Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks *

Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@csail.mit.edu

ABSTRACT

We present a new technique for enabling computations to survive errors and faults while providing a bound on any resulting output distortion. A developer using the technique first partitions the computation into tasks. The execution platform then simply discards any task that encounters an error or a fault and completes the computation by executing any remaining tasks. This technique can substantially improve the robustness of the computation in the face of errors and faults. A potential concern is that discarding tasks may change the result that the computation produces.

Our technique randomly samples executions of the program at varying task failure rates to obtain a quantitative, probabilistic model that characterizes the distortion of the output as a function of the task failure rates. By providing probabilistic bounds on the distortion, the model allows users to confidently accept results produced by executions with failures as long as the distortion falls within acceptable bounds. This approach may prove to be especially useful for enabling computations to successfully survive hardware failures in distributed computing environments.

Our technique also produces a timing model that characterizes the execution time as a function of the task failure rates. The combination of the distortion and timing models quantifies an accuracy/execution time tradeoff. It therefore enables the development of techniques that purposefully fail tasks to reduce the execution time while keeping the distortion within acceptable bounds.

1. INTRODUCTION

With standard program execution platforms, an error anywhere within the computation may propagate to cause the entire computation to fail. Such failures can be problematic because they deny the user access to the output that the computation would otherwise have provided.

*This research was supported in part by the Singapore-MIT Alliance, DARPA Cooperative Agreement FA 8750-04-2-0254, NSF Grant CCR-0086154, NSF Grant CCR-0341620, NSF Grant CCF-0209075, and NSF Grant CCR-0325283.

ICS06, June 28-30, Cairns, Queensland, Australia.

We propose a new approach to handling such errors. The developer starts with a program written in a standard programming language such as C or Java. He or she then uses a metalanguage (in our case, the Jade metalanguage [13]) to partition the computation into tasks. When a task encounters a software error or hardware fault, the execution platform simply discards the task, then continues on to complete the computation by running the remaining tasks.

1.1 Distortion and Timing Models

A potential complication is that discarding tasks may unacceptably change the output. We address this complication by developing a *probabilistic distortion model* that characterizes the distortion of the output as a function of the task failure rates in the computation. Instead of confronting the user with an output that has unknown distortion, the model provides a probabilistic bound on the distortion. The user can then examine this bound to determine if the output satisfies the accuracy requirements within an acceptable level of certainty.

In addition to the distortion model, we also provide a timing model that characterizes the execution time as a function of the task failure rates. In many cases higher task failure rates correspond to lower execution times — when a computation discards tasks, it often performs less work. The presence of both a distortion model and a timing model enables the deployment of techniques that purposefully fail tasks to reduce the execution time while keeping the resulting distortion within acceptable bounds.

1.2 Basic Approach

Our basic approach consists of the following steps:

- **Task Decomposition:** The developer uses the metalanguage to identify *task blocks* in the program. Each task block consists of a block of code whose execution corresponds to a task in the computation. Note that a given task block may execute many times during the course of a computation and may therefore generate many tasks into the computation.
- **Baseline:** We obtain several sample inputs for which the program is known to generate correct output. We run the program on those inputs and record the outputs that it generates.
- **Criticality Testing:** We configure the execution platform to randomly fail executions of selected task blocks at target failure rates. We then select each task block in the program in turn, fail executions of that task block at a targeted rate, and observe the resulting output distortion. If the failures produce unacceptable distortion, we mark the task block as *critical*, otherwise we mark the task block as *failable*.

- **Distortion Model:** Given a set of failable task blocks, we run repeated trials in which we randomly select a target task failure rate for each failable task block, execute the computation, then record both the observed task failure rates and the resulting output distortion. We then use regression [7] to obtain a probabilistic model that estimates the distortion as a function of the task failure rates.

- **Timing Model:** For each trial we also record the execution time of the program. We then use regression to obtain a model that estimates the execution time as a function of the task failure rates.

We have applied this approach to several benchmark applications selected from the Jade benchmark suite [13]. Our results show that the models are quite accurate (they have good statistical properties and usually explain almost all of the observed variation in the distortion and execution time data) and that the estimated distortion and confidence intervals are small enough to provide useful distortion bounds in practice. And it is possible to use models developed on one set of inputs to accurately predict both timing and distortion properties of executions on other inputs.

1.3 Potential Uses

We see several potential uses for our models. Some of these uses simply make existing programs more robust against errors and faults. Other uses enable developers to deliberately introduce failures or accept the presence of errors to achieve other goals.

1.3.1 Hardware Faults

The distortion model can provide distortion bounds for techniques that make software more robust against hardware faults by discarding tasks that experience such faults. We anticipate transient hardware faults for which the recovery mechanism is simply to discard the task, then continue on to execute other tasks on the same hardware platform. We also anticipate tolerating partial failures within a distributed computing platform hosting a distributed (and potentially parallel) execution of the program.

1.3.2 Software Errors

Another potential use is to provide distortion bounds for techniques that tolerate otherwise fatal software errors by discarding tasks that experience such errors. Once developers experience the ability of this technique to provide tight distortion bounds for computations that acceptably recover from inadvertent software errors by discarding tasks, they may start to consciously exploit the technique by purposefully omitting complex code otherwise required to handle rare special cases. We are already aware of several systems that apply such approaches on an ad-hoc basis with no distortion bounds [11, 5]. We anticipate that the availability of distortion bounds will make such approaches more acceptable and therefore more widely used. The potential benefits include reduced development time and simpler delivered software systems.

1.3.3 Reducing Computation Time and Resources

It is possible to analyze the distortion and timing models to obtain strategies that optimally fail tasks to obtain the maximum execution time reduction while minimizing the resulting output distortion. Users with large computations may use these strategies to reduce the amount of time or hardware resources required to obtain a desired result while staying within appropriate accuracy bounds.

The timing models assume that failed tasks consume no execution time. This assumption does not hold if the task failures are caused by software errors and/or hardware faults — in this case, tasks may execute for some time before failing. The primary anticipated use of the timing models is therefore to enable strategies that manage the time/accuracy tradeoff by preemptively failing tasks before they begin execution.

1.3.4 Assumptions and Fault Model

Because our statistical models use the number of failed tasks to predict the distortion, they are appropriate for situations in which there is no correlation between the likelihood that a task will fail and the effect of that failure on the distortion. If there is such a correlation, the statistical model may not be accurate. This could happen with hardware faults, for example, if tasks that take a longer time to execute (and therefore have a greater chance of experiencing a hardware fault) have a larger effect on the final result. It could happen with software errors, for example, if the magnitude of a task's contribution to the final result is correlated with the likelihood that the task will encounter a software error. As with any statistical technique, one must understand the underlying assumptions to appropriately apply the technique to the situation at hand.

We obtain our statistical models by sampling executions of the program running on several different training inputs. We anticipate usage scenarios in which these models are then used to provide distortion bounds estimates for the program running on (different) production inputs. Our approach is therefore appropriate for situations in which the effect of task failures on the distortion is not systematically different for the production inputs as compared with the training inputs.

1.4 Scope

We have implemented a system that automatically produces distortion and timing models and applied this system to a collection of programs written in the Jade implicitly parallel programming language [13]. Based on our experience with these programs, we have identified a general computational pattern that interacts well with our approach. Many scientific computations first generate many contributions to a final result, then combine the contributions to obtain the result. This pattern is also present in many graphics and information retrieval programs [11, 5].

