, reexecution enables Topaz to recognize that the result is correct (even though it is an outlier) and include that correct result into the main computation.
- **More Conservative Outlier Detection:** Because reexecution enables Topaz to recognize and include correct outlier tasks, Topaz can use a more conservative outlier detector that detects more incorrect tasks (in comparison with an outlier detector that simply discards outlier tasks).

The drawback of reexecution, of course, is the energy required to reexecute the task. Topaz must therefore reexecute few enough tasks to deliver significant energy savings.

1.3 Outlier Detector Adaptive Control

The current Topaz implementation works with a range-based outlier detector that maintains a minimum and maximum value for each component of the results that the tasks produce. The detector accepts results that fall within these values and rejects results that fall outside these values. Larger ranges reexecute fewer correct tasks but accept more incorrect results; smaller ranges accept fewer incorrect results but reexecute more incorrect tasks. The fact that Topaz has no a priori information about the values that correct tasks produce complicates its discovery of an effective outlier range.

Topaz uses an adaptive feedback control algorithm [35] to obtain an effective outlier detector. This algorithm is given a target reexecution rate for correct tasks and dynamically adjusts the minimum and maximum outlier detector values to obtain a detector that delivers this target rate. To obtain a responsive controller that can lock on to the target rate quickly while avoiding phenomema such as overshoot and oscillation, Topaz uses a proportional-integral-derivative (PID) control algorithm [35].

Our current implementation uses a benchmark-specific target task reexecution rate, which is divided equally across the components that a task produces (so if our target reexecution rate is 1% and each task produces three numbers, the target rate for each of these numbers is 0.333 percent). Topaz maintains one outlier detector for each component that the task produces and controls each correct task reexecution rate separately for each outlier detector. Results from our set of benchmark applications indicate that this approach delivers the desirable combination of negligible reexecution energy loss and few accepted incorrect results (and as described further below, even these accepted incorrect results come with probabilistic error bounds).

1.4 Probabilistic Error Bounds

Topaz uses a hardware fault model and task characteristics (such as the number of potentially faulty operations that the tasks perform) to probabilistically bound the number of incorrect tasks in a given taskset. Subtracting off the number of directly observed incorrect tasks (i.e., rejected tasks whose reliable reexecution produces a different correct result) provides a probabilistic bound on the number of accepted incorrect tasks whose results are integrated into the main computation.

With outlier detection and reexecution, any error in the computation is caused only these accepted incorrect tasks (whose results fall within the outlier detector range). It is possible to compute the likelihood that the (not computed and not used) correct results from these accepted incorrect tasks fall inside the minimum and maximum observed results from correct tasks (Topaz observes such minimum and maximum results when it reexecutes correct outlier tasks). For our set of benchmark applications this likelihood is close to certain (because the ratio of correct to incorrect tasks is very high). Topaz therefore combines 1) the probabilistic bound on the number of accepted incorrect tasks and 2) the difference between the minimum and maximum observed correct results to compute, for each taskset, a probabilistic bound on the approximation error caused by accepting incorrect results from incorrect tasks.

For each executed taskset we obtain 1) the percentage of rejected incorrect tasks, 2) a probabilistic bound on the number of accepted incorrect tasks, and 3) a probabilistic bound on the total error integrated into the main computation from the accepted incorrect tasks. These numbers can help a Topaz user better understand the accuracy consequences of the incorrect tasks and the overall acceptability of the resulting approximate computation.

1.5 Evaluation

We evaluate Topaz by developing a set of approximate benchmark programs in Topaz, then executing these programs on a (simulated) approximate computing platform. The approximate computing platform provides two processors, an accurate processor that executes the main computation and an approximate processor that executes the Topaz tasks. The approximate processor features an unreliable but energy-efficient cache for floating point numbers (this memory may, with some probability, deliver an incorrect value for a fetched floating point number) [42].

The two relevant metrics are accuracy and energy consumption. Our results show that Topaz can deliver good energy savings (depending on the benchmark application, from 9 to 13 percent out of a maximum possible savings of 14 percent) and acceptably accurate results. We note that, for all of our benchmark applications, the Topaz outlier detector is required for Topaz for produce acceptably accurate results. See Section 5 for more details.

1.6 Contributions

This paper makes the following contributions:

- **Topaz:** It presents Topaz, a task-based language for approximate computation. Topaz supports a reliable main computation and approximate tasks that can execute on energy-efficient approximate computing platforms. Each Topaz task takes parameters from the main computation, executes its approximate computation, and returns the results to the main computation. If the results are acceptably accurate, the main computation integrates the results into the final results that it produces.
- **Outlier Detection:** Approximate computations must deliver acceptably accurate results. To enable Topaz tasks to execute on approximate platforms that may produce arbitrarily inaccurate results, the Topaz implementation uses an outlier detector to ensure that unacceptably inaccurate results are not integrated into the main computation.

  The outlier detector enables Topaz to work with approximate computation platforms that may occasionally produce arbitrarily inaccurate results. The goal is to give the hardware designer maximum flexibility when designing hardware that trades accuracy for energy.
- **Reliable Reexecution:** Instead of simply discarding outlier tasks, Topaz reliably reexecutes outliers and integrates the resulting correct results into the computation. Reexecution enables Topaz to detect and correct incorrect outliers, include correct outliers into the main computation, and use more conservative outlier detectors that detect and correct more incorrect results.
- **Adaptive Control:** Because Topaz has no a priori information about the values that correct tasks produce, it uses an adaptive outlier detector that, given a target correct task reexecution rate, produces a range that delivers that target rate. To avoid overshooting and oscillation, Topaz deploys a proportional-integral-derivative controller [35] that uses feedback from the observed correct task reexecution rate to appropriately adjust the outlier detector range.
- **Probabilistic Error Bounds:** For each taskset, we probabilistically bound the number of incorrect results integrated into the main computation and the total error associated with these incorrect results. These numbers can help Topaz users better understand the accuracy consequences of the Topaz approximation and the resulting overall acceptability of the Topaz computation.
- **Experimental Evaluation:** We evaluate Topaz with a set of benchmark Topaz programs executing on an approximate computing platform. This platform features a reliable processor and an approximate processor. The approximate processor uses an unreliable floating-point cache to execute floating-point intensive tasks more energy-efficently than the reliable processor.

  Our results show that Topaz enables our benchmark computations to effectively exploit the approximate computing platfom, consume less energy, and produce acceptably accurate results.

2. Example

We next present an example that illustrates the use of Topaz in expressing an approximate computation and how Topaz programs execute on approximate hardware platforms.

2.1 Example Topaz Program

Figure 1 presents an example Topaz program. This program computes the sum from `0` to `n-1` of `f(i,v)`, where `n` and `v` are parameters of the `compute` function. The Topaz `taskset` construct (we have implemented Topaz as an extension to C; the `taskset` construct is the only additional construct) creates `n` tasks, each of which computes one of the values `f(i,v)`.

In general, each Topaz task has a list of `in` parameters. In our example, each task takes a single `in` parameter `d`, which is set to `v` when the task is created. Each task also has a list of `out` parameters. In our example, each task has a single `out` parameter `result`, which is set to the computed value of `f(i,d)`.

When the task finishes, its `combine` block executes to incorporate the results from the `out` parameters into the main Topaz com-

```
double compute(int n, double v) {
  double sum = 0.0;
  taskset add(int i = 0; i < n; i++) {
    compute
        in  (double d = v)
        out (double result)
    {
      result = f(i,d);
    }
    combine
    {
      sum += result;
    }
  }
  return sum;
}
```

Figure 1: Example Topaz Program

putation. In our example the `combine` blocks simply compute the sum of the results from the tasks. The `compute` function returns this sum.

2.2 Approximate Execution

Topaz works with a model of computation with a precise main Topaz computation (which executes fully reliably) and approximate tasks. When a Topaz task executes, it may execute on an approximate processor that may produce only an approximation of the precise result. So in our example, the `results` returned from the approximate tasks and therefore the final `sum` returned from the `compute` function may only approximate the actual precise sum.

Topaz is designed to work with approximate computation platforms that may occasionally produce arbitrarily inaccurate results. A potential issue is that such arbitrarily inaccurate results may become integrated into the main computation so that a result of the main computation (for example, the value of `sum` returned from `compute`) may also become unacceptably inaccurate.

2.3 Adaptive Outlier Detection

Topaz therefore deploys an *outlier detector* that is designed to detect tasks that produce very inaccurate results. The outlier detector operates under the principle that the majority of the accurate executions will produce results within a range that the outlier detector can adaptively lock on to, while inaccurate executions will tend to produce results that are outside that range. An ideal outlier detector would accept all correct results and reject all incorrect results. In practice this is infeasible because the ranges of correct and incorrect results overlap. The outlier detector must therefore strike a balance between rejecting too many correct tasks and accepting too many incorrect tasks. One complication is that, at the start of the execution, the outlier detector has no information about what a reasonable range might be.

The outlier detector therefore operates with a target correct task rejection rate and dynamically adapts the range to meet that rate. The goal is to obtain a tight range that accepts typical correct results but rejects extreme correct results so that it can also reject and correct extreme incorrect results. The basic principle behind the adaptation is as follows. If the detector is rejecting too many results below the minimum, it decreases the minimum to accept more of these results (and similarly for the maximum). If, on the other hand, it is accepting too many correct results, it increases the minimum to reject more results (and similarly for the maximum). To obtain a detector that locks on to the target rate quickly while avoiding overshoot and oscillation, the outlier detector uses a proportional-integral-derivative (PID) control algorithm [35] (see Section 4.4 for more details).

In our example, the outlier detector starts with the minimum and maximum of its range set to zero. It then dynamically adapts the minimum and maximum in response to the correct results that it observes, initially reexecuting many tasks until it finds a range that places the correct task reexecution rate under control. When the task reexecution rate is under control, the outlier detector reexecutes 1% of the correct tasks.

2.4 Reliable Reexecution

When the detector rejects a task, the current Topaz implementation reliably reexecutes the task to obtain the correct results and integrate these results into the main computation. This mechanism ensures that the outlier detector adaptation algorithm works only with correct results. It also maximizes the accuracy of the overall computation and enables the outlier detector to use a tight range that rejects both extreme correct and incorrect results (because the reexecution will ensure that these extreme correct values are integrated into the main computation).

3. The Topaz Language

We have implemented Topaz as an extension to C. Topaz adds a single construct, the *taskset* construct. When a taskset construct executes, it creates a set of approximate tasks that execute to produce results that are eventually combined back into the main computation. Figure 2 presents the general form of this construct.

```
taskset name(int i = l; i < u; i++) {
  compute in  (d1 = e1; ... dn = en)
       out (o1; ... om) {
    <task body>
  }
  combine { <combine body> }
}
```

Figure 2: Topaz Taskset Construct

Each taskset construct creates an indexed set of tasks. Referring to Figure 2, the tasks are indexed by `i`, which ranges from `l` to `u-1`. Each task uses an `in` clause to specify a set of in parameters `x1` through `xn`, each declared in corresponding declarations `d1` through `dn`. Topaz supports scalar declarations of the form `double x`, `float x`, and `int x`, as well as array declarations of the form `double x[N]`, `float x[N]`, and `int x[N]`, where `N` is a compile-time constant. The value of each `in` parameter `xi` is given by the corresponding expression `ei` from the `in` clause.

Each task also uses an `out` clause to specify a set of out parameters `y1` through `ym`, each declared in corresponding declarations `o1` through `om`. As for `in` parameters, `out` parameters can be scalar or array variables.

Topaz imposes the requirement that the `in` and `out` parameters are disjoint and distinct from the variables in the scope surrounding the `taskset` construct. Task bodies have no externally visible side effects — they write all of their results into the `out` parameters (which are then integrated into the computation via the execution of the combine body). All of the task bodies for a given taskset are therefore independent and can, in theory (although not in our current Topaz implementation) execute in parallel — only the combine bodies may have dependences. More importantly for our current implementation, the absence of externally visible side effects enables the transparent Topaz task reexecution mechanism (which increases the accuracy of the computation by transparently reexecuting outlier tasks to obtain guaranteed correct results).

3.1 Conceptual Execution Model

We next present the conceptual execution model for the `taskset` construct. The `taskset` construct iterates over all of the values of the task index `i` from `l` to `u-1`. At each iteration it creates a new task, evaluates the `in` expressions `e1` through `en` for that task, and copies the values of the `in` expressions into the corresponding `in` variables `x1` through `xn`. It then creates the naming context for the task, which includes the task index `i`, the `in` variables `x1` through `xn`, and the `out` variables `y1` through `ym`.

The task then executes the `task body`. When the task body finishes, the `out` variables may be set to arbitrary values (although, in practice, the Topaz implementation attempts to deliver values that are identical or close to the values that the `task body` would have generated had it executed precisely).

The next step is to optionally execute the `combine body` — in other words, the `combine body` may or may not execute. In practice, of course, Topaz implementations should make a best-effort attempt to obtain reasonable values and execute the `combine body` for every task. The current Topaz implementation executes the `combine body` for every task — if the outlier detector accepts the approximate result, it executes the combine body with that result, otherwise it reexecutes the task precisely and executes the combine body with the resulting correct result.

Unlike the `task body`, the `combine body` executes precisely. The naming context for the `combine body` includes the naming context surrounding the `taskset` construct augmented with the task index `i` and the `out` parameters `y1` through `ym`. The `combine body` integrates the results from the task into the main computation, typically by combining the results into a data structure within the main computation.

3.2 Design Rationale

Topaz is designed to support a computational pattern that is in our experience characteristic of many approximate computations [17, 37–39, 47]. This computational pattern consists of multiple independent tasks that generate contributions that are then combined to obtain the final result (in Topaz the `out` variables contain these contributions). In this pattern, the operation that combines the contributions into the final result is typically relatively simple, with the vast majority of the computation taking place in the tasks. By exposing the structure present in this computational pattern to the Topaz implementation, Topaz enables the implementation to identify and exploit the approximation opportunities inherently present in the application.

We note that in many aspects, the basic Topaz execution model more closely resembles a networking API specification (no guarantees, best-efforts execution) than a standard precise programming language semantics. Indeed, Topaz and standard networking APIs share a common motivation — they are both designed to work with underlying hardware platforms (networks and approximate computation platforms) that can typically offer few, if any, precise performance or quality guarantees. The Topaz best efforts execution model is therefore designed to give the Topaz implementation the freedom it needs to execute Topaz programs successfully on emerging and future approximate computing platforms. This flexible model of computation also gives the Topaz implementation the freedom it needs to apply techniques (such as outlier detection and reexecution, see Section 4.4) that enhance the accuracy of results that the bare approximate hardware provides.

4. Topaz Implementation

We next discuss the current Topaz implementation. The Topaz front end translates Topaz programs into C programs that invoke the featherweight Topaz runtime APIs. The Topaz runtime provides the support required to execute Topaz programs on the underlying target approximate computing platform. The Topaz outlier detector prevents unacceptably inaccurate results from corrupting the results that the Topaz main computation produces.

4.1 Target Topaz Approximate Computing Platform

The current Topaz implementation is designed to run on an approximate computing platform with at least one precise processor and at least one approximate processor. The precise processor executes the main Topaz computation, which maps the Topaz tasks onto the approximate processors for execution. The current Topaz implementation assumes a distributed model of computation with separate processes running on the precise and approximate processors. The current Topaz implementation uses MPI [23] as the communication substrate that enables these separate processes to interact. The lifetime of a Topaz task therefore comprises the following events:

- **Creation:** The main Topaz computation executing on a precise processor creates the task.
- **Approximate Processor Assignment:** The Topaz implementation assigns the Topaz task to an approximate processor for execution. It sends an MPI message from the precise processor to the approximate processor that will execute the task. This message contains the `in` parameters and other information required to execute the task.
- **Execution:** The approximate processor receives the task information and executes the task to produce the results.
- **Return Results:** The approximate processor sends an MPI message back to the precise processor running the Topaz main computation. This message contains the (potentially arbitrarily inaccurate) results that the approximate execution of the Topaz task produced.
- **Outlier Detection and Optional Reexecution:** The precise processor runs the outlier detector. If any of the return values is an outlier, the Topaz implementation reexecutes the task on the precise processor to obtain correct results. If all of the return values are not outliers, the Topaz implementation accepts the result from the approximate execution. The current Topaz implementation maintains a separate outlier detector for each return value.
- **Result Integration:** The precise processor running the main Topaz computation executes the `combine` block of the task. This `combine` block integrates the results from the task into the results that the main Topaz computation will produce.

4.2 Topaz Front End

As is standard in implementations of task-based languages that are designed to run on distributed computing platforms [39], the Topaz implementation contains a front end that translates the Topaz `taskset` construct into API calls inside the Topaz runtime that marshall the `in` parameters, the task index `i`, and other information (such as the identifier of the function that implements the Topaz task) required to execute the task. The front end also generates the code required to marshall the task results and send them back to the processor running the main Topaz computation.

4.3 Topaz Runtime

The Topaz runtime is a featherweight library that provides a data marshalling API and coordinates the movement of tasks and results between the precise processor running the main Topaz implementation and the approximate processor that runs the Topaz tasks. It also contains the task dispatch and management code required to coordinate the transfer of tasks and data between the precise and approximate processors.

4.4 Outlier Detection Control Algorithm

The Topaz outlier detector works with a target task reexecution rate r for each taskset. All tasks in the taskset return a result with the same number of components m. There is an outlier detector for each of the m components, with the target correct task reexecution rate for each component set to $r_{targ} = r/m$. The goal of the outlier detector for each component is to reject exactly $r_{targ}\%$ of the values that correct tasks return in that component.

Each outlier detector maintains three values: μ, l, and r. It initializes these values by reliably reexecuting the first 10 tasks (this number is configurable) and computing the mean μ, minimum l, and maximum r of the observed values for the component that the outlier detector is responsible for. The values l and r are controlled by two distinct PID control systems C_{right} and C_{left}; μ is unchanged throughout the remaining computation.

The outlier detector rejects a task if the value of its component falls outside the range $[\mu - l, \mu + r]$. The control system C_{left} is responsible for rejecting tasks that fall to the left μ; C_{right} is responsible for rejecting tasks that fall to the right μ. Each control system is given a target reexecution rate of r_{targ} for the subset of tasks that it controls. The two control systems, working together, therefore have a target reexecution rate of r_{targ} for all of the tasks, with the percentage of rejected tasks split between the two control systems in proportion to the number of tasks that fall to the left and right of the outlier detector's range. Given a value v for a given component, C_{left} is updated when $v < \mu$, C_{right} is updated when $v > \mu$.

We formalize the control system C_{left} as follows (C_{right} is formalized symmetrically):
Control System State:

- n: number of tasks that have been executed.
- r_{left}: number of tasks rejected by outlier detector.
- e^*_{left}: The exponentially decayed sum (with decay D=.99) of all previous observed reexecution rate errors.
- e_{left}: The last reexecution rate error
- $e'_{left} = \frac{r_{left}}{n} - r_{targ}$: The error between the target reexecution rate and the current reexecution rate.

The outlier detector uses the following PID equation to update the bound l:

$$u(t) = K_{pres} \cdot e_{left} + K_{int} \cdot (e'_{left} + D \cdot e^*_{left}) + K_{der} \cdot (e'_{left} - e_{left})$$

Here we determine the constants $K_{pres}, K_{int}, K_{der}$ empirically.

Updating the Control System
We discuss two algorithms for updating the control system — one when the system reexecutes outliers to obtain the correct value, and one when the system simply discards outliers.
The next state of the control system is calculated as follows:

With Reexecution: Given a value v for its component, C_{left} is updated under the following circumstances:

- value $v \in [\mu - l, \mu]$, accepted $\rightarrow$
  $r_{left} = r_{left} + 1, n = n + 1, l = l + u(t)$.
- value $v < \mu - l$, rejected, is verified correct on re-execution $\rightarrow$
  $n = n + 1, l = l + u(t)$.

The outlier detector also maintains the observed minimum v_{min} and maximum v_{max} values for its component. When it updates l and r, the reexecution outlier detection system imposes the additional constraint that:
$v_{min} >= \mu - l$
$v_{max} <= \mu + r$

If the value v is accepted, the outlier detector updates the mean μ and adjusts the l,r bounds to account for the change in μ. This process occurs as follows:

1. if value $v \in [\mu - l, \mu]$, accepted $\rightarrow$
   $\mu' = \mu \leftarrow v$, the mean with v incorperated.
   $l = l + (\mu' - \mu), r = r - (\mu' - \mu), \mu = \mu'$

With Discard: C_{left} is updated under the following circumstances:

1. value $v \in [\mu - l, \mu]$, accepted $\rightarrow$
   $r_{left} = r_{left} + 1, n = n + 1, l = l + u(t)$
2. value $v < \mu - l$, rejected $\rightarrow$
   $n = n + 1, l = l + u(t)$

4.5 Probabilistic Accuracy Bounds

We next discuss how we obtain probabilistic bounds on the number of accepted incorrect tasks and the size of the total error that these tasks induce. We assume we have a count of the number of potentially faulty operations performed during the computation of a taskset. Our experimental results in Section 5 work with approximate hardware in which floating point loads and stores may be faulty. We assume that the approximate processor comes with standard hardware event counters that provide the total number of executed floating point loads and stores. If such information is not available the task execution time can be used to estimate the number of potentially faulty operations. Our formalism works with the following parameters:

- $e_{sram} = 1 \cdot 10^{-5}, 1 \cdot 10^{-3}$: probability of read/write error occurring for a given floating point load or store. When enacting our formalization, we choose one of the two cache error rates listed above.

