Performability Modeling of Coordinated Software and Hardware Fault Tolerance

Ann T. Tai
IA Tech, Inc.
Los Angeles, CA 90024, USA
E-mail: a.t.tai@ieee.org

William H. Sanders
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: whs@crhc.uiuc.edu

1 Motivation and Problem Description

Quantitative system evaluation concerning both software design faults and hardware operational faults has not yet received enough attention. Although a few studies considered such dependability analysis, analytic evaluation of systems with integrated software and hardware fault tolerance remains a challenge. In particular, the need to distinguish the effects of software design faults from those of random hardware faults often makes problems very complex. Such evaluation may become even more difficult when we are concerned with 1) distributed computing environments, 2) results other than instantaneous steady-state measures, 3) fault tolerance beyond simple application of resource redundancy, and/or 4) combined assessments of performance and dependability, such as performability.

In this paper, we tackle the problem of performability evaluation for a scheme of coordinated software and hardware fault tolerance we developed earlier [1]. The scheme involves 1) a time-based (TB) checkpointing protocol developed by Neves and Fuchs for tolerating hardware faults, and 2) our message-driven confidence-driven (MDCD) protocol for software error containment and recovery. The two protocols coordinate through their checkpointing activities, guaranteeing that when recovering from a software or hardware error in a distributed computing environment, the system will always reach a global state that is not only consistent but also *valid* (i.e., it does not reflect the receipt of not-yet-validated messages from low-confidence processes).

The system in question comprises two application processes that interact via message passing. One of the processes is created from a low-confidence software component; this process, call it P_1^{low}, is escorted by the MDCD protocol, which lets a high-confidence version, P_1^{hi}, run in the background to enable error recovery. Thus the system has two active interacting processes, P_1^{low} and P_2 (i.e., the second application process, which is a high-confidence component), and a shadow process P_1^{hi}.

The protocol coordination scheme emphasizes avoiding potential interference between software and hardware fault tolerance techniques and enabling them to be mutually supportive. Specifically, the MDCD protocol is responsible for establishing a checkpoint in volatile storage upon the occurrence of a message-passing event that lowers our confidence in the correctness of a process state to enable software error containment and recovery. The duty of the TB protocol is to save consistent and valid checkpoints to stable storage, based on periodically resynchronized timers, to tolerate transient hardware faults. Moreover, the TB protocol requires processes to start a blocking period of a minimum length (during which a checkpoint is saved to stable storage) upon timer expiration, to ensure global state consistency. Since the required blocking period is an increasing function of clock drift, which in turn is an increasing function of the elapsed time since clock synchronization, the system must periodically undergo resynchronization to prevent the blocking period from becoming longer than the time required to save a checkpoint to stable storage.

When a process fails to pass an AT, software error recovery will be invoked and a process will roll back to its volatile-storage checkpoint if its `dirty_bit` equals 1; otherwise, the process will roll forward. However, when a process's host encounters a transient hardware error, all the processes will roll back to their stable-storage checkpoints.

The coordination scheme does not rely on costly message exchange among the participating protocols, which preserves the performance advantages of the original MDCD and TB protocols. Accordingly, it is important to verify that the scheme indeed enhances a system's performability.

2 Reward Model Approach

2.1 Performability Measure

We define a performability measure that quantifies a process's *effective work* that is accumulated in a given interval of time. More specifically, the measure quantifies the expected amount of computation time that a process devotes to the application during a resynchronization cycle.

It can be shown that the number of stable-storage checkpointing intervals between two resynchronization points is a constant N, given that the system is error-free during the cycle. Furthermore, since occurrence of a detected software or hardware error will also cause all the interacting processes to be resynchronized at a consistent global state, N can be considered the *least upper bound* of the number

of stable-storage checkpointing intervals that the system undergoes before it reaches the next resynchronization point. Accordingly, we use N_{max} to denote this upper bound and use $G[N_{max}]$ to denote the performability measure defined above. Furthermore, we choose P_2 to be the process in question, since P_2 is an active process and is involved in all types of error containment and recovery activities.

2.2 Upper-Layer Reward Model

The upper-layer model consists of the following states:

- S_0: The state in which the error contamination source P_1^{low} has not encountered fault manifestation, and P_2 is not considered potentially contaminated.
- S_1: The state in which P_1^{low} has not encountered fault manifestation, but P_2 is considered potentially contaminated.
- S_2: The state in which P_1^{low} has encountered fault manifestation, but P_2 is not considered potentially contaminated.
- S_3: The state in which P_1^{low} has encountered fault manifestation, and P_2 is considered potentially contaminated.
- S_4: The state in which one of the processors in the system encounters a transient error, so that the system undergoes hardware error recovery and resynchronization.
- S_5: The state in which a software error is detected so that the system undergoes software error recovery and resynchronization.

