Appeared in: Proceedings of the IEEE Workshop on Imprecise and Approximate Computation,
Phoenix, AZ, Dec. 1992, pages 45–49.

Checkpointing Imprecise Computation

Riccardo Bettati
Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

Nicholas S. Bowen, Jen–Yao Chung
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

1 Introduction

The imprecise-computation model has been proposed in [1, 2, 3] as a means to provide flexibility in scheduling time-critical tasks. In this model, tasks are composed of a mandatory part, where an acceptable result is made available, and an optional part, where this initial result is improved monotonically to reach the desired accuracy. At the end of the optional part of a task, an exact result is produced. This model allows one to tradeoff computation accuracy against computation-time requirements.

Whenever a failure occurs in time-critical system, several actions have to be taken. The fault has to be identified and isolated. Recovery has to be invoked. Some tasks that were running at the time when the failure occurred may have to be restarted. The system experiences a transient increase in workload. In some cases the accumulated workload during the failure and successive recovery may cause a temporary overload in the system, and an increase of tasks that miss their deadline. Means must be found to reduce the effect of this temporary overload on the ability of the system to terminate time-critical tasks in time. Imprecise computation offers the flexibility to temporarily settle for a lower degree of computation accuracy. In this way, the computation-time requirements are lowered and the effective workload therefore temporarily reduced. Hence, an overload condition can be better handled. Providing imprecise results in the presence of failures is therefore a viable method to enhance the conventional fault-tolerance techniques such as checkpointing.

The workload accumulated during a failure is the set of tasks that were executing at the time when the failure occurred. If no provisions have been taken, these tasks have to be repeated after the recovery and therefore add to the transient overload. One way to reduce the amount of work to be repeated after a recovery is to regularly checkpoint the state of the running tasks to stable storage. In this way, only those portions of the tasks that have not been checkpointed have to be repeated. Checkpointing can therefore be viewed as another method to reduce the temporary overload caused by failures.

Due to its limited rollback and its predictable recovery behavior, checkpointing is – besides various parallel redundancy and replication schemes [4, 5] – a widely used technique for fault tolerance in real-time systems. Traditionally, checkpointing in real-time systems was considered from a task-oriented view [6, 7]. Once a task acquires a computational resource, it is supposed to run until it finishes, or until a failure occurs. No multiprogramming is therefore assumed in this model. The problem of devising a checkpointing scheme for real-time tasks typically reduces to determining the optimal inter-checkpoint interval to minimize the expected execution time of tasks, a problem that has found broad attention in the literature [8, 9]. However, most of todays real-time operating systems are multiprogrammed. The execution of a task can be preempted by other tasks. In such systems, minimizing the expected execution time of tasks is still an important means to meet timing constraints. In addition to that, the problem of how to schedule both the execution of tasks and their recovery in case of a failure, becomes an important issue.

In this paper we investigate ways to combine imprecise computation and traditional checkpointing to provide fault tolerance in time-critical systems. We propose the model of *checkpointed imprecise computation* to achieve dependability in time-critical systems. In Section 2 we review the traditional imprecise computation model. In Section 3 we describe checkpointing time-critical tasks. An approach will be presented on how to determine optimal checkpoint intervals with a fixed number of failures. In Section 4 we propose the checkpointed-imprecise-computation model. We describe an approach to schedule checkpointed imprecise tasks with a given upper bound on the number of failures from which the system has to recover during the execution of any given task. In Section 5 we use this approach to schedule tasks in a transaction-processing system. Simulation results are given to measure its performance under various loads and failure rates. The last section summarizes the proposed method and points to future work.

