When Piecewise Determinism Is *Almost* True

Edith Cohen   Yi-Min Wang   Gaurav Suri*

Abstract

Most existing log-based recovery techniques assume perfect piecewise determinism. In practice, however, the behavior of certain events is determined by the execution environment, and is not replayble. In this paper, we introduce a probabilistic piecewise deterministic model, and develop a probability tracking mechanism to achieve high-confidence output commits. The paper also addresses the issue of scalability by proposing an efficient on-line algorithm that accurately estimates the probability of determinism by piggybacking only constant-size vectors.

1 Introduction

Checkpointing and rollback recovery are well-known techniques to provide fault tolerance to long-running distributed applications. During failure-free execution, processes periodically save their states onto stable storage. In the event of a failure, the saved checkpoints are used to roll back the system to a consistent state which minimizes the lost work. Applying rollback recovery techniques to service-providing applications further requires the following issues to be addressed: *localized recovery*, *nonreplayable events*, *continuous availability*, and *high-confidence output commit*.

When a server process fails, it is desirable that the failed process can recover locally without affecting the clients. One common way to achieve that is to use log-based recovery techniques [1, 2] which employ event logging in addition to checkpointing so that a failed process can recover more pre-failure states by replaying event logs beyond the latest checkpoint. Most existing log-based recovery techniques assume perfect piecewise determinism; i.e., every nondeterministic event can be replayed. In practice, service-providing applications often contain nonreplayable events, the behavior of which is determined by the execution environment, for example, resource-related events. Due to the requirement of continuous availability, a service-providing application cannot simply exit when the assumption of piecewise determinism fails; it needs to take a different execution path, for example, by entering a degraded mode of operation. The existence of nonreplayable events also complicates the issue of output commit because the state from which an output is generated may never be guaranteed to be recreatable. Therefore, the challenge is to achieve high-confidence output commit with a reasonable overhead.

Assuming perfect piecewise determinism is overly optimistic for service-providing applications. On the other hand, abandoning event replay once a nonreplayable event is encountered is overly pessimistic because such an event may behave the same way before and after the failure with a high probability. In this paper, we propose a **probabilistic piecewise deterministic model**. By estimating the probability of each nonreplayable event behaving the same way before and after a failure, and by propagating the probabilities through message piggybacking, each process can on-line calculate the *probability of determinism* associated with its current execution. This *probability of determinism* forms the basis for a **probabilistic output commit protocol**. In the protocol, each output commit point is assigned an *undo cost* based on the impact of its rollback on service quality. A cumulative expected undo cost is calculated based on all the output commit points since the previous coordinated checkpoint. Output-driven logging [3], which usually has a lower overhead, is performed if the cost is below a global threshold; otherwise, a higher overhead coordinated checkpointing session is initiated. Finally, we address the scalability issue for the transitive probability vectors. An algorithm is derived to compute accurate estimates of the *probability of determinism* by piggybacking only constant-size vectors.

2 Log-based Rollback Recovery

We consider a message-passing system consisting of N communicating processes. A *system state* with N process states, one from each process, is *consistent* if it does not record the receiving event of any message m without also recording the sending event of m. For example, if process P_1 in Figure 1 rolls back from RB to checkpoint B which does not record the sending events of m_b and m_a, then P_0 should also roll back to checkpoint A to exclude the receiving events of m_b and m_a from the system state. Event logging based on piecewise determinism can be used to localize P_1's rollback. The piecewise deterministic model divides process execution into a number of deterministic *state intervals* separated by nondeterministic events. By logging and replaying the nondeterministic events, a failed process can recover more pre-failure states. For example, in Figure 1, process P_1 can replay the event logs of m_d, m_c and e to reconstruct its state up to RB without forcing P_0 to roll back as well.

In order to focus on the issue of non-perfect piecewise determinism, we base our presentation on the following assumptions: (1) periodic coordinated checkpointing is used to simplify the garbage collection of checkpoints and message logs, and asynchronous event logging is employed to achieve a maximum recoverable state beyond the set of coordinated checkpoints; (2)

*The authors are with AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

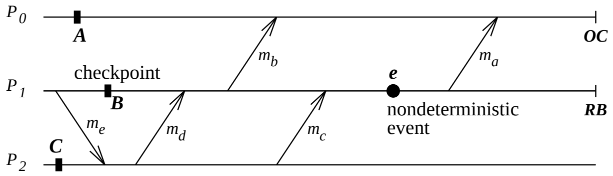

Figure 1: Checkpointing and message logging.

there exists a central server CS[1] responsible for initiating coordinated checkpoints, calculating the maximum recoverable state, selecting the target machine for process migration, and deciding the appropriate action in case of an output commit.