On average, there are $n^{fp}_{sram} \cdot e_{sram}$ erroneous tasks in the taskset. Assuming an even distribution of floating point load and store operations across the tasks, we predict there are $e_{task} = \frac{n^{fp}_{sram} \cdot e_{sram}}{n_{task}}$ erroneous operations per task. We use the inverse Poisson distribution with $\lambda = e_{task}$ to determine the lowest frequency of errors that may occur with probability q:

$$e'_{task} = Poiss^{-1}(q, \lambda = e_{task})$$

We then determine the corresponding bound on the total number of erroneous tasks in the taskset as:

$$n^{err}_{task}\prime = n_{task} \cdot e'_{task}$$

Recall n^{err}_{rej} is available at runtime since rejected tasks are reexecuted. We compute the bound on the number of accepted erroneous tasks as:

$$n^{err}_{acc}\prime = n^{err}_{task}\prime - n^{err}_{rej}$$

The outlier detector tracks the observed minimum and maximum values bounds $[v_{min}, v_{max}]$ for each task component. We estimate the maximum error from each accepted erroneous task as:

$$f_{max} - f_{min}$$

The corresponding bound on the total error is:

$$\bar{e}_{out} = (f_{max} - f_{min}) \cdot \frac{n^{err}_{acc}\prime}{n_{task}}$$

Benchmark	% Target Reject Rate	% Accepted Correct	% Rejected Correct	% Accepted Error	% Rejected Error	Reject Rate
blackscholes	3.0%	96.989%	2.852%	0.048%	0.111%	2.964%
scale	4.0%	96.714%	0.564%	0.255%	2.466%	3.031%
barnes	3.0%	86.813%	1.797%	9.233%	2.157%	3.955%
water interf	5.9%	94.564%	4.984%	0.211%	0.241%	5.225%
water poteng	5.9%	96.061%	3.675%	0.118%	0.146%	3.822%
search	3.6%	97.223%	1.639%	0.562%	0.576%	2.215%

Table 1: Overall Outlier Detector Effectiveness

5. Experimental Results

We present experimental results for Topaz implementations of our set of five benchmark computations:

- **Scale:** A computation that scales an image using bilinear interpolation [11].
- **BlackScholes:** A financial analysis application that solves a partial differential equation to compute the price of a portfolio of European options [47]. As is standard when using this application as a benchmark, we run the computation multiple times.
- **Water:** A computation that simulates liquid water [37].
- **Barnes-Hut:** A computation that simulates a system of N interacting bodies (such as molecules, stars, or galaxies). At each step of the simulation, the computation computes the forces acting on each body, then uses these forces to update the positions, velocities, and accelerations of the bodies [6, 36].
- **Search:** A computation that uses a Monte-Carlo technique to simulate the interaction of an electron beam with a solid at different energy levels. The result is used to measure the match between an empirical equation for electron scattering and a full expansion of the wave equation.

5.1 Experimental Setup

We perform our experiments on a computational platform with one precise processor, which runs the main Topaz computation, and one approximate processor, which runs the approximate Topaz tasks. Given a Topaz program, the Topaz compiler produces two binary executables: a reliable executable that runs the main Topaz computation on the precise processor and an approximate executable that runs the approximate Topaz tasks on the approximate processor. The two executables run in separate processes, with the processes communicating via MPI as discussed in Section 4.

In our experiments we run both processes under the control of the Pin [30] binary instrumentation system. For the main Topaz process Pin does not affect the semantics. For the approximate process, we use Pin to simulate an approximate processor with two caches: a reliable cache that holds integer data and a more energy-efficient but unreliable cache that holds floating point data. The integer and floating point caches are the same size. For the unreliable floating point cache we use the Medium cache model and Aggressive cache model in [42], Table 2. The Medium cache writes an incorrect result (for a floating point number) with probability $1*10^{-4.94}$ and reads an incorrect result with probability $1*10^{-7.4}$. This cache yields energy savings of 80% over a fully reliable cache. Together, the two caches consume 35% of the total CPU energy. The approximate processor therefore consumes (.8*.35%)/2 = 14% less energy than the precise processor.[1]

The Aggressive cache reads and writes an incorrect result (for a floating point number) with probability $0.5 * 10^{-3}$ and energy savings of 90% over a fully reliable cache. Together, the two caches consume 35% of the total CPU energy. The approximate processor therefore consumes (.9*.35%)/2 = 15.75% less energy than the precise processor.

For the MPI communication layer that the Topaz implementation uses, each communication incurs a fixed cost plus a variable cost per byte. To amortize the fixed cost, our experiments work with *batched tasks* — instead of sending single tasks from the main processor to the approximate processor, Topaz sends a batch of tasks for each communication. When the batch of tasks finishes, Topaz sends all of the results from the approximate processor back to the main processor with a single batched message. For our benchmark set of applications, this approach successfully amortizes the communication overhead.

5.2 Benchmark Executions

We present experimental results that characterize the accuracy and energy consumption of the Topaz benchmarks under a variety of scenarios. We perform the following executions:

- **Precise:** We execute the entire computation, Topaz tasks included, on the precise processor. This execution provides the fully accurate results that we use as a baseline for evaluating the accuracy and energy savings of the approximate executions (in which the Topaz tasks execute on the approximate processor).
- **Full Computation Approximate:** We attempt to execute the full computation, main Topaz computation included, on the approximate processor. For Scale, BlackScholes, Barnes, Water, and Search, this computation terminates with a segmentation violation.
- **Full Reexecution:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with outlier detection as described in Section 4. Instead of reliably reexecuting only outlier tasks, we reliably reexecute all tasks. This enables us to classify each task as either 1) Accepted Correct, 2) Rejected Correct, 3) Accepted Error, or 4) Rejected Error. For each benchmark, we specify a computation-specific target reexecution rate.
- **No Outlier Detection:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with no outlier detection. We integrate all of the results from approximate tasks into the main computation.
- **Outlier Detection With Reexecution:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with outlier detection and reexecution as described in Section 4.
- **Outlier Detection With Discard:** We execute the Topaz main computation on the precise processor and the Topaz tasks on the approximate processor with outlier detection. Instead of

[1] Our experiments conservatively use the higher write error rate for reads in the Medium cache from [42].

reexecuting outliers and including the resulting correct results in the computation, we discard outliers.

Benchmark Benchmark	Target Reexecution Rate	Observed Reexecution Rate
scale	1.333%	1.5988%
	1.333%	1.4938%
	1.333%	1.4297%
blackscholes	3.00%	2.9927%
barnes	0.375%	0.463$
	0.375%	0.454%
	0.375%	0.494%
	0.375%	0.414%
	0.375%	0.411%
	0.375%	0.448%
	0.375%	0.000%
	0.375%	3.03 %
water [inter mol]	0.28%	0.28%
	0.28%	0.43%
	0.28%	0.32%
	0.28%	0.28%
	0.28%	0.30%
	0.28%	0.31%
	0.28%	0.29%
	0.28%	0.33%
	0.28%	0.29%
	0.28%	0.28%
	0.28%	0.33%
	0.28%	0.30%
	0.28%	0.28%
	0.28%	0.31%
	0.28%	0.32%
	0.28%	0.30%
	0.28%	0.43%
	0.28%	0.32%
	0.28%	0.35%
	0.28%	0.13%
	0.28%	0.28%
water [pot eng]	1.97%	0.0017 %
	1.97%	1.97%
	1.97%	1.98%
search	0.720%	0.721%
	0.720%	0.768%
	0.720%	0.934%
	0.720%	0.720%
	0.720%	0.719%

Table 2: Individual Outlier Detector Reexecution Rates.

5.3 Outlier Detector Effectiveness

We evaluate the effectiveness of the outlier detector using results from the Full Reexecution runs. Table 1 presents the results from these runs. There is a row for each benchmark application. The first column (Benchmark) is the name of the benchmark. The second column (Target Reject Rate) are the benchmark-specific target reexecution rates we will use for our experiments. The third column (Accepted Correct) is the percentage of correct tasks that the outlier detector accepts. The fourth column (Rejected Correct) is the percentage of correct tasks that the outlier detector rejects. The fifth column (Accepted Error) is the percentage of error tasks that the outlier detector accepts. The sixth column (Reject Rate) is the actual percentage of reexecuted tasks. These tasks are the only source of error in the computation. The number of such tasks is small and, because the results fall within the outlier detector range, the introduced error is bounded with high likelihood. The fifth column (Rejected Error) is the percentage of error tasks that the outlier detector rejects. The final column (Reexecuted) is the percentage of tasks that are reexecuted (this column is the sum of Rejected Correct and Rejected Error). These numbers highlight the overall effectiveness of the outlier detector. The vast majority of tasks integrated into the main computation are correct; there are few reexecutions and few incorrect tasks.

To evaluate how well the Topaz control algorithm is able to deliver the target task reexecution rates for each result component, we next consider the effectiveness of the individual outlier detectors. Table 2 presents the task reexecution rates for each outlier detector in each application. Recall from 4 that there is a separate outlier detector for each output component that the tasks produce, where the target task reexecution rate is evenly distributed across these outlier detectors.

If the value of any result component falls outside the range of its outlier detector, the entire task is rejected. If each task had at most one component that fell outside the range, the Rejected Correct number from Table 1 would equal the sum of the number of tasks that each individual outlier detector rejected. In practice, multiple outlier detectors may reject a single task, so the Rejected Correct percentage from Table 1 provides a lower bound on the sum of the individual outlier detector reexecution percentages.

- **Scale:** Scale has one taskset; the tasks in this taskset return three numbers (the RGB values for each pixel). This taskset therefore has three outlier detectors, each with a target correct task reexecution rate of 1.333%. The results show that the Topaz control algorithm is able to hit this target rate for each outlier detector. The sum of the reexecution rates for the individual outlier detectors is 4.5223%, and the percentage of rejected tasks (Reject Rate) from Table 1 is 3.031%. We attribute the difference between the two values to the fact that a non-negligible fraction of the tasks are rejected by multiple outlier detectors.
- **BlackScholes:** BlackScholes has a single taskset with a single return value and therefore a single outlier detector. The reexecution rate for this outlier detector is therefore the Reject Rate from Table 1.
- **Barnes:** Barnes has a single taskset with tasks that return eight components. All outputs but the last one have reexecution rates that are are sufficiently close to the target. The reexecution rate for the seventh detector is zero because all tasks return a fixed value for that component. The summation of the individual reexecution rates is 5.71%, which exceeds the task reexecution 3.955%, suggesting that, in some scenarios, multiple outlier detectors reject the same task. Notice the last output never reaches the target reexecution rate. The eighth component grows consistently throughout the execution of the application, so the outlier detector control algorithm can never lock on to an effective range and must continually grow its range in response to the new values that the tasks produce.
- **Water:** Water has two tasksets — the first has 21 components, the second has three components. The Water outlier detectors are all able to (essentially) hit their target rates. The reexecution rates of the outputs from the first task sum up to 6.18$, which exceeds the observed task reexecution rate of 5.225 from Table 1. Similarly, the reexecution rates of the outputs from the second task sum up to 3.95%, which exceeds the observed task reexecution rate of 3.822. Therefore, in both tasks, there are some cases where multiple outlier detectors reject the same task.

Benchmark	Accepted Error Task Bound (95%)	Actual Accepted Error
scale	3.32%	0.258%
blackscholes	0.206%	0.087%
barnes	15.63%	9.23%
water [inter mol]	0.842%	0.2112%
water [pot eng]	0.642%	0.118%
search	1.691 %	0.5619%

Table 3: Probabilistic Error Bounds Results

Benchmark	Percent Silent Errors
Scale	15.4%
Blackscholes	19.96%
Barnes	19%
Water [Inter Mol]	40.68%
Water [Pot Eng]	52.00%
Search	19.98%

Table 4: Probabilistic Error Bounds, Corrected for Silent Errors

- **Search:** Search has five outputs. The outlier detectors are able to (essentially) hit their target re-execution rates. The individual re-execution rates sum up to 3.862, which is significantly larger than the observed rejection rate of 2.215. Therefore, a non-negligible portion of tasks are rejected by multiple outlier detectors.

These numbers illustrate the overall general effectiveness of the Topaz reexecution rate control algorithm for our set of benchmark applications.

5.4 Probabilistic Error Bounds Analysis

Table 3 presents the results for the 95% error bounds analysis described in Section 4.5. There is one row in the table for each benchmark. The first column presents the name of the benchmark. The second column (Accepted Error Task Bound (95%)) presents the 95% bound on the predicted number of accepted error tasks. As described above in Section 4.5, we compute this number based on the number of potentially faulty operations in the taskset. For our approximate computing platform these operations are floating point cache loads and stores. In production these counts would be supplied by standard hardware event counters; in our experiments our Pin emulator provides these counts. We note that the number of accepted tasks is, for all benchmarks, significantly below the 95% error bounds. There are two relevant factors: the first is the overestimate required to obtain the 95% bound, the second is the fact that a significant number of errors are silent (i.e., do not affect the final result that the task produces). Table 4 presents the percent silent errors for every benchmark.

5.5 End-to-End Application Accuracy

We next present end-to-end accuracy results from our benchmark applications. For each application we obtain an appropriate end-to-end accuracy metric: for Scale we report the PSNR of the scaled image; for BlackScholes we report the percentage error of the portfolio of options; for Water we report the error in the positions of the Water molecules; for Barnes we report the error in the positions of the bodies; for Search we report the error between optimal parameter sets. For all the benchmark, the comparison points for the percentage differences are the results from the Precise executions.

Benchmark	No Topaz	No Outlier Detection	Outlier Detection With Discard	Outlier Detection With Reexecute
blackscholes	crash	6.04e34%	0.18%	0.03%
scale	crash	24.7	19.3	47.3
barnes	crash	inf	1.54e-2%	4.29e-2%
water	crash	inf	1.46e-2%	6.13e-4%
search	crash	43.25%	24.5%	2.5%

Table 5: End-to-end Application Accuracy Measurements. For Scale, the table presents the PSNR for the output image. For PSNR, a larger value indicates the generated image is closer to the solution image. For BlackScholes the table presents the error as a percentage of the total value of the option portfolio. For Water the table presents the mean relative error in the positions of the Water molecules (in comparison with the positions computed by the precise execution). For Barnes the table presents the mean relative error in the positions of the simulate bodies (in comparison with the positions computed by the precise execution). For Search, the table presents the percent parameter error (average percent error for each parameter). For all other benchmarks (excluding scale), smaller errors indicate better quality results.

Table 5 presents these metrics for the No Outlier Detector, Outlier Detector with Reexecution, and Outlier Detector with Discard runs. For Scale outlier detection with reexecution significantly increases the PSNR from 24.7 to 47.3 (we use the image produced by the Precise execution as the baseline to calculate this PSNR). The discard PSNR is lower than the no outlier detection PSNR because discarded tasks yield black pixels in the image, which degrade the PSNR substantially.

For BlackScholes outlier detection is critical to obtaining acceptable accuracy — outlier detection reduces an otherwise enormous error to reasonable size, with reexecution delivering an almost additional two orders of magnitude reduction in the relative error in comparison with discarding outliers. For Water both outlier detection and reexecution are required to obtain acceptably accurate results. Without outlier detection the molecule positions quickly become not a number (nan). Reexecuting outliers (instead of discarding outliers) delivers roughly a factor of one hundred increase in accuracy. For Barnes outlier detection is also critical to obtaining acceptable accuracy — without outlier detection, some of the positions become so large (inf) that our Python data processing script starts generating overflows when it attempts to calculate the relative error. Reexecuting outliers (instead of discarding outliers) delivers roughly a factor of three increase in accuracy. For Search both outlier detection and re-execution are necessary to obtain a set of parameters that is acceptably close to the correct result. Without outlier detection, the average parameter error is almost 50%. With reexecution, the average parameter error is 2.5% - 20x better than the no outlier detection case. In fact, the percent error on the incorrect parameter is so small, it is within one adjustment of the correct parameter (search makes adjustments by increasing, decreasing a parameter by 20%). Discarding tasks yields a average parameter error of 24.5%; a 2x improvement on no outlier detection, but still 10x worse than reexecution.

5.6 Energy Model

We next present the energy model that we use to estimate the amount of energy that we save through the use of our approximate computing platform. We model the total energy consumed by the

Benchmark	No Reexecute	Reexecute
blackscholes	$0.903 = 0.31 + 0.01 + 0.86 * 0.679$	$0.902 = 0.31 + 0 + 0.86 * 0.679$
scale	$0.894 = 0.37 + 0 + 0.845 * 0.621$	$0.903 = 0.37 + 0.009 + 0.845 * 0.621$
barnes	$0.865 = 0.04 + 0 + 0.86 * 0.959$	$0.866 = 0.04 + 0.001 + 0.86 * 0.959$
water	$0.856 = 0.16 + 0 + 0.86 * 0.809$	$0.887 = 0.16 + 0.031 + 0.86 * 0.809$
search	$0.872 = 0.12 + 0 + 0.86 * 0.874$	$0.878 = 0.12 + 0.006 + 0.86 * 0.874$

Table 6: Energy Consumption Equations for Benchmarks

computation as the energy spent executing the main computation and any reliable reexecutions on the precise processor plus the energy spent executing tasks on the approximate processor. We report the total energy consumption for the Outlier Detection with Reexecution runs normalized to energy consumption for the Precise execution runs (which execute the entire computation reliably on the precise processor). Our equation for the normalized total energy for the approximate execution is therefore $M + T_r + F * T_t$, where M is the proportion of instructions dedicated to the precise execution of the main computation, T_t is the proportion of instructions dedicated to the approximate execution of Topaz tasks, T_r is the proportion of instructions dedicated to the approximate reexecution of Topaz tasks, and F reflects the reduced energy consumption of the approximate processor. In the medium energy model, the approximate processor consumes 14% less power than the precise processor, and $F = 0.86$. In the aggressive energy model, the approximate processor consumes 15.75% less power than the precise processor and $F = 0.845$. See Section 5.1 for details.

Benchmark	Model	Maximum	No Reexecute	Reexecute
blackscholes	med	14.0%	9.641%	9.489%
scale	agg	15.75%	9.984%	9.831%
barnes	med	14.0%	13.44%	13.425%
water	med	14.0%	11.726%	11.281%
search	med	14.0%	12.38%	12.296%

Table 7: Energy Consumption Table. 'med' is the medium energy model, and 'agg' is the aggressive energy model.

Table 6 presents the energy consumption equations for our set of benchmark applications. There is one row for each application; that row presents the energy consumption equation for that application. The second column of Table 6 are the energy consumption equations for outlier detection without reexecution. The third column of 6 are the energy consumption equations for outlier detection with reexecution. The approximate Scale execution, for example, consumes 90% of the energy required to execute the precise Scale computation — the approximate execution spends 37% of its execution time in the main computation, spends 0.1% of the time reexecuting, and spends 62% of its execution time in approximate tasks for a total energy savings of 10% relative to the reliable execution, and similarly for the other applications. Starting from a maximum possible energy savings of 14% and 15.75% for medium and aggressive energy models respectively, the applications exhibit between a 13% and 9% energy savings.

Table 7 presents the energy savings in a condensed form. Each benchmark has a corresponding row, labeled with the benchmark name in the first column. The second column lists the energy savings for each benchmark without re-execution, the third column lists the energy savings with re-execution. Notice that the energy overhead from re-executing tasks is relatively small.

6. Related Work

We discuss related work in software approximate computing, approximate memories, and approximate arithmetic operations for energy-efficient hardware.

6.1 Software Approximate Computing

In recent years researchers have investigated various software mechanisms that produce less accurate (but still acceptably accurate) results in return for performance increases or reductions in the energy required to execute the computation. Examples of these mechanisms include skipping tasks [37], early termination of barriers in parallel programs with load imbalances [28, 38], approximate function memoization [16], loop perforation [34, 47], loop approximation and function approximation [5], randomized reduction sampling and function selection [51], pattern-based approximate program transformations [41], and autotuners that automatically select combinations of algorithms and algorithm parameters to optimize the tradeoff between performance and accuracy [2, 20, 50]. Researchers have also developed static program analyses for approximate program transformations [10, 16, 33, 49, 51] and analyses for computations that operate on uncertain inputs [7, 14–16, 19, 31, 44, 45]. It is also possible to improve performance by eliminating synchronization in multithreaded programs, with the resulting data races introducing only acceptable inaccuracy [32, 36].

Like task skipping [37] and early phase termination [38], Topaz operates at the granularity of tasks and exploits the ability of approximate computations to tolerate inaccurate or missing approximate tasks. One unique and novel contribution of Topaz is its use of outlier detection to identify outliers with obviously inaccurate results. Unlike many outlier detectors, which simply discard outliers, Topaz replaces outliers with the mean of previously observed results. This mechanism enables Topaz to salvage and incorporate accurate results from tasks that produce multiple results when only one or some of the produced results is an outlier.

Our experimental results show that outlier detection and replacement enables approximate computations to profitably use approximate computational platforms that may occasionally produce arbitrarily inaccurate outputs. Topaz therefore supports a larger range of approximate platforms and gives hardware designers significantly more freedom to deploy aggressive energy-efficient designs.

6.2 Approximate Memories

Researchers have previously exploited approximate memories for energy savings in approximate computations. Empirically even approximate computations have state and computation that must execute exactly for the overall computation to produce acceptable results [11, 12, 29, 37, 42, 47]. Proposed architectures therefore provide both exact and approximate memory modules [11, 29, 42, 43]. In some cases the assumption is that approximate memory can be (in principle) arbitrarily incorrect, with the program empirically producing acceptable results with the proposed memory implementation [29, 42]. Researchers have also developed techniques that make it possible to reason quantitatively about how often the approximate memory may cause the computation to produce an incorrect result [11].

Like all of these systems, Topaz leverages the reduced energy consumption of approximate memories (in our experiments, the reduced memory consumption enabled by unreliable SRAM cache memory) to reduce the overall energy consumption of the computation. Topaz goes beyond all of these systems in that it can detect when the approximate memory has caused the computation to generate an unacceptably inaccurate result. In this case Topaz, unlike these previous systems, takes steps to preserve the integrity of the computation and prevent the unacceptable corruption of the overall result that the unacceptably inaccurate result would otherwise produce. Specifically, the Topaz outlier detection and reexecution mechanism enables the approximate computation to produce acceptably accurate results even in the presence of arbitrarily inaccurate results (caused, for example, by approximate memories).

