Appeared in: Proceedings of the Fifth Euromicro Workshop on Real-Time Systems,
Oulu, Finland, June 1993, pages 238–243.

On-Line Scheduling for 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

Abstract

When a failure occurs in a real-time system, the temporary loss of service and the recovery can cause a transient overload with an increase in the number of tasks that can not meet their timing constraints. The imprecise-computation technique allows one to trade off computation accuracy with computation time and offers therefore the necessary scheduling flexibility required during the recovery process after a failure. In this paper we investigate how the imprecise-computation approach can be combined with checkpointing; the result is a technique for fault-tolerance for real-time system. We define optimality criteria for the checkpointed imprecise-computation model. In an earlier work we have described algorithms to statically schedule imprecise tasks to meet these criteria. These approaches are conservative for systems with very rare failures. We take advantage of new results in on-line scheduling of imprecise computation to design an algorithm that dynamically adapts to failure occurrences. Simulations are described to evaluate the performance of this algorithm.

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. The optional part can be partially or entirely skipped, with a resulting reduction of accuracy in the result of the task. This model therefore allows one to tradeoff computation accuracy against computation-time requirements.

The ability to temporarily lower the amount of required computation time can be naturally used in recovery schemes for time-critical systems. Whenever a failure occurs in such a 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 must be repeated that have not been checkpointed. 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. In [8] we described ways to combine imprecise computation and traditional checkpointing

to provide fault tolerance in time-critical systems. We proposed the model of *checkpointed imprecise computation* to achieve dependability in time-critical systems. The basic algorithms to schedule checkpointed imprecise computation 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. In Section 2 we review the traditional imprecise computation model. In Section 3 we describe checkpointing time-critical tasks. The $\bar{k}$-tolerance model is described to design a system for a fixed number of failures. In Section 4 we propose the checkpointed-imprecise-computation model. We describe an approach to schedule checkpointed imprecise tasks on-line 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 present simulation results that describe the effect of varying loads and failure rates on the performance of the approach. The underlying system is supposed to be a transaction-processing system that uses checkpointed imprecise computation. 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. Whenever not enough time is available to execute task T_i to completion, either the entire optional part, or portions of it, can be skipped, without affecting the correctness of the system. By skipping portions of the optional part of T_i, a cost is incurred in form of the *error* introduced in the result of T_i. 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 sum of the errors e_i for all tasks in the schedule.

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.

Shih et al. [6, 7, 9] have developed several scheduling algorithms that address the problem of scheduling imprecise task sets to minimize the total error. In [6] they formulate it as a network-flow problem. In [7] much faster algorithms are found, that are based on a variation of the traditional earliest-deadline-first algorithm. Recently Shih and Liu [9] developed an algorithm to schedule imprecise tasks on-line. The imprecise-computation model and its applications in real-time systems are described in detail in [1, 2, 3].

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.

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 $\bar{k}$-*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. 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 $w_i = \tau_i + \lfloor \tau_i/s_i \rfloor (s_i+c_i) + k_i(s_i+c_i)$.

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. In [8] we showed $\tilde{s}_i = \sqrt{\tau_i c_i / k_i}$ to be optimal.

The problem of deriving optimal checkpoint intervals has been extensively discussed under a variety of assumptions. 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. The checkpoint interval 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 checkpointed-computation model, we want to generate schedules where all tasks and their recoveries meet the timing constraints to avoid timing faults. In a $\bar{k}$-tolerant schedule therefore enough time must be reserved for both the execution of each task T_i, the checkpointing overhead, and for the additional k_i recoveries.

In the checkpointed-imprecise-computation model, only the mandatory part of each task must be guaranteed to complete before the deadline. In generating $\bar{k}$-tolerant schedules for such a system, we have to consider one additional goal; the total error of the schedule must be minimized. Moreover, the total error of a schedule varies, depending on whether any specific failure does or does not occur. 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 minimizes the total error an *optimal* $\bar{k}$-tolerant schedule of $\mathcal{T}$.