Figure 1 illustrates the transitions among the states. According to their definitions, S_4 and S_5 are the absorbing states in which no reward will be accumulated, whereas S_3 is "virtually absorbing" in the sense that the system loses its ability to perform effective work before the renewal at the next resynchronization point. On the other hand, upon the system enters S_1, reward accumulation will tentatively halt, and will 1) resume if the system subsequently returns to S_0 or enters S_2, or 2) remain seized if the system stays in S_1, or enters S_3 or S_4. It is worth noting that in the former case, reward must be "compensated" for the time interval during which the system was in S_1.

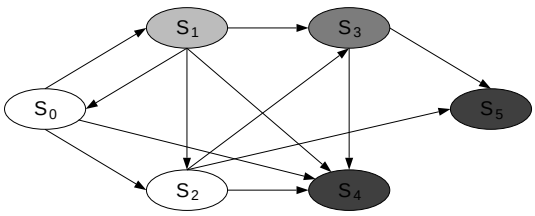

Figure 1: State Transition Diagram

Since the TB protocol enforces a blocking period τ at the end of each checkpointing interval φ during which no effective work will be performed, and the MDCD protocol introduces a performance overhead ρ which is the fraction of time for checkpointing and AT, the amount of effective work that can be accomplished by a process in an error-free checkpointing interval will be $\gamma(\varphi - \tau)$, where $\gamma = 1 - \rho$.

Figure 2 illustrates the system behavior concerning 1) volatile-storage and stable-storage checkpoint establishment, and 2) software- and hardware-error recovery. In the figure, the time horizon represents a resynchronization cycle, the thinner shaded region of the time horizon represents the interval during which the system is in S_1, and the thicker shaded region indicates that the system is in S_3. Note that the ending portion of the time horizon is depicted by a dashed segment, which represents the interval during which P_1^{low} has an erroneous state due to its fault manifestation (a necessary condition for the system to enter S_3). In addition, a flash sign[1] denotes the occurrence of a hardware transient error, while each line with an arrow pointing to the left denotes the process's rollback distance. Note also that some of those lines have dashed segments, which indicate that while the process rolls back to its stable-storage checkpoint of the nth checkpointing interval, the process is indeed brought back further to an earlier (clean) state. The reason is that the process was considered potentially contaminated when its timer expired at $n\varphi$, and thus the stable-storage checkpoint established then was a copy of the process's latest volatile-storage checkpoint (relative to $n\varphi$). This type of rollback scenario will happen if error recovery occurs when the process is in S_1 or S_3. Further, if the faults that occur in intervals CI-3 and CI-N_{max} (as marked in Figure 2) are detected software faults, the process's rollback distances would be the same as in the hardware transient error case.

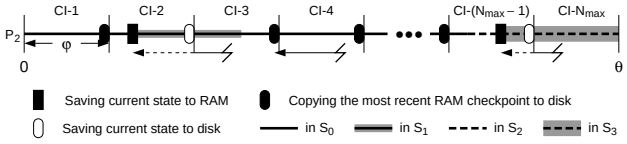

Figure 2: Checkpointing and Rollback Error Recovery

The above discussion suggests that we can solve for $G[N_{max}]$ in terms of an "impulse reward." Specifically, we view the amount of reward accumulated in a stable-storage checkpointing interval as the magnitude of an impulse, and at the end of each interval we choose one of the following actions:

1) add such an impulse to the reward that is computed at the end of the previous interval,

[1] In order to contrast different rollback recovery scenarios, we let three types of rollback action be shown in the same line representing a single resynchronization cycle. In reality, any error recovery will terminate a resynchronization cycle.

2) tentatively (or definitely) halt reward accumulation, or

3) adjust the reward that is computed at the end of the previous interval, and resume reward accumulation by adding the reward impulse of the current interval to the adjusted reward.

For example, if the system is in S_0 at the beginning of the nth checkpointing interval and we find that the system remains in S_0 at the end of the interval, then we add $\gamma(\varphi - \tau)$ to the reward computed at the end of the $(n-1)$th interval. As another example, if at the end of the nth interval, the system is in S_1 (or S_3), in which no reward should be accumulated, but the reward accrued prior to the nth interval must be carried on, then we simply let the reward at the end of the nth interval retain the value of the reward computed at the end of the $(n-1)$th interval. As a third example, if the system is in S_1 at the beginning of the nth interval and we find that the system has moved to S_0 by the end of the interval, then we restore the reward by letting the reward at the end of the nth interval have a value of $\gamma(n(\varphi - \tau))$, since this state transition implies that the process has never actually been contaminated and thus has performed effective work up to the end of the nth interval.