2 Imprecise Computation

Our model of an imprecise-computation system consists of a set $\mathcal{T}$ of n tasks, that is to be executed on a single processor. Each task T_i in $\mathcal{T}$ has a execution time τ_i, and consists of a mandatory part of length m_i and an optional part of length $o_i = \tau_i - m_i$. The task T_i is said to have reached an *acceptable* level of accuracy after executing for m_i units of time. During the optional part, the result is improved until a *precise* result is reached after o_i units of execution. The *error* e_i of the result of T_i describes the amount of accuracy that is lost if the task can not execute to the end of its optional part. The *error function* $e_i(\sigma)$ describes the error of T_i in a schedule where σ is the amount of time that the schedule has assigned to the execution of the optional part of T_i. The *total error* e of a schedule is the weighted sum of the errors e_i for all tasks in the schedule, that is, $e = \sum_{i=1}^{n} \alpha_i e_i$. If we want to model a linear error behavior, for instance, we choose $e_i = (o_i - \sigma_i)$ and $\alpha_i = 1/o_i$. The total error is then $e = \sum_{i=1}^{n} (o_i - \sigma_i)/o_i$, the normalized sum of the amount of computation that has been discarded.

Each task T_i is subject to timing constraints, which are given as *release time* r_i and *deadline* d_i. They are the points in time after which T_i can start its execution and before which T_i must terminate, respectively. If the deadline of a task is reached, the portion of the task that has not been executed yet is discarded. If any portion of the mandatory part has not been executed, a *timing fault* is said to occur.

3 Checkpointing Time-Critical Tasks

We assume a fault model where faults are transient. Tasks do not communicate with each other. Therefore the effect of a fault is confined to the task that was executing at the time when the fault occurred.

Each task T_i is checkpointed every s_i units of execution. It takes c_i units of execution to generate a checkpoint. We call s_i the *checkpoint interval* and c_i the *checkpoint cost*. While the checkpoint is generated, a sanity check of the computation is made and the status of the computation is written to stable storage. During the sanity check, the state of the computation is analyzed and checked for correctness. Sanity checks are assumed to not fail. Whenever a failure occurs, it is detected by the next sanity check, and recovery is initiated. During the recovery, the state of the computation at the time of the last checkpoint is loaded, and execution is resumed from there. The task is said to be *rolled back* to the beginning of the checkpoint interval.

When the task is scheduled, provisions must be made for the case that failures occur during its execution. Analysis of checkpointing strategies typically assume a stochastic error model, usually in terms of an inter-failure distribution. Modeling failure occurrences as stochastic events makes the design and analysis of deterministic scheduling policies difficult. To determine the schedulability of a task set, a worst-case number of failures must be assumed. In this way, the necessary recovery time can be allocated and scheduled. In our model, we assume that a task T_i can fail up to k_i times. If it fails more than k_i times, it is considered "erratic", and special measures have to be taken. For example, the T_i could be allowed to continue after it fails more than k_i times if there is no other task waiting to be executed; otherwise it would be aborted and discarded from the schedule. For different tasks T_i and T_j, the values for k_i and k_j can be different, reflecting such aspects as the execution times of the tasks and their importance, and the availabilities of the resources they access.

In general, the scheduler has to reserve enough time for a task to recover from its failures. We call a schedule k_i-*tolerant* for task T_i if enough computation time has been reserved for T_i to recover from k_i failures without any task in $\mathcal{T}$ missing its deadline. More generally, a schedule for the task set $\mathcal{T}$ is $(k_1, k_2, \ldots, k_n)$-*tolerant* (or $\bar{k}$-*tolerant* where $\bar{k}$ is the vector $(k_1, k_2, \ldots, k_n)$) if it is k_1-tolerant for T_1, k_2-tolerant for T_2, and so on. In traditional checkpointing, a schedule that is k_i-tolerant for T_i is assigned k_i additional intervals of length $s_i + c_i$ to the execution of T_i. This is to allow for k_i rollbacks and recoveries. Sometimes we will call the total length of the k_i additional intervals the *recovery time* h_i of T_i in the k_i-tolerant schedule. The total time scheduled to execute T_i, assuming that k_i failures occur, is the *total execution time* w_i of T_i, and $w_i = \tau_i + \lfloor \tau_i / s_i \rfloor (s_i + c_i) + k_i(s_i + c_i)$.