During failure-free execution, server CS periodically broadcasts a marker to initiate checkpoint coordination [5]. Every nondeterministic event is temporarily logged in a volatile buffer. The volatile logs of a number of events are periodically flushed to stable storage. Each process P_i maintains a size-N *transitive dependency vector* D_i [1], where $D_i[j]$ records the highest index of any state interval of P_j on which P_i's current state interval (with index $D_i[i]$) transitively depends. The vector D_i is piggybacked on every message that P_i sends out so that the receiver can properly update its own vector by taking the coordinate-wise maximum of the piggybacked vector and its current vector. When a process needs to perform an output commit, all the events that its current state transitively depends on have to be logged on stable storage. For example, if process P_0 in Figure 1 needs to perform an output commit at OC, all the messages and the event e have to be logged.

Upon a failure, some of the volatile event logs may be lost. Server CS first migrates the failed processes, then collects the information about the available message logs and their dependency vectors. The maximum recoverable state which minimizes the amount of total rollback is calculated and broadcast. Every process whose current state interval does not belong to the maximum recoverable state is required to roll back to its previous checkpoint and replay its event logs up to the maximum recoverable state.

3 Nonreplayable Events

The behavior of an application can be affected not only by the program itself and the input data, but also by the execution environment. Figure 2 shows the typical execution environment (nonshaded components) for a single-thread Unix application linked with some software libraries (shaded components). A program behaves deterministically as long as the execution flow is completely determined by the shaded components. Nondeterminism is introduced when the application program interacts with the execution environment either directly or indirectly through the libraries, and such interactions go through the operating system. For example, message receiving events are nondeterministic events because an application needs to interact with the "communication channel environment"; they are *replayable* events because fixing message receiving order does not violate any constraints of the physical environment.

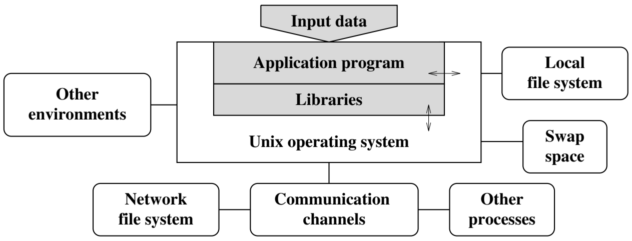

Figure 2: Application programs interact with the execution environment through the operating system.

In general, a nonreplayable nondeterministic event exists whenever a program construct similar to the following one appears:

```
if (expression) {Action A} else {Action B}
```

where the value of `expression` depends on the runtime execution environment and cannot (or should not) be controlled by the applications. For example, if `expression` is testing a memory allocation failure, then the program flow is determined by whether or not the system has sufficient remaining swap space to satisfy the memory request. `Action A` can be an exception handler that either performs memory garbage collection or produces suboptimal solutions, while `Action B` is the normal routine that delivers optimal services. However, not every memory allocation request is a nondeterministic event. Usually, a Unix program uses the library function `malloc()` to allocate memory, and `malloc()` provides a caching mechanism that manages previously allocated and then freed memory blocks. If a memory request can be satisfied by the cached memory blocks, then it is not a nondeterministic event. A nonreplayable nondeterministic event is created when `malloc()` decides that it is necessary to invoke the Unix system call `sbrk()` to ask for more memory, because the result of `sbrk()` depends on the "swap space environment." In Section 4, we provide some heuristics which can form the basis of the detection and classification of nonreplayable events.

In this paper, we distinguish between two types of nonreplayable nondeterministic events: *Heisenevents* (whose behavior is likely to change across retries) [6] and *Bohrevents*. The classification is based on the desire to have every nondeterministic event behave the same way *before* and *after* the failure in order to make more state intervals recoverable.

Heisenevents

A nonreplayable nondeterministic event is called a **Heisenevent** if, when its behavior during recovery is different from its pre-failure behavior, there is a good chance that the former will become the same as the latter within a number of retries. For example, an `sbrk()` system call may succeed in the original execution but fail during recovery due to a temporary machine overload. The same call may succeed after a number of

[1] The central server has a backup and is supported by a fail-over mechanism based on heartbeats and checkpointing [4] to avoid single point of failure.

retries if any of the applications sharing the same machine exits and releases its occupied memory blocks. In mobile computing, communication over wireless links in a possibly wide-area setting is expected to introduce more transient problems that constitute Heisenevents.

