Another *Two-Level* Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency*

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: vaidya@cs.tamu.edu

Technical Report 94-068
December 1994†

Abstract

This report deals with the design and evaluation of a "two-level" failure recovery scheme for distributed systems. In our previous work [30, 32], we motivated a "two-level" recovery approach that tolerates the more probable failures with a low overhead, and less probable failures with possibly higher overhead. The *two-level* approach can achieve a smaller overhead as compared to traditional recovery schemes. The contributions of this report are summarized below:

- We present and evaluate a "two-level" recovery scheme that is suitable for a network of workstations, each workstation having a local disk. The recovery scheme presented in the report can tolerate transient processor failures with a low overhead, while other failures require a larger overhead. The report presents analysis of the *average* (expected) task completion time using the proposed scheme. This scheme has been implemented on a workstation cluster. Our analysis indicates that the proposed two-level recovery scheme can achieve better performance as compared to existing "one-level" recovery schemes.
- The report also evaluates the impact of *checkpoint latency* on the performance of the recovery scheme. To our knowledge, no analysis of the performance impact of checkpoint latency has been carried out previously.
- Experimental measurements of checkpoint *latency* and checkpoint *overhead* for four applications are presented.

*References [32, 30] present material related to this report. The interested reader can obtain these references via anonymous ftp from ftp.cs.tamu.edu:/pub/vaidya.

†This report was revised several times in January 1995. The purpose of these revisions was to add Sections 10 and 11, and to revise Section 1.

1 Introduction

Many applications require massive parallelism to solve the problem in a reasonable amount of time. Despite the increase in hardware reliability, such applications encounter a high failure rate due to large multiplicity of hardware components. In the absence of a failure recovery scheme, the task must be restarted (from beginning) whenever a failure occurs. This leads to unacceptable performance overhead for long-running applications. Some failure recovery scheme must be used to minimize the performance overhead. During failure-free operation, failure recovery schemes periodically save information such as process state and messages; this information is used to recover from a failure. The performance overhead of a recovery scheme consists of two components:

- Overhead during failure-free operation (*failure-free overhead*), e.g., checkpointing and message logging.
- Overhead during recovery (*recovery overhead*), e.g., loss of computation due to rollback.

This report analyzes an approach to reduce the average performance overhead.

The design principle *"make the common case fast"* has been successfully used in designing many components of a computer system (e.g., cache memory, RISC [18]) and some aspects of checkpointing and rollback [21, 36]. However, the designers of distributed rollback recovery schemes have largely ignored this guideline. In any system, some failure scenarios have a greater probability of occurring as compared to other failure scenarios. In the context of failure recovery, the *"common case"* consists of the *more probable* failure scenarios. The above guideline suggests that a recovery scheme should provide low-overhead protection against *more probable* failures, providing protection against other failures with, possibly, a higher overhead. We refer to recovery schemes having this capability as *two-level* recovery schemes. This approach can be generalized to *multi-level* recovery [30]. It was recently brought to our attention [2] that, for transaction-oriented systems, Gelenbe [8] previously proposed an approach similar to the *multi-level* recovery approach. Gelenbe's work is summarized in Section 11.

Most existing recovery schemes are *"one-level"* in the sense that their actions during *failure-free* execution are designed to tolerate the worst case failure scenario. For example, the traditional consistent checkpointing algorithms are designed to tolerate simultaneous failure of all components in the system [11, 20]. The two-level recovery approach can achieve lower overhead than one-level schemes by differentiating between the more probable failures and the less probable failures.

Previously, we demonstrated that, it is often advantageous to use two-level recovery schemes as compared to traditional one-level recovery schemes [30, 32]. In this report we present design and analysis of a new two-level recovery scheme. Also, the report summarizes the two-level recovery scheme analyzed in our previous work [30, 32]. This report achieves three objectives:

- The report carries out a detailed analysis of the proposed two-level recovery scheme and presents performance results. It demonstrates that two-level recovery can achieve a better performance than a one-level recovery scheme. Although a large number of researchers have analyzed checkpointing and recovery [3, 6, 7, 9, 10, 12, 13, 17, 22, 25, 26, 28, 33, 34], to our knowledge, except for Gelenbe [8], no analysis of *two-level* recovery schemes has been attempted by other researchers.

- Another objective of this report is to analyze the impact of *checkpoint latency* on the performance of the recovery schemes. Checkpoint latency is the duration of time it takes to establish a checkpoint. For example, if a consistent checkpoint of a distributed system is initiated at time t_1 and completed at time t_2, then checkpoint *latency* is $(t_2 - t_1)$.