If each task either generates or combines contributions, the net effect of task failures is simply to discard some of the contributions. Our results indicate that the distortion associated with discarding some of these contributions is often quite small and that our approach is quite effective at enabling computations that exhibit this pattern to execute through task failures to produce outputs with good distortion bounds. We anticipate that our approach will also work well for other computations that have this general pattern.

Finally, we note that our technique obtains the accuracy and timing models by sampling many different executions of the program running on a several training inputs. We anticipate usage scenarios in which this sampling overhead is profitably amortized over the production runs, either because the production runs take substantially longer to execute than the training runs or because the program will be used for many more production runs than training runs. It may also be feasible to perform the training runs during a lead time between when the program is developed and when it is deployed into production.

1.5 Contributions

This paper makes the following contributions:

- **Basic Concept:** It introduces, for the first time, the concept of using continuous distortion and timing models to characterize the behavior of software systems in the face of failures.

- **Methodology:** It shows how to apply the basic concept to existing programs by:
  - **Task Decomposition:** using a metalanguage to divide the computation up into tasks,
  - **Criticality Testing:** using random sampling to find critical tasks whose failures impose unacceptable distortion costs, and
  - **Distortion and Timing Models:** using random sampling on the failure rates of the remaining failable tasks in combination with regression to obtain distortion and timing models that characterize the behavior of the program in the face of task failures.
- **Evaluation:** It presents our experience applying our technique to several scientific computations. Our results show that we are able to obtain accurate models that precisely capture the distortion and timing responses of these computations as a function of the task block failure rates.
- **Characterization:** We analyze the underlying properties of the programs that are responsible for their behavior and identify a general computational pattern that interacts well with our approach.

The remainder of the paper is structured as follows. In Section 2 we provide an example that illustrates how our technique operations. Section 3 presents the methodology we use to obtain our models. Section 4 presents our experience applying our technique to a set of scientific computations. We present related work in Section 5 and conclude in Section 6.

2. EXAMPLE

Figure 1 presents a simple Jade program that we use to illustrate our approach. This program computes the sum of `n` numbers, storing the result in the global variable `sum`. The task block on lines `8` and `9` uses the Jade `withonly` construct to specify that the computation of each number and its accumulation into `sum` is a task. The `withonly` construct on line `8` uses the access specification operations `rd(&sum)` and `wr(&sum)` to specify that the code in the body of the task (in this case the computation of the numbers and their accumulation into `sum`) may, when it executes, read and/or write `sum`. We assume that the `add` procedure is part of a larger program that invokes `add` with `sum` equal to 0, then prints the value of `sum` as the result of the computation.

```
 1: int shared sum;
 5: void add(int n) {
 6:   int i;
 7:   for (i = 0; i < n; i++) {
 8:     withonly { rd(&sum); wr(&sum); }
 9:     do (i) { sum = sum + f(i); }
10:   }
11: }
```

Figure 1: Example Jade Program

When it executes this program, the Jade implementation creates a new task for every execution of the task block on lines `8` and `9`. It dynamically analyzes the data dependences between tasks to exploit any available concurrency; Jade implementations exist for a variety of parallel computing platforms [13].

To support the development of the distortion and timing models, we modified the Jade implementation to accept a target task failure rate for each task block. When the program runs, the Jade implementation randomly fails the corresponding tasks at the specified rates.

To construct the distortion model, our technique first runs the program on several inputs for which it is known to produce correct output. It records the output, then runs a sequence of trials that randomly fail executions of each task block at a randomly selected failure rate. For each trial it records the output of the computation and the actual task failure rate for each task block.

To quantify the impact of the failures on the output, the technique computes the *distortion* associated with each trial. In our example the distortion is simply $|(o - \hat{o})/o|$, where o is the correct output and $\hat{o}$ is the observed output from the trial with failures. Dividing the difference $o - \hat{o}$ by o makes it possible to meaningfully compare distortions from executions with different correct outputs.

The result of the sampling phase is a set of observations x, d, where x is the actual task failure rate for the task block in the program and d is the observed distortion. Our technique takes this set of observations and uses regression [7] to obtain a linear model $\hat{d}(x) = c_0[\pm e_0] + c_1[\pm e_1]x$ of the distortion. Here c_0 and c_1 are the regression coefficients and e_0 and e_1 provide the confidence bounds for these coefficients. For the program in our example, the regression produces the following distortion model.

$$\hat{d}(x) = 0[\pm 0.0002] + 1.0[\pm 0.0007]x$$

In this model $c_0 = 0$, which correctly estimates that there should be no distortion if there are no task failures. The coefficient $c_1 = 1$, which indicates that every increase in the task failure rate produces the same increase in the distortion. So, for example, an increase of 10% in the task failure rate would produce an increase in the distortion of 0.10.[1]

In addition to the model, the regression algorithm provides a variety of statistics that evaluate how well the model fits the data. In our example R^2 is 1, which means that the model perfectly explains the variation in the data. Finally, given a task failure rate x, the regression can provide a confidence bound e around the estimated distortion $c_0 + c_1 x$. In our example the maximum 95% confidence bound over all of the 2064 sampled task failure rate points is 0.0025, which provides a tight confidence interval around the estimated distortion.

We anticipate the following usage scenario. The user has obtained the model and now runs the program on another input. The program comes back with an output $\hat{o}$. It also informs the user that several tasks failed during the execution, and that the actual failure rate was x for that execution. The user plugs the task failure rate into the model to get an estimated distortion $c_0 + c_1 x$ and confidence bound e. The user then evaluates the distortion and confidence bound to determine if, with high enough likelihood, the distortion is acceptable. Assume, for example, that the user runs the example program and obtains the output 791792064 at an observed task failure rate of 0.01 or 1%. The estimated distortion is therefore 0.01. Adding the confidence bound 0.0025 gives 0.0125 as a reasonable upper bound on the distortion. Applying the distortion equation yields an expectation that the correct output should occur within the range [782016853,801814748]. The correct output is, in fact, 799980000.

[1]It turns out that for this example it is possible to apply the bias compensation technique discussed in Section 3.8 to obtain an estimator with an expected distortion of 0.0 for task failure rates within the sampled range of 0.0 to 0.75. This technique enables the program to acceptably tolerate much higher task failure rates.

3. METHODOLOGY

We obtain and evaluate our distortion model and timing model for each program as follows.

3.1 Failure-Free Executions

Our methodology applies to programs that produce an output of the form $o_1, \ldots, o_m$, where each output component o_i is a number. We obtain several test inputs for which the program is known to execute without failures, run the program on these inputs, and record the correct output $o_1, \ldots, o_m$ for each input.

3.2 Distortion Definition

In subsequent steps we run the program with failures and measure how much the failures affect the accuracy of the outputs. Given a correct output $o_1, \ldots, o_m$ and an observed output $\hat{o}_1, \ldots, \hat{o}_m$ from an execution with failures, the following quantity d, which we call the *distortion*, measures the accuracy of the observed output:

$$d = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{o_i - \hat{o}_i}{o_i} \right|$$

The closer the distortion d is to zero, the less the failures distort the output.