The process of collecting reward impulse at the end of each checkpointing interval can thereby be viewed as "delayed reward accumulation," which enables us to capture dependencies between sample paths and accrued reward. Figure 3 illustrates the concept of delayed reward accumulation. Note that each arc in the diagram is labeled by the conditional probability, in the form of conditional probability distribution function (PDF) $F_{ij}(\varphi)$, that the system enters state S_j by the end of an interval with a length of φ given that the system is in S_i at the beginning of the interval. Relating the diagram shown in Figure 1 to this diagram, each arc in the latter represents a "cross-interval state transition" which comprises one or more of the state transitions depicted in Figure 1. More specifically, while the state-transition process shown in Figure 1 is continuous-time in nature and enumerates all the possible transitions among states, the reward model in Figure 3 is discrete-time in nature (time is indexed on the sequence number of the stable-storage checkpointing interval) and illustrates the cross-interval state transitions, each of which is composed by the transitions that occur within φ. We view that a cross-interval transition will not conclude until the end of the interval, at which point a reward impulse is generated accordingly. The reward accumulation process can then serve as an upper-layer, discrete-time reward model that enables us to compute expected reward at the end of each stable-storage checkpointing interval.

To solve for the performability measure, we let $G[n]$ denote the expected value of accumulated effective work (AEW) at $n\varphi$ and define a set of functions $\{G_k[n] \mid 0 \leq k \leq 5\}$, in which

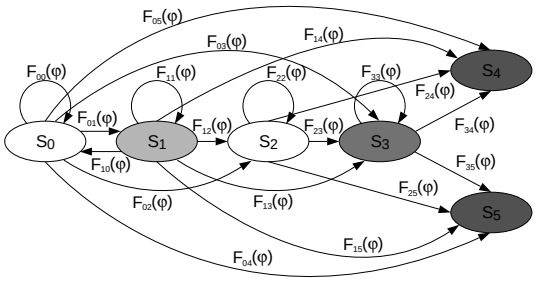

Figure 3: Upper-Layer Reward Model

$$G_k[n] = E[\text{AEW at } n\varphi, \text{ given that the system is in } S_k] \cdot P(\text{system in } S_k \text{ at } n\varphi) \quad (1)$$

Then, by the theorem of total expectation, the performability measure, namely the system's expected reward accumulated between two consecutive resynchronization points, can be expressed as follows:

$$G[N_{max}] = \sum_{k=0}^{5} G_k[N_{max}] \quad (2)$$

Based on the above analysis, we derive recursive solutions for $\{G_k[n] \mid 0 \leq k \leq 5\}$:

$$\begin{aligned}
G_0[i] &= F_{00}(\varphi)\,(G_0[i-1] + H_0[i-1]\,\gamma\,(\varphi - \tau)) + \\
&\quad F_{10}(\varphi) H_1[i-1]\,\gamma\,(i\,(\varphi - \tau)) \\
G_1[i] &= \sum_{j=0}^{1} F_{j1}(\varphi)\, G_j[i-1] \\
G_2[i] &= \sum_{j \in \{0,2\}} F_{j2}(\varphi)(G_j[i-1] + H_j[i-1]\,\gamma\,(\varphi - \tau)) + \\
&\quad F_{12}(\varphi) H_1[i-1]\,\gamma\,(i\,(\varphi - \tau)) \\
G_3[i] &= \sum_{j=0}^{3} F_{j3}(\varphi)\, G_j[i-1] \\
G_4[i] &= \sum_{j=0}^{3} F_{j4}(\varphi)\, G_j[i-1] + G_4[i-1] \\
G_5[i] &= \sum_{j=0}^{3} F_{j5}(\varphi)\, G_j[i-1] + G_5[i-1]
\end{aligned}$$

By definition, $G_k[0] = 0$ for $0 \leq k \leq 5$. Furthermore, in the above equations, $H_k[n]$ denotes the probability that by the end of the nth stable-storage checkpointing interval, the system will be in state S_k, and $F_{ij}(\varphi)$ is the conditional probability that the system will be in state S_j at the end of a stable-storage checkpointing interval of length φ, given that the system is in state S_i at the beginning of the interval.

The transient (after n intervals) state probabilities $\{H_k[n] \mid 0 \leq k \leq 5\}$ are also solved by recursive functions. The interval-of-time, conditional state transition probabilities $\{F_{ij}(\varphi) \mid i,j \in \{0 \ldots 5\}\}$ and performance overhead ρ are evaluated by the SAN model at the lower layer.