In [8] we described an algorithm (Algorithm C) to generate optimal $\bar{k}$-tolerant schedules in an imprecise-computation-system. The general idea is to increase the mandatory part m_i of each task T_i by the time necessary for the checkpointing overhead and for the necessary recoveries in a $\bar{k}$-tolerant schedule. An algorithm (such as described in [6, 7]) to schedule this modified imprecise task set is applied to generate an optimal $\bar{k}$-tolerant schedule.

---

1. For as long as no event occurs, execute the task at the head of the task queue Q_T. If no such task is present, execute the task at the head of the optional queue Q_O.
2. When an event occurs:

   Event 1: the current task completes or is terminated at its deadline:
   - remove the current task from the task queue Q_T;
   - cancel the reservation of the current task; goto step 1.

   Event 2: the beginning of the first reserved interval is reached:
   - if there is time reserved for the current task, cancel the reservation for the current task; else terminate the current task and remove it from the task queue;
   - if any part of the task remains to be executed, insert it into the queue Q_O; goto step 1.

   Event 3: a new on-line task T_i arrives:
   - update the reservation of the current task;
   - make reservation for T_i (w_i units of time);
   - insert this task into the task queue Q_T; goto step 1.

Figure 1: Algorithm C_{OL}.

The assumption that all K failures do occur is conservative. Under normal circumstances, very few failures occur, if any at all. 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 unfinished tasks for their execution. In its basic form, Algorithm $\mathcal{C}$ does not make use of this additional time. The low complexity of Algorithm $\mathcal{C}$ would allow the scheduler to generate dynamically adjusted schedules when less than k_i failures occur during the execution of any task T_i. Whenever the mandatory part of a task T_i terminates after less than k_i failures, Algorithm C is executed to generate an optimal $(k_1, \ldots, k_i, k_{i+1}, \ldots, k_n)$-tolerant schedule for the remaining tasks.

We introduce a much more efficient approach based on Shih and Liu's recent results on on-line scheduling of imprecise computation [9]. Shih and Liu describe various algorithms to schedule imprecise task systems with tasks whose parameters are only known

after the processor starts executing some tasks. We describe here how their algorithm NORA can be extended to generate optimal $\bar{k}$-tolerant schedules. Algorithm NORA has been proved in [9] to minimize the total error for imprecise task systems with no off-line tasks and on-line tasks that are ready upon arrival. This algorithm maintains a reservation list for all tasks that have arrived but are not yet completed. The reservation list is derived from a feasible schedule of the unfinished portions of the mandatory parts of the tasks. It can be generated by backward scheduling according to the latest-ready-time-first rule. This reservation list is updated each time a new task arrives. The scheduling is done according to an earliest-deadline-first (EDF) policy. When a new task arrives, the reservation list is updated, the new task is put into the EDF-ordered task queue Q_T, and the first task is scheduled for execution. Figure 1 gives the pseudo code of Algorithm C_{OL}, an extension to Algorithm NORA, that allows to optimally schedule checkpointed imprecise task sets in a $\bar{k}$-tolerant way. When a new task T_i arrives, enough time must be reserved on the processor for the total execution time w_i (assuming k_i failures) of T_i. Since, during the execution of T_i, less than k_i failures can occur, some previously reserved time may not be used by T_i. The elegance of this algorithm in combination with a checkpointing scheme is, that any unused recovery time is automatically made available for the execution of other tasks. Since the reservation is deleted at the moment when T_i terminates, any unused recovery time can directly be used for the execution of the optional part of the tasks or for the next task in the task queue Q_T. Algorithm NORA generates $\bar{k}$-tolerant schedules if enough time is reserved for the execution, checkpoint overhead, and recovery of the arriving task. The addition of the FIFO queue Q_O for the unfinished optional parts in Algorithm C_{OL} is enough to guarantee optimal $\bar{k}$-tolerance. The queue Q_O contains any unfinished optional part of previous tasks. It allows the execution of those portions of tasks in the case that no executing or waiting task can use any unneeded recovery time that becomes available when planned failures do not occur. We note that the queue Q_O acts as an EDF queue by virtue of how the optional parts are entered into the queue.