Note that because each difference $o_i - \hat{o}_i$ is scaled by the corresponding correct output component o_i and because the sum is divided by the number of output components m, it is possible to meaningfully compare distortions d obtained from executions on different inputs even if the inputs cause the failure-free executions of the program to produce outputs with different numbers of components m and different correct component values o_i.

Note that the distortion equation weights each output value o_i equally. It is possible to use a set of weights w_i to generalize the distortion equation for programs whose outputs are not all equally important. Each weight w_i would capture the importance of the corresponding output o_i:

$$d = \frac{1}{m} \sum_{i=1}^{m} w_i \left| \frac{o_i - \hat{o}_i}{o_i} \right|$$

where the w_i satisfy $m = \sum_{i=1}^{m} w_i$.

3.3 Criticality Testing

It turns out that some programs have task blocks that must always execute without failures for the program to produce acceptably accurate output. We experimentally detect these *critical* task blocks as follows. We first configure the underlying execution engine (in our case the Jade runtime system) to randomly fail executions of a selected task block at a specified rate (our criticality testing executions fail 10% of the executions of the selected task block). We then select each task block in turn and run the program at least ten times with the execution engine randomly failing that task block at the specified rate. If any of these runs does not produce any output at all (typically because the program failed) or if the mean distortion from all of the runs is larger than the specified acceptable distortion for criticality testing (we use an acceptable distortion of 0.10), we identify the task block as a critical task block that must execute without failure. Otherwise, we consider the task block to be a *failable* task block. The purpose of the remaining steps is to characterize the effect that failures of failable task blocks have on the acceptability of the outputs.

3.4 Distortion Sampling Runs

We next run a set of trials in which we randomly select a target failure rate for each failable task block, run the program with the execution engine randomly failing executions of each task block at its target failure rate, then record the distortion and actual failure rate for each failable task block for that run. If we have n failable task blocks, the result is a set of observations $x_1^i, \ldots, x_n^i, d^i$, where d^i is the distortion and $x_1^i, \ldots, x_n^i$ are the actual failure rates for the n failable task blocks in the i'th trial.

We run multiple sampling runs for each of our test inputs. In our experiments the target failure rates for our distortion sampling runs range from 0 (no execution of the task block fails) to 0.75 (three out of every four executions of the task block fail). The number of sampling runs for each test input varies between approximately 500 and 5000 depending on the program. We simply set up a script that repeatedly executed each program on each test input, using a pseudo-random number generator to select the target failure rates for each run. While we did not attempt to closely control the amount of time spent performing the distortion sampling runs, we typically let the script run for about a day for each program before collecting the data and computing the distortion and timing models.

3.5 Distortion Model

Given the results from the distortion sampling runs, we use multiple linear regression [7] to compute a linear least-squares distortion model of the form

$$\hat{d}(x_1, \ldots, x_n) = c_0[\pm e_0] + \sum_{i=1}^{n} c_i[\pm e_i]x_i$$

where the c_i are the least-squares coefficients for the regression and the e_i provide the confidence intervals for these coefficients (we use 95% confidence intervals in this paper). Given failure rates x_i for all of the failable task blocks, this model produces a distortion estimate $\hat{d}(x_1, \ldots, x_n)$ of the expected accuracy of the result produced by a computation with task block failure rates $x_1, \ldots, x_n$.

The regression also produces an F value that assesses how well the model predicts the data and an R^2 value that indicates how much of the variation in the data the model accounts for. Moreover, given a specific point $x_1, \ldots, x_n$ in the failure rate space, the regression can produce a confidence interval around the distortion estimate $\hat{d}(x_1, \ldots, x_n)$ at that point. It is possible to obtain confidence intervals for whatever alpha level one desires; in this paper we use an alpha level of 0.05, which produces a 95% confidence interval. We use the SAS system to compute the regression [7].

3.6 Timing Model

We also observe the running times of the various executions and use these running times to similarly obtain a timing model that captures the effect of failures on the running time. To obtain this model, we first run the program several times without failures to obtain a mean failure-free execution time t for each input. We then divide the observed running times t^i from the distortion sampling runs to obtain a scaled time observation $s = t^i/t$ for each execution. This scaling makes it possible to meaningfully compare timing results from executions on different inputs with different failure-free execution times. Finally, we use multiple linear regression to obtain a least-squares timing model of the form

$$\hat{s}(x_1, \ldots, x_n) = c_0[\pm e_0] + \sum_{i=1}^{n} c_i[\pm e_i]x_i$$

for the scaled running time $\hat{s}(x_1, \ldots, x_n)$ of the program as a function of the task block failure rates $x_1, \ldots, x_n$. This model is useful for calculating expected time/accuracy trade-offs and can, in some circumstances, guide a strategy that purposefully fails tasks to move the execution toward a more desirable point in this trade-off space.

3.7 Using the Model

One goal of the distortion model is to allow a user running the program in the presence of failures to obtain an estimate of how the failures in that execution affected the accuracy of the result that the program produced. In particular, the user would take the observed failure rates $x_1, \ldots, x_n$, apply the distortion model to obtain an estimated distortion $\hat{d}(x_1, \ldots, x_n)$ along with its associated confidence interval to evaluate whether the failures were likely to have unacceptably distorted the result. In this scenario, several issues are likely to be of interest:

- **Distortion:** How quickly does the distortion grow as a function of the task block failure rates? We evaluate this issue by examining the model coefficients c_i. The smaller the model coefficients, the less the results are affected by failures of the corresponding task blocks.

- **Bounds:** How small are the confidence intervals? We evaluate this issue by computing the minimum and maximum sizes of the upper confidence intervals at all of the $x_1, \ldots x_n$ points observed during the distortion sampling runs. For a user to accept a distorted result, both the distortion estimate and the upper confidence interval must be small enough to make the likelihood that an unacceptably large actual distortion has occurred remote enough for the user to accept. The upper confidence interval provides the appropriate bound for this purpose — the distortion is inherently bounded below by zero and becomes more acceptable the closer it gets to this bound.

- **Predictive Power:** We use our test inputs to obtain the regression model. We anticipate that users will apply the model to executions of the program running on other inputs. The issue is whether a model derived from executions on one set of inputs can accurately predict the distortion for other inputs.

  We evaluate this issue by applying a model derived from executions on some of our inputs to executions on other inputs. Specifically, we partition our test inputs into two categories: learning inputs and unseen inputs. We use the executions on the learning inputs to obtain a model, then calculate the proportion of observed executions on unseen inputs whose distortion d falls below the upper confidence interval of the distortion estimate $\hat{d}(x_1, \ldots, x_n)$ from the learning input model. If the proportion of executions that falls below the upper confidence interval is consistent with the alpha level of that confidence interval, the model has good predictive power.

3.8 Bias Definition and Use

The distortion measures the absolute error induced by a set of failures. It is also sometimes useful to consider whether there is any systematic direction to the error. The following quantity b measures the *bias* of the outputs:

$$b = \frac{1}{m} \sum_{i=1}^{m} \frac{o_i - \hat{o}_i}{o_i}$$

Note that this is the same formula as the distortion with the exception that it preserves the sign of the summands. Errors with different signs may therefore cancel each other out in the computation of the bias instead of accumulating as for the distortion.

If there is a systematic bias, it may be possible to compensate for the bias to obtain a more accurate result. Consider, for example, the special case of a program with a single output component o. If we know that the bias at a certain point is b, we can simply divide the observed output $\hat{o}$ by $(1 - b)$ to obtain an estimate of the correct output whose expected distortion is 0.