One way to increase the schedulability of T_i, that is, the probability for it to be feasibly scheduled, is to minimize its worst-case execution time. Under the assumption that k_i failures occur, a checkpoint interval $\tilde{s}_i$ can be determined that minimizes the worst case execution time of T_i. We call $\tilde{s}_i$ the *optimal* checkpoint interval.

Theorem 1. For a task T_i with execution time τ_i and checkpoint cost c_i, the optimal checkpoint interval $\tilde{s}_i$ in a k_i-tolerant schedule is $\tilde{s}_i = \sqrt{\tau_i c_i / k_i}$.

Proof: If exactly k_i failures occur, we have $w_i = \tau_i + \lfloor \tau_i / s_i \rfloor (s_i + c_i) + k_i(s_i + c_i)$. This expression is minimized when $s_i = \sqrt{\tau_i c_i / k_i}$. □

If less than k_i failures occur, the total execution time is naturally smaller, since the amount of time used for recovery is smaller.

The problem of deriving optimal checkpoint intervals has been extensively discussed under a variety of assumptions, for example by Young [8], Gelenbe [9], Coffman and Gilbert [10], Nicola et al. [11], and Grassi et al. [7]. Most previous research assumes stochastic failure occurrences, mostly in form of Poisson processes. Our definition of an optimal checkpoint interval, however, assumes a maximum number of failures,

s_i is chosen to minimize the worst case execution time when there are k_i failures that occur during the execution of the task T_i.

4 Checkpointed Imprecise Computation

In the traditional imprecise-computation model, we want to generate schedules where two goals are met, namely: (1) all mandatory parts meet the timing constraints to avoid timing faults, and (2) the total error e is minimized. Shih et al. [12, 13] have developed several scheduling algorithms that address this problem. In [12] they formulate it as a network-flow problem. In [13] much faster algorithms are found, that are based on a variation of the traditional earliest-deadline-first algorithm.) If we want to generate a $\bar{k}$-tolerant schedule for a checkpointed-imprecise-computation system, on the other hand, we have to consider one additional goal; (3) for every task T_i, enough time h_i must be reserved to execute k_i additional recoveries. Moreover, when we minimize the total error (the second goal), we have to consider that the total error of a schedule varies, depending on whether any specific failure does or does not occur. The total error of a $\bar{k}$-tolerant schedule has to be defined more precisely. In the following discussion, by total error we mean the total error of a schedule, assuming that all $K = k_1 + k_2 + \ldots + k_n$ failures do occur. We call a $\bar{k}$-tolerant schedule of $\mathcal{T}$ that meets the three goals stated earlier an *optimal* $\bar{k}$-tolerant schedule of $\mathcal{T}$.

In this section we describe a technique to determine an optimal $\bar{k}$-tolerant schedule in an imprecise computation system. We assume that tasks can be preempted at any time, even during the checkpoint generation. We assume the total error e of a schedule to be the weighted sum of the amount of computation that has been discarded, that is, $e = \sum_{i=1}^{n} (o_i - \sigma_i)/o_i$. This definition of a total error is said to define a *linear error behavior*. Shih et al [12, 13] distinguished two cases of linear error behavior. In the simpler case, called the *unweighted* case, all weights in the total error are identical. In the more general *weighted* case, they may vary.

Figure 1 shows a basic algorithm (Algorithm $\mathcal{C}$) to optimally schedule $\bar{k}$-tolerant checkpointed-imprecise computations. The following argument shows that Algorithm $\mathcal{C}$ is optimal in the way we defined earlier: As shown in [13], Step 2 either generates a feasible schedule for the mandatory part or declares failure. Goal (1) is therefore met. Since we included the recover time in the mandatory part in Step 1, goal (3) is also met. Since Step 2 minimizes the total error for $\mathcal{T}^{\bar{k}}$ (the task set $\mathcal{T}$ with all K failures occurring,) it minimizes the total error for the case where all K failures occur, and hence satisfies goal (2).

If one of the fast algorithms described in [13] is used, Algorithm $\mathcal{C}$ has a complexity of $\mathcal{O}(n^2 logn)$ and

---

Algorithm $\mathcal{C}$:

Input: Task set $\mathcal{T}$ defined by mandatory parts m_i, optional parts o_i, a vector $\bar{k}$, recovery time h_i and timing constraints r_i and d_i.