6.3 Approximate Arithmetic Operations

For essentially the entire history of the field, digital computers have used finite-precision floating point arithmetic to approximate computations over (conceptually) arbitrary-precision real numbers. The field of numerical analysis is devoted to understanding the consequences of this approximation and to developing techniques that maximize the accuracy of computations that operate on floating point numbers [3].

Motivated by the goal of reducing energy consumption, researchers have proposed energy-efficient hardware that may produce reduced precision/incorrect results in return for energy savings [13, 21]. Researchers have also investigated methods that tune the amount of precision to the needs of the computation at hand. Hardware approaches include mantissa bitwidth reductions with significant energy savings [25, 48] or fuzzy memoization [1]. It is well-known that replacing selected double precision floating point with single precision floating point can yield energy improvements and acceptable accuracy [4, 8, 9, 22, 24, 27]. Building on this common knowledge, researchers have developed techniques that automatically perform this transformation when it preserves acceptable accuracy [26, 40]. It is also possible to deploy a wider range of transformations in a randomized optimizer that aims only to produce acceptably accurate bit patterns and is agnostic to whether it obtains these bit patterns using floating point operations or other means [46].

7. Conclusion

As energy consumption becomes an increasingly critical issue, practitioners will increasingly look to approximate computing as a general approach that can reduce the energy required to execute their computations while still enabling their computations to produce acceptably accurate results.

Topaz enables developers to cleanly express the approximate tasks present in their approximate computations. The Topaz implementation then maps the tasks appropriately onto the underlying approximate computing platform and manages the resulting distributed approximate execution. The Topaz execution model gives approximate hardware designers the freedom and flexibility they need to produce maximally efficient approximate hardware — the Topaz outlier detection and reexecution algorithms enable Topaz computations to work with approximate computing platforms even if the platform may occasionally produce arbitrarily inaccurate results. Topaz therefore supports the emerging and future approximate computing platforms that promise to provide an effective, energy-efficient computing substrate for existing and future approximate computations.

References

[1] C. Álvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Computers*, 54(7):922–927, 2005.

[2] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *PLDI*, pages 38–49, 2009.

[3] K. Atkinson. *An Introduction to Numerical Analysis*. Wiley, 1989.

[4] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[5] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[6] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[7] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain: a first-order type for uncertain data. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 51–66. ACM, 2014.

[8] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Exploiting mixed precision floating point hardware in scientific computations. In *High Performance Computing Workshop*, pages 19–36, 2006.

[9] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *IJHPCA*, 21(4):457–466, 2007.

[10] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[11] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, pages 33–52, 2013.

[12] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, pages 37–48, 2010.

[13] L. N. Chakrapani, K. K. Muntimadugu, L. Avinash, J. George, and K. V. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation. In *CASES*, pages 187–196, 2008.

[14] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–220. ACM, 2014.

[15] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[16] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 102–112. ACM, 2011.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, pages 301–312, 2012.

[19] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[20] M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[21] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *CASES*, pages 158–168, 2006.

[22] D. Goeddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007.

[23] W. Gropp and E. Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[24] J. Hogg and J. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Soft.*, 37(2):1–24, 2010.

[25] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. In *ISSCC*, pages 182–184, 2012.

[26] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *ICS*, pages 369–378, 2013.

[27] X. Li, M. Martin, B. Thompson, T.Tung, D. Yoo, J. Demmel, D. Bailey, G.Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, and A. Kapur. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Soft.*, 28(2):152–205, 2002.

[28] T.-H. Lin, S. Tarsa, and H. Kung. Parallelization primitives for dynamic sparse computations. In *HOTPAR*, 2013.

[29] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, pages 213–224, 2011.

[30] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[31] R. Majumdar and I. Saha. Symbolic robustness analysis. In *RTSS '09*.

[32] S. Misailovic, D. Kim, and M. C. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embedded Comput. Syst.*, 12(2s), 2013.

[33] S. Misailovic, D. Roy, and M. Rinard. Probabilistically accurate program transformations. SAS, 2011.

[34] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[35] K. Ogata. *Modern Control Engineering*. 2009.

[36] M. Rinard. Parallel synchronization-free approximate data structure construction. In *HOTPAR*, 2013.

[37] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[38] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, pages 369–386, 2007.

[39] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[40] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *SC*, page 27, 2013.

[41] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: pattern-based approximation for data parallel applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 35–50. ACM, 2014.

[42] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[43] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36. ACM, 2013.

[44] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 14. ACM, 2014.

[45] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, 2013.

[46] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, 2014.

[47] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[48] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3):273–286, 2000.

[49] E. Westbrook and S. Chaudhuri. A semantics for approximate program transformations. *arXiv preprint arXiv:1304.5531*, 2013.

[50] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

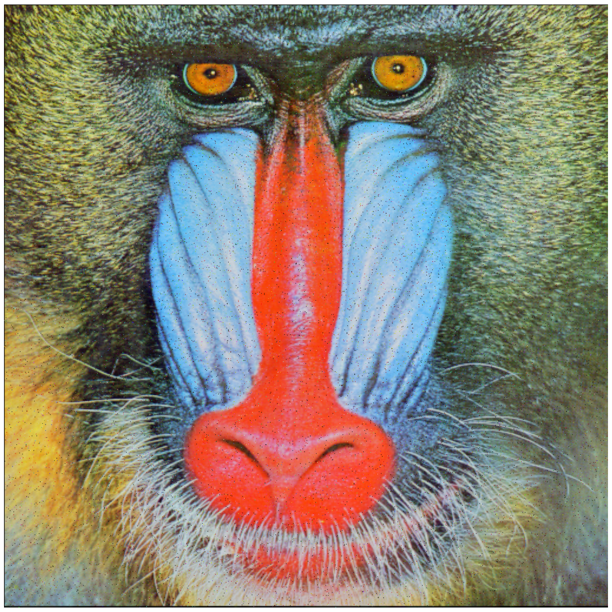
[51] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.

8. Appendix I: Alternative Output Representations

8.1 Visual Representations of Selected Outputs

8.1.1 Image Output for Scale

Below are the output images generated from the scale benchmark for the following scenarios: (1) No Outlier Detector (2) Outlier Detector with Reexecution (3) Outlier Detector with Discard. The raw output image is on the left side of each entry. An image diff between the generated image and the correct image is on the right side of each entry. In the image diff, identical pixels show up as white, and incorrect pixels show up as varying shades of red (depending on magnitude of error).


(a) Scaled Baboon Image

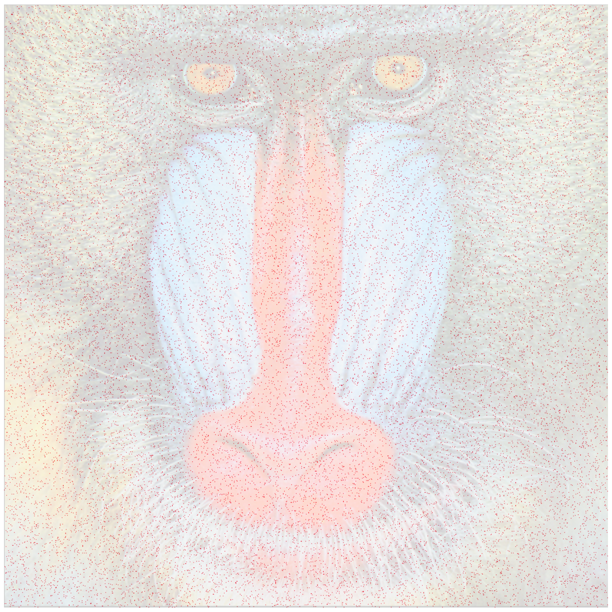
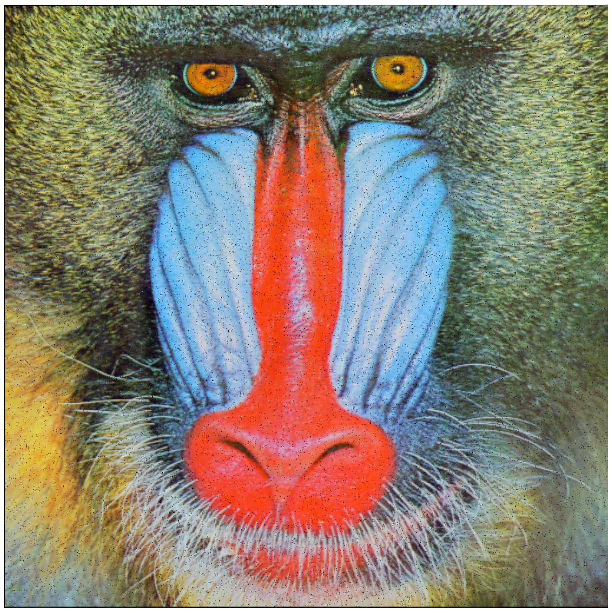

(b) Diff of Scaled Baboon Image with Correct Image

Figure 3: No Outlier Detector


(a) Scaled Baboon Image

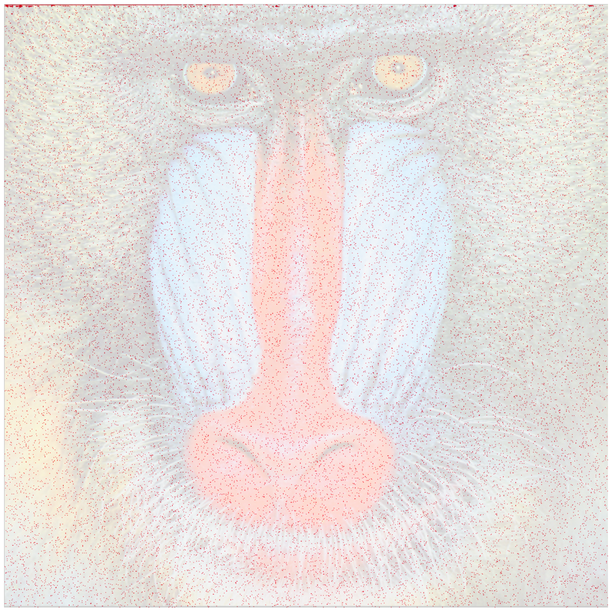
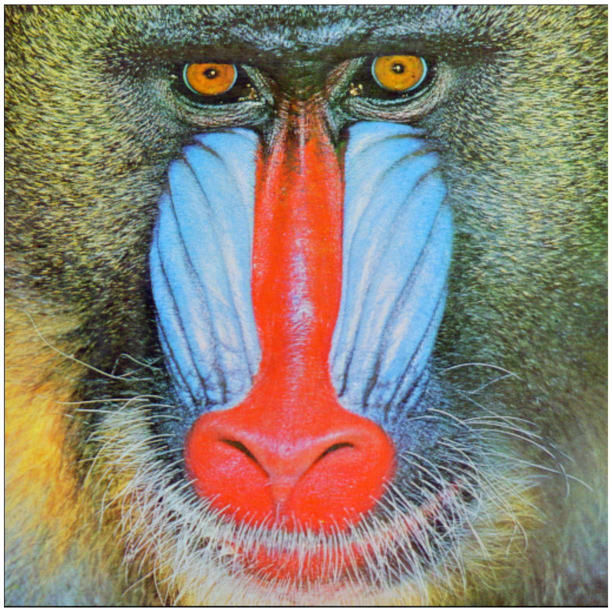
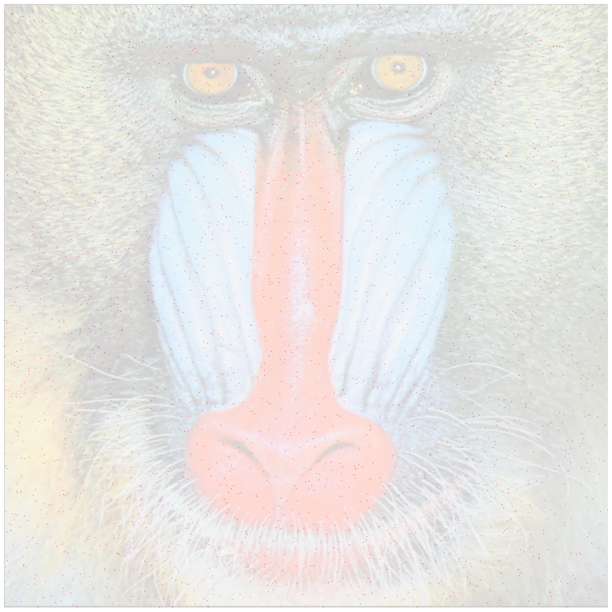

(b) Diff of Scaled Baboon Image with Correct Image

Figure 4: Outlier Detector with Discard Strategy



(a) Scaled Baboon Image



(b) Diff of Scaled Baboon Image with Correct Image

Figure 5: Outlier Detector with Reexecution Strategy

8.1.2 Output Positions for Water

Below are the final positions for the water molecules generated from the scale benchmark for the following scenarios: (1) No Outlier Detector (2) Outlier Detector with Reexecution (3) Outlier Detector with Discard. The raw output image is on the left side of each entry. In each entry, the upper left plot is the left hydrogen molecule, the upper right plot is the right hydrogen molecule and the bottom plot is the oxygen molecule. The generated positions are black 'x's and the correct positions are red '+'s.

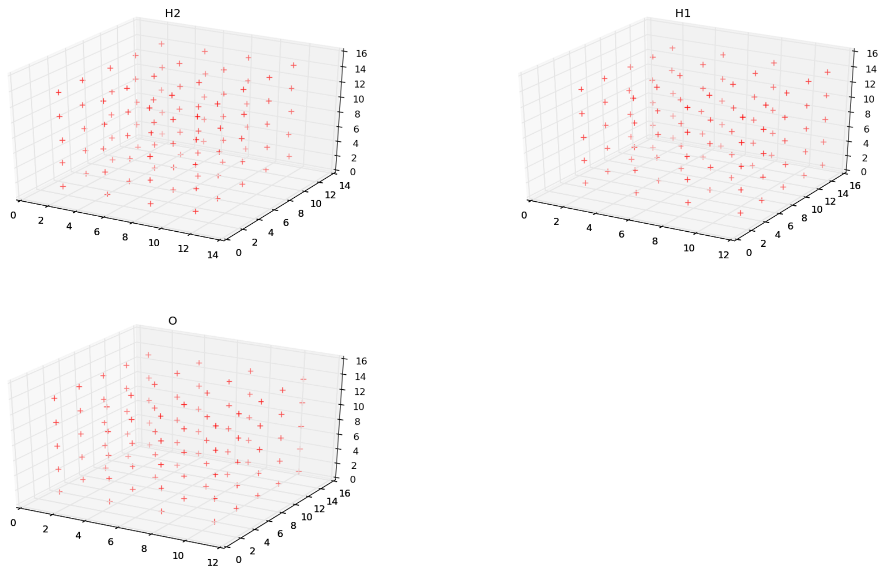


Figure 6: Molecule Positions at Last Timestep for No Outlier Detector [red:correct, blue:generated]* all generated points are NaNs, and are therefore not rendered.

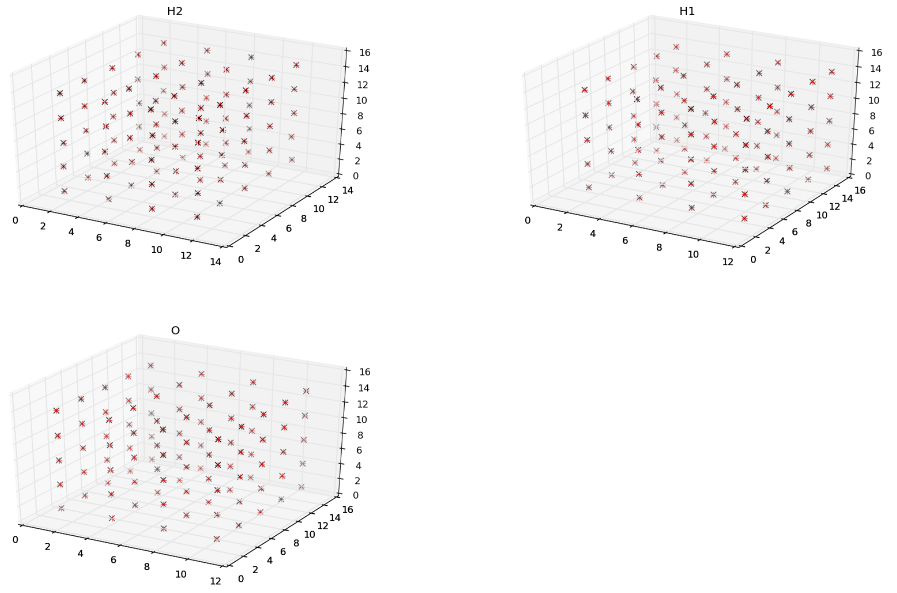


Figure 7: Molecule Positions at Last Timestep for Outlier Detector with Discard Strategy [red:correct, blue:generated]

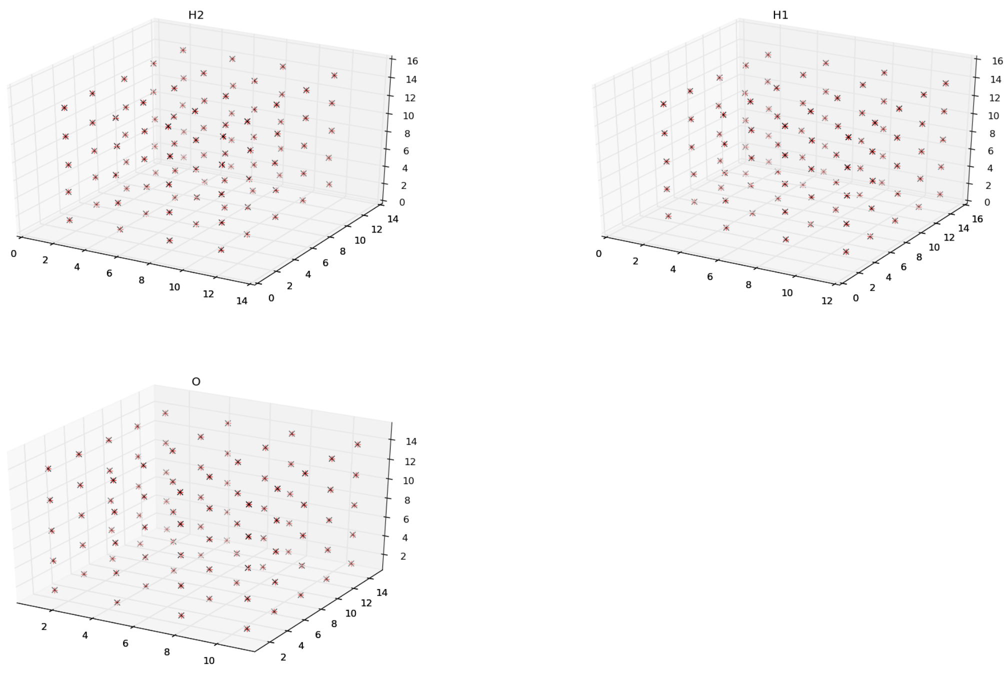


Figure 8: Molecule Positions at Last Timestep for Outlier Detector with Reexecute Stategy [red:correct, blue:generated]

8.1.3 Output Positions for Barnes

Below are the final positions for the planets generated from the scale benchmark for the following scenarios: (1) No Outlier Detector (2) Outlier Detector with Reexecution (3) Outlier Detector with Discard. The raw output image is on the left side of each entry. The generated positions are black ’x’s and the correct positions are red ’+’s.

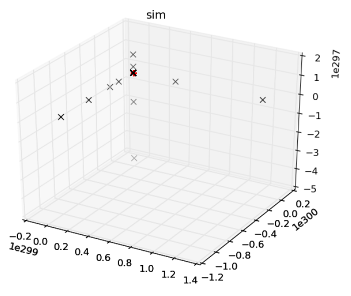


Figure 9: Planetary Positions at Last Timestep for No Outlier Detector [red:correct, black:generated]* all generated points are NaNs, and are therefore not rendered.