5 Evaluation

In this section we describe the evaluation of the online scheduling approach as realized in Algorithm C_{OL} with a series of simulations. The underlying model is supposed to be a transaction-processing system. In on-line transaction-processing systems, bounded-response-time and high-availability requirements team up to request for both fault-tolerance and real-time techniques to be applied. In our simulations the system contains a single processor that executes transactions modeled as tasks. The task arrival is a Poisson process with rate λ. The processing time τ is normally distributed and consists of a mandatory part $m = \mu\tau$ and an optional part $o = (1 - \mu)\tau$, where μ denotes the fraction of mandatory computation. All tasks have the same checkpoint cost c, checkpoint interval s, and number of planned failures k. The timing constraints are defined as the limit D on the response time of tasks. If the mandatory part of task T_i is not finished by D time units after its arrival, T_i missed its deadline and causes a timing fault. The processor is subject to intermittent failures, which are modeled by a Poisson process 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.

Algorithm C_{OL} is used to schedule the arriving tasks. Whenever a task experiences more than k failures, it is declared optional and enqueued in the queue of optional portions Q_O. In the following, the processing times are normally distributed with mean 1.0 and standard deviation 0.3. The average error is defined as the unweighted sum of the lengths of the optional parts that were discarded, divided by the total length of the optional parts of all tasks that were accepted by the scheduler (i.e. without counting the tasks that were rejected at arrival time.) The following parameters are constant throughout the simulations: $D = 10.0$, $c = 0.01$, $s = 0.1$, and $k = 1$.

Figure 2 shows the average error for different fractions μ of mandatory computation. The failure rate is $\rho = 0.1$. Figure 3a and Figure 3b show the miss rate and the average error for different failure rates. The miss rate represents the fraction of both tasks that have been rejected by the scheduler upon arrival, and tasks whose mandatory part could not be completed by the deadline. The fraction μ of mandatory computation is 70%. In Figure 4 the same data is plotted against the processor utilization as measured during the simulation. We see that – given the same processor utilization – the results are significantly worse for the higher failure rate. Increasing the processor utilization by adding workload and by increasing the recovery activity has not the same effect on the amount of tasks that miss their deadlines and on the average error. We plan to further investigate this matter. In all simulations the 90% confidence intervals are below 1.0% for both the miss rates and the average error.

6 Summary

In this paper we described 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 natural 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 envision the checkpointed imprecise-computation model being an integral part of an imprecise system architecture such as [10], where system-directed checkpointing guarantees fault-tolerance with a minimum amount of error. The number k of planned failures would then be a parameter that is determined at service-negotiation time.

In [8] we presented two algorithms to optimally schedule checkpointed imprecise task sets. These algorithms are not suitable for systems with very low failure rates, however. In this paper, we present an on-line algorithm to schedule checkpointed imprecise task sets. This algorithm guarantees that the task set is schedulable with a specific number of failures and generates a schedule that minimizes the average error. Moreover, it automatically adapts to failure occurrences. We are currently evaluating the performance of the checkpointed imprecise-computation approach in general, and of this algorithm 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.)

Acknowledgement

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

References

[1] D. J. Lin, S. Natarajan, J. W.-S. Liu, and T. Krauskopf, "Concord: a System of Imprecise Computations," *Proceedings of the 1987 IEEE Compsac*, Japan, October 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] 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.

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

[8] Bettati, R., N. S. Bowen and J. Y. Chung, "Checkpointing Imprecise Computation", *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation,* Phoenix, Arizona, December 1992.

[9] Shih, W. K. and J. W. S. Liu, "On-line Scheduling of Imprecise Computations to Minimize Error," *Proceedings of the 13th IEEE Real-Time Systems Symposium,* Phoenix, Arizona, December 1992.

[10] Shih-Wei Liao, Kwei-Jay Lin, "Implementing a Fault-Tolerant Imprecise Computation Server on Mach", *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation,* Phoenix, Arizona, December 1992.