The reasoning in this example generalizes to handle programs with multiple output components $o_1, \ldots, o_m$ — the key is to generalize our methodology to obtain a separate distortion and bias model for each different output component o_i. It is then possible to correct each output component individually to eliminate the bias for that component. If the output components do, in fact, exhibit a systematic bias in the face of failed task blocks, the primary obstacle to applying this technique is the number of output components. For programs with large numbers of output components it may be difficult to perform the number of trials required to obtain a useful model of the distortion and bias for each individual output component.

4. EXPERIMENTAL RESULTS

We apply our methodology to four scientific computations:

- **Water:** Water evaluates forces and potentials in a system of water molecules in the liquid state. Water is derived from the Perfect Club benchmark MDG [2] and performs the same computation.

- **Search:** Search [4] is a program from the Stanford Electrical Engineering department. It simulates the interaction of several electron beams at different energy levels with a variety of solids. It uses a Monte-Carlo technique to simulate the elastic scattering of each electron from the electron beam into the solid. The result of this simulation is used to measure how closely an empirical equation for electron scattering matches a full quantum-mechanical expansion of the wave equation stored in tables.

- **SOR:** SOR uses an iterative method to solve a set of spatial partial differential equations [18]. It stores the state of the system in several two-dimensional arrays. On every iteration the solver recomputes each element of the array using a standard five-point stencil algorithm. The new value of the element depends on its old value and on the values of the four elements above it, below it, to the left of it and to the right of it. The solver terminates when the differences between the old values and the new values drop below a given threshold.

- **String:** String [10] uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geological medium between two oil wells. Each element of the velocity model records how fast sound waves travel through the corresponding part of the medium. The seismic data are collected by firing non-destructive seismic sources in one well and recording the seismic waves digitally as they arrive at the other well. The travel times of the waves can be measured from the resulting seismic traces. The application uses the travel-time data to iteratively compute the velocity model.

In addition to these computations, the Jade benchmark suite contains Panel Cholesky, which performs a Cholesky factorization of a sparse matrix, and Volume Rendering, which generates a sequence of images of a set of volume data. We do not report results for Panel Cholesky because it operates on synthetic data rather than the actual data in its benchmark sparse matrices. Any distortion results would therefore be meaningless for this computation. We were not able to run Volume Rendering on our current computational platform because of input data incompatabilities. Other than these two applications, we report experimental results for all computations in the Jade benchmark suite.

4.1 Water

We ran Water on four different inputs; the inputs vary in the number of molecules they cause Water to simulate. Specifically, the inputs produce simulations of 343, 512, 729, and 1000 molecules. Water calculates several values of potential interest, including the total energy, kinetic energy, and potential energy. We choose to measure the distortion of the total energy (in part because it includes contributions from all of the other partial energy calculations); it would be possible to extend this measure to include the different partial energy values explicitly.

4.1.1 Task Blocks and Criticality Testing

Water has four task blocks. Executions of the first task block compute the intermolecular forces between pairs of molecules, storing their intermediate results into blocks of storage allocated for that purpose. Executions of the second task block sum up all of the intermediate results to produce the final intermolecular forces. Similarly, executions of the third task block compute the intermolecular contributions to the potential energy, storing intermediate results into blocks of storage allocated for that purpose. Executions of the fourth task block sum up the intermediate results to produce the final intermolecular potential energy.

Our criticality testing experiments revealed no critical task blocks — all task blocks produce a mean distortion of less than 0.1 when failed at the target 10% rate during our criticality testing runs.

4.1.2 Distortion Models

Figure 2 presents the distortion models for Water. The first model is for 343 molecules, the second for 512, et cetera; x_1 is the failure rate for the first task block, x_2 is the failure rate for the second task block, et cetera. The model $\hat{d}_{comp}$ is the composite model obtained by combining the observations from 343, 512, and 1000 inputs (see Section 4.1.3). We present each regression coefficient in the form $c[\pm e]$, where c is the coefficient itself and $[\pm e]$ provides the 95% confidence bounds for that coefficient. The F values for all of these models are in the tens of thousands, the corresponding p values are less than 0.0001, and the R^2 values are above 98%, which indicates that the model explains over 98% of the variation in the data.[2]

The coefficients are relatively large, which indicates that the distortion increases relatively quickly in the face of failures. For example, the coefficient for x_3 is either 0.55 or .56 for all of the models. So, for example, a 10% failure rate for task block 3 results in approximately 0.05 increase in the estimated distortion. If the failure rate is 50% for all task blocks, the estimated distortion is approximately 0.6 — in other words, the observed output with failures is estimated to be more than a factor of two different from the correct output! The models would therefore appear to indicate that the vast majority of the tasks in Water must execute without failure for the program to produce an acceptable output. Note, however, that Section 4.1.4 describes a way to correct the bias in the output to obtain a corrected output with an estimated distortion of zero. With this correction, Water can tolerate much higher task block failure rates.

[2]See a standard statistics book for definitions and more detailed interpretations of these quantities [7].

Intuitively, one would expect the y intercept values c_0 in the distortion models to be zero — after all, there should be no distortion at all with no failures. While the y intercept values c_0 are all close to zero, they are not exactly zero. The reason for this anomaly is not immediately clear. A slight non-linearity in the distortion as a function of the task failure rate (in which the distortion increased slightly more than linearly for small task failure rates and slightly less than linearly for large task failure rates) would explain the results.

Note that the models are close to identical for all of the different inputs, which (because the models are linear) indicates that the models do not depend on the input to the program. It should therefore be feasible to use models developed from one set of inputs to predict distortions for other inputs.

4.1.3 Predictive Power and Confidence Bounds

We evaluate the predictive power of our technique by obtaining a composite model for our learning inputs (we selected 343 molecules, 512 molecules, and 1000 molecules) and applying the model to the observations for our unseen input (729 molecules, chosen arbitrarily as the third out of four inputs). Specifically, we concatenated our observations from our learning inputs, used regression to obtain a composite model, then calculated the number of observations $x_1, \ldots, x_4, d$ from the distortion sampling runs for 729 molecules whose observed distortion d fell above the upper 95% confidence bound of the distortion estimate $\hat{d}(x_1, \ldots, x_4)$ from the composite model. Of the 3305 observations from the distortion sampling runs for the unseen input with 729 molecules, 99 (or approximately 3%) fell above the upper 95% confidence interval from the composite model, which indicates that the composite model was able to successfully predict the distortion behavior of the unseen input.

A final question is the size of the confidence bounds. To address this question we used the composite model to obtain 95% upper confidence bounds for all of the 13219 sampled task block failure points $x_1, \ldots, x_4$ in the distortion sampling runs. The maximum upper confidence bound was 0.0586. Given a task block failure point $x_1, \ldots, x_4$, the quantity $\hat{d}(x_1, \ldots, x_4) + 0.0586$ provides an appropriate upper bound for the actual distortion.