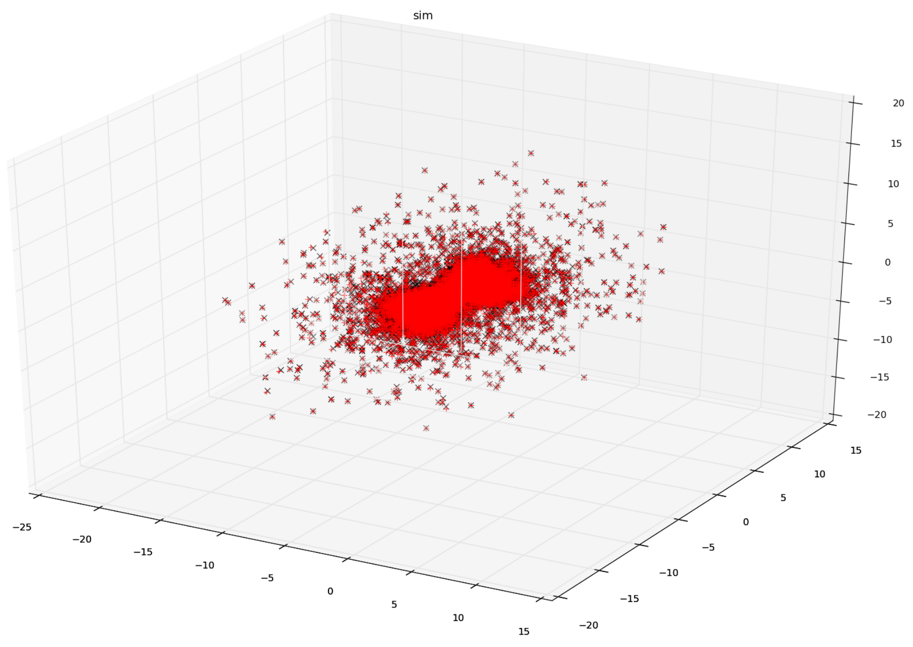


Figure 10: Planetary Positions at Last Timestep for Outlier Detector with Discard Strategy [red:correct, black:generated]

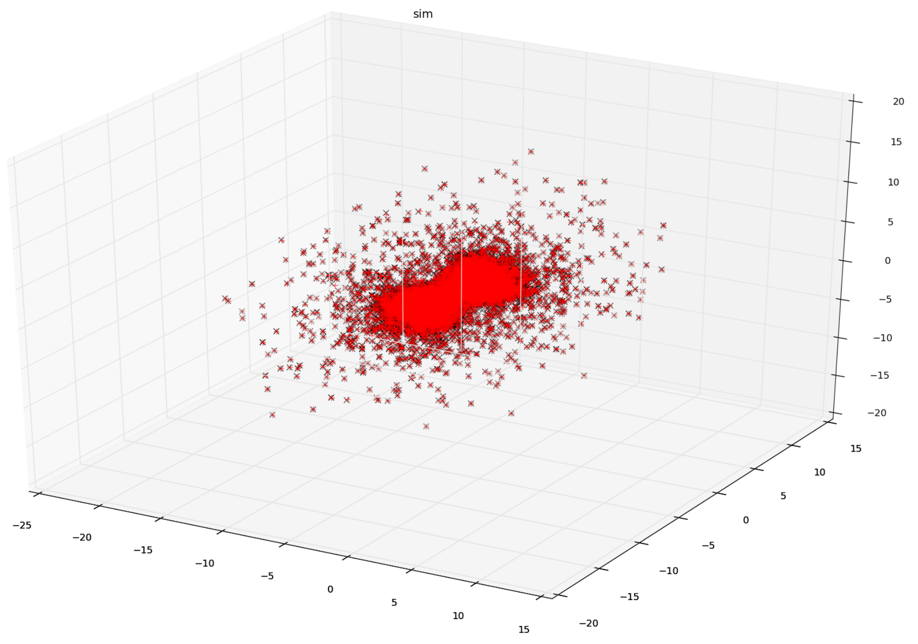


Figure 11: Planetary Positions at Last Timestep for Outlier Detector with Reexecute Strategy [red:correct, black:generated]

9. Appendix II: Outlier Detector Visualizations

Below are diagrams illustrating the outlier detector control dynamics for each benchmark output in the following scenarios:

1. No Outlier Detector
2. Outlier Detector with Reexecution
3. Outlier Detector with Discard

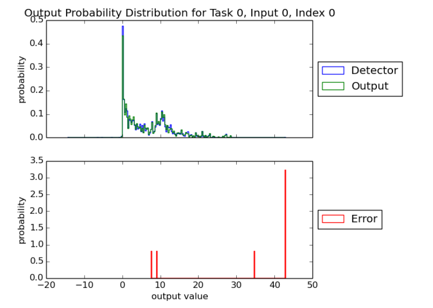
For each entry, there are three top-level visualizations:

1. **Distribution Visualization(Left)**: In this plot, the error, correct and generated (under the current scenario) are shown. The y-axis for both plots is the frequency, and the x axis is the output value.
   - *Correct Distribution (Top, Green)*: The green distribution on the top subplot is the correct distribution for a particular output.
   - *Generated Distribution (Top, Blue)*: The blue distribution on the top subplot is the distribution generated, given the current scenario.
   - *Error Distribution (Bottom, Red)*: The red distribution on the bottom subplot is the error distribution. Both the top and the bottom subplots share the same x axis - which is labelled under the bottom subplot.
2. **Outlier Detector Bounds(Middle)**: In this plot, the evolution of the outlier detector bounds over time and the reexecution rate are shown. In both plots, the x axis is time. In the upper plot, the y axis is output value. In the lower plot the y axis is the reexecution rate.
   - *Upper Bound (Top, Green)*: The green line is the upper bound of the outlier detector over time.
   - *Average (Top, Blue)*: The blue line is the mean of the output over time.
   - *Lower Bound (Top, Red)*: The red line is the lower bound of the outlier detector over time.
   - *Right of Mean Reexecution Rate (Bottom, Green)*: The green line is reexecution rate of outputs that fall right of the mean.
   - *Average Reexecution Rate (Bottom, Blue)*: The blue line is the average reexecution rate.
   - *Left of Mean Reexecution Rate (Bottom, Red)*: The red line is the reexecution rate of outputs that fall left of the mean..

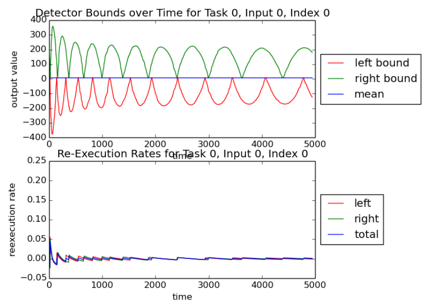
3. **Outlier Detector Bounds overlay on Distributions (Right)**: In this plot, the evolution of the outlier detector bounds over time superimposed on the output,generated and error distributions is shown. In both plots, the x axis is output value. In the both plots, the y axis is frequency of each output for the output distributions (in Red, Blue, Green).
   - *Correct Distribution (Top, Green)*: The green distribution on the top subplot is the correct distribution for a particular output.
   - *Generated Distribution (Top, Blue)*: The blue distribution on the top subplot is the distribution generated, given the current scenario.
   - *Error Distribution (Bottom, Red)*: The red distribution on the bottom subplot is the error distribution. Both the top and the bottom subplots share the same x axis - which is labelled under the bottom subplot.
   - *Upper Bound (Top; Bottom, Teal)*: The upper bound of the outlier detector over time (y axis).
   - *Average (Top; Bottom, Black)*: The mean of the output over time.
   - *Lower Bound (Top; Bottom, Pink)*: The lower bound of the outlier detector over time (y axis).

9.1 Outlier Detector with Discard Strategy

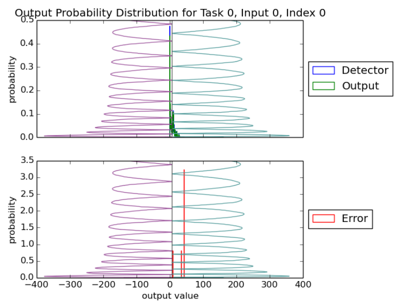
9.1.1 BlackScholes



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



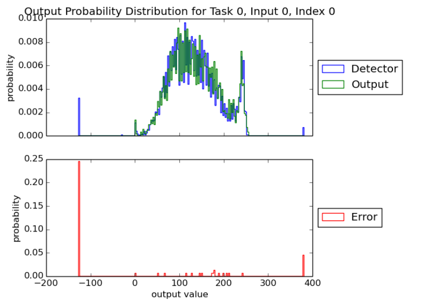
(b) Bounds Evolution and Reexecution Rate over Time



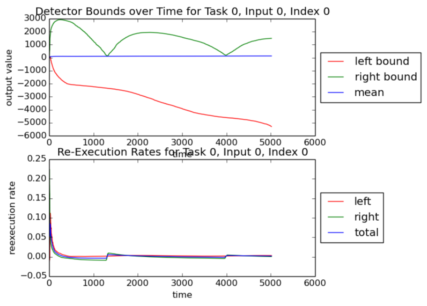
(c) Bounds Evolution over Distribution

Figure 12: BlackScholes: Outlier Detector with Discard Strategy Behavior for Price

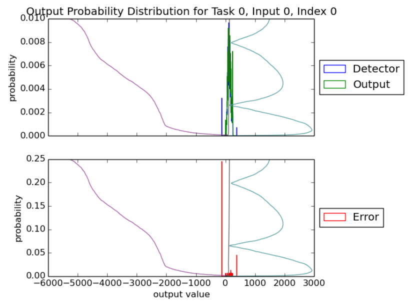
9.1.2 Scale



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

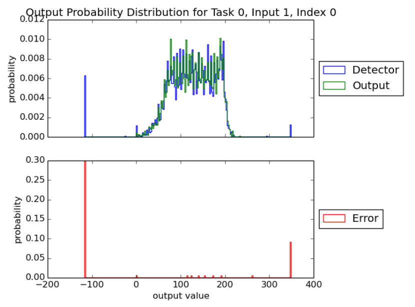


(b) Bounds Evolution and Reexecution Rate over Time

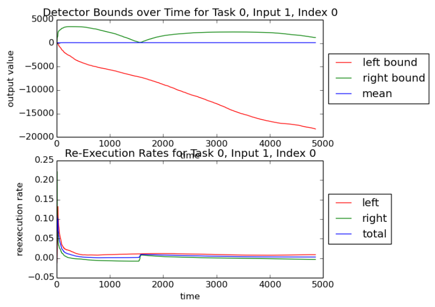


(c) Bounds Evolution over Distribution

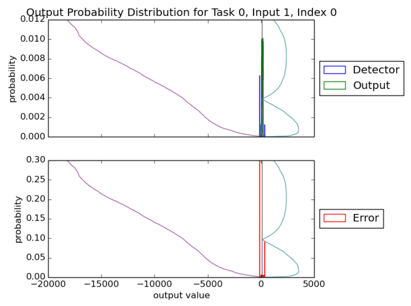
Figure 13: Scale: Outlier Detector with Discard Strategy Behavior for Red Component



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

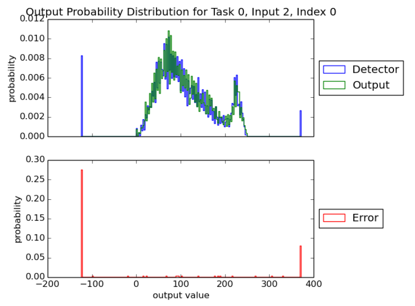


(b) Bounds Evolution and Reexecution Rate over Time

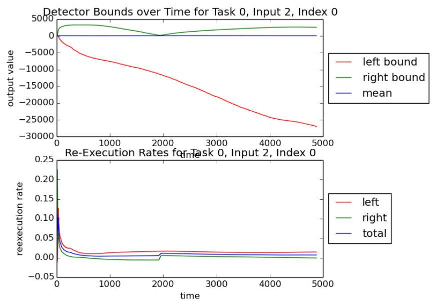


(c) Bounds Evolution over Distribution

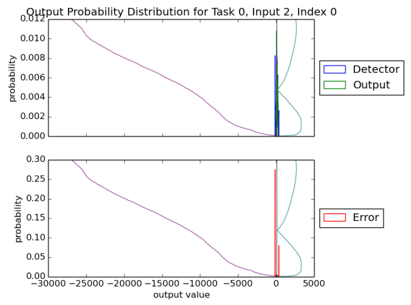
Figure 14: Scale: Outlier Detector with Discard Strategy Behavior for Blue Component



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



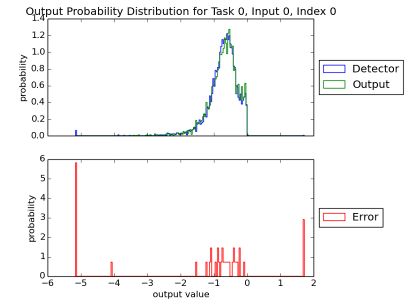
(b) Bounds Evolution and Reexecution Rate over Time



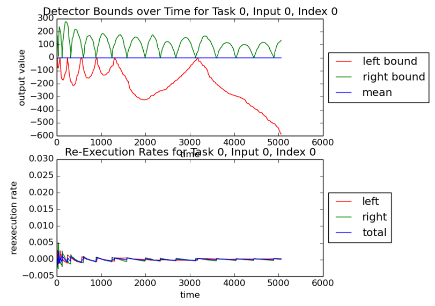
(c) Bounds Evolution over Distribution

Figure 15: Scale: Outlier Detector with Discard Strategy Behavior for Green Component

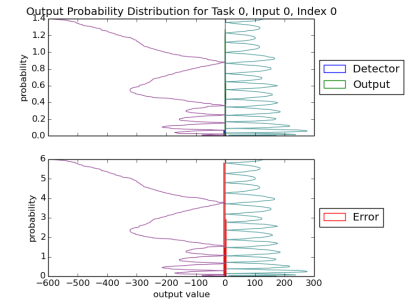
9.1.3 Search



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

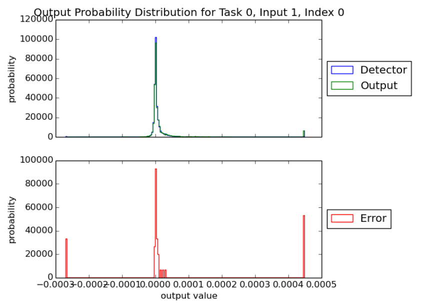


(b) Bounds Evolution and Reexecution Rate over Time

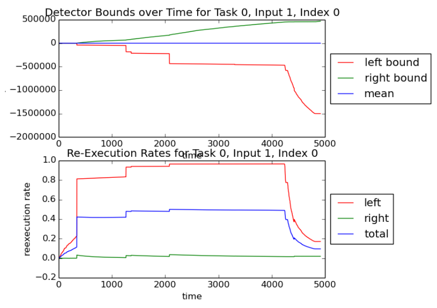


(c) Bounds Evolution over Distribution

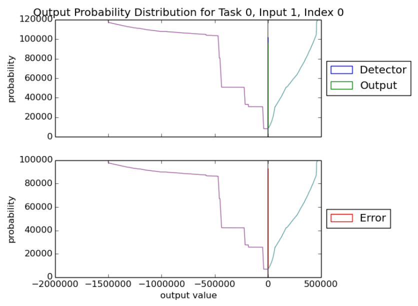
Figure 16: Search: Outlier Detector with Discard Strategy Behavior for Partial Eng Sum



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

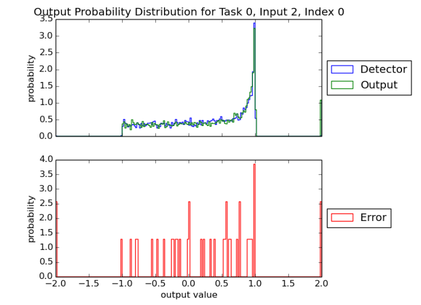


(b) Bounds Evolution and Reexecution Rate over Time

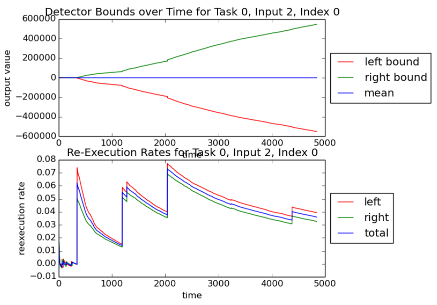


(c) Bounds Evolution over Distribution

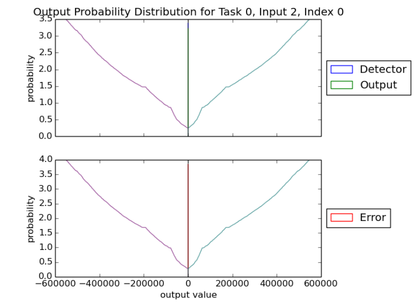
Figure 17: Search: Outlier Detector with Discard Strategy Behavior for Partial ZCorr Sum



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

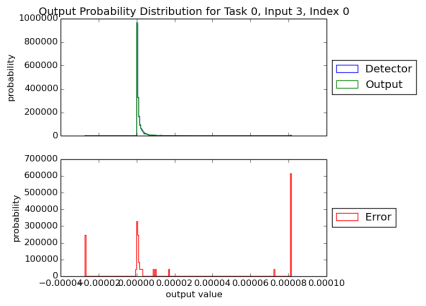


(b) Bounds Evolution and Reexecution Rate over Time

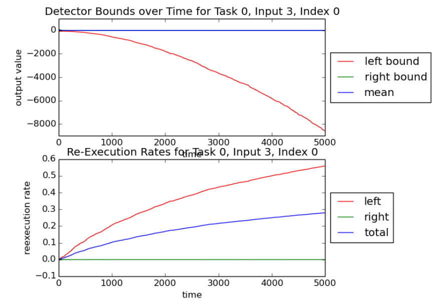


(c) Bounds Evolution over Distribution

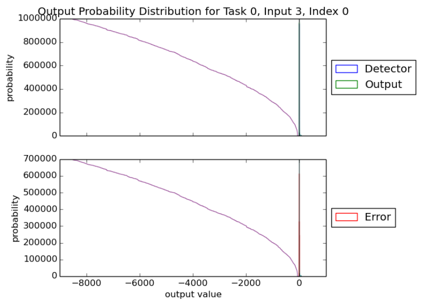
Figure 18: Search: Outlier Detector with Discard Strategy Behavior for CZ



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

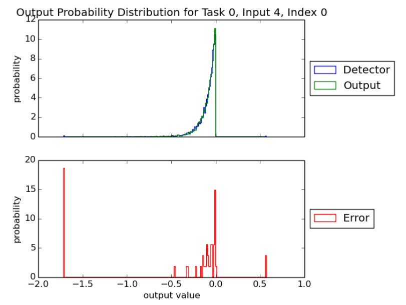


(b) Bounds Evolution and Reexecution Rate over Time

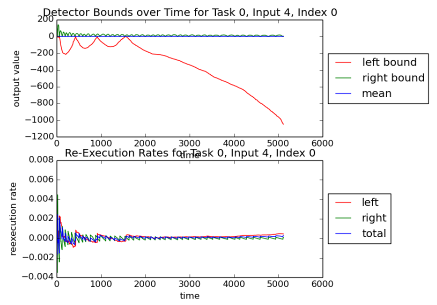


(c) Bounds Evolution over Distribution

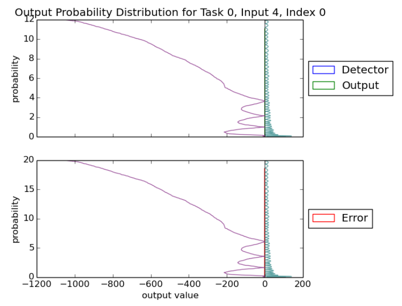
Figure 19: Search: Outlier Detector with Discard Strategy Behavior for Step



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



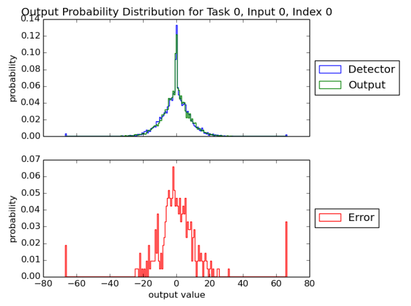
(b) Bounds Evolution and Reexecution Rate over Time



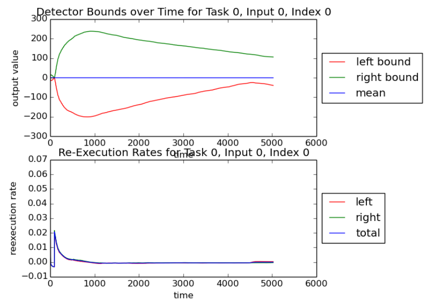
(c) Bounds Evolution over Distribution

Figure 20: Search: Outlier Detector with Discard Strategy Behavior for DE

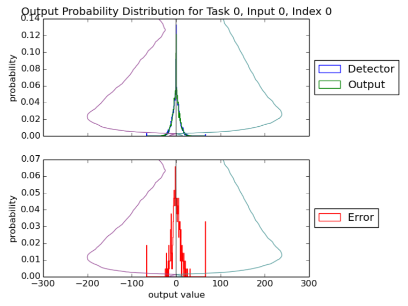
9.1.4 Barnes



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

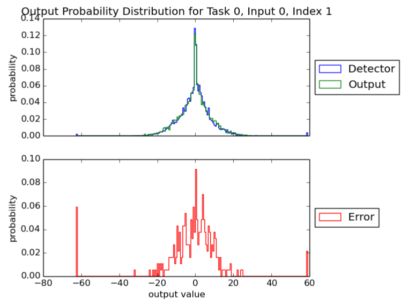


(b) Bounds Evolution and Reexecution Rate over Time

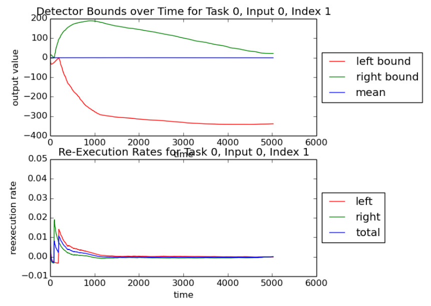


(c) Bounds Evolution over Distribution

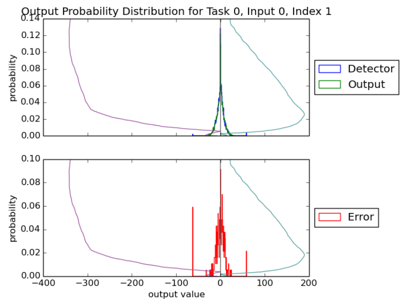
Figure 21: Barnes: Outlier Detector with Discard Strategy Behavior for Velocity[0]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

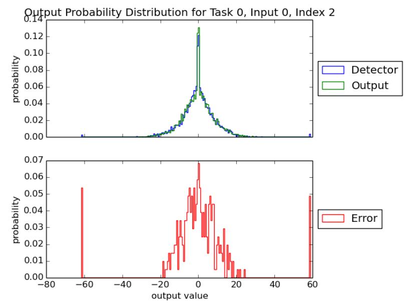


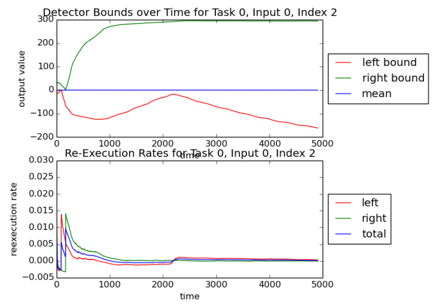
(b) Bounds Evolution and Reexecution Rate over Time



(c) Bounds Evolution over Distribution

Figure 22: Barnes: Outlier Detector with Discard Strategy Behavior for Velocity[1]