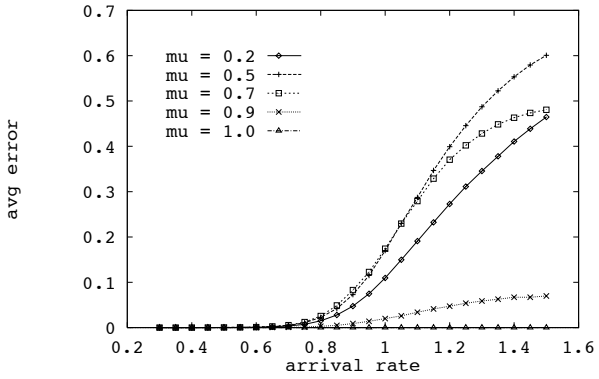

Figure 2: Varying the amount of mandatory part.

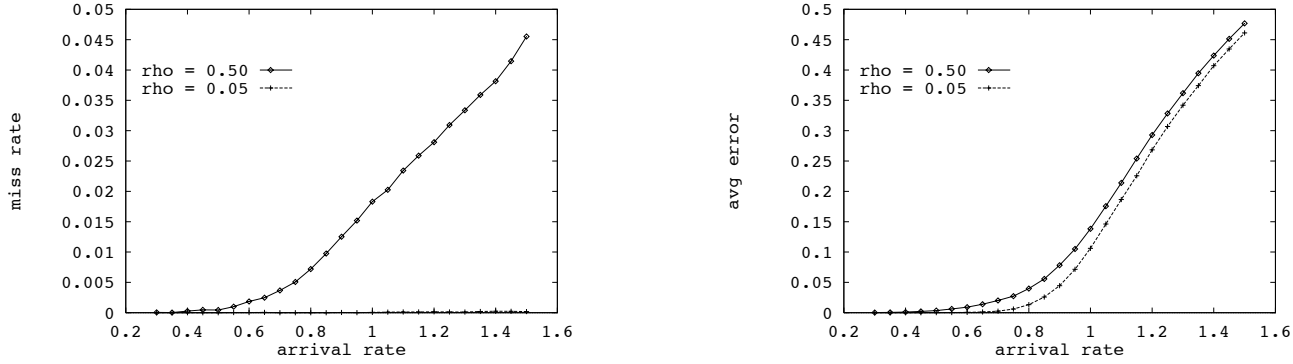

Figure 3: Varying the failure rate.

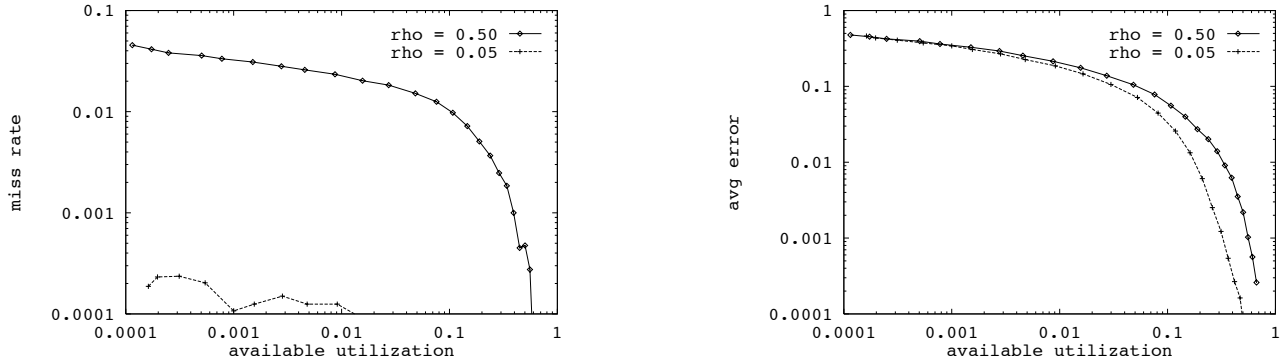

Figure 4: Varying failure rate: comparison against processor utilization.