Bohrevents

A nonreplayable nondeterministic event is called a **Bohrevent** if, when a discrepancy is detected between its pre-failure and recovery-time behaviors, any retry will most likely not be able to resolve the difference. If a program's execution flow depends on the resource configuration or other attributes of the machine that it is running on, then the program will certainly behave in a different way when being restarted on another machine with a different resource availability or on the same mobile host but in a totally new environment. For example, when a mobile host moves far away from a stationary information server, it may decide to provide approximate answers based on the slightly out-of-date information cached locally; applications running on a mobile host may switch to a different operational mode that requires less computation, communication and disk access in order to conserve power. Time-dependent events can be another example of Bohrevents. A less obvious example is when system calls such as `sbrk()` fail in the original execution but succeed during recovery. Although it is possible to ask a machine to "pretend" that it has run out of memory, it is usually not desirable. Therefore, such events should be treated as Bohrevents.

4 Probabilistic Piecewise Determinism

It becomes clear that an important recovery issue for service-providing applications is to properly handle nonreplayable nondeterministic events without losing the benefits of piecewise determinism. One way to allow a mix is to ask a process to turn off event logging/replay whenever a nonreplayable nondeterministic event is encountered until the next checkpoint is taken [3]. However, it may be too drastic an approach if the number of nondeterministic events is large but most of them are likely to behave the same way before and after a failure. We propose a *probabilistic piecewise deterministic model* based on the following assumptions: (1) Every nondeterministic event e that can be identified is logged. If the event is replayable, the log will be used for replay; otherwise, it is recorded as either a Heisenevent or a Bohrevent with a predicted probability p_e that the event will be **well-behaved**, i.e. it will behave the same way in case of a recovery as it did during failure-free execution. The log will be used to check against its recovery-time behavior to verify the determinism; (2) Every nondeterministic event starts a new state interval and every state interval s is associated with a *probability of determinism* $\mathcal{P}_d(s)$ which is defined to be the probability of s being recreatable if all N processes roll back to the latest coordinated checkpoints and replay all the event logs that s depends on.

Let $\mathcal{H}(s)$ and $\mathcal{B}(s)$ denote the set of Heisenevents and Bohrevents since the latest coordinated checkpoint, on which s depends. Our algorithm uses the following recovery protocol. In the process of event replay towards the calculated maximum recoverable state, if a Bohrevent b behaves differently from its logged counterpart, then a new maximum recoverable state is calculated by discarding all the event logs that depend on b since any retry would not help. In contrast, when a Heisenevent has a different behavior, the current recovery process is restarted towards the same maximum recoverable state. Let k denote the predetermined maximum number of such retries, $\mathcal{P}_{\mathcal{B}}(s) = \prod_{b\in\mathcal{B}(s)} p_b$, and $\mathcal{P}_{\mathcal{H}}(s) = \prod_{h\in\mathcal{H}(s)} p_h$. The *probability of determinism* $\mathcal{P}_d(s)$ can be calculated as follows:

$$\mathcal{P}_d(s) = \prod_{b\in\mathcal{B}(s)} p_b \left(1 - \left(1 - \prod_{h\in\mathcal{H}(s)} p_h\right)^k\right) \qquad (1)$$

$$= \mathcal{P}_{\mathcal{B}}(s)\left(1 - (1-\mathcal{P}_{\mathcal{H}}(s))^k\right) .$$

That is, state interval s is recreatable if all its depended-on Bohrevents are well-behaved and all its depended-on Heisenevents are well-behaved in one of the k retries. We first describe an on-line tracking mechanism for maintaining the *probability of determinism* based on message piggybacking, and then describe its application to a high-confidence output commit protocol.