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

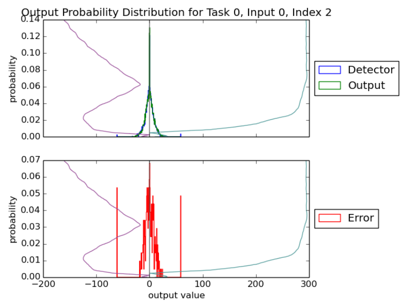
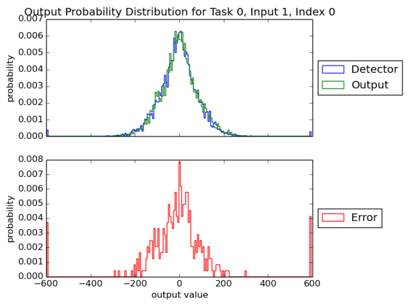
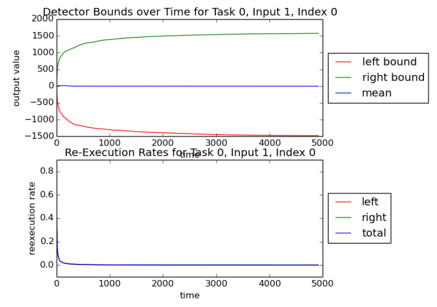

(c) Bounds Evolution over Distribution

Figure 23: Barnes: Outlier Detector with Discard Strategy Behavior for Velocity[2]


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

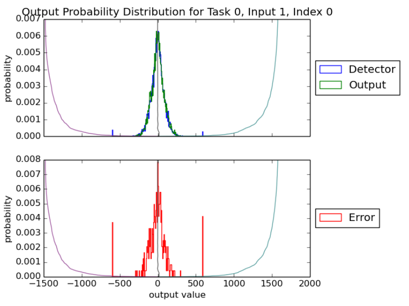
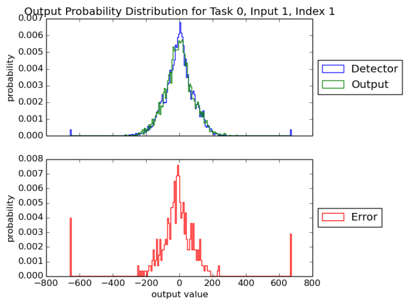
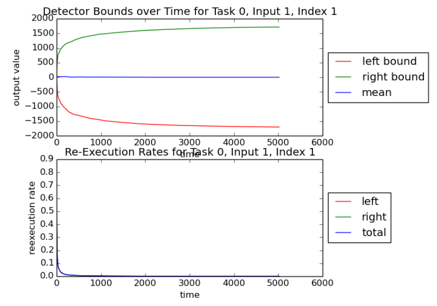

(c) Bounds Evolution over Distribution

Figure 24: Barnes: Outlier Detector with Discard Strategy Behavior for Acceleration[0]


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

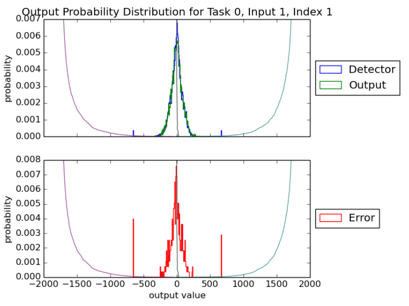
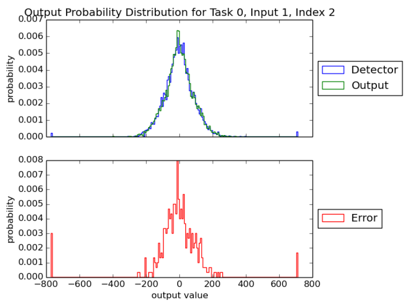
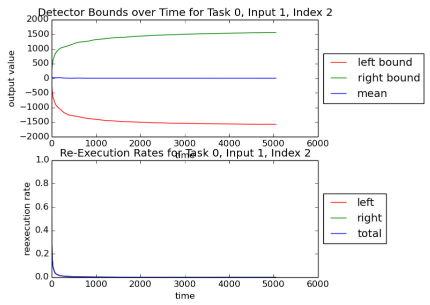

(c) Bounds Evolution over Distribution

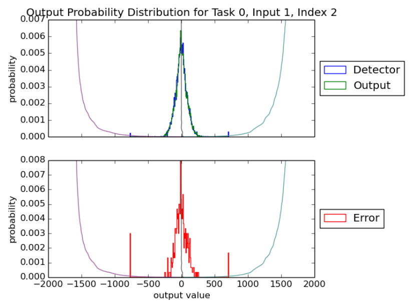
Figure 25: Barnes: Outlier Detector with Discard Strategy Behavior for Acceleration[1]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

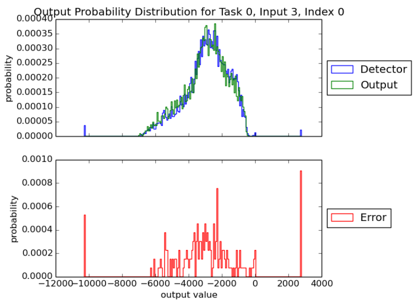


(b) Bounds Evolution and Reexecution Rate over Time

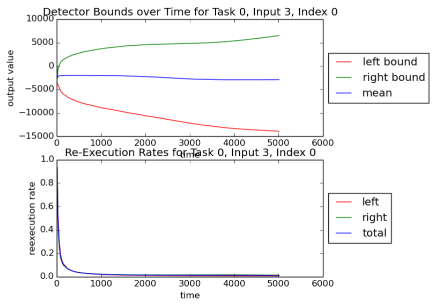


(c) Bounds Evolution over Distribution

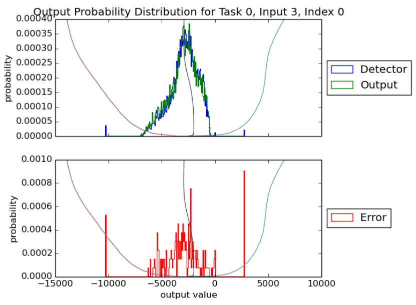
Figure 26: Barnes: Outlier Detector with Discard Strategy Behavior for Acceleration[2]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



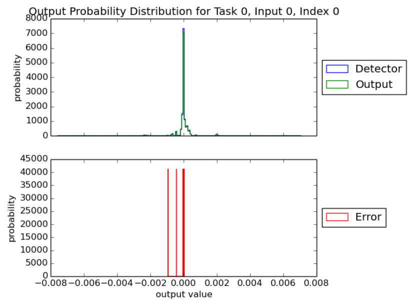
(b) Bounds Evolution and Reexecution Rate over Time



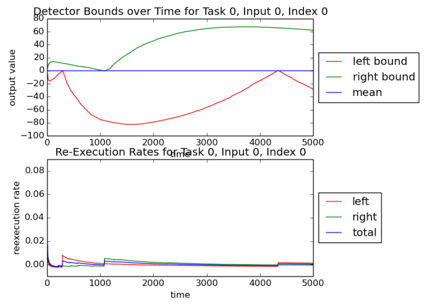
(c) Bounds Evolution over Distribution

Figure 27: Barnes: Outlier Detector with Discard Strategy Behavior for Phi

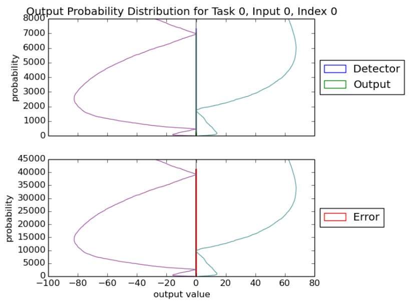
9.1.5 Water



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

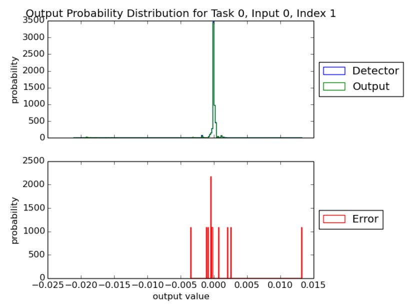


(b) Bounds Evolution and Reexecution Rate over Time

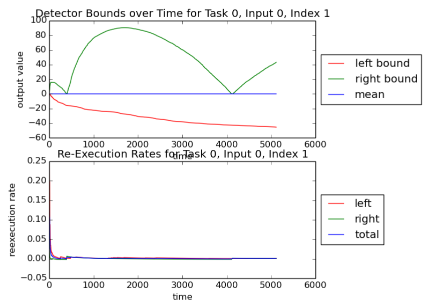


(c) Bounds Evolution over Distribution

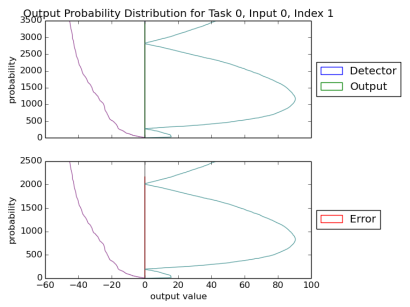
Figure 28: Water: Outlier Detector with Discard Strategy Behavior for Res1[0] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

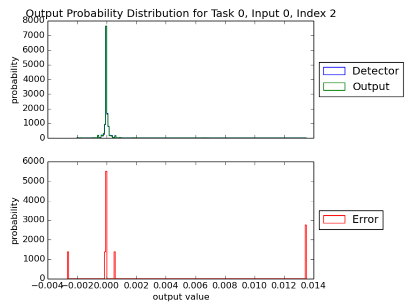


(b) Bounds Evolution and Reexecution Rate over Time

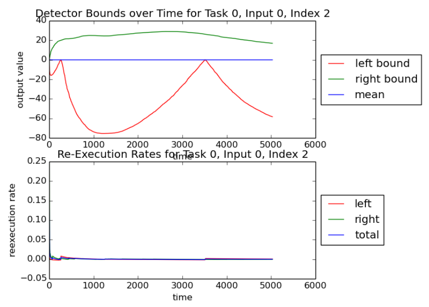


(c) Bounds Evolution over Distribution

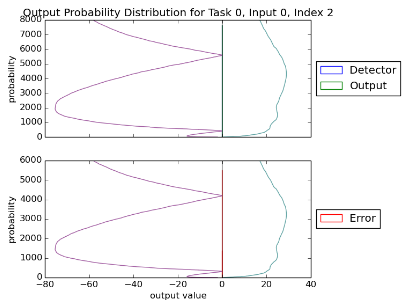
Figure 29: Water: Outlier Detector with Discard Strategy Behavior for Res1[1] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

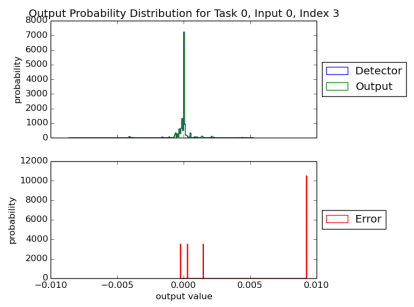


(b) Bounds Evolution and Reexecution Rate over Time

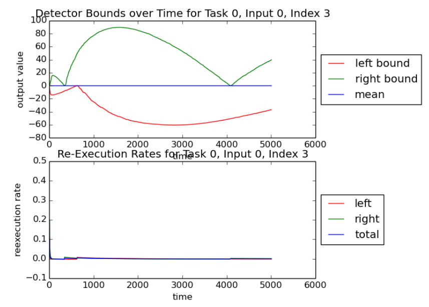


(c) Bounds Evolution over Distribution

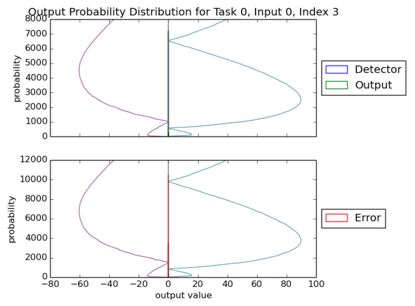
Figure 30: Water: Outlier Detector with Discard Strategy Behavior for Res1[2] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

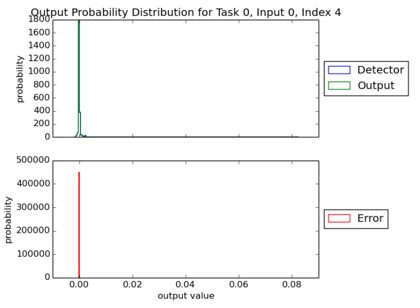


(b) Bounds Evolution and Reexecution Rate over Time

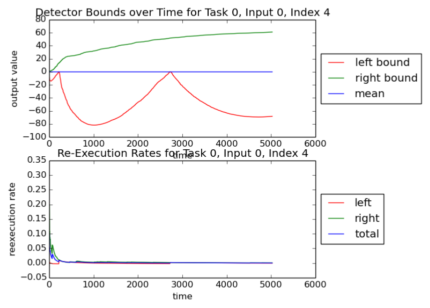


(c) Bounds Evolution over Distribution

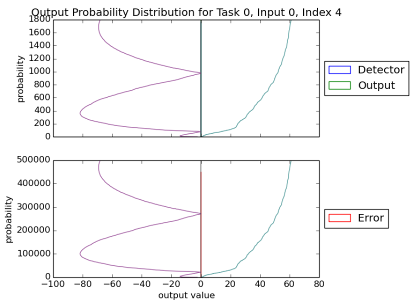
Figure 31: Water: Outlier Detector with Discard Strategy Behavior for Res1[3] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

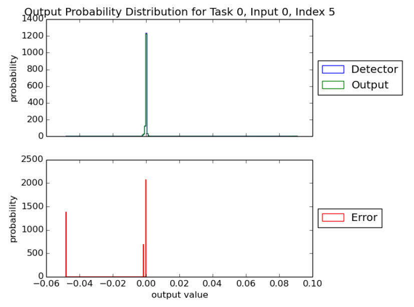


(b) Bounds Evolution and Reexecution Rate over Time

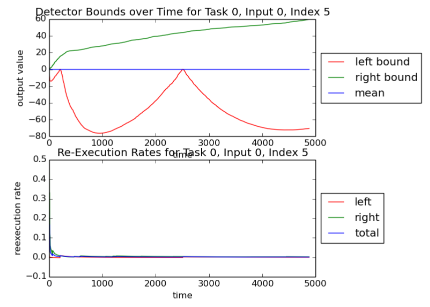


(c) Bounds Evolution over Distribution

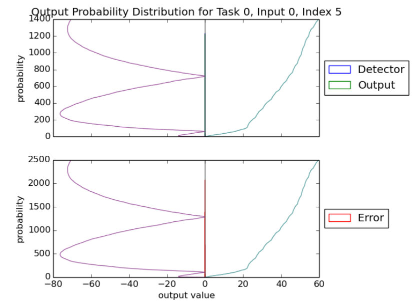
Figure 32: Water: Outlier Detector with Discard Strategy Behavior for Res1[4] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

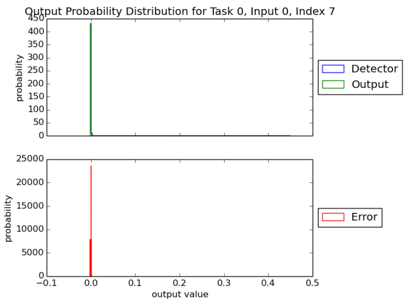


(b) Bounds Evolution and Reexecution Rate over Time

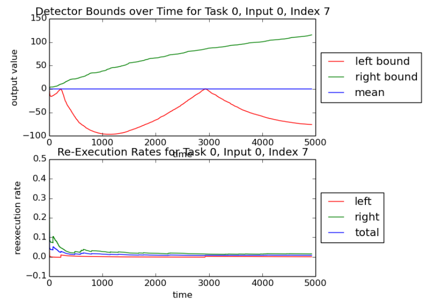


(c) Bounds Evolution over Distribution

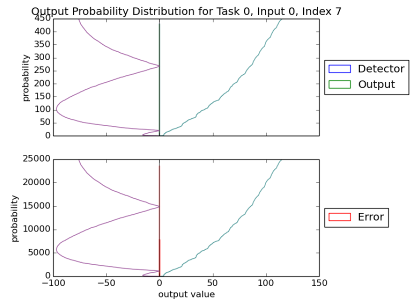
Figure 33: Water: Outlier Detector with Discard Strategy Behavior for Res1[5] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

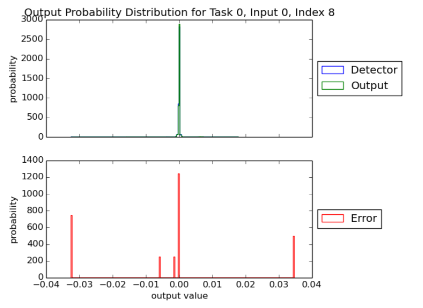


(b) Bounds Evolution and Reexecution Rate over Time

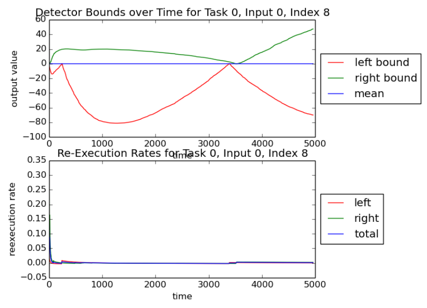


(c) Bounds Evolution over Distribution

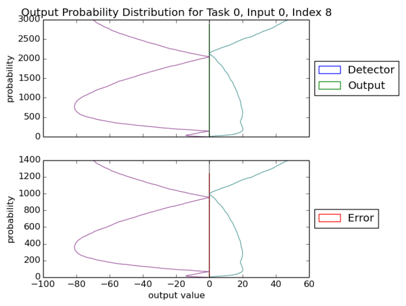
Figure 34: Water: Outlier Detector with Discard Strategy Behavior for Res1[7] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

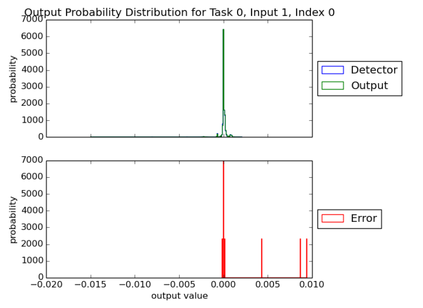


(b) Bounds Evolution and Reexecution Rate over Time

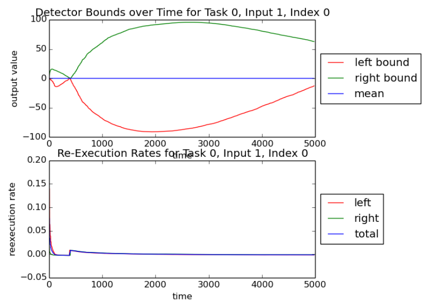


(c) Bounds Evolution over Distribution

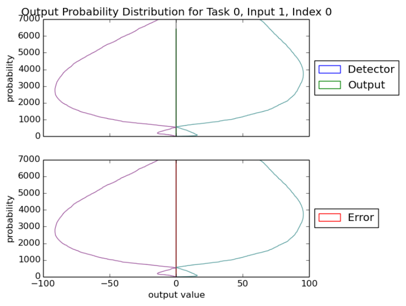
Figure 35: Water: Outlier Detector with Discard Strategy Behavior for Res1[8] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

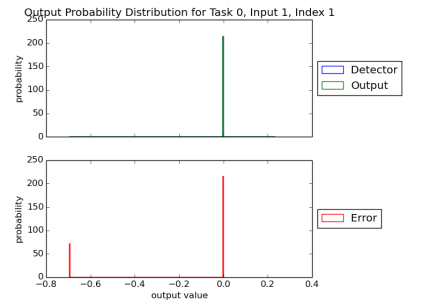


(b) Bounds Evolution and Reexecution Rate over Time

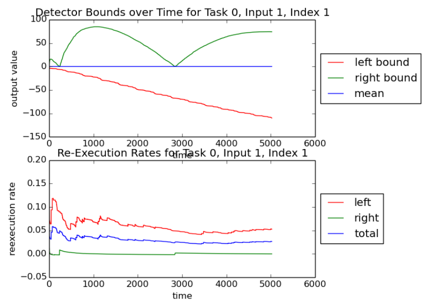


(c) Bounds Evolution over Distribution

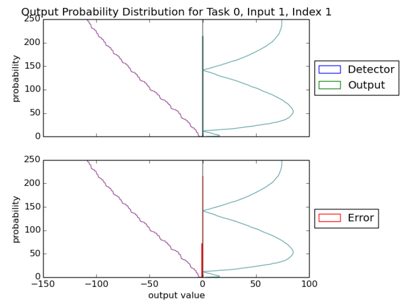
Figure 36: Water: Outlier Detector with Discard Strategy Behavior for Res2[0] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

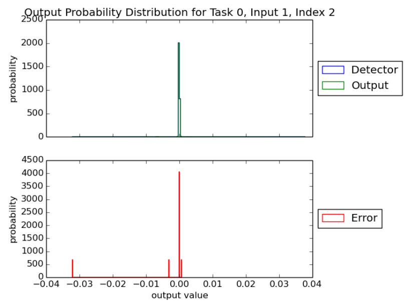


(b) Bounds Evolution and Reexecution Rate over Time

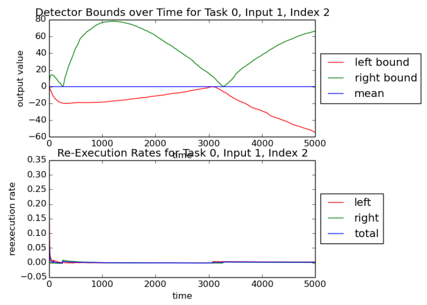


(c) Bounds Evolution over Distribution

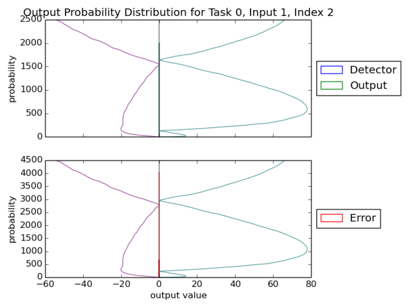
Figure 37: Water: Outlier Detector with Discard Strategy Behavior for Res2[1] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