Output: An optimal $\bar{k}$-tolerant schedule $\mathcal{S}$ or the conclusion that the tasks in $\mathcal{T}$ cannot be scheduled to both be $\bar{k}$-tolerant and meet the timing constraints.

Step 1: Transform the task set $\mathcal{T}$ into a task set $\mathcal{T}^{\bar{k}}$ by modifying the mandatory part m_i of each task T_i according to the following rule:

- $m_i^{\bar{k}} = m_i + h_i$

Step 2: Apply an algorithm to schedule the imprecise task set $\mathcal{T}^{\bar{k}}$ to minimize the total error.

Figure 1: Algorithm $\mathcal{C}$.

$\mathcal{O}(nlogn)$ for the weighted and unweighted case, respectively. Its low cost makes it suitable for on-line scheduling. Whenever a new task is released and its parameters become known, the scheduler recomputes a new schedule, including this new task along with the remaining portions of other tasks.

The assumption that all K failures do occur is conservative. Under normal circumstances, very few failures occur, if any at all. In the following, let q_i in $\bar{q} = (q_1, q_2, \ldots, q_n)$ denote the number of failures actually experienced by T_i during its execution. If task T_i terminates successfully after experiencing $q_i \leq k_i$ failures, the recovery time for the remaining $k_i - q_i$ failures could be made available to the remaining tasks for their execution. In its basic form, Algorithm $\mathcal{C}$ does not make use of this additional time. The low complexity of Algorithm $\mathcal{C}$ allows the scheduler to generate dynamically adjusted schedules when less than k_i failures occur during the execution of any task T_i. This idea is used in the following Algorithm $\mathcal{C}1$ (see Figure 2) that dynamically adjusts the schedule to the occurrence of failures. We note that Algorithms $\mathcal{C}$ and $\mathcal{C}1$ are identical when all K planned failures occur. The following theorem states that Algorithm $\mathcal{C}1$ generates the optimal $\bar{k}$-tolerant schedule, independently of how many failures actually occur during the execution of the schedule.

Theorem 2. For every $\bar{k} = (k_1, k_2, \ldots, k_n)$ and $\bar{q} = (q_1, q_2, \ldots, q_n)$ with $q_i \leq k_i$, Algorithm $\mathcal{C}1$ generates a $\bar{k}$-tolerant schedule $\mathcal{S}$ that is optimal among all the $\bar{q}$-tolerant schedules.

Algorithm $\mathcal{C}1$:

Input: Task set $\mathcal{T}$ defined by mandatory parts m_i, optional parts o_i, two vectors $\bar{k}$ and $\bar{q}$ with $q_i \leq k_i$, recovery times h_i, and timing constraints r_i and d_i.

Output: An schedule $\mathcal{S}$ that is both $\bar{k}$-tolerant and optimally $\bar{q}$ tolerant, or the conclusion that the task set $\mathcal{T}$ cannot be scheduled to both be $\bar{k}$-tolerant and meet the timing constraints.

Step 1: Use Algorithm $\mathcal{C}$ to generate an initial $\bar{k}$-tolerant schedule $\mathcal{S}_1$.

Step 2: Whenever the mandatory part of task T_i successfully terminates at time t_i after q_i failures, use Algorithm $\mathcal{C}$ to generate a $(k_1, k_2, \ldots, k_{i-1}, k_{i+i}, \ldots, k_n)$-tolerant schedule $\mathcal{S}_{i+1}^C$, starting at time t_i, of the remaining parts of the tasks. Define the schedule $\mathcal{S}_{i+1}$ to be the sequence of $\mathcal{S}_i$ up to t_i and $\mathcal{S}_{i+1}^C$ from t_i on.

Step 3: Return the schedule $\mathcal{S}_n$.

Figure 2: Algorithm $\mathcal{C}1$.