  Checkpoint overhead is the increase in the execution time of an application due to a checkpoint. In simple-minded implementations of checkpointing, checkpoint latency equals the checkpoint overhead. However, in some (more efficient) implementations, checkpoint latency is much larger than the checkpoint overhead (e.g., copy-on-write [15]). In this report, we study the impact of checkpoint latency on the average performance overhead. To our knowledge, there has not been any previous work on modeling and analysis of checkpoint latency.

  Terminology: Plank [20] uses the term checkpoint *time* to denote what we call checkpoint *latency*. Plank uses the term checkpoint *latency* to mean something else.

  Note: As will be elaborated in Section 10, it turns out that checkpoint *latency* can sometimes be smaller than checkpoint *overhead*. This is somewhat counter-intuitive. However, in any viable implementation of checkpointing, latency will be at least as large as the overhead. Therefore, in the analysis, we do <u>*not*</u> consider the case where checkpoint latency is smaller than checkpoint overhead.

- The report presents experimental measurements of checkpoint latency and checkpoint overhead for four applications. (The proposed two-level scheme has been implemented on a network of workstations.)

This report is organized as follows. Section 2 summarizes a two-level scheme that we had proposed previously. Section 3 presents the system model used in designing the two-level checkpointing scheme proposed in this report. The proposed checkpointing scheme is discussed in Section 4. Section 5 discusses the recovery algorithm. Performance analysis of the proposed scheme is discussed in Section 7. Section 8 presents some numerical results to illustrate the benefit of the proposed approach. Impact of checkpoint latency is analyzed in Section 9. Section 10 discusses an experimental implementation of the proposed scheme. Related work is discussed in Section 11. The report concludes with Section 12.

2 Brief Description of a Two-Level Scheme [30, 32]

This section summarizes a two-level recovery scheme that was proposed and analyzed previously [30, 32]. This two-level recovery scheme is useful in an environment consisting of disk-less workstations that can access a stable storage over the network. In the environment under consideration, small number of failures are more probable than a large number of failures. Specifically, *single processor failures* are more probable than all other failure scenarios. The two-level recovery scheme consists of two *components*, one *component* recovery scheme designed for single failure tolerance, and the second component scheme designed for tolerating all other failure scenarios. For this scheme, it is assumed that a single process is scheduled on each processor.[1] (This assumption is not necessary for the scheme proposed and analyzed later in this report.) The two component recovery schemes are summarized here:

- The first *component* is the single process failure tolerance scheme presented in [1]. In this scheme, the processes periodically take checkpoints (which need not be consistent with each other). The checkpoint of a process can be saved in any volatile storage except that of its own processor. The messages are saved by their senders in their volatile storage.

  This *component* scheme is capable of tolerating only a single failure. To tolerate a single failure, the faulty process is rolled back to its previous checkpoint (which is saved on a non-faulty processor). Subsequently, the messages that the faulty process had received before failure are re-sent to recover its state. These messages are available in the volatile memory of the message senders.

  If a second failure occurs before the system has recovered from the first failure, it is possible that the system may not be able to recover from the failure.

  We refer to the checkpoints taken by this *component* scheme as *1-checkpoints*, as they are useful to recover from single failures only. A *checkpoint interval* is the duration between two adjacent checkpoint. For this scheme, the failure-free overhead per checkpoint interval is denoted by C_1. C_1 is the increase in the execution time of a checkpoint interval due to the use of this recovery scheme.

- The second component recovery scheme periodically saves *consistent*[2] checkpoints on the stable storage. To establish the checkpoint, the processes coordinate with each other and ensure that their states saved on the *stable* storage are consistent with each other. Such a checkpoint is useful to recover from an arbitrary number of failures.

[1] This limitation can be eliminated using the single *processor* failure tolerance scheme presented in [31] instead of a single *process* failure tolerance scheme, as described below.

[2] A *consistent* checkpoint consists of one checkpoint per process such that a message sent after the checkpoint of one process is not received by another process before taking its checkpoint [4, 11].

Therefore, these checkpoints are called *N-checkpoints*. For this *component* scheme, the failure-free overhead per checkpoint interval is denoted by C_N.

Volatile storage access is cheaper than accessing the shared stable storage. Therefore, C_1 is expected to be smaller than C_N.