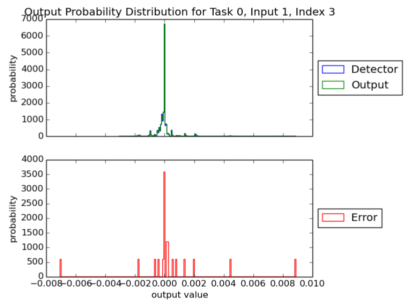


(b) Bounds Evolution and Reexecution Rate over Time

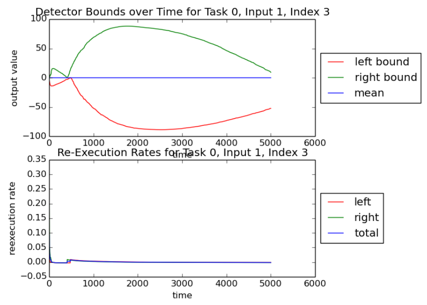


(c) Bounds Evolution over Distribution

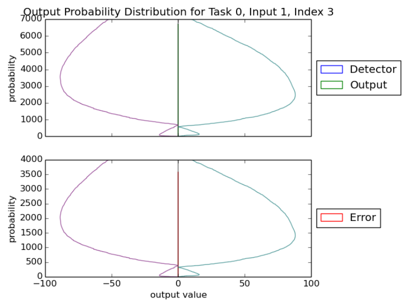
Figure 38: Water: Outlier Detector with Discard Strategy Behavior for Res2[2] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

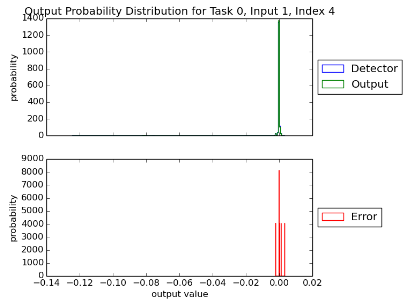


(b) Bounds Evolution and Reexecution Rate over Time

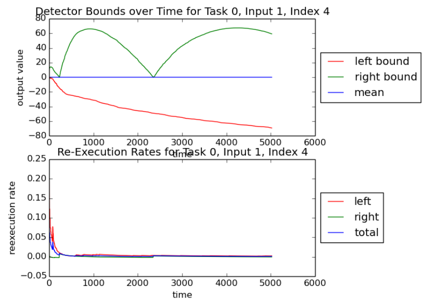


(c) Bounds Evolution over Distribution

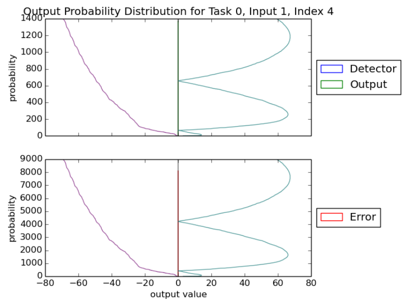
Figure 39: Water: Outlier Detector with Discard Strategy Behavior for Res2[3] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

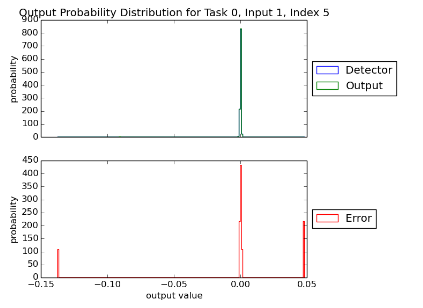


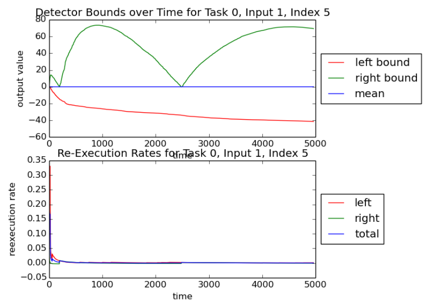
(b) Bounds Evolution and Reexecution Rate over Time



(c) Bounds Evolution over Distribution

Figure 40: Water: Outlier Detector with Discard Strategy Behavior for Res2[4] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

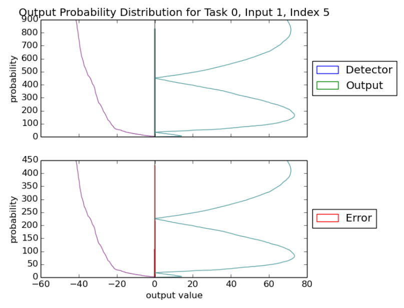
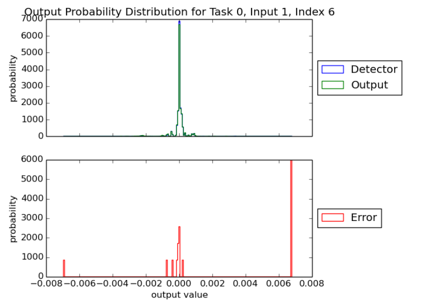
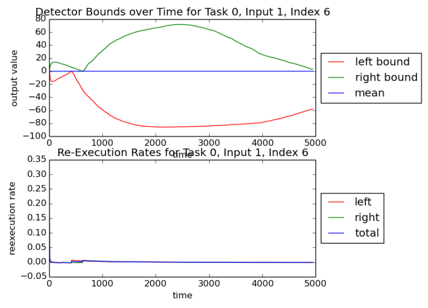

(c) Bounds Evolution over Distribution

Figure 41: Water: Outlier Detector with Discard Strategy Behavior for Res2[5] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

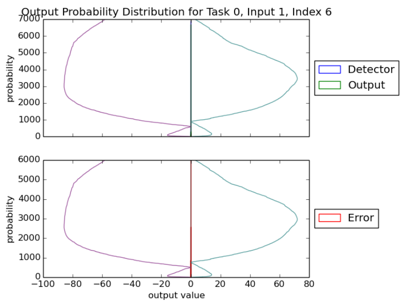
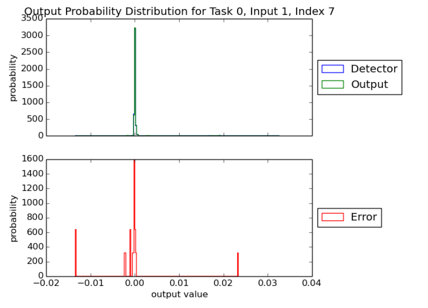
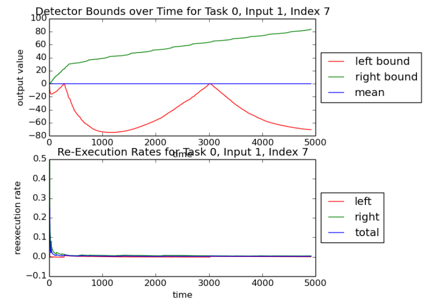

(c) Bounds Evolution over Distribution

Figure 42: Water: Outlier Detector with Discard Strategy Behavior for Res2[6] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

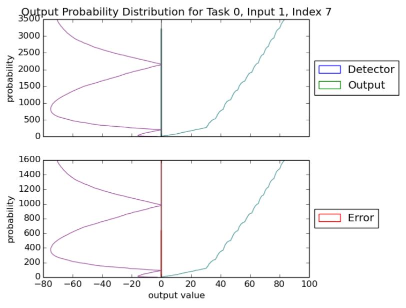
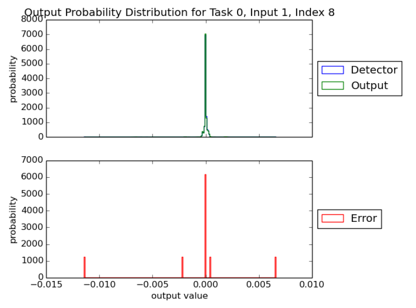
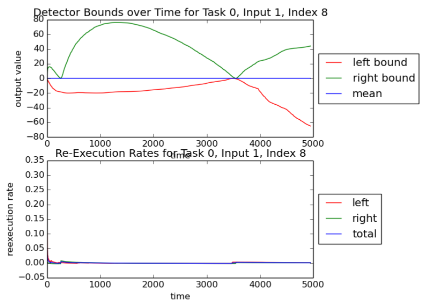

(c) Bounds Evolution over Distribution

Figure 43: Water: Outlier Detector with Discard Strategy Behavior for Res2[7] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

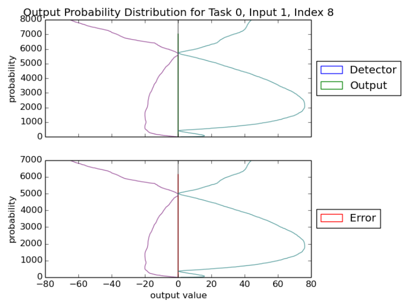
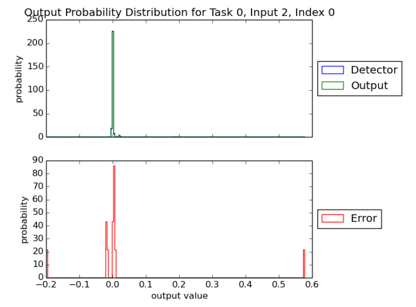
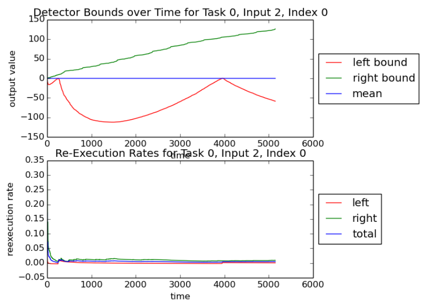

(c) Bounds Evolution over Distribution

Figure 44: Water: Outlier Detector with Discard Strategy Behavior for Res2[8] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

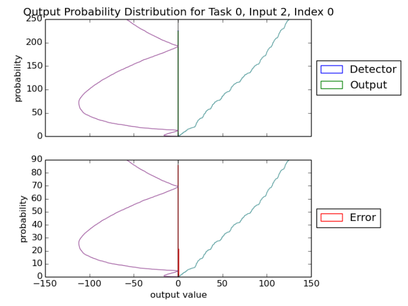

(c) Bounds Evolution over Distribution

Figure 45: Water: Outlier Detector with Discard Strategy Behavior for Incr in Intermolecular Forces Computation

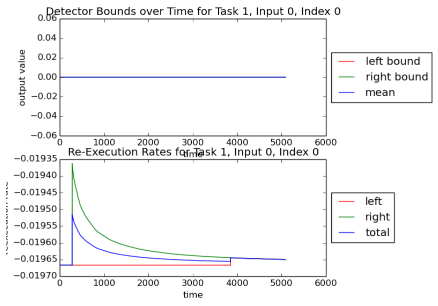
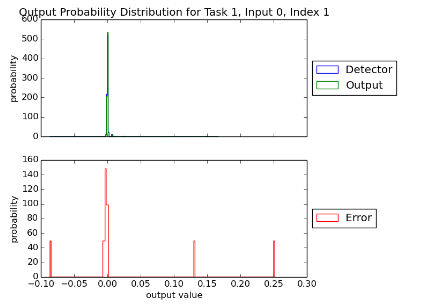
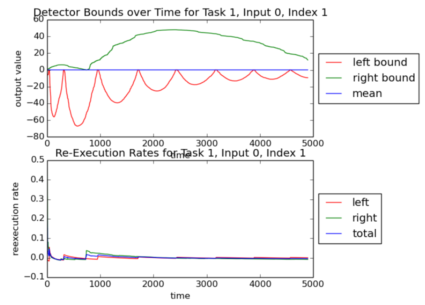

(a) Bounds Evolution and Reexecution Rate over Time

Figure 46: Water: Outlier Detector with Discard Strategy Behavior for Result[0] in Potential Energy Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

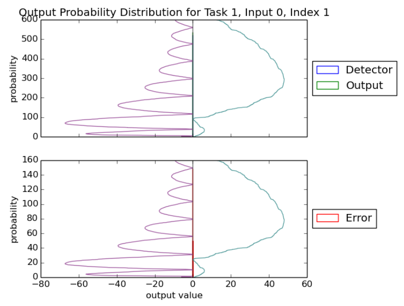
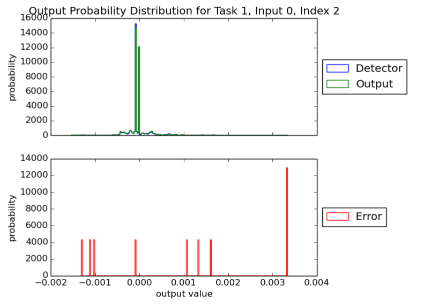
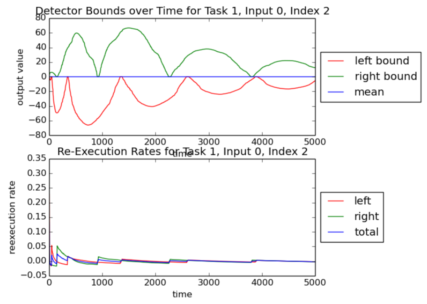

(c) Bounds Evolution over Distribution

Figure 47: Water: Outlier Detector with Discard Strategy Behavior for Result[1] in Potential Energy Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

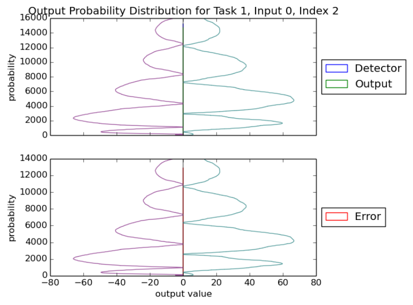
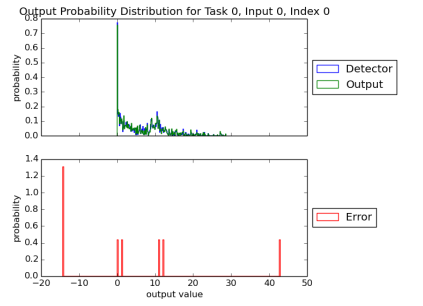
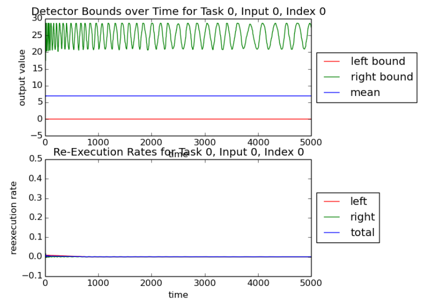

(c) Bounds Evolution over Distribution

Figure 48: Water: Outlier Detector with Discard Strategy Behavior for Result[2] in Potential Energy Computation

9.2 Outlier Detector with Reexecution Strategy

9.2.1 BlackScholes


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

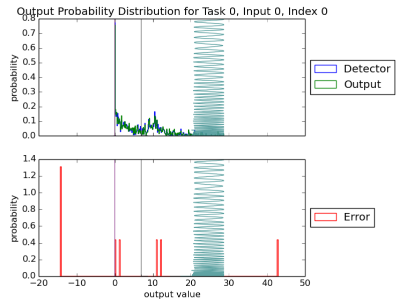
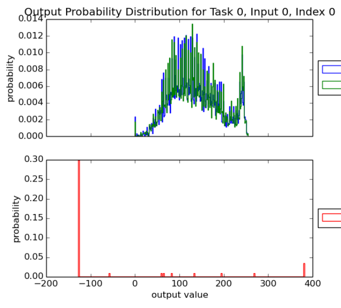
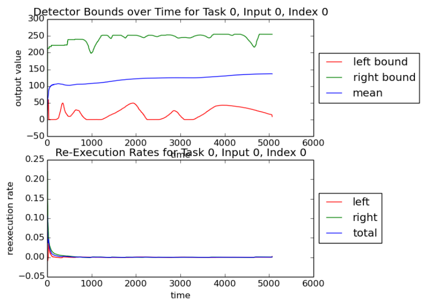

(c) Bounds Evolution over Distribution

Figure 49: BlackScholes: Outlier Detector with Reexecution Strategy Behavior for Price

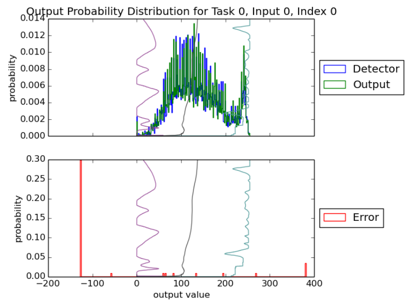
9.2.2 Scale



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

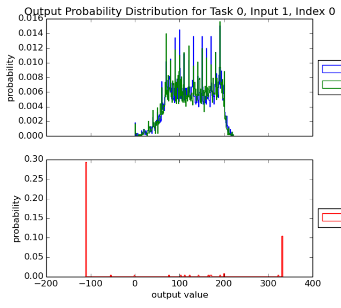


(b) Bounds Evolution and Reexecution Rate over Time

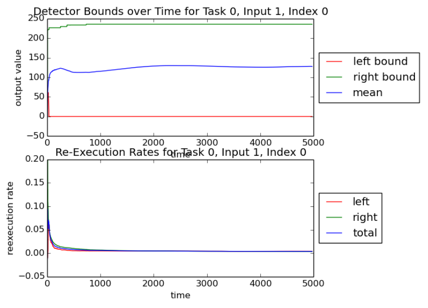


(c) Bounds Evolution over Distribution

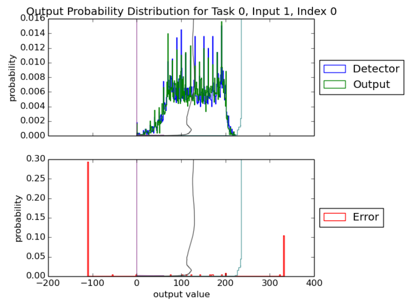
Figure 50: Scale: Outlier Detector with Reexecution Strategy Behavior for Red Component



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

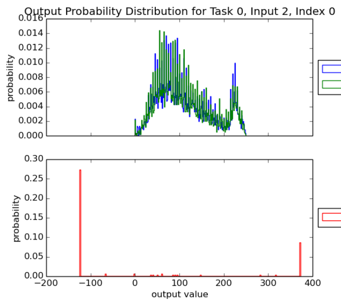


(b) Bounds Evolution and Reexecution Rate over Time

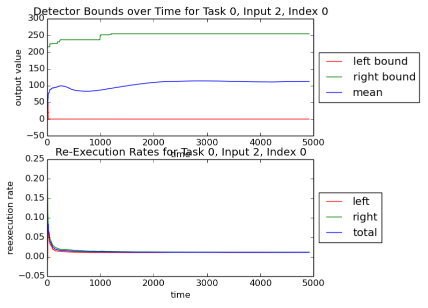


(c) Bounds Evolution over Distribution

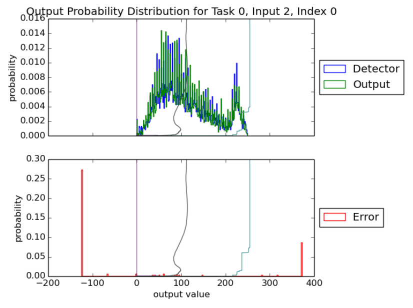
Figure 51: Scale: Outlier Detector with Reexecution Strategy Behavior for Blue Component



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



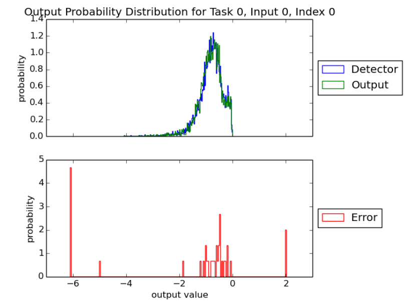
(b) Bounds Evolution and Reexecution Rate over Time



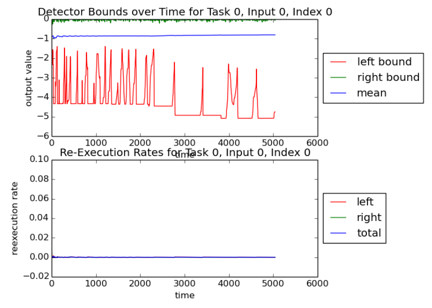
(c) Bounds Evolution over Distribution

Figure 52: Scale: Outlier Detector with Reexecution Strategy Behavior for Green Component

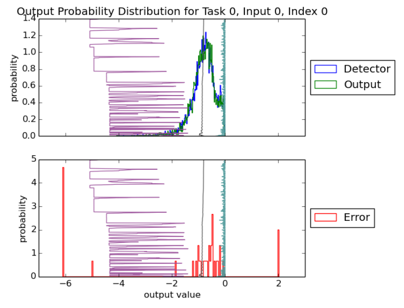
9.2.3 Search



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

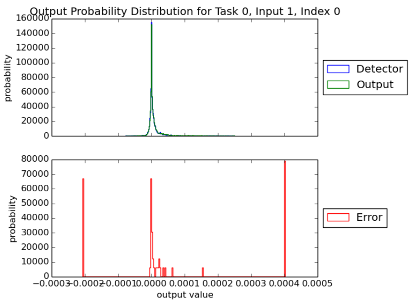


(b) Bounds Evolution and Reexecution Rate over Time

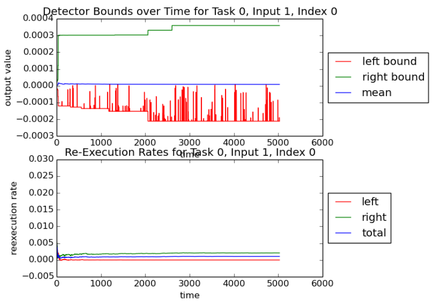


(c) Bounds Evolution over Distribution

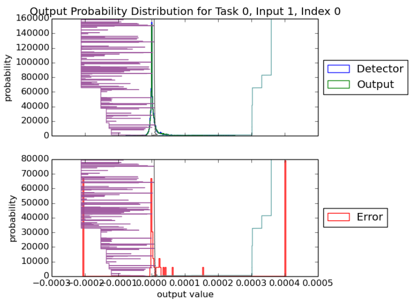
Figure 53: Search: Outlier Detector with Reexecution Strategy Behavior for Partial Eng Sum