2.3 Lower-Layer SAN Reward Model

The SAN reward model at the lower layer is composed of a performance model that computes ρ and a dependability model that evaluates the conditional state transition probabilities $\{F_{ij}(\varphi) \mid i,j \in \{0 \ldots 5\}\}$. The SAN dependability model is itself composed of two submodels, as shown in Figure 4. We solve those models using $UltraSAN$. In particular, $\{F_{ij}(\varphi) \mid i,j \in \{0 \ldots 5\}\}$ and ρ are solved by specifying "predicate-reward pairs." By applying the same predicate-reward pair but different initial markings, we are able to conveniently evaluate the conditional probabilities $F_{ij}(\varphi)$ and $F_{kj}(\varphi)$, $i \neq k$.

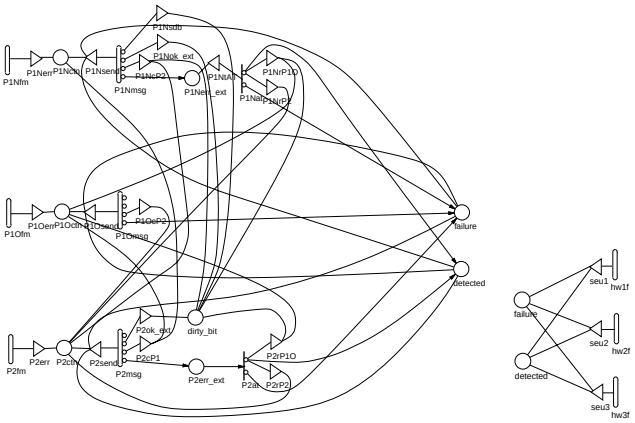

Figure 4: SAN Reward Model

3 Evaluation Results

Applying the hierarchical reward model developed in the previous section, we evaluate the performability measure $G[N_{max}]$. The curves in Figure 5 demonstrate that when the hardware transient fault rate is fixed, a smaller software fault manifestation rate (of the low-confidence software component) will provide a greater value of $G[N_{max}]$. This is a reasonable result, because when performance overhead is fixed, a greater value of $G[N_{max}]$ in general implies that the system is less likely to experience an earlier resynchronization due to error recovery. In this particular study, a lower software fault manifestation rate implies a lower probability that the system will undergo error recovery due to a detected software error within a resynchronization cycle and a lower probability that the system will encounter a failure because of an undetected error.

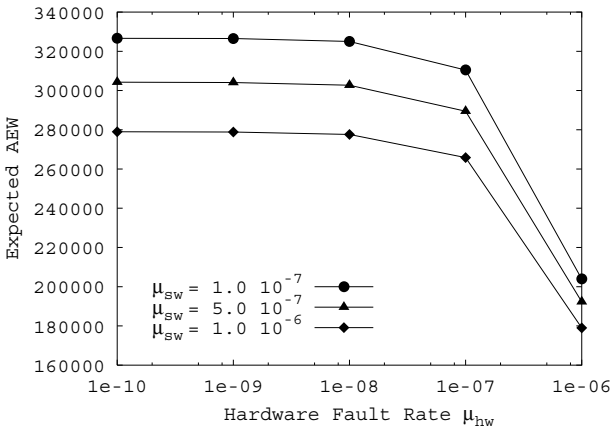

Figure 5: Results of Performability Measure (I)

We also evaluate the performability measure for the cases in which the TB and MDCD protocols are used alone. The results are displayed in Figure 6, in which we duplicate the curve with solid diamonds from Figure 5 for comparison. The results confirm that the coordination scheme is always superior to the schemes that use "stand-alone" protocols. Meanwhile, the curves illustrate that among the three schemes, the TB protocol is least effective when software faults dominate the system's failure behavior, whereas the MDCD protocol exhibits the worst performance when hardware faults become the dominant role in weakening the system's ability to perform effective work.

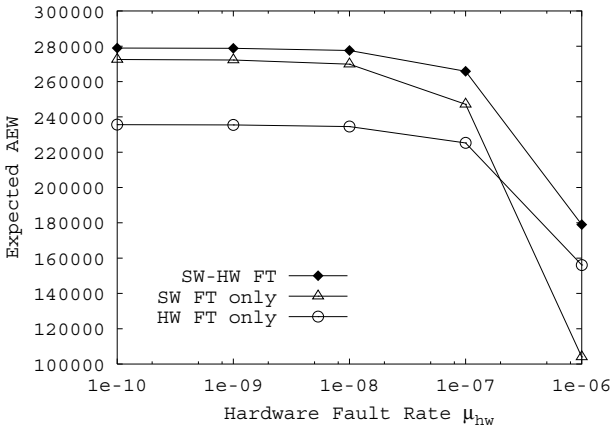

Figure 6: Results of Performability Measure (II)

References

[1] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "Synergistic coordination between software and hardware fault tolerance techniques," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2001)*, (Göteborg, Sweden), pp. 369–378, July 2001.