The two-level recovery scheme consists of the above two components [30, 32]. The two-level scheme takes 1-checkpoints more frequently and N-checkpoints less frequently. As the 1-checkpoints are taken more frequently, recovery overhead for a single processor failure is smaller. Also, overhead of taking 1-checkpoints is lower than that of N-checkpoints. As demonstrated in [30, 32], the two-level scheme can achieve better performance as compared to either component recovery scheme.

To further clarify the concept of *two-level* recovery, the tables below present an analogy of the two-level recovery scheme with cache memory organizations.

Cache and main memory (two-level) hierarchy

access type	served by	latency
address in cache	cache	small
address not in cache	main mem.	large
average access latency = small		

Two-level recovery scheme

failure scenario	failure tolerated by	overhead
single failure	1st component scheme	small
other	2nd component scheme	large
average performance overhead = small		

Ziv and Bruck [36] present a checkpointing and rollback scheme for duplex systems, that also takes two types of checkpoints, similar to the above two-level scheme. Section 11 discusses their scheme.

The rest of this report presents another two-level recovery scheme and its performance evaluation. The next section discusses the system model assumed while designing the proposed two-level recovery scheme.

3 System Model

The system architecture is illustrated in Figure 1. The system is assumed to consist of N processors. Each processor may execute one or more processes. Each processor has access to a *memory* (such as a RAM) and a *local storage* (such as a disk), and it can also access a *stable storage* over the network. (More than one stable storage may also be accessible.) Accessing the *local storage* incurs less overhead as compared to the *stable storage*, as the stable storage access is made over the network.

Memory and *local storage* are both accessible to the processor locally, i.e., without going over the network. Thus, one can potentially consider the *memory* and the *local storage* together as a single *locally accessible storage*. The reason for separating the locally accessible storage into two types (i.e., *memory* and *local storage*) is that the failure of a processor *always*

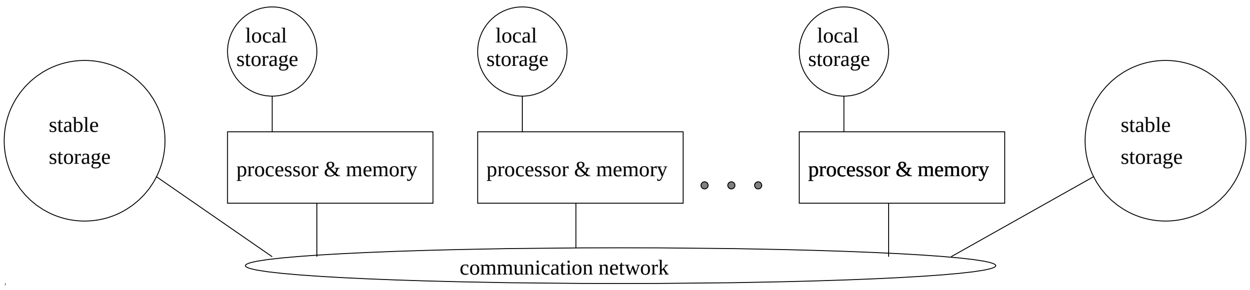

Figure 1: System Architecture

results in the loss of *memory* contents, but not necessarily in the loss of *local storage* contents. We will address this issue again shortly.

The above architecture is suitable for workstation clusters, where each workstation has a memory (RAM), a local storage (a local disk), and each workstation can also access a stable storage by going over the network.

Alternate architectures: The discussion in this report often refers to a system consisting of workstations with local disks. However, the above model and the analysis in this report is also applicable to any system where a processor has a local access to two types of memory storages, such that failure of one memory storage is weakly correlated (or uncorrelated) with that of the processor. Thus, both the storages may be RAMs, but one of them may be on the same card as the processor, and another on a different card. If the probability of correlated failure of the two cards is small, then the scheme proposed in the report is useful for this system.

Failure Model

In this report, we consider only *fail-stop* failures [24]. We first describe the failure model assumed in this report, followed by another failure model that may be applicable to some systems. The analysis presented in the report assumes the first model, however, the analysis is applicable to the second model with a minor modification.

A processor is subject to transient as well as permanent failures. Failure of a processor results in the loss of its *memory* contents, however, it does not *cause* a failure of its local storage. Thus, the contents of the *local storage* can survive the failure of the associated processor. The *local storage*, however, is not a stable storage.

A local storage is also subject to transient and permanent failures. Failure of the local storage corrupts the information stored on the local storage. Subsequent to a transient failure, the local storage can still be written to, though the data stored before the failure is lost.