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

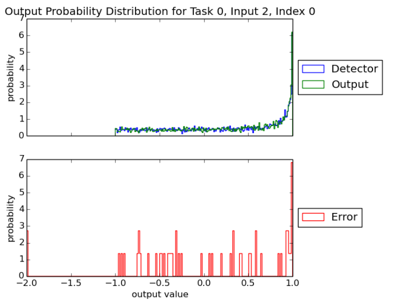


(b) Bounds Evolution and Reexecution Rate over Time

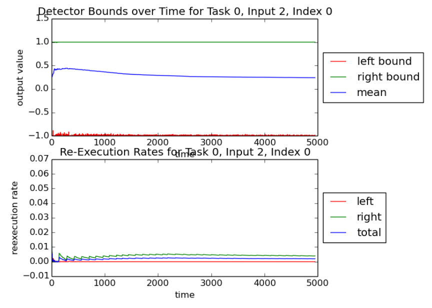


(c) Bounds Evolution over Distribution

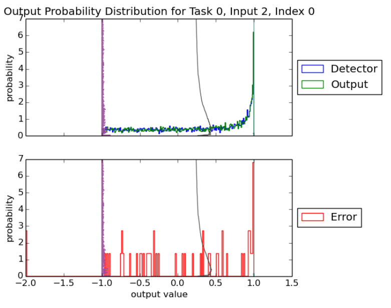
Figure 54: Search: Outlier Detector with Reexecution Strategy Behavior for Partial ZCorr Sum



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

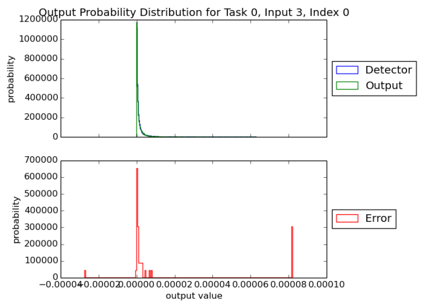


(b) Bounds Evolution and Reexecution Rate over Time

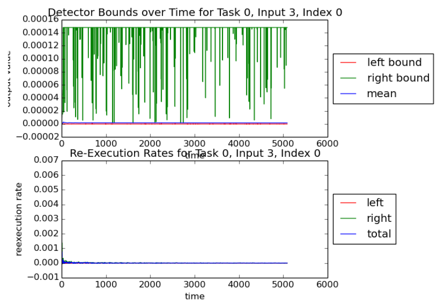


(c) Bounds Evolution over Distribution

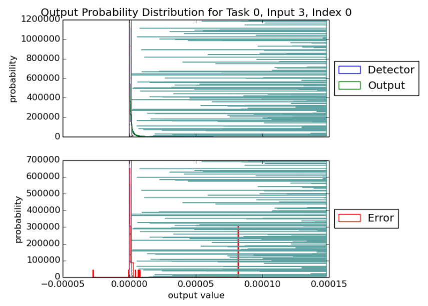
Figure 55: Search: Outlier Detector with Reexecution Strategy Behavior for CZ



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

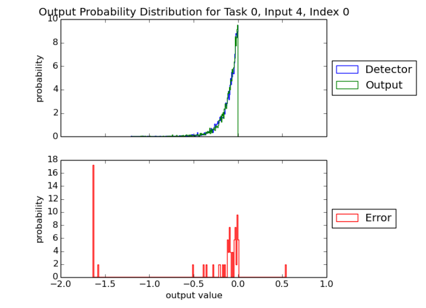


(b) Bounds Evolution and Reexecution Rate over Time

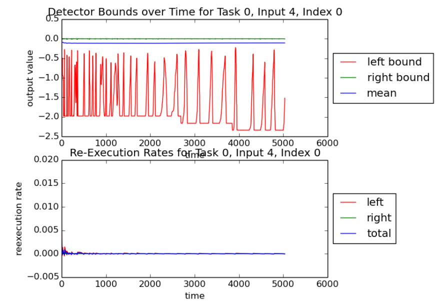


(c) Bounds Evolution over Distribution

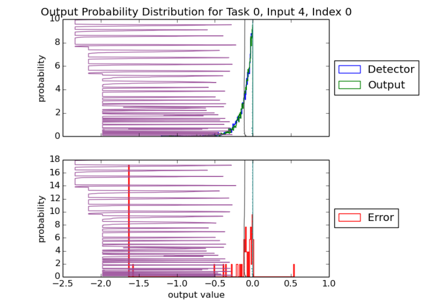
Figure 56: Search: Outlier Detector with Reexecution Strategy Behavior for Step



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



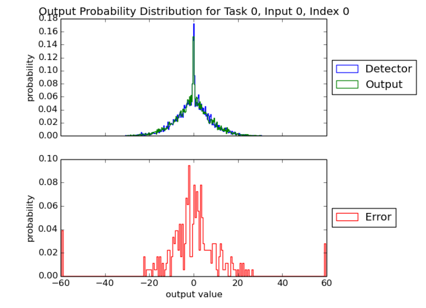
(b) Bounds Evolution and Reexecution Rate over Time



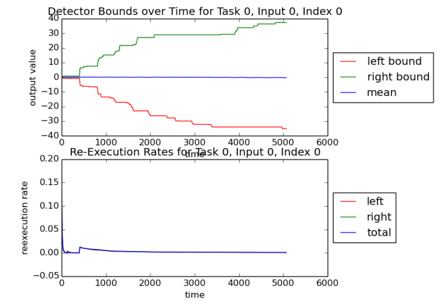
(c) Bounds Evolution over Distribution

Figure 57: Search: Outlier Detector with Reexecution Strategy Behavior for DE

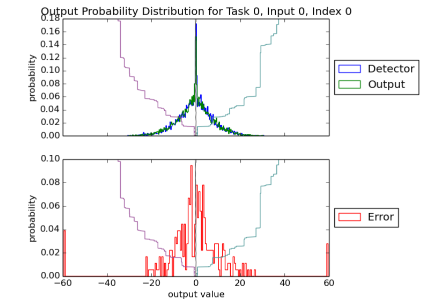
9.2.4 Barnes



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

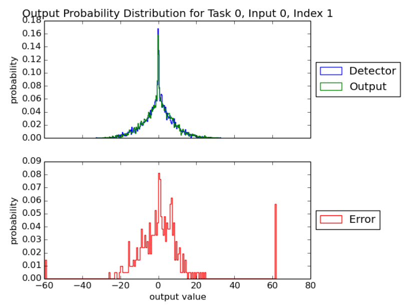


(b) Bounds Evolution and Reexecution Rate over Time

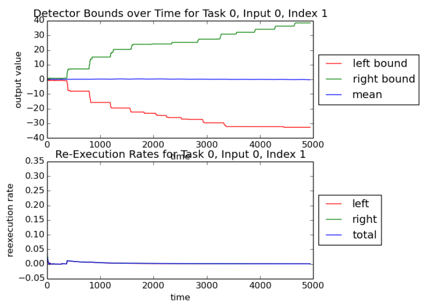


(c) Bounds Evolution over Distribution

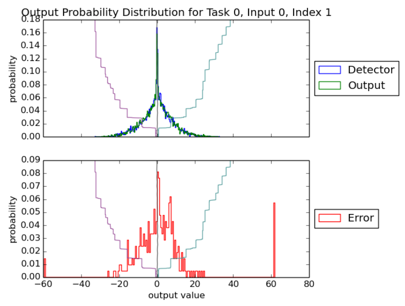
Figure 58: Barnes: Outlier Detector with Reexecution Strategy Behavior for Velocity[0]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

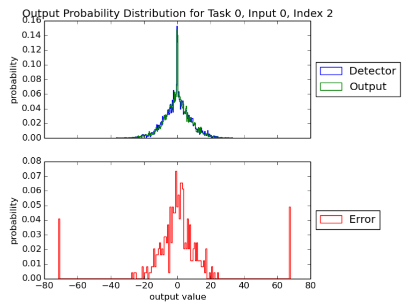


(b) Bounds Evolution and Reexecution Rate over Time

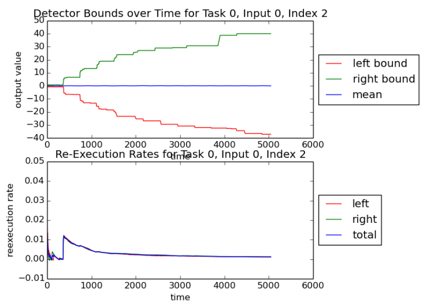


(c) Bounds Evolution over Distribution

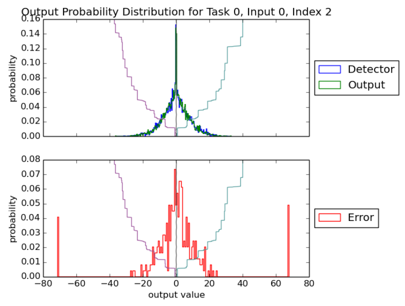
Figure 59: Barnes: Outlier Detector with Reexecution Strategy Behavior for Velocity[1]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

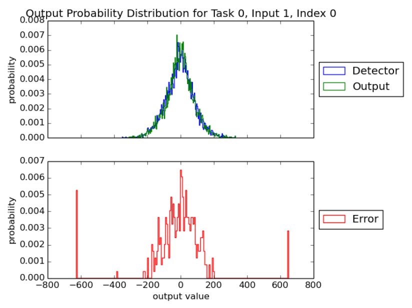


(b) Bounds Evolution and Reexecution Rate over Time

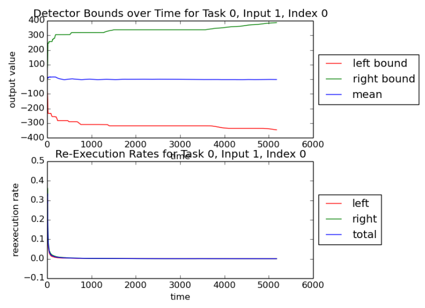


(c) Bounds Evolution over Distribution

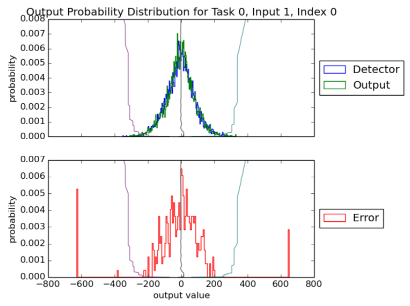
Figure 60: Barnes: Outlier Detector with Reexecution Strategy Behavior for Velocity[2]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

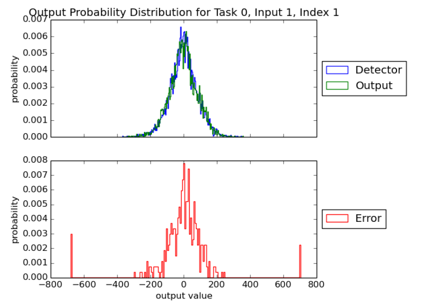


(b) Bounds Evolution and Reexecution Rate over Time

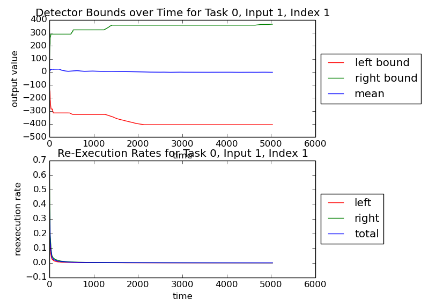


(c) Bounds Evolution over Distribution

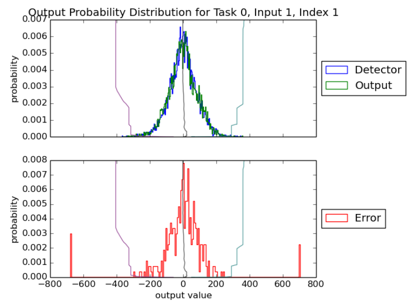
Figure 61: Barnes: Outlier Detector with Reexecution Strategy Behavior for Acceleration[0]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

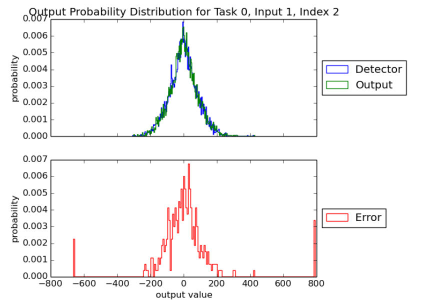


(b) Bounds Evolution and Reexecution Rate over Time

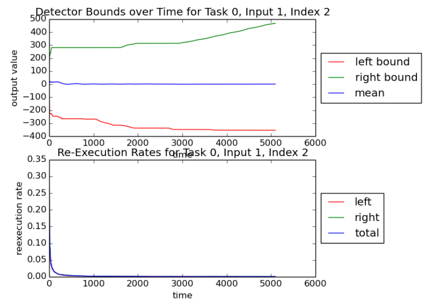


(c) Bounds Evolution over Distribution

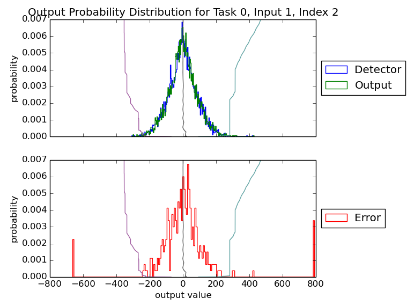
Figure 62: Barnes: Outlier Detector with Reexecution Strategy Behavior for Acceleration[1]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

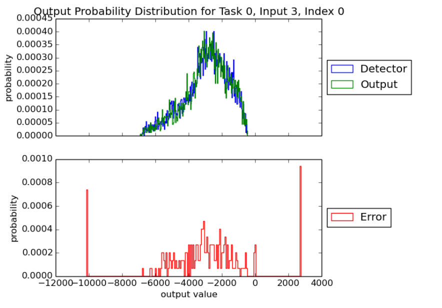


(b) Bounds Evolution and Reexecution Rate over Time

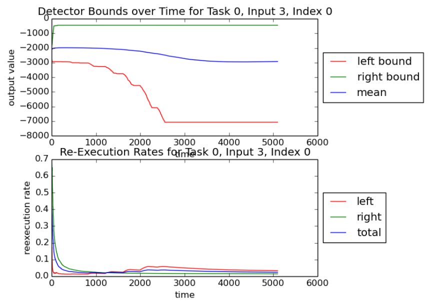


(c) Bounds Evolution over Distribution

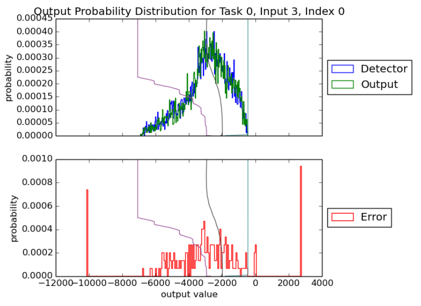
Figure 63: Barnes: Outlier Detector with Reexecution Strategy Behavior for Acceleration[2]



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



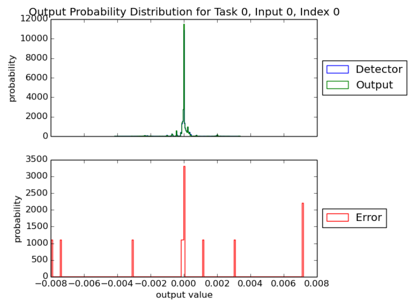
(b) Bounds Evolution and Reexecution Rate over Time



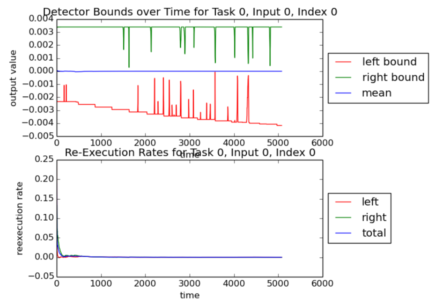
(c) Bounds Evolution over Distribution

Figure 64: Barnes: Outlier Detector with Reexecution Strategy Behavior for Phi

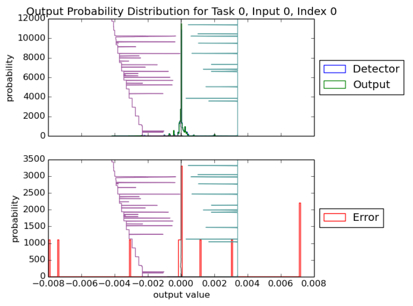
9.2.5 Water



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

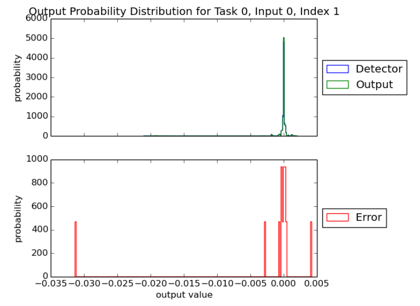


(b) Bounds Evolution and Reexecution Rate over Time

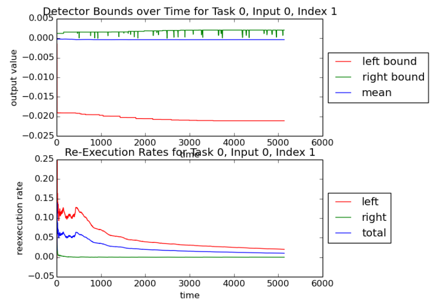


(c) Bounds Evolution over Distribution

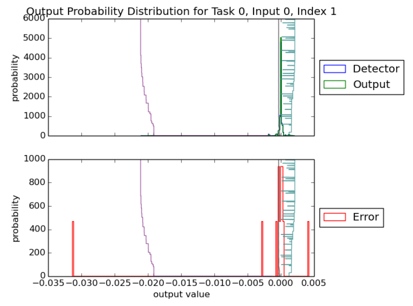
Figure 65: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[0] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

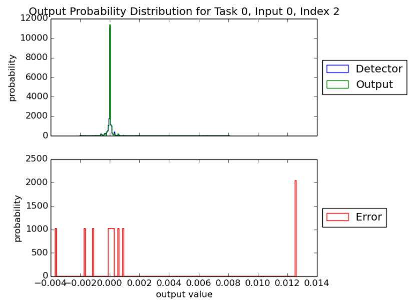


(b) Bounds Evolution and Reexecution Rate over Time

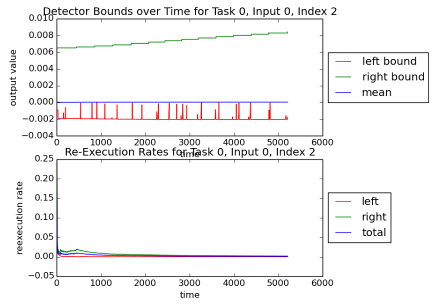


(c) Bounds Evolution over Distribution

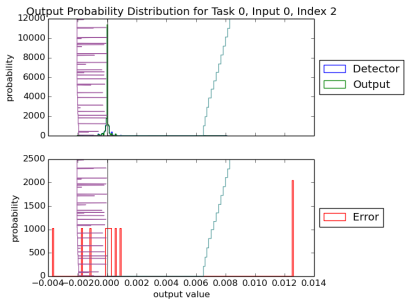
Figure 66: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[1] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

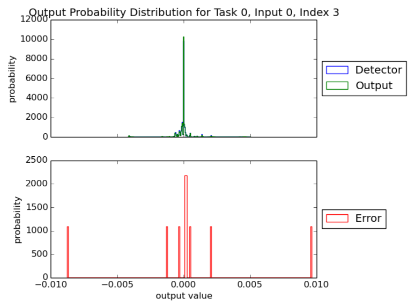


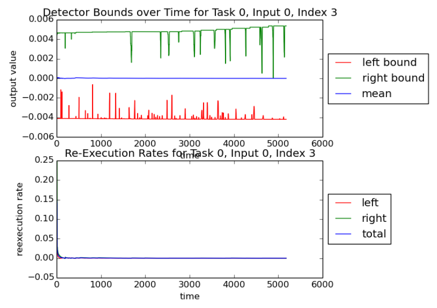
(b) Bounds Evolution and Reexecution Rate over Time



(c) Bounds Evolution over Distribution

Figure 67: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[2] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

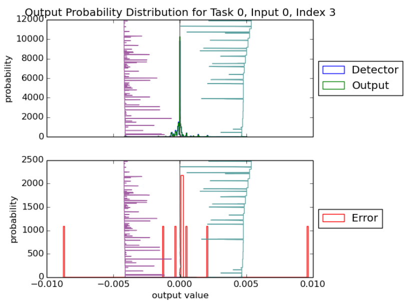
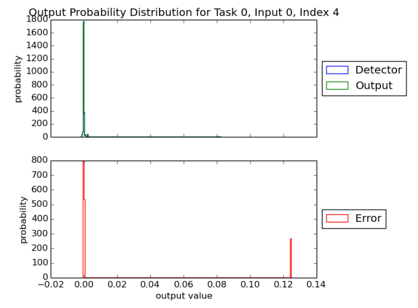
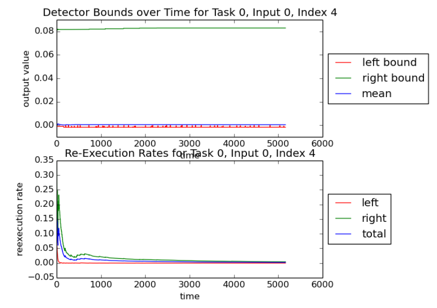

(c) Bounds Evolution over Distribution

Figure 68: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[3] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