Proof: We assume that the tasks in $\mathcal{T}$ are sorted according to increasing termination time t_i (as generated by Algorithm $\mathcal{C}1$,) i.e. for T_i and T_j, $i < j$ iff $t_i < t_j$. We define $t_0 = 0$. For every j, the schedule $\mathcal{S}_j$ is identical to the schedules $\mathcal{S}_{j-1}$ in the interval $[t_0, t_j]$ and $\mathcal{S}_j^C$ in the interval $[t_j, t_n]$. $\mathcal{S}_{j-1}$ is optimally $(q_1, \ldots, q_j)$-tolerant for the portions of the tasks that are scheduled in the interval $[t_0, t_j]$. $\mathcal{S}_j^C$ is optimally $(k_{j+1}, \ldots, k_n)$-tolerant for the remaining parts of the task set. By virtue of the linearity of the total error, the schedule $\mathcal{S}_j$ must be optimally $(q_1, \ldots, q_j, k_{j+1}, \ldots, k_n)$-tolerant for the entire task set $\mathcal{T}$. □

5 Results

In this section, we evaluate the checkpointed-imprecise-computation model in the simulation of a transaction-processing system. On-line transaction-processing systems are a good example of an area where fault-tolerance and real-time techniques are applied to achieve bounded-response-time and high-availability requirements. Our model contains a single processor that executes transactions, modeled as tasks. The time between the arrival of tasks is exponentially distributed with rate λ. The service time (i.e. the processing time τ_i) of task T_i is normally distributed and is partitioned into a mandatory and an optional part according to a factor μ, so that $m_i = \mu\tau_i$ and $o_i = (1 - \mu)\tau_i$. All tasks have identical checkpoint cost c, checkpoint interval s, and number of planned failures k. Each task is subject to timing constraints; the release time r_i is identical to the arrival time. The deadline d_i is defined as $r_i + D$, where D is a constant denoting the upper bound on the response time for all tasks. If the mandatory part of the task T_i is not terminated at time $r_i + D$, T_i missed its deadline and causes a timing fault. It is discarded from the system. The processor is allowed to fail, and the time to failure is exponentially distributed with rate ρ. Whenever a failure occurs, it is detected at the next sanity check of the currently running task, which is rolled back to its last checkpoint.

In the following simulations we use Algorithm $\mathcal{C}$ to generate a new schedule whenever a new task arrives. If the task cannot be feasibly scheduled, it is rejected at scheduling time. Whenever a task experiences more than k failures, it is assigned the lowest priority among all the tasks. This is done by declaring the remaining portion of the mandatory part to be optional. In the following, the processing times are normally distributed with mean 1.0 and standard deviation 0.3. The error e is defined to be the unweighted sum of the lengths of the optional parts that were discarded. The following parameters are constant throughout the simulations: $D = 10.0$, $c = 0.01$, and $k = 1$.

Figure 3 shows the effect of the checkpoint interval s on the performance of the system. As predicted in Section 3, for tasks with mean processing time of 1.0, $c = 0.01$, and $k = 1$, the checkpoint interval $\tilde{s} = \sqrt{\tau c/k}$ results in the lowest miss rates. The failure rate is $\rho = 0.3$. The dotted line is used as reference and represents the case where no failures occur and no checkpointing is performed.

6 Summary

In this paper we introduced the model of checkpointed imprecise computation. It uses the imprecise-computation model as a technique to increase the flexibility required when scheduling recoveries in a checkpointed real-time system. This is especially suitable in systems with very low failure rates, where most of the time reserved for recovery could be used to perform optional computation. In addition, checkpointing is an integral part of the imprecise computation model. Whenever a new, more accurate result has been calculated, either at the end of the mandatory part, or during the optional part, the system may store it to stable storage. We may think of it as a checkpoint being generated.

We have presented two basic algorithms to schedule checkpointed imprecise task sets. Both algorithms guarantee that the task set is schedulable with a specific number of failures and generate a schedule

that minimizes the average error. We are currently evaluating the performance checkpointed imprecise-computation approach in general, and of the algorithms in specific for a transaction-based model through simulation. The performance evaluation does not consider several important aspects at this stage. We want to evaluate the performance of the algorithms for systems with very small failure rates. We are currently looking into general techniques to evaluate systems with very rare event occurrences. We also want to analyze if – and how – fluctuations in the failure rate affect the performance of our approach differently than fluctuations in the basic workload (in terms of arrival rate.)