4.1 Probability tracking

It is not sufficient to piggyback $\mathcal{P}_d(s)$ on each outgoing message for maintaining the *probability of determinism* on-line because a state interval s_1 may reach another state interval s_2 through more than one dependency chains. We use a probability tracking scheme that is similar to the transitive dependency tracking. Let $\mathcal{B}_j(s)$ and $\mathcal{H}_j(s)$ denote the set of events of P_j contained in $\mathcal{B}(s)$ and $\mathcal{H}(s)$, respectively. In addition to the transitive dependency vector D_i, each process P_i maintains two size-N *probability vectors* BV_i and HV_i:

$$BV_i[j] = \prod_{b\in\mathcal{B}_j(s)} p_b, \qquad HV_i[j] = \prod_{h\in\mathcal{H}_j(s)} p_h \qquad (2)$$

where s is P_i's current state interval. Both BV_i and HV_i are initialized to all 1's when P_i takes a (coordinated) checkpoint, and are piggybacked on each outgoing message. Upon receiving a message m with tags BV_i and HV_i, and m was not sent before the previous coordinated checkpoint, process P_k updates BV_k by taking the coordinate-wise *minimum* of BV_k and BV_i, and updates HV_k in a similar manner.

With the above probability tracking mechanism, the *probability of determinism* for process P_i's current checkpoint interval s can be computed by first calculating

$$\mathcal{P}_{\mathcal{B}}(s) = \prod_{0\le j\le N-1} BV_i[j], \quad \mathcal{P}_{\mathcal{H}}(s) = \prod_{0\le j\le N-1} HV_i[j] \qquad (3)$$

and then substituting $\mathcal{P}_{\mathcal{B}}(s)$ and $\mathcal{P}_{\mathcal{H}}(s)$ in Eq. (1) to obtain $\mathcal{P}_d(s)$.

4.2 Probabilistic output commit protocol

Most existing work on log-based recovery assumes that any output sent to the outside world cannot be revoked because the outside world does not have the capability of performing rollbacks. For most service-providing applications, however, the outside world *can* be rolled back if necessary but it is undesirable and may involve a certain penalty depending on its damage to the overall service quality. For example, when a failure of a server process destroys a customer's screen, it is possible for the customer to "roll back" by doing a re-login; but the rollback annoys the customer and so its frequency should be minimized to reduce the risk of losing the customer to other competitors. Furthermore, in the presence of nonreplayable nondeterministic events, it is in general not possible to guarantee 100% output commits by using mere event logging and replay. This motivates a **probabilistic output commit protocol** based on the probabilistic piecewise deterministic model.

We assume that each output commit point ϑ belonging to a state interval s is associated with an undo cost C_ϑ, which is the expected penalty for rolling back the commit point, based on its potential impact on service quality. When there is more than one output commit point between two consecutive set of coordinated checkpoints, the proper output-driven action may depend on *all previous output commit points since the latest coordinated checkpoints.* For example, it is possible that, although the Heisen and Bohr events that an output commit point ϑ_1 depends on are all well-behaved, event replay is always aborted before ϑ_1 is reached because of an ill-behaved event that another output commit point ϑ_2 depends on.

The central server CS maintains the cumulative probability vectors CHV and CBV corresponding to the Heisenevents and Bohrevents, respectively. It also records the cumulative undo cost of all the output commit points that have occurred since the last coordinated checkpoint. CS also maintains T_g, which is a global threshold cost signifying the penalty that the system can tolerate if some outputs are rolled back.

Whenever a process P_i reaches an output commit point ϑ it sends its vectors HV_i, BV_i and its undo cost C_ϑ to the CS. The CS combines these vectors with the cumulative vectors that it maintains, and updates the expected undo cost. If the cost is more than the threshold value, it initiates a coordinated checkpoint; otherwise output-driven logging is performed. The details of the protocol are given in Figure 3.

4.3 Simplified Implementation Model

The first issue that arises for an implementation of the proposed algorithm is the ability to identify events. Another requirement that may be hard to satisfy in some environments is the ability to classify events into Heisenevents and Bohrevents. In some application environments, it may be difficult to assign p_e for each nonreplayable nondeterministic event. In this section, we present heuristics that can be used to estimate these parameters.

Event Identification: As shown in Figure 2, a nondeterministic event can occur whenever a process interacts with the execution environment. Thus a simple heuristic is to assume each *system call* to be a nondeterministic event, since in a Unix-like environment all interactions with the execution environment take place through system calls.

Event Classification: The motivation behind the classification of events into Bohrevents and Heisenevents is that we can save on recovery time by not retrying more than once if we know an event to be a Bohrevent. A conservative heuristic is thus to assume all events to be Heisenevents. This heuristic also reduces the space requirements of the protocol, since only one size-N probability vector now needs to be tagged to messages instead of two.