For small task failure rates it is possible to obtain a tighter confidence bound. First, we discarded all observations $x_1, \ldots, x_4, d$ with a task failure rate $x_1, \ldots, x_4$ greater than 1%. We next used regression on the remaining observations to obtain another model (the *1% failure rate model*) for this region of the task failure rate space. We then used the 1% failure rate model to obtain 95% upper confidence bounds for all of the remaining 3509 sampled task block failure points with failure rates $x_1, \ldots x_4$ less than or equal to 1%. The maximum 95% confidence bound for any of these remaining sampled task block failure points is 0.0082. An appropriate upper bound on the distortion for points $x_1, \ldots, x_4$ where $x_1, \ldots, x_4$ are all less than or equal to 1% is therefore $\hat{d}(0.01, \ldots, 0.01) + 0.0082$ or 0.03.

4.1.4 Bias Correction

It turns out that, for every execution of Water, the bias is the same as the distortion (which implies that the bias estimator $\hat{b}(x_1, \ldots, x_4)$ equals the distortion estimator $\hat{d}(x_1, \ldots, x_4)$). Because Water has a single output, it is possible to correct for the bias by simply dividing the observed output by $(1 - \hat{d}(x_1, \ldots, x_4))$ to obtain an output estimator with an expected distortion of zero and a confidence interval of the same size as the confidence interval of the distortion.

$$\hat{d}_{343}(x_1,\ldots,x_4) = 0.011[\pm 0.0014] + 0.053[\pm 0.0074]x_1 + 0.11[\pm 0.0075]x_2 + 0.56[\pm 0.0075]x_3 + 0.53[\pm 0.0075]x_4$$
$$\hat{d}_{512}(x_1,\ldots,x_4) = 0.010[\pm 0.0013] + 0.052[\pm 0.0070]x_1 + 0.11[\pm 0.0071]x_2 + 0.56[\pm 0.0072]x_3 + 0.54[\pm 0.0071]x_4$$
$$\hat{d}_{729}(x_1,\ldots,x_4) = 0.011[\pm 0.0013] + 0.049[\pm 0.0067]x_1 + 0.11[\pm 0.0068]x_2 + 0.55[\pm 0.0069]x_3 + 0.54[\pm 0.0068]x_4$$
$$\hat{d}_{1000}(x_1,\ldots,x_4) = 0.010[\pm 0.0012] + 0.053[\pm 0.0066]x_1 + 0.11[\pm 0.0067]x_2 + 0.56[\pm 0.0067]x_3 + 0.54[\pm 0.0067]x_4$$
$$\hat{d}_{comp}(x_1,\ldots,x_4) = 0.010[\pm 0.0076] + 0.053[\pm 0.0041]x_1 + 0.11[\pm 0.0041]x_2 + 0.56[\pm 0.0041]x_3 + 0.54[\pm 0.0041]x_4$$

Figure 2: Distortion Models for Water

$$\hat{s}_{343}(x_1,\ldots,x_4) = 1.0[\pm 0.00008] + -0.45[\pm 0.0004]x_1 + -0.030[\pm 0.0004]x_2 + -0.42[\pm 0.0004]x_3 + -0.004[\pm 0.0004]x_4$$
$$\hat{s}_{512}(x_1,\ldots,x_4) = 1.0[\pm 0.00007] + -0.47[\pm 0.0004]x_1 + -0.027[\pm 0.0004]x_2 + -0.40[\pm 0.0004]x_3 + -0.008[\pm 0.0004]x_4$$
$$\hat{s}_{729}(x_1,\ldots,x_4) = 1.0[\pm 0.00007] + -0.49[\pm 0.0003]x_1 + -0.025[\pm 0.0003]x_2 + -0.41[\pm 0.0003]x_3 + -0.005[\pm 0.0003]x_4$$
$$\hat{s}_{1000}(x_1,\ldots,x_4) = 1.0[\pm 0.00005] + -0.50[\pm 0.0003]x_1 + -0.025[\pm 0.0003]x_2 + -0.42[\pm 0.0003]x_3 + -0.004[\pm 0.0003]x_4$$

Figure 3: Timing Models for Water

4.1.5 Time/Accuracy Trade-Offs

Figure 3 presents the timing models for Water (these models are in the same format as the distortion models). The F values for all of these models are in the hundreds of thousands or more, the corresponding p values are less than 0.0001, and the R^2 values are above 99%, which indicates that the models almost perfectly explain the variation in the data.

Several aspects of these models are of interest. First, the coefficient c_0 is always 1, indicating that the estimated execution time with no failures is the same as the corresponding measured failure-free execution times. Second, the other coefficients c_1 through c_4 are all negative, indicating that the execution time decreases as the corresponding task block failure rate increases. This is as expected — for this program, failing a task does not affect the amount of computation that the remaining computation performs. The execution time therefore decreases by an amount proportional to the amount of computation that the failed task would otherwise have performed. This property also implies that the sum of the coefficients $c_1 + c_2 + c_3 + c_4$ is the proportion of the computation performed by failable tasks (for Water the remaining computation takes place in the main task). The models indicate that roughly 90% of the computation takes place in failable tasks.

There is a significant difference in the execution time decreases associated with failures of different task block executions. For example, c_1 varies between -0.4 and -0.5, which indicates that a 10% failure rate for task block 1 translates into an approximately 4% or 5% reduction in the execution time. But c_4 is much smaller (between -0.004 and -0.008), indicating that the failing executions of task block 4 have little or no impact on the execution time.

A comparison of the coefficients from the distortion and timing models makes it possible to quantify the time/accuracy trade-offs associated with failing different task blocks. Conceptually, the goal is to minimize the amount of distortion one must pay to obtain a given decrease in the execution time. The ratios of the distortion model coefficients to the corresponding timing model coefficients estimate how much distortion one must pay to obtain a unit decrease in execution time by failing executions of the corresponding task block. For example, this ratio for the coefficient c_1 is roughly $(0.052/-.48) = -.11$ while the ratio for c_4 is roughly $(0.54/-.006) = -90$. To obtain a given reduction in the execution time from failing task block 4, one must accept over 900 times the distortion associated with a corresponding reduction in the execution time from failing task block 1. Clearly any strategy that purposefully fails tasks to achieve a given execution time reduction goal is much better off failing executions of task block 1.

Given an execution time reduction goal, it is possible to use the two models to calculate target task block failure rates for each task block that deliver the desired execution time reduction while minimizing the resulting distortion — simply maximize the failure rates for the task blocks in order of their corresponding distortion/time coefficient ratios (while staying within the limits of the model) until the model indicates that the desired execution time reduction goal has been achieved. This calculation provides a target task block failure rate for each task block. Applying the distortion model to these failure rates provides an estimate of the distortion caused by the failures that achieve the execution time reduction. This calculation indicates if it is possible to achieve a given execution time reduction goal while staying within a given distortion bound.

The preceding calculation starts with a target execution time reduction and minimizes the distortion that results from this reduction. It is also possible to run the calculation in the other direction to minimize the execution time subject to a given distortion bound.

4.2 Search

We ran Search on four different inputs; the inputs vary in the backscattering parameters. The program calculates and outputs a backscattering coefficient for 51 different solid/energy level pairs; this coefficient indicates the percentage of the electrons that escape back out of the solid. We take the resulting sequence of 51 backscattering coefficients as the output of the program.

4.2.1 Task Blocks and Criticality Testing

Search has one task block which uses a Monte-Carlo simulation to trace the paths of the electrons through the solid. Our criticality testing experiment revealed that this task block was failable since it produced a mean distortion of less than 0.1 when failed at the target 10% rate during our criticality testing runs.