The basic algorithms presented here schedule the task sets to guarantee in a conservative way that the system can recover from a worst-case number of failures. In systems with very low failure rates, this either limits the workload that can be feasibly scheduled, or becomes prohibitively expensive (in terms of time to generate the schedule) when the schedule is adapted whenever a failure does not occur. We are currently evaluating algorithms that take an opposite approach. They allocate the minimum number of recovery time at any given point in time and adapt the schedule only in the rare event that the system experiences a failure.

Acknowledgement

We thank Prof. Jane Liu for her comments and suggestions. This work was partially supported by US Navy Office of Naval Research Contract No. NVY N00014 89-J-1181.

References

[1] Liu, J. W. S., K. J. Lin and C. L. Liu, "A position paper for the IEEE 1987 Workshop on Real-Time Operating Systems," Cambridge, Mass., May 1987.

[2] Lin, K. J., S. Natarajan, J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," *Proceedings of the IEEE 8th Real-Time Systems Symposium,* San Jose, California, December 1987.

[3] Chung, J. Y. and J. W. S. Liu, "Algorithms for scheduling periodic jobs to minimize average error," *Proceedings of the 9th IEEE Real-Time Systems Symposium,* Huntsville, Alabama, December 1988.

[4] Muppala, J. K., S. P. Woolet and K. S. Trivedi, "Real-Time-Systems Performance in the Presence of Failures," *IEEE Computer,* May 1991.

[5] Ramamritham, K. and J. A. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software,* Vol. 1, No. 3, 1984.

[6] Shin, K. G., T.-H. Lin and Y.-H. Lee, "Optimal Checkpointing of Real-Time Tasks," *IEEE Transactions on Computers,* Vol. C-36, No. 11, November 1987.

[7] Grassi, V., L. Donatiello and S. Tucci, "On the Optimal Checkpointing of Critical Tasks and Transaction-Oriented Systems," *IEEE Transactions on Software Engineering,* Vol. 18, No. 1, January 1992.

[8] Young, J. W., "A first order approximation to the optimum checkpoint interval," *Commun. ACM,* Vol. 17, No. 9, 1974.

[9] Gelenbe, E., "On the optimum checkpoint interval," *J. ACM,* Vol. 26, No. 2, 1979.

[10] Coffman, E. G. and E. N. Gilbert, "Optimal Strategies for Scheduling Checkpoints and Preventive Maintenance," *IEEE Transactions on Reliability,* Vol. 39, No. 1, April 1990.

[11] Nicola, V. F. and J. M. van Spanje, "Comparative Analysis of Different Models of Checkpointing and Recovery," *IEEE Transactions on Software Engineering,* Vol. 16, No. 8, August 1990.

[12] Shih, W.K., J. Y. Chung, J. W. S. Liu, and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *ACM Operating Systems Review,* July 1989.

[13] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Algorithms for scheduling tasks to minimize total error," *SIAM Journal of Computing,* 1991.

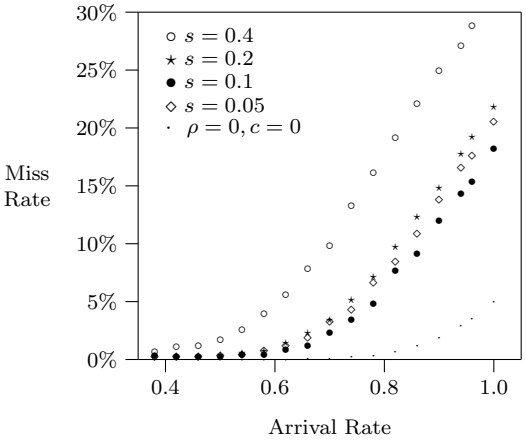

Figure 3: Effect of checkpoint interval s.