Probability Assignment: In situations where it is difficult to assign a value to p_e, one can use an integer-valued *risk factor* as an approximation to probability. The risk factor for an event indicates the likelihood of an event being *ill-behaved.* The risk factor is a relative measure and can be calculated by first subdividing the system calls into subsets based on the execution environment, and then assigning a risk factor to each subset. For example, network based system calls have a higher risk factor than system calls that operate on files which in turn have a higher risk factor than those operating on memory.

With the above simplifications, each process P_i now maintains a size-N Risk Factor Vector (RFV_i) which is initialized to all 0's at a checkpoint and updated to the coordinate-wise maximum upon receiving a message. Whenever a nonreplayable nondeterministic event occurs, the risk factor of that event is added to $RFV_i[i]$. Server CS updates its RFV vector using the coordinate-wise maximum operation.

The determination of when a coordinated checkpoint is due is done by the central server CS. It compares the cumulative risk factor to a threshold T, and initiates a coordinated checkpoint when the risk factor is above T. Two approaches can be used to determine the value of T. (1) **Individual output commit guarantees,** Each output commit point has a pre-defined and independent risk-factor tolerance that is sent to CS. The server CS sets the threshold T to be the minimum risk-factor tolerance of any of the output commit points. (2) **Global undo-cost considerations,** The server CS maintains the global *undo cost* parameter C_{oc} and utilizes the constant T_g, as suggested in the previous section. The intuition is that if C_{oc} increases, our expected loss from a failure becomes larger. Hence, to balance that, we decrease the tolerated risk factor.

5 Efficient Probability Estimation for Large Scale Applications

When N is large, the per-message overhead of piggybacking the transitive dependency vector and probability vectors (all of size-N) may become unacceptable. To address the scalability issue for dependency tracking, Lowry et al. [7] introduced the concept of *recovery unit gateways* to compress the vector at the cost of introducing false dependencies. Direct dependency tracking [1, 3] can be used to piggyback only the sender's current state interval index. When tran-

Application Process P_i

Initialization, Coordinated Checkpoint

for $j = 0, N-1$ do $\{BV_i[j] = HV_i[j] = 1;\}$

Output Point

send BV_i, HV_i, C_v to CS;

Message Receive with tags MBV and MHV

for $j = 0, N-1$ do

$BV_i[j] = min(BV_i[j], MBV[j])$;

$HV_i[j] = min(HV_i[j], MHV[j])$;

endfor;

Nondeterministic event

if (Heisenevent) $HV_i[i] *= p_{event}$

else $BV_i[i] *= p_{event}$;

Legend

HV_i, BV_i: prob. vectors for Heisen and Bohr events

C_v: undo cost for output v

p_{event}: prob. of event being well-behaved

Central Server

Initialization, Coordinated Checkpoint

for $j = 0, N-1$ do $\{CBV[j] = CHV[j] = 1;\}$

$C_{oc} = 0$;

Output Point Message Receive with tags BV_{oc} and HV_{oc}

for $i = 0, N-1$ do

$CBV[i] = min(CBV[i], BV_{oc}[i])$;

$CHV[i] = min(CHV[i], HV_{oc}[i])$;

endfor;

$C_{oc} += C_v$;

Compute P_d using Eq. (1) and (3);

$C_e = C_{oc} * (1 - P_d)$;

if ($C_e \geq T_g$) initiate coordinated checkpoint;

else perform output-driven logging;

Legend

CHV, CBV: cumulative prob. vectors

C_{oc}: cumulative penalty for outputs since last coordinated chkpt

C_e: expected cost of outputs since last coordinated chkpt.

T_g: global penalty threshold

Figure 3: Pseudo-code for probabilistic output commit protocol

sitive dependencies are needed for an output commit, the direct dependencies must be "assembled" either at a central server or through several rounds of message exchange. To address the scalability issue for probability tracking, we develop an algorithm that requires piggybacking only a constant-size vector on each message to allow high-confidence, accurate on-line estimates of the *probability of determinism*.