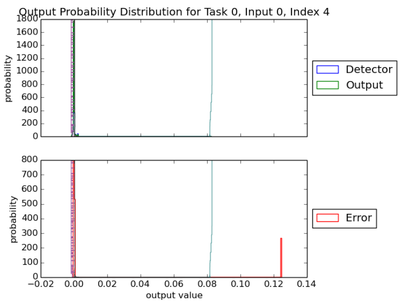
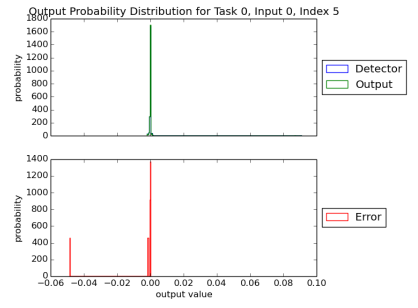
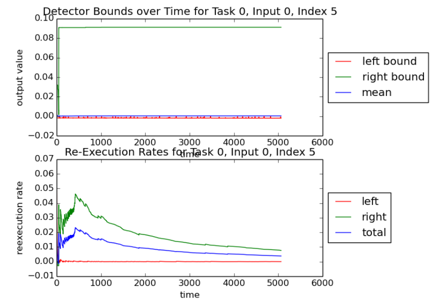

(c) Bounds Evolution over Distribution

Figure 69: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[4] in Intermolecular Forces Computation


(a) Correct (green), Error (red), Accepted (blue) Distributions over Time


(b) Bounds Evolution and Reexecution Rate over Time

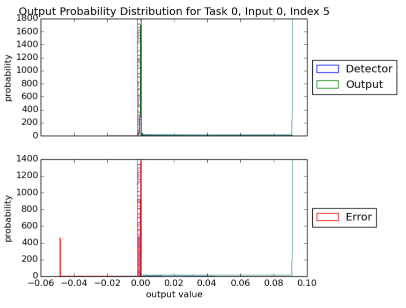
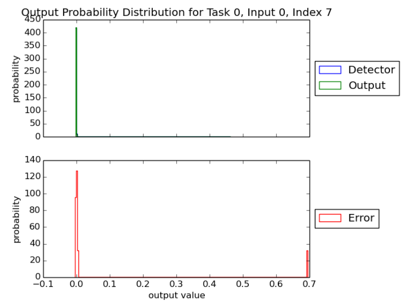
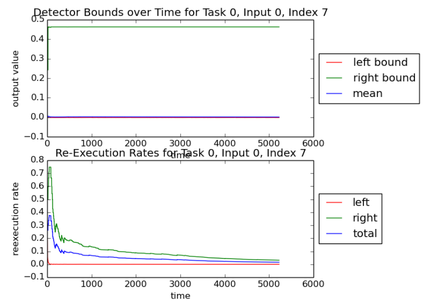

(c) Bounds Evolution over Distribution

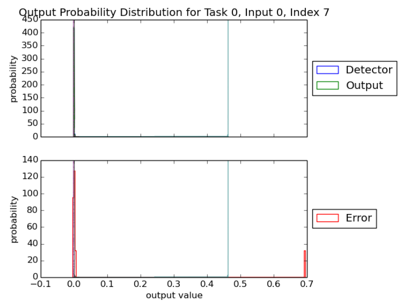
Figure 70: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[5] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

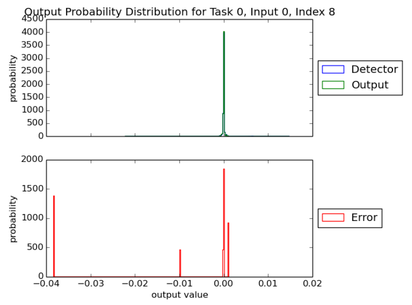


(b) Bounds Evolution and Reexecution Rate over Time

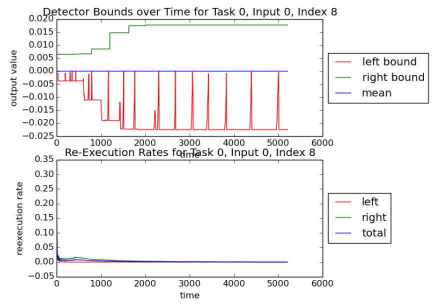


(c) Bounds Evolution over Distribution

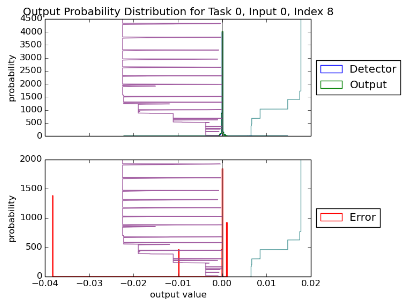
Figure 71: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[7] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

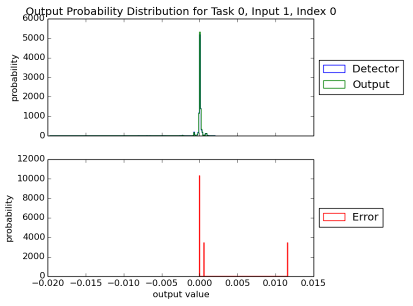


(b) Bounds Evolution and Reexecution Rate over Time

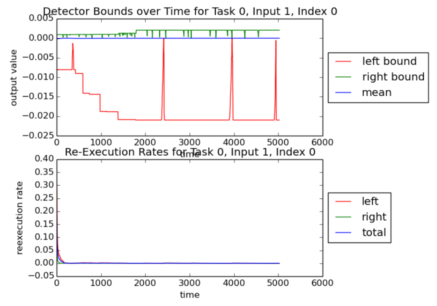


(c) Bounds Evolution over Distribution

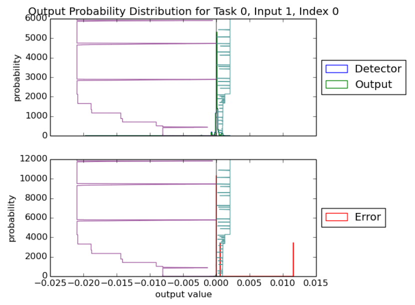
Figure 72: Water: Outlier Detector with Reexecution Strategy Behavior for Res1[8] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

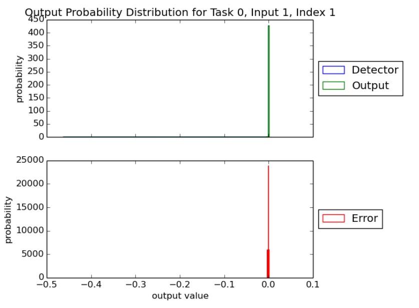


(b) Bounds Evolution and Reexecution Rate over Time

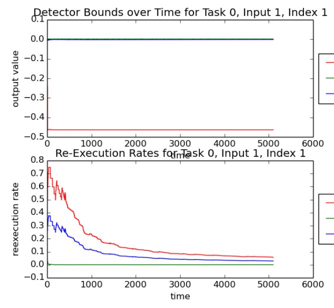


(c) Bounds Evolution over Distribution

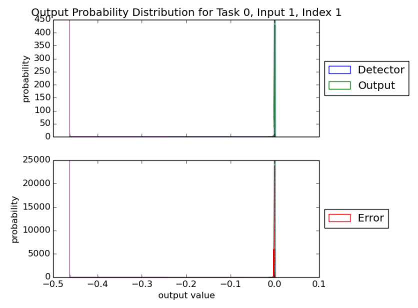
Figure 73: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[0] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

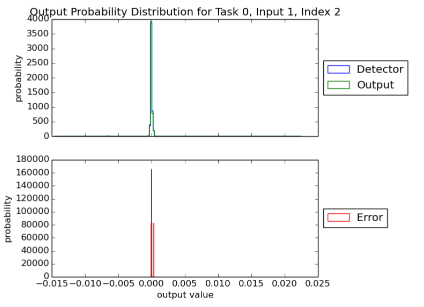


(b) Bounds Evolution and Reexecution Rate over Time

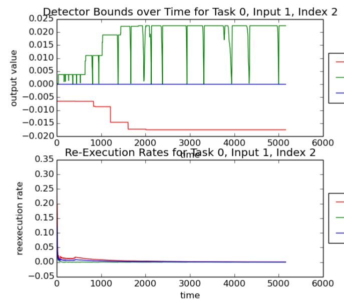


(c) Bounds Evolution over Distribution

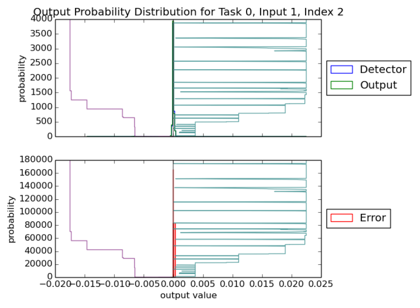
Figure 74: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[1] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

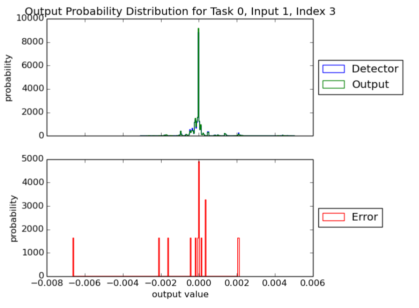


(b) Bounds Evolution and Reexecution Rate over Time

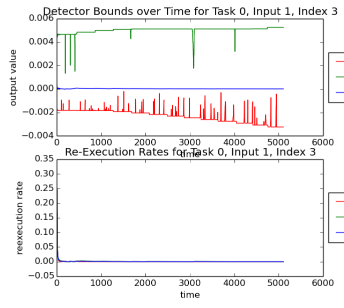


(c) Bounds Evolution over Distribution

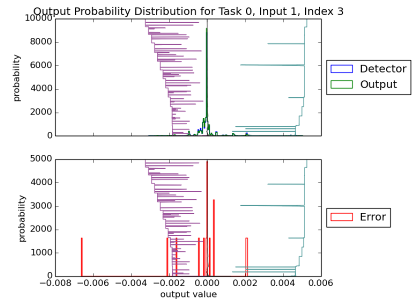
Figure 75: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[2] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

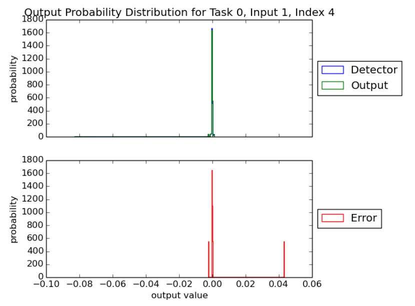


(b) Bounds Evolution and Reexecution Rate over Time

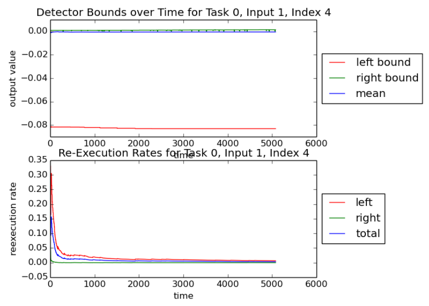


(c) Bounds Evolution over Distribution

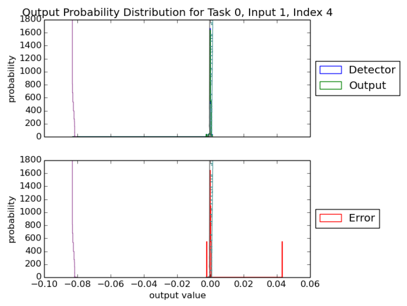
Figure 76: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[3] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

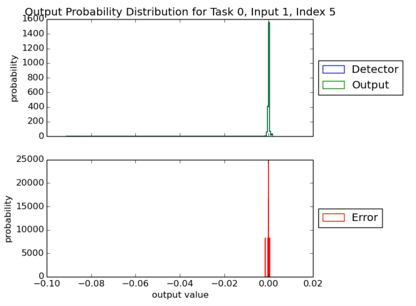


(b) Bounds Evolution and Reexecution Rate over Time

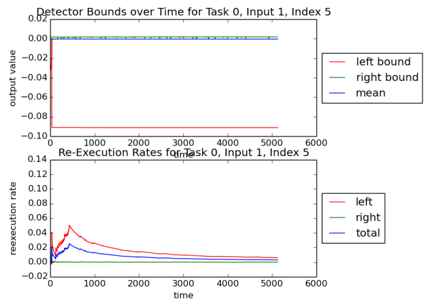


(c) Bounds Evolution over Distribution

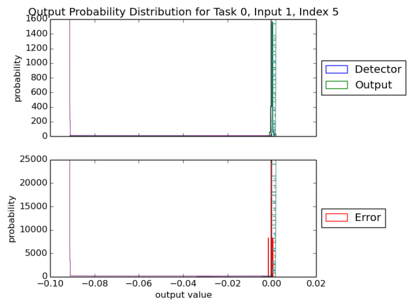
Figure 77: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[4] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

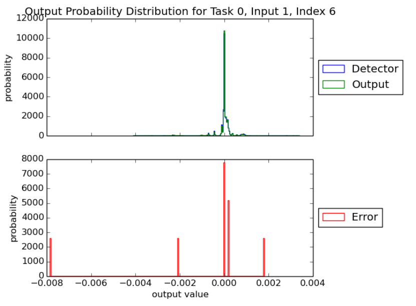


(b) Bounds Evolution and Reexecution Rate over Time

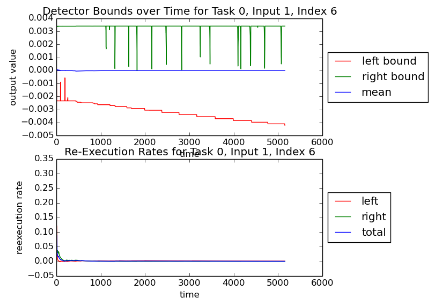


(c) Bounds Evolution over Distribution

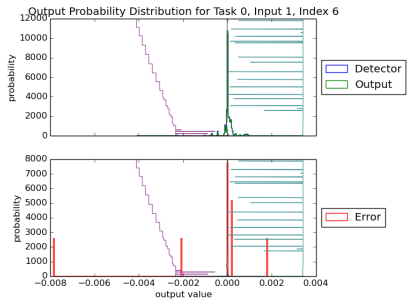
Figure 78: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[5] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

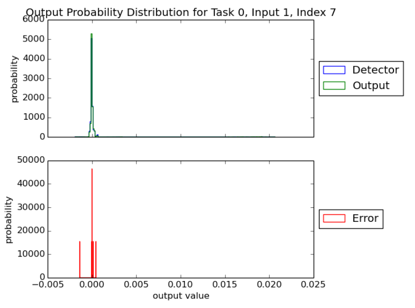


(b) Bounds Evolution and Reexecution Rate over Time

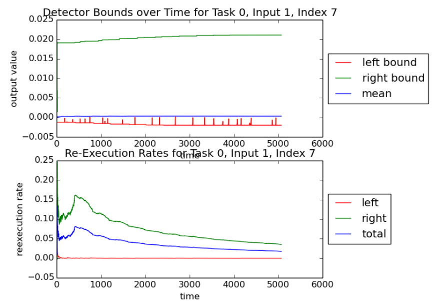


(c) Bounds Evolution over Distribution

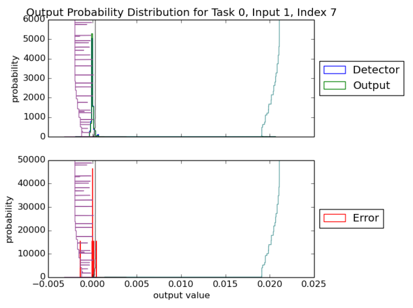
Figure 79: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[6] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

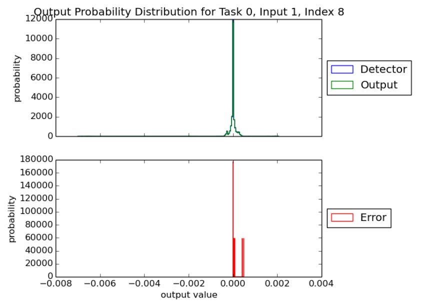


(b) Bounds Evolution and Reexecution Rate over Time

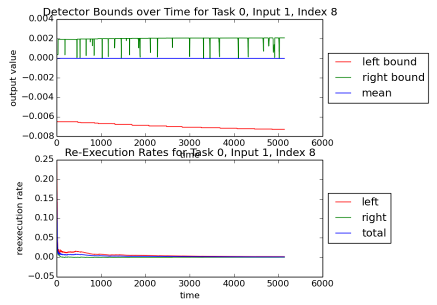


(c) Bounds Evolution over Distribution

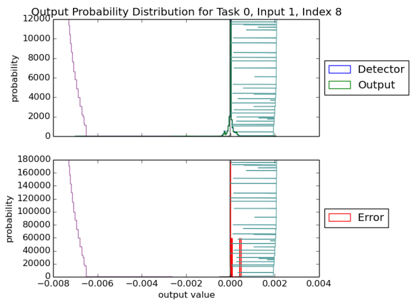
Figure 80: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[7] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

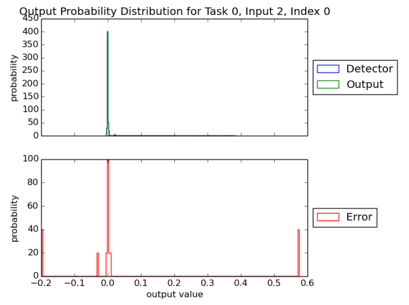


(b) Bounds Evolution and Reexecution Rate over Time

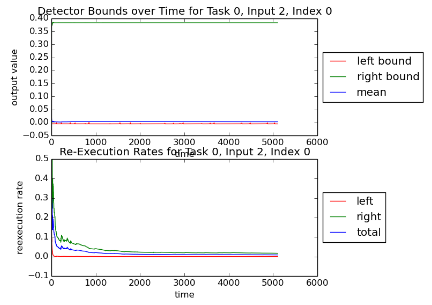


(c) Bounds Evolution over Distribution

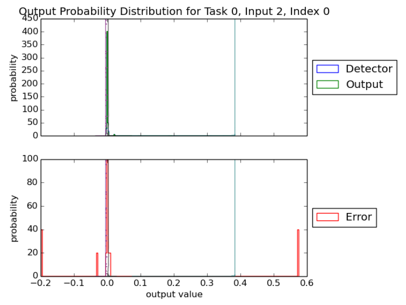
Figure 81: Water: Outlier Detector with Reexecution Strategy Behavior for Res2[8] in Intermolecular Forces Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

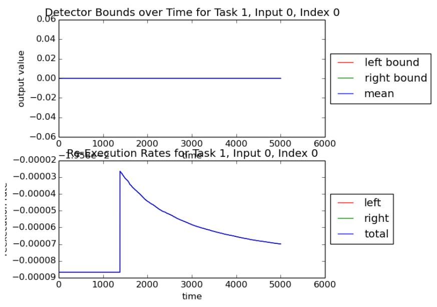


(b) Bounds Evolution and Reexecution Rate over Time



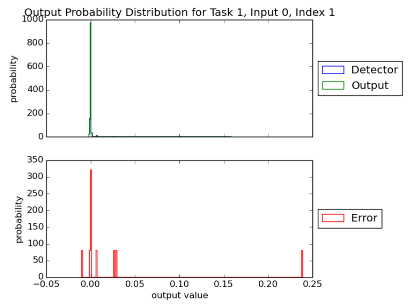
(c) Bounds Evolution over Distribution

Figure 82: Water: Outlier Detector with Reexecution Strategy Behavior for Incr in Intermolecular Forces Computation

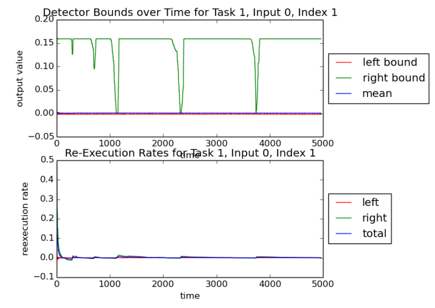


(a) Bounds Evolution and Reexecution Rate over Time

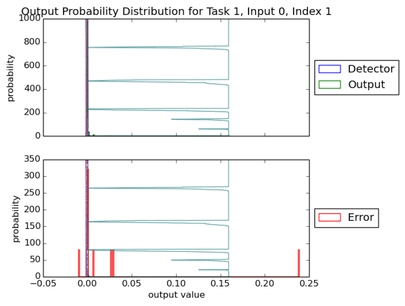
Figure 83: Water: Outlier Detector with Reexecution Strategy Behavior for Result[0] in Potential Energy Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time

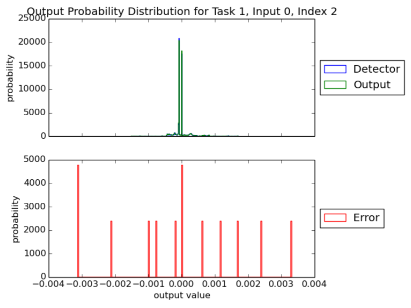


(b) Bounds Evolution and Reexecution Rate over Time

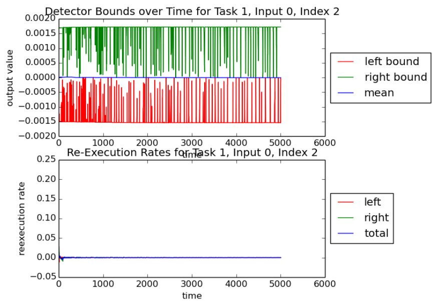


(c) Bounds Evolution over Distribution

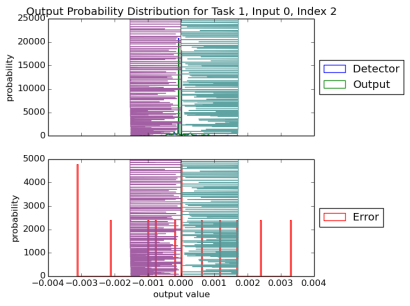
Figure 84: Water: Outlier Detector with Reexecution Strategy Behavior for Result[1] in Potential Energy Computation



(a) Correct (green), Error (red), Accepted (blue) Distributions over Time



(b) Bounds Evolution and Reexecution Rate over Time



(c) Bounds Evolution over Distribution

Figure 85: Water: Outlier Detector with Reexecution Strategy Behavior for Result[2] in Potential Energy Computation