Subsequent to a permanent failure, the local storage cannot be accessed. This necessitates that the processes on the corresponding processor be moved to another processor. It turns out that, for the purpose of our analysis, there is no need to differentiate between transient and permanent failure of a local storage. Note that to be able to access the *local storage* of a processor, the processor itself must be operational. Permanent failure of a processor makes its *local storage* inaccessible to other processors.

We assume that the failure (transient or permanent) of a *local storage* always crashes the associated processor. This assumption is quite accurate in the case of workstations. The *local disk* of a workstation often stores swapped out process memory, temporary files accessed by an application as well as many files that are accessed by the operating system. Failure of the local disk is, therefore, likely to crash the system. When the above assumption does not hold, our analysis can be revised to reflect the accurate system model.

Failures of the N processors are independent of each other, similarly the local storage failures are independent of each other. Let the inter-failure interval for a processor be governed by an exponential distribution with mean $1/\lambda_p$. The probability that a processor failure is *permanent* is denoted by p, $(1-p)$ being the probability that a processor failure is transient. Let the time interval between detection of consecutive local storage failures be governed by an exponential distribution with mean $1/\lambda_l$. Let λ denote $\lambda_p + \lambda_l$. It is expected that, in practice, transient processor failures will be more probable than permanent processor failures or local storage failures.

In the event of a permanent processor failure or any *local storage* failure, the process scheduled on that processor must be rescheduled on another processor. This overhead is included in the overhead of *rollback*. We discuss the various overheads in more detail, in the following.

Alternate failure model: As noted above, a *permanent* processor failure makes its local storage inaccessible to other processors. In some systems, it is possible that a *transient* failure of the processor may also cause a *correlated* local storage failure. In such systems, define r as the probability that a processor failure either makes the local storage inaccessible or causes a correlated local storage failure. It is clear that $r \geq p$. The analysis presented here becomes applicable to this model if p is replaced by r in all the expressions derived in the report.

4 Checkpointing Scheme

The processes periodically take *consistent* checkpoints using some consistent checkpointing algorithm, for example, Chandy-Lamport [4]. (For uni-process applications, trivially, any checkpoint of the process is a "consistent" checkpoint.) The consistent checkpoints are assumed to be equidistant. (In practice, the checkpoints will not be exactly equidistant, but can be made approximately equidistant.) Every k-th consistent checkpoint is stored on the stable storage, all other checkpoints being stored on local storages. No checkpoint is taken at the beginning or at the completion of the task. We use the term *local checkpoint* to refer to a checkpoint that is stored on a local storage. Similarly, the term *stable checkpoint* refers to a checkpoint that is stored on the stable storage. Figure 2 illustrates local and stable checkpoints for $k = 3$. The horizontal line depicts task execution. Observe that checkpoints CP3, CP6 and CP9 are *stable* checkpoints, while the other checkpoints are *local* checkpoints. (In this figure, we assume that checkpoint *overhead* and *latency* are identical, this assumption will soon be relaxed.)

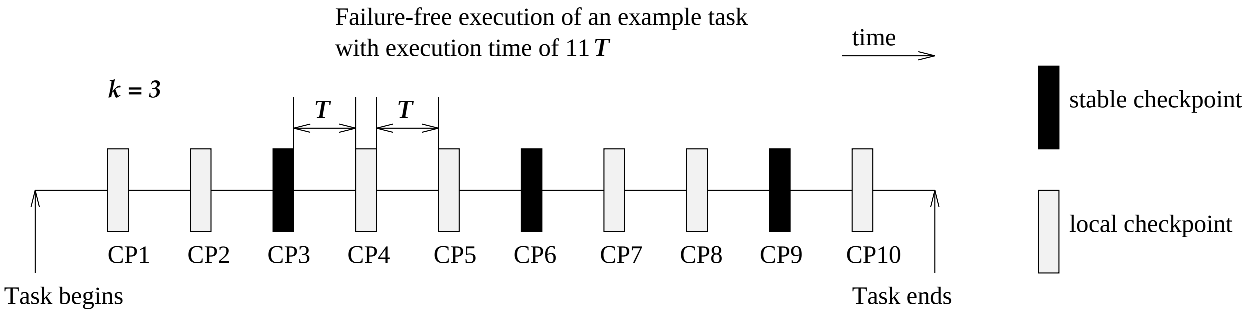

Figure 2: *Local* and *stable* checkpoints

The checkpointing operation incurs some *overhead* (e.g., messages required for establishing consistency, overhead in saving the state on the storage, etc.). Let the overhead in taking a local checkpoint be C_l. This implies that if the application takes one consistent checkpoint, where each processor stores its state on the *local storage*, then the total execution time of the task will be increased by C_l. Similar to C_l, let the overhead in taking a stable checkpoint be C_s. In most systems, one would expect C_l to be significantly smaller than C_s.