4.2.2 Distortion Models

Figure 4 presents the distortion models for Search; $\hat{d}_{comp}(x_1)$ is the composite model for inputs 1, 2, and 4 (see Section 4.2.3). The F values for all of these models are in the tens of thousands, the corresponding p values are less than 0.0001, and the R^2 values are above 96%, which indicates that the model explains over 96% of the variation in the data.

The coefficient c_1 is relatively small, which indicates that the distortion increases relatively slowly as that task fails. In particular, a 10% failure rate for task block 1 results in approximately 0.01 increase in the estimated distortion. The program is therefore relatively resilient to failures.

Note that the models are close for all of the different inputs, which (because the models are linear) indicates that the models are

$$\hat{d}_1(x_1) = 0.005[\pm 0.0002] + 0.11[\pm 0.0009]x_1$$
$$\hat{d}_2(x_1) = 0.006[\pm 0.0003] + 0.13[\pm 0.0011]x_1$$
$$\hat{d}_3(x_1) = 0.005[\pm 0.0003] + 0.12[\pm 0.0011]x_1$$
$$\hat{d}_4(x_1) = 0.003[\pm 0.0003] + 0.08[\pm 0.0007]x_1$$
$$\hat{d}_{comp}(x_1) = 0.005[\pm 0.0001] + 0.11[\pm 0.0005]x_1$$

Figure 4: Distortion Models for Search

$$\hat{s}_1(x_1) = 1.0[\pm 0.00006] + -0.99[\pm 0.0002]x_1$$
$$\hat{s}_2(x_1) = 1.0[\pm 0.00006] + -0.99[\pm 0.0002]x_1$$
$$\hat{s}_3(x_1) = 1.0[\pm 0.00006] + -0.99[\pm 0.0002]x_1$$
$$\hat{s}_4(x_1) = 1.0[\pm 0.00006] + -0.99[\pm 0.0002]x_1$$

Figure 5: Timing Models for Search

largely independent of the input to the program. It should therefore be feasible to use models developed from one set of inputs to predict distortions for other inputs.

The observed biases for the distortion sampling runs are all close to zero, which indicates that failing tasks does not systematically bias the outputs in any direction.

4.2.3 Predictive Power and Confidence Bounds

We evaluate the predictive power of our technique by obtaining a composite model for our learning inputs (we selected inputs 1, 2, and 4) and applying the model to the observations for our unseen input (input 3). Specifically, we concatenated our observations from our learning inputs, used regression to obtain a composite model, then calculated the number of observations x_1, d from the distortion sampling runs for the unseen input 3 whose observed distortion d fell above the upper 95% confidence bound of the distortion estimate $\hat{d}(x_1)$ from the composite model. Of the 1832 observations from the distortion sampling runs for the unseen input 3, 57 (or approximately 3%) fell above the upper 95% confidence interval from the composite model, which indicates that the composite model was able to successfully predict the distortion behavior of the unseen input.

A final question is the size of the confidence bounds. The maximum upper 95% confidence bound for any sampled task block failure point x_1 in any of the 7328 distortion sampling runs was 0.0136. An appropriate bound on the distortion at a 25% task failure rate is therefore approximately 0.039.

For small task failure rates it is possible to obtain a tighter confidence bound. Specifically, we discarded all observations x_1, d with x_1 greater than 1% and used regression on the remaining observations to obtain the 1% failure rate model. We then used this model to compute the upper 95% confidence bound for all of the 2568 points with x_1 less than or equal to 1%. The maximum bound was 0.00157. An appropriate bound on the distortion for task failure rates below 1% is therefore approximately 0.0066.

4.2.4 Time/Accuracy Trade-Offs

Figure 5 presents the timing models for Search. The F values for all of these models are in the millions, the corresponding p values are less than 0.0001, and the R^2 values are 100%, which indicates that the models perfectly explain the variation in the data.

The fact that the coefficient c_1 is 0.99 indicates that the program spends almost all of its time in task block 1 and that the running time is directly proportional to the percentage of these executions of this task blocks that do not fail. The distortion/timing model coefficient ratio is approximately -0.12, which indicates (for example) that a decrease in the execution time of 10% entails a distortion cost of roughly 0.01. As for Water, one can use these two models to navigate the execution time/accuracy trade-off space to move to a more desirable location in this space.

4.3 String

We ran String on two pairs of inputs (for a total of four inputs). Each pair has different seismic data and starting velocity models. Within each pair the inputs differ on the ray tracing parameters used to compute the velocity model. The output is the velocity model for the geology between the oil wells. This output is a large matrix of numbers whose size varies depending on the input.

4.3.1 Task Blocks and Criticality Testing

String has five task blocks. Executions of the first task block shoot rays through the current velocity model, storing their intermediate results into blocks of storage allocated for that purpose. Executions of the second task block combine these results into a single block of storage that executions of the third task block use to compute a new velocity model. The fourth task block creates data structures used in the remaining computation; the fifth task block deallocates these data structures at the end of the computation. Our criticality testing runs revealed that the first two task blocks are failable, while the third and fourth task blocks are critical. We attribute the criticality of the third task block to the fact that failing its executions leaves some of the values of the old velocity model in place, which causes significant distortion. The fourth task block is critical because failures in its executions leave the remaining computation without a place to store some of its results. Finally, failures of the fifth task have no effect at all on the result, but leave the data structures allocated at the end of the computation.

4.3.2 Distortion Models

Figure 6 presents the distortion models for String; $\hat{d}_{comp}(x_1, x_2)$ is the composite model for inputs 1, 2, and 4 (see Section 4.3.3). The F values for all of these models are in the thousands, the corresponding p values are less than 0.0001, and the R^2 values vary from around 80% (for inputs 1 and 3) to around 90% (for inputs 2 and 4). The models for inputs 1 and 3 are similar to each other but differ from the models for inputs 2 and 4, which are again similar to each other. We attribute this variation to the variation in the inputs — inputs 1 and 3 share a seismic data set and starting velocity model, as do inputs 2 and 4. The differences in these models underscore the importance of including representative inputs in the test input set that elicit the full range of program failure and timing behavior. The R^2 value for the composite model is around 44%, which reflects the differences in the data used to construct the model. The coefficients c_1 and c_2 are very small, which indicates that the distortion is hardly affected by task failures. It is therefore possible to fail a very large proportion of the tasks without incurring substantial distortion.

The observed biases for the distortion sampling runs are all close to zero, which indicates that failing tasks does not systematically bias the outputs in any direction.

4.3.3 Predictive Power and Confidence Bounds

We evaluate the predictive power of our technique by obtaining a composite model for our learning inputs (we selected inputs 1, 2, and 4) and applying the model to the observations for our unseen input (input 3). Of the 769 observations from the distortion sampling runs for the unseen input 3, 38 (or approximately 5%) fell above the upper 95% confidence interval from the composite model, which indicates that the composite model was able to successfully predict

$$
\begin{aligned}
\hat{d}_1(x_1,x_2) &= -0.0002[\pm 0.0009] + 0.056[\pm 0.005]x_1 + 0.060[\pm 0.004]x_2 \\
\hat{d}_2(x_1,x_2) &= 0.0007[\pm 0.0001] + 0.014[\pm 0.007]x_1 + 0.016[\pm 0.007]x_2 \\
\hat{d}_3(x_1,x_2) &= 0.0006[\pm 0.0009] + 0.052[\pm 0.004]x_1 + 0.047[\pm 0.004]x_2 \\
\hat{d}_4(x_1,x_2) &= 0.0004[\pm 0.00009] + 0.009[\pm 0.0005]x_1 + 0.011[\pm 0.0004]x_2 \\
\hat{d}_{comp}(x_1,x_2) &= 0.0003[\pm 0.0006] + 0.027[\pm 0.003]x_1 + 0.029[\pm 0.003]x_2
\end{aligned}
$$

Figure 6: Distortion Models for String

$$
\begin{aligned}
\hat{s}_1(x_1,x_2) &= 1.0[\pm 0.0010] + -0.81[\pm 0.005]x_1 + -0.09[\pm 0.005]x_2 \\
\hat{s}_2(x_1,x_2) &= 1.0[\pm 0.0013] + -0.63[\pm 0.007]x_1 + -0.14[\pm 0.006]x_2 \\
\hat{s}_3(x_1,x_2) &= 1.0[\pm 0.0002] + -0.93[\pm 0.001]x_1 + -0.036[\pm 0.001]x_2 \\
\hat{s}_4(x_1,x_2) &= 1.0[\pm 0.0007] + -0.87[\pm 0.004]x_1 + -0.047[\pm 0.003]x_2
\end{aligned}
$$

Figure 7: Timing Models for String

the distortion behavior of the unseen input. A final question is the size of the confidence bounds. The maximum upper 95% confidence bound for any sampled task block failure point x_1, x_2 in any of the 3076 distortion sampling runs was 0.0229. An appropriate probabilistic bound on the distortion at a 50% task failure rate for executions of task blocks 1 and 2 is therefore approximately 0.063.

For small task failure rates it is possible to obtain a tighter confidence bound. Specifically, we discarded all observations x_1, x_2, d with x_1, x_2 greater than 1% and used regression on the remaining observations to obtain the 1% failure rate model. We then used this model to compute the upper 95% confidence bound for all of the 761 points with x_1, x_2 less than or equal to 1%. The maximum bound was 0.00171. An appropriate bound on the distortion for task failure rates below 1% is therefore approximately 0.0026.

4.3.4 Time/Accuracy Trade-Offs

Figure 7 presents the timing models for String. The F values for all of these models are in the tens to hundreds of thousands; the corresponding p values are less than 0.0001, and the R^2 values are close to 99%, which indicates that the models almost perfectly explain the variation in the data.

The values of the timing model coefficients c_1 and c_2 indicate that the program spends most of its time in task block 1, although there is some variation between the different inputs. The distortion/timing model coefficient ratios clearly favor failing executions of task block 1. The distortion model coefficients c_1 and c_2 are comparable, indicating that comparable failure rates of tasks from the two task blocks cause comparable distortions. But the timing model coefficient c_1 is significantly smaller than c_2. Failures of executions of task block 1 therefore provide a much larger reduction in execution time than failures of executions of task block 2.

4.4 SOR

We ran SOR on six different inputs: 96ga, 96gb, 192ga, 192gb, 384ga, and 384gb. All of the inputs are taken from the SPLASH benchmark Ocean [19, 18], which simulates the role of eddy and boundary currents in influencing large-scale ocean movements.

4.4.1 Task Blocks and Criticality Testing

Search has three task blocks. Executions of the first task block copy the input into a work array. Executions of the second task block operate on the work array to solve the system of equations. Executions of the third task block copy the solution from the work array into the output array. Both the first and third task blocks are critical — failures in executions of these task blocks either cause SOR to solve the wrong set of equations or to fail to copy out part of the solution. In either case the result is significant distortion. As we discuss further below, the second task block is failable.

4.4.2 Distortion

An examination of the distortion sampling runs shows that failures of executions of task block 2 do not affect the distortion — in fact, SOR exhibits essentially zero distortion regardless of the task failure rate! An examination of the application reveals the reason for this behavior — the solution algorithm contains a termination test that iterates the solver until all errors in the solution fall below a certain precision bound.

4.4.3 Execution Times

Figures 8 through 13 plot the execution times of the distortion sampling runs as a function of the task failure rate in those runs. Unlike all of the other applications, increasing the task failure rate increases, rather than decreases, the execution time of the computation. The reason for this increase is that failing tasks causes the algorithm to execute more iterations before it converges. The net effect is an increase in the computation time. For this program, it is possible to fail tasks to tolerate hardware faults with no loss in accuracy. But failing tasks in an attempt to reduce the execution time is counterproductive.

4.5 Discussion

One of the benefits of our approach is that it can identify critical parts of the computation whose failures have a large effect on the accuracy of the result. The clear pattern that emerges is that tasks that move or copy data are critical — if they fail, the downstream computation that reads the data is usually unable to generate an acceptable output because its input diverges significantly from the correct input. Tasks that create data structures are even more critical — a program that attempts to access an incompletely constructed data structure usually fails because of an addressing exception and is unable to produce any result at all. We expect this pattern to also hold for different programs.

4.5.1 Failable Tasks

In retrospect, it is possible to reconstruct the reasons why the task failures cause the applications in our study behave the way they do. The failable tasks in Water, Search, and String all either compute a set of contributions that are combined to obtain a final result, or combine these contributions to produce the final result. Although the programs themselves perform complex, detailed computations, it is possible to come up with a relatively simple high-level characterization of the behavior of each program that explains the observed results.

At a high level, Water essentially computes sums of positive numbers. The net effect of failing tasks is to remove some of the positive numbers from the sums. On average, the resulting relative reduction in the values of the sums will be roughly proportional

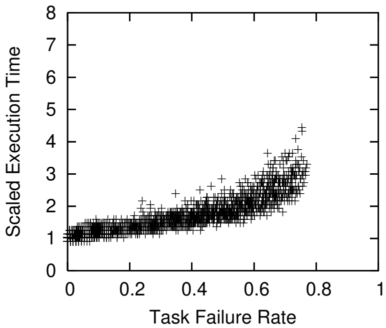

Figure 8: SOR 96ga

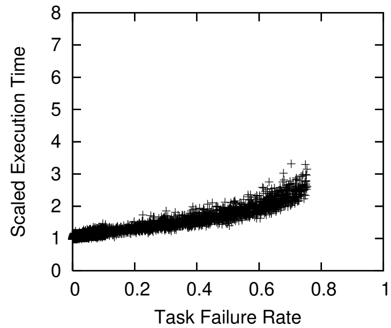

Figure 9: SOR 192ga

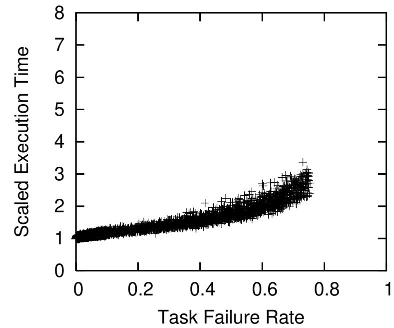

Figure 10: SOR 384ga

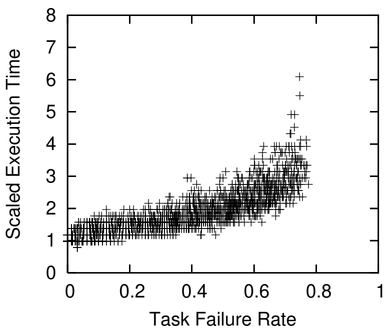

Figure 11: SOR 96gb

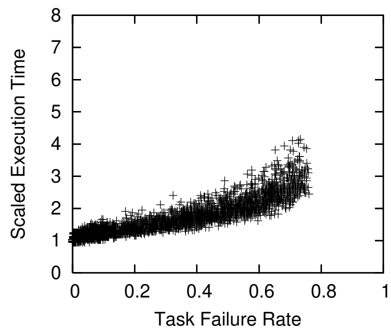

Figure 12: SOR 192gb

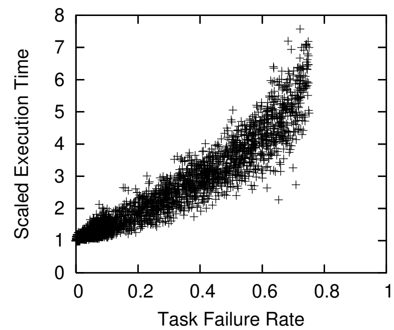

Figure 13: SOR 384gb

to the percentage of numbers removed from the sums. The coefficients in the distortion model capture the relative contribution of each partial sum to the final output total energy of the system of water molecules. Because all of the summed numbers are positive, it is possible to model the bias as a linear function of the task failure rates and apply that bias to correct the output. One can view the resulting computation as randomly selecting a subset of the numbers to sum, computing the sum of that subset, then using the size of the subset to extrapolate the partial sum to obtain an estimate of the sum of all of the numbers.

At a high level, the tasks in Search essentially sample a population of electron paths. The net effect of failing a task is to discard the samples that the task would have performed. The net effect of performing fewer samples is that the resulting estimate of the property of interest in the population may be somewhat less accurate. The coefficient c_1 in Search's distortion model indicates that failing half the tasks (in effect, performing half of the number of samples) can cause a distortion of around 0.06. To place this loss of accuracy in perspective, consider that simply changing the random number seed that drives the Monte Carlo simulation in Search can cause a distortion of 0.08 in the failure-free computation.

At a high level, the failable tasks in String either sample a population of rays projected through the velocity model of the geology between two oil wells or combine the results of this sampling process. Task failures therefore have the effect of discarding projected rays. Because the combination operator averages the contributions, there is no bias and failures have little effect on the accuracy of the final computation.

SOR is a special case in our set of sample programs. Because of the termination test in the solution algorithm, the algorithm is, in effect, already set up to tolerate failures. The underlying mathematics guarantees that the algorithm will eventually converge even in the face of failures. The only question is how long this convergence will take. One interesting aspect of our methodology is that it reveals the presence of this property in the computation.

4.5.2 *Implications for Other Programs*

In general, we anticipate that many computations will turn out to have the same general pattern as Water, String, and Search. Many graphics computations have this high-level pattern [11], as do information retrieval computations [5]. And of course other scientific computations share this pattern [1]. We anticipate that our technique can be applied to all of these kinds of programs, to make the programs more robust in the face of both software and hardware errors, and to reduce the execution time while maintaining an acceptable distortion.

5. RELATED WORK

Dealing with hardware failures in long-running scientific computations is a well-known problem. A standard approach to this problem is checkpoint/restore [12], which periodically checkpoints the state of the computation on stable storage. If the hardware running the computation fails, the state is restored from stable storage when the system comes back up and the computation continues from the last checkpoint. Researchers have also developed techniques for making distributed scientific computations robust in the face of partial failures. The basic technique is to farm tasks out to a pool of machines, then respond to failures of these machines by reexecuting the failed tasks on other machines [16, 3]. The goal is to obtain a computation that survives hardware failures to produce the same result as the computation without failures. Unlike our proposed technique, they are not designed to enable computations to survive software errors, nor do they open up the possibility of reducing the execution time while staying within accuracy bounds.

We know of several software systems that use ad-hoc techniques to survive software errors. The Lucent 5ESS telephone switch and

IBM MVS operating systems, for example, both apply hand-coded data structure inconsistency detection and repair to recover from errors that corrupt the data structures. The reported results indicate an order of magnitude increase in the reliability of the system [9]. MapReduce discards records that cause the record processing task to fail multiple times [5]. We know of a graphics rendering algorithm that discards computations associated with problematic triangles instead of including complex special-case code that attempts to render the triangle into the scene [11]. These systems illustrate the value of recovery techniques that allow systems to execute through errors and faults to generate acceptable but potentially distorted results. None of these systems provides any indication of how accurate the resulting outputs are likely to be.

Researchers have recently developed several general techniques that allow computations to execute through software errors without failing. Examples include failure-oblivious computing [15], acceptability-oriented computing [14], data structure repair [6], and transactional function termination [17]. While the empirical results indicate that these techniques often enable computations to continue executing successfully, they provide no indication of how accurate or correct the resulting outputs are likely to be.

Asynchronous iteration [8] relaxes standard ordering constraints in iterative solvers to allow parallel processors to proceed without synchronization, typically as they operate on different regions of the problem that potentially share border elements. The lack of synchronization introduces nondeterminism into the computation as different processors asynchronously read and write the same (typically border) elements. Our technique, in contrast, completely eliminates some computations rather than running computations asynchronously at variable execution rates.

To the best of our knowledge, our research is the first to provide probabilistic accuracy bounds for program outputs in the face of failures within the computation.

6. CONCLUSION

Software errors and hardware faults can substantially impair the ability of a given program to deliver acceptable results to its users. Standard program execution platforms can leave programs quite vulnerable to errors and faults — a standard response upon encountering an error or failure is to terminate the computation.

We have developed an alternate approach in which tasks provide an effective fault containment mechanism — when a task encounters an error or fault, the execution platform simply discards the task and continues on to complete the remaining tasks in the computation. Probabilistic distortion models bound the resulting distortion in the output; probabilistic timing models enable the user to predict the effect on the overall execution time of the computation.

The distortion model allows users to quantitatively evaluate the effect of any failures on the accuracy of the output. In many cases the model may allow users to confidently accept results produced by computations that have encountered failures during their execution; without such a model, the user would have no good way of estimating the potential distortion and therefore no confidence in the result. We anticipate that the availability of the distortion model will make techniques that discard tasks when they encounter faults much more acceptable to users and therefore much more widely used. The net effect will be a significant increase in the reliability and robustness of the corresponding computations.

7. REFERENCES

[1] J. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324(4):446–449, Dec. 1986.

[2] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[3] R. Blumofe and P. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symposium*, Anaheim, CA, Jan. 1997.

[4] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Low-energy electron/atom elastic scattering cross sections for 0.1-30keV. *Scanning*, 17(4):250–253, July/August 1995.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[6] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering*, St. Louis, MO, May 2005.

[7] R. Freund and R. Littell. *SAS System for Regression*. SAS Publishing, 2000.

[8] A. Frommer and D. Szyld. On asynchronous iterations.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[10] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[11] T. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–78, Aug. 1986.

[12] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.

[13] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1994.

[14] M. Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session*, Oct. 2003.

[15] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[16] D. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*. ACM, New York, Nov. 1994.

[17] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks. Technical Report CUCS-031-04, Columbia University Computer Science Department, Sept. 2004.

[18] J. Singh and J. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results and implications. *Journal of Parallel and Distributed Computing*, 15(1), May 1992.

[19] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Comput. Arch. News*, 20(1):5–44, Mar. 1992.