5.1 The on-line estimation framework

We first describe an algorithm for estimating the sum of risk factors, based on the size-estimation framework previously developed by Cohen [8], and then extend it to handle the on-line estimation of the *probability of determinism*. Given a checkpoint and communication pattern, we construct a weighted directed graph as follows. Divide a process execution at checkpoint events, output commit events, message sending and receiving events. Each interval v between two consecutive events of the same process, called an *E-interval*, is represented by a node with its weight w_v equal to *the sum of the risk factors of the nonreplayable nondeterministic events in* v. An edge is drawn from node A to node B if (1) interval B immediately follows interval A on the same process or (2) a message is sent from the end of interval A and received at the beginning of interval B. The problem of "estimating the sum of the risk factors of the nonreplayable nondeterministic events that an output commit point ϑ depends on" translates into the problem of "estimating the sum of the weights of the nodes that can reach the node representing the interval beginning with ϑ."

We select a positive integer n according to the desired accuracy/confidence tradeoffs. Each process P_i maintains a size-n vector V_i and the *accumulated risk factor* w for the current E-interval. All entries of V_i are initialized to $+\infty$ and reinitialized to the same whenever a checkpoint is taken. At the end of an E-interval, the value of w becomes the weight of the node representing the interval. If the ending event is not a checkpoint, then a size-n vector A is drawn based on the weight w as follows: each entry of A is sampled independently from the probability distribution with a density function $w\exp(-wt)$ (an alternative is to sample y uniformly from $[0,1]$ and use the value $-\ln(y)/w$); the V_i vector is updated to the *coordinate-wise minimum* of V_i and A. If the ending event is a message send, the message is tagged with V_i; if it is a message receive, the receiver P_j's V_j vector is updated to the coordinate-wise minimum of V_j and the V_i vector tagged on the message; if it is an output commit, the V_i vector and the threshold are sent to server CS. Server CS updates the V_{CS} vector to be the coordinate-wise minimum of V_{CS} and the newly received V_i. To estimate "the sum of the weights of the nodes that can reach any of the output commit points since the latest checkpoint" (denoted by $W(\Theta)$), server CS calculates the following value:

$$\hat{W}(\Theta) = n / \sum_{j=1}^{n} V_{CS}[j] \tag{4}$$

The server CS compares the estimate $\hat{W}(\Theta)$ on the risk factor to a monotonically-decreasing threshold parameter T. The server CS orders a coordinated checkpoint when $\hat{W}(\Theta)$ exceeds the threshold T. Let W_s be the accumulated risk factor when we stop, that is, when $\hat{W}(\Theta) = T$. To analyze the estimate quality we consider the random variables $y \equiv \frac{W_s}{T}$ and the relative error $\frac{|W_s - T|}{T} = |1 - y|$.

THEOREM 1 *The density and distribution functions of y are independent of W_s and are given by*

$$f_n(y) = \frac{n^n}{(n-1)!} y^{n-1} e^{-ny}$$

$$F_n(y) = 1 - e^{-ny}\left(1 + \sum_{i=1}^{n-1} \frac{(ny)^i}{i!}\right)$$

for $y \geq 0$.

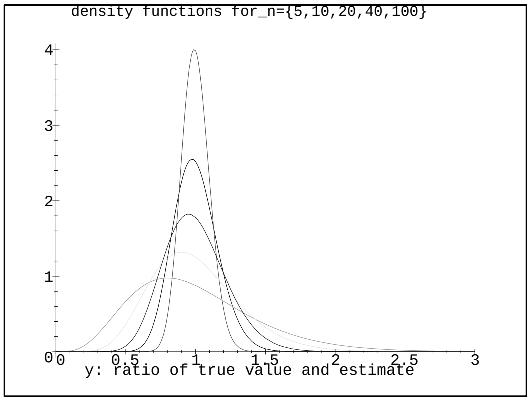

Figure 4: density $f_n(y)$

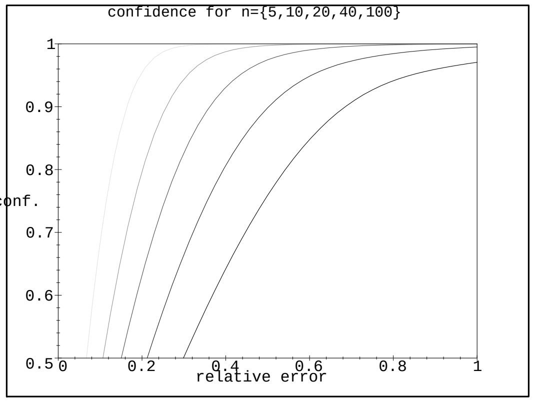

Figure 5: accuracy vs. confidence

Plots of $f_n(y)$ for $n = \{5, 10, 20, 40, 100\}$ are provided in Figure 4.