It is clear that, in practice, C_l is likely to be different for different checkpoints. However, in this analysis we assume C_l to be the *average* overhead of a local consistent checkpoint.[3] We use averages for some other parameters as well (including C_s).

Another parameter that may affect performance is the *checkpoint latency*. Checkpoint latency is the duration, from the instant at which a checkpoint operation is initiated, till the instant when the checkpoint operation is completed. *Checkpoint overhead* is the increase in the execution time of the application due to a checkpoint operation. Checkpoint *latency* is usually at least as large as the checkpoint *overhead*. (As stated in Section 1, our analysis does not consider implementations where the latency is smaller than checkpoint overhead.)

[3] Our assumption is similar to [10]. Some researchers [7] assume that the checkpoint overhead is exponentially distributed, though, to our knowledge there has been no experimental justification of this assumption.

For many implementations of checkpointing (e.g., copy-on-write), *checkpoint latency* is larger than the *checkpoint overhead.*[4] Let the *checkpoint latency* for a local checkpoint be L_l and that for a stable checkpoint be L_s. Note that a checkpoint is an image of the system state at the beginning of the latency period.

For the purpose of the analysis, it will suffice to assume that all the overhead of a checkpoint is incurred at the start of the *checkpoint latency* period. This is illustrated in Figure 3. As shown in the figure, we will assume (without loss of correctness in the analysis) that the first C_l time units, during latency L_l of a local checkpoint, are spent in saving the state on local storage, while the remaining $L_l - C_l$ time units are spent doing useful work. However, the checkpoint is *not* considered to have been *established* until the end of the checkpoint *latency* period. (We will return to this issue when discussing recovery.)

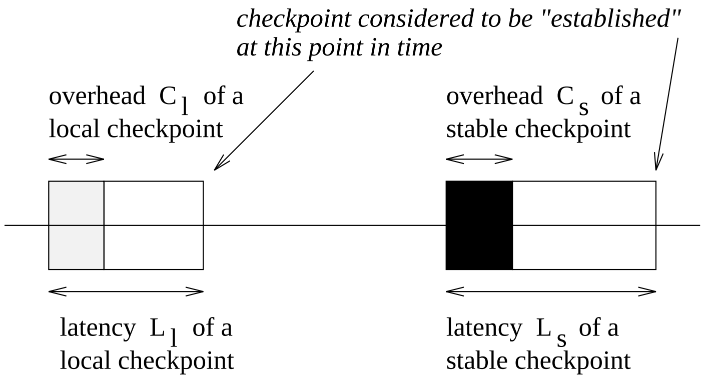

Figure 3: Checkpoint latency and checkpoint overhead

Measurements: The proposed checkpointing scheme has been implemented on a network of workstations. Results of its experimental evaluation are included in Section 10. As the analysis is independent of the implementation details, we defer discussion of the experimental measurements until Section 10.

5 Recovery Algorithm

When a failure occurs, the system recovers by rolling back to a previous checkpoint. The choice of checkpoint to be used (to rollback) depends on what is faulty and the timing of the failure. Failures can occur during normal operation, during checkpointing or during recovery. (A failure can occur before the system has recovered from a previous failure.) Recovery is

[4]Example: In some implementations, when a process wants to take a checkpoint, it forks a child process [19]. The child process saves the state (which is identical to the parent process' state when it executed *fork*), while the parent process continues to perform computation. With this approach, the child process requires a longer duration of time to save the state, as compared to the overhead incurred by the parent process.

initiated immediately after a processor failure or a local storage failure is detected. The following three cases of failures are possible.

Case 1: *A processor has a transient failure*: In this case, the failure is recovered by rolling back to the most recent checkpoint (which may be a local or a stable checkpoint), as shown in Figure 4. When a processor has a transient failure, its local storage contents are not corrupted, therefore, the local checkpoint can be used for recovery. Note that a transient processor failure any time during the *checkpoint latency* period requires a rollback to the *previous* checkpoint. For example, in Figure 4(b), a processor fails during the latency period for checkpoint $CP2$. As checkpoint $CP2$ is not *established* by the time of failure, to recover from the failure, the processors must roll back to checkpoint $CP1$.

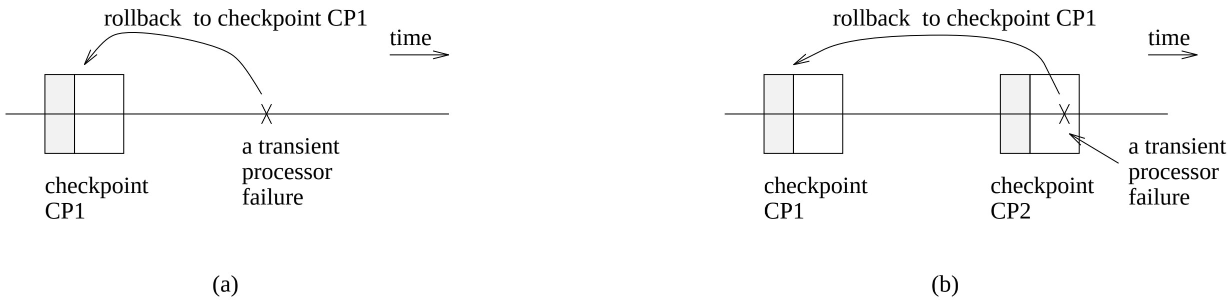

Figure 4: Case 1: A processor has a *transient* failure

Case 2: *A processor has a permanent failure*: In this case, the local storage of the faulty processor is inaccessible, as the processor failure is permanent. Thus, the system cannot roll back to the previous *local* checkpoint, and recovery requires that the processors roll back to the most recent *stable* checkpoint. This is illustrated in Figure 5. (If no stable checkpoint is established before the failure, then the task must be restarted from the beginning.) As noted earlier, if a failure occurs during the checkpoint latency period for a checkpoint, say CP, then checkpoint CP cannot be used for recovery. Figure 5(b) illustrates this.

Case 3: *A* local storage *has a failure (transient or permanent)*: In this case, failure of the local storage will also crash the corresponding processor. To recover from this failure, the processors must rollback to the most recent *stable* checkpoint (because a *local* checkpoint cannot be recovered). The recovery in case 3 is identical to that in case 2 above.

In cases 2 and 3 both, the processors roll back to the most recent stable checkpoint. Whenever a rollback to the stable checkpoint occurs, a fault-free processor loads its state from the stable storage. The state of the faulty processor and the executable both must be loaded to a new processor (if the failure is permanent) or to the same processor (if the failure is transient). In both cases, the overhead is likely to be identical.

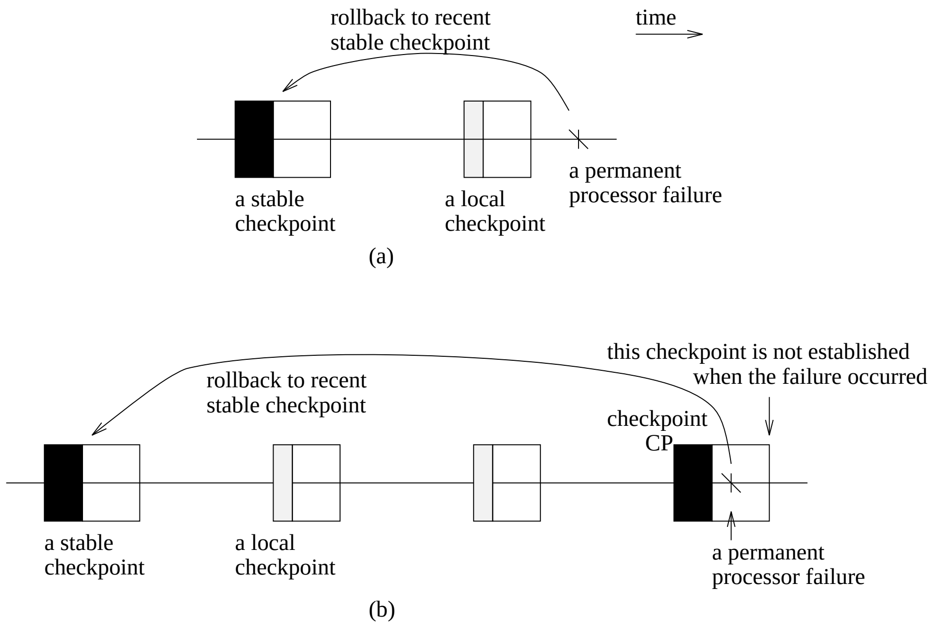

Figure 5: Case 2: A processor has a *permanent* failure

6 Some Terminology

Computation performed between two consecutive consistent checkpoints is called a *checkpoint interval*. Let the total number of checkpoint intervals during the execution of the task be μ. Thus, a total of $(\mu - 1)$ consistent checkpoints will be taken (no checkpoint is taken at the end of the last checkpoint interval). Every k-th consistent checkpoint is stored on the stable storage. Thus, of the $(\mu-1)$ checkpoints, $\lfloor(\mu-1)/k\rfloor$ checkpoints will be *stable* checkpoints, and the remaining checkpoints will be *local* checkpoints.

Two extreme cases occur when $k = 1$ or $k = \mu$. When $k = 1$, all the consistent checkpoints are *stable* checkpoints – this corresponds to traditional implementations of consistent checkpointing (e.g. [5]). When $k = \mu$, the recovery scheme takes only *local* checkpoints. In this case, if a permanent processor failure or a local storage failure occurs, the application must be restarted from the beginning.

As shown in Figure 6, some portion of a checkpoint interval is spent during the *checkpoint latency* period for the checkpoint preceding the checkpoint interval. The only exception to this is the first checkpoint interval, as there is no checkpoint taken at the beginning of the task.

Length of a task (application) is the execution time of the task in a failure-free environment (without using any recovery scheme). Length of the task is denoted by Υ. The length of each checkpoint interval is identical, and is denoted by T. Length of the task Υ is

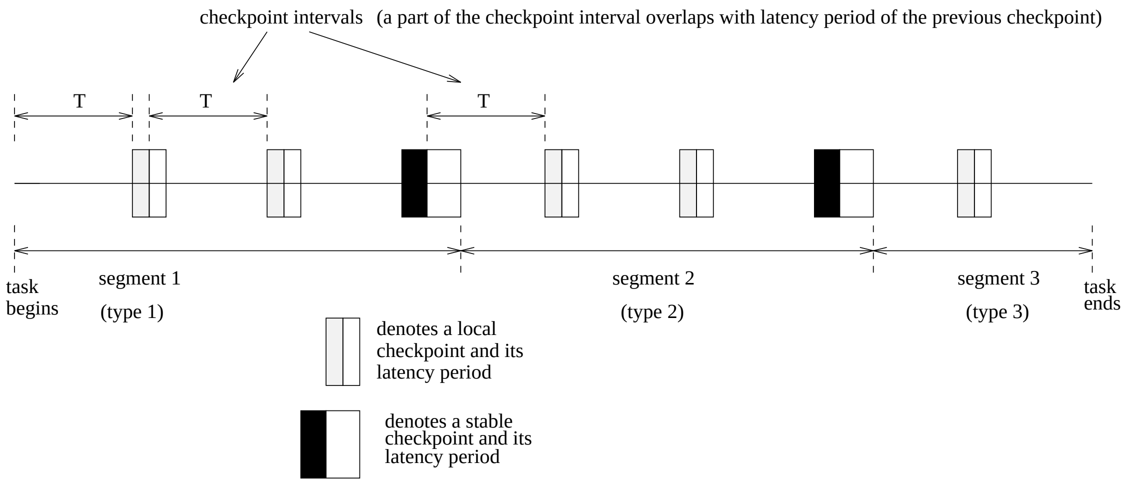

Figure 6: Failure-free execution of an example task

an integral multiple of T, specifically, $T = \Upsilon/\mu$ (where μ is an integer). However, length of the task is *not* necessarily an integral multiple of kT.

The time required to perform a rollback to a stable checkpoint is denoted by R_s. (This does not include the time required for re-execution.) Similarly, the time required to perform a rollback to a local checkpoint is denoted by R_l. R_s will be typically larger than R_l, for two reasons: (i) rolling back to a stable checkpoint requires checkpoint transfers over the network, (ii) rolling back to a stable checkpoint requires loading of the executable code of any faulty processes.

Similar to the checkpoint overhead, the rollback overhead is likely to be different for different checkpoints. However, in this analysis we assume them to be constant (essentially, R_l and R_s denote the *average* rollback overheads).

The number of checkpoint intervals μ may not be an integral multiple of k. This implies that the number of checkpoint intervals *after* the last stable checkpoint may be less than k. The number of checkpoint intervals between two consecutive *stable* checkpoints is always k. For example, in Figure 6, length of the task is $8T$ and $k = 3$. Therefore, the computation time between adjacent stable checkpoints is $3T$. However, the computation time from the last stable checkpoint to the completion of the task is $2T$.

The execution of the task is divided into certain number of *segments*. A *segment* terminates either with a stable checkpoint or with the completion of the task. For example, in Figure 6, the task is divided into three segments. Segments 1 and 2 terminate with stable checkpoints, whereas segment 3 terminates with task completion. The segments are divided into three *types*. The first segment of the task is of type 1, the last segment of the task is of type 3, and all other segments in the middle are of type 2. A type 1 segment begins at the start of the task, and ends when the first stable checkpoint is *established*.

Recall that a checkpoint is said to be *established* only at the end of the checkpoint latency period. Thus, for a checkpoint to be established, the task must execute the latency period once. (Part of the computation in the latency period may be repeated if a failure occurs after establishing the checkpoint.) A type 2 segments begins immediately after a stable checkpoint is established and ends when the next stable checkpoint is established. A type 3 segment begins after the last stable checkpoint is established and ends when the task is completed. The example task in Figure 6 has 3 segments. In general, a task contains $\lceil \mu/k \rceil$ segments.

Note: A degenerate case occurs when the task contains only one segment. In this case, no stable checkpoints are taken during the execution of the task. This segment neither begins nor ends with a stable checkpoint. Such a segment is said to be a type 4 segment.

Re-execution time: Consider a failure that can be tolerated by rolling back to a certain checkpoint CP. If the failure is detected when t time units of computation was performed after checkpoint CP, then it is assumed that t units of execution is required to re-do the lost computation (in absence of further failures), excluding checkpoint overhead. In the past, many researchers have assumed (e.g., [3]) that the time required to re-do the computation is βt for some constant β. Thus, we assume $\beta = 1$ here. This assumption is reasonable for parallel applications of interest. Our analysis can be easily revised when $\beta \neq 1$.

7 Performance Analysis

The performance metric of interest here is the *average overhead* of the recovery scheme. Let Γ denote the time required to complete the task using the given recovery scheme. Then, $E(\Gamma)$ is the expected (or average) task completion time. The average overhead is evaluated as a fraction of Υ (task length). Specifically, *average overhead* is defined as

$$\frac{E(\Gamma)}{\Upsilon} - 1$$

Average *percentage* overhead is obtained by multiplying the average overhead by 100. (We will denote the expected or average value of any random variable x as $E(x)$.)

This section presents an analysis of the average overhead. The results of the analysis have been verified using simulations. The simulation results are within less than 1% of the analytical results, therefore, the simulation results are not presented separately.

The average overhead can be obtained once we know the average execution time $E(\Gamma)$. To evaluate $E(\Gamma)$, we first evaluate average execution time for each *segment* and then add them to obtain $E(\Gamma)$. Recall that the task contains $\lceil \mu/k \rceil$ segments. Let S_i denote the execution time for the i-th segment. Then,

$$E(\Gamma) = \sum_{i=1}^{\lceil \mu/k \rceil} E(S_i).$$

The analysis of the average execution time of a segment is somewhat different for the segments of the four types. In this report, we present detailed analysis for type 2 segments, the other segments can be analyzed similarly (as elaborated later).

7.1 Average Execution Time of a Type 2 Segment

Recall that a type 2 segment begins after a stable checkpoint is established, and ends when the next stable checkpoint is established. During the execution of the segment, $(k-1)$ local checkpoints are also established. Figure 7 illustrates the execution of a type 2 segment, assuming $k = 3$. Figure 7(a) illustrates a failure-free execution. Observe that, in the absence of a failure, the computation performed before the first checkpoint in the segment requires $(T - L_s + C_s)$ time units (as shown in the figure). (Note: $L_s - C_s$ units of computation in the first checkpoint interval of the segment is performed during the latency period of the previous stable checkpoint.) The meaning of the various *states* in the figure will be explained later.

Figure 7(b) illustrates an execution of the segment where three failures occur. The first failure is a transient processor failure, and it occurs soon after the segment begins execution. The transient processor failure requires a rollback to the recent checkpoint, which happens to be a stable checkpoint. The rollback incurs an overhead of R_s. After the rollback, T units of computation is required before the next checkpoint can be initiated. (Note: After the rollback, the $L_s - C_s$ units of computation performed during the latency period of the previous checkpoint must also be repeated.) The second failure is also a transient processor failure. This failure is tolerated by rolling back to the most recent checkpoint, which happens to be a *local* checkpoint. This rollback incurs an overhead of R_l time units. The third failure is a *local storage* failure. This failure necessitates a rollback to the stable checkpoint at the start of the segment. No failure occurs after this rollback. (The next section explains the meaning of various *states* in Figure 7.)

7.1.1 Markov Chain for the Execution of a Type 2 Segment

To evaluate the expected execution time of the task, we construct a finite-state Markov chain [27]. Markov chains have been