The relative error of the estimate at the stopping point is

$$\text{Prob}\left\{\frac{|W_s - T|}{T} \geq \epsilon\right\} = F_n(1-\epsilon)+1-F_n(1+\epsilon)\ . \quad (5)$$

The relation of the confidence and relative error $\frac{|W_s-T|}{T}$ for various vector sizes are plotted in Figure 5 (using Equation 5). The accuracy increases for larger vector sizes. The asymptotic convergence follows using Chernoff bounds and an analysis similar to the one provided in [8]. $\text{Prob}\{|1-y| \geq \epsilon\} = \exp(-\Omega(\epsilon^2 n))$.

By integrating we obtain the following:

COROLLARY 1 *The estimate is unbiased. The expected relative error and the variance are as follows.* $E(y) - 1 = 0$, $E(|1-y|) = \int_0^\infty |1-y| f_n(y) dy = \frac{2n^n}{e^n n!} \approx \sqrt{\frac{2}{\pi n}}$, $E((1-y)^2) = 1/n$.

5.2 On-line estimate of $\mathcal{P}_d$

We first rewrite the products in Eq. (1) in a way that the estimator of Eq. (4) can be applied. Two separate directed graphs are constructed, one each for Heisenevents and Bohrevents. Since the estimation procedures for both are basically the same, we will only discuss in detail the estimator for $\mathcal{P}_\mathcal{B}(s) = \prod_{b\in\mathcal{B}(s)} p_b$. For each E-interval v, let $p_v = \prod_{b\in v} p_b$, $R(v)$ denote the set of nodes that can reach node v (excluding v itself), and define the weight $w_v = 1 - p_v$. We then have $\mathcal{P}_\mathcal{B}(v) = \prod_{u\in R(v)}(1 - w_u)$. Since the probability estimates will be used for the purpose of output commit, we are only interested in the cases with high *probabilities of determinism* or equivalently $w_u \ll 1$. We can then use the following approximation:

$$\mathcal{P}_\mathcal{B}(v) = \prod_{u\in R(v)} (1 - w_u) \approx 1 - \sum_{u\in R(v)} w_u. \quad (6)$$

We note that an alternate way is to assign weights $w_v = -\log p_v$ and use $\mathcal{P}_\mathcal{B}(v) = \exp(-\sum_{u\in R(v)} w_u)$.

Let $\hat{W}_\mathcal{B}$ denote the estimate of $\sum_{u\in R(v)} w_u$ in Eq. (6) by applying the estimator of Eq. (4) with the random vector A being drawn based on the weight $w_v = 1-p_v$. The estimate $\hat{W}_\mathcal{H}$ for Heisenevents is similarly defined and obtained. From Eq. (1), the final estimate $\hat{W}_p$ that server CS uses to compare against the thresholds can then be calculated as:

$$\hat{W}_p = (1 - \hat{W}_\mathcal{B})(1 - \hat{W}_\mathcal{H}^k) \approx 1 - \hat{W}_\mathcal{B} - \hat{W}_\mathcal{H}^k \quad (7)$$

6 Summary

Handling nonreplayable nondeterministic events is important for service-providing applications. We have addressed the issue by presenting a probabilistic piecewise deterministic model, and a probabilistic output commit protocol which chooses between output-driven event logging and coordinated checkpointing based on the probability of determinism and the impact of revoking output on service quality. We have also presented an on-line probability estimation algorithm that addresses the scalability issue of the protocol.

References

[1] A. P. Sistla and J. L. Welch, “Efficient distributed recovery using message logging,” in *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pp. 223–238, Aug. 1989.

[2] E. N. Elnozahy and W. Zwaenepoel, “Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit,” *IEEE Trans. Comput.*, Vol. 41, No. 5, pp. 526–531, May 1992.

[3] D. B. Johnson, “Efficient transparent optimistic rollback recovery for distributed application programs,” in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 86–95, Oct. 1993.

[4] Y. Huang and C. Kintala, “Software implemented fault tolerance: Technologies and experience,” in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 2–9, June 1993.

[5] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

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

[7] A. Lowry, J. R. Russell, and A. P. Goldberg, “Optimistic failure recovery for very large networks,” in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 66–75, 1991.

[8] E. Cohen, “Estimating the size of the transitive closure in linear time,” in *Proc. 35th IEEE Annual Symposium on Foundations of Computer Science*, pp. 190–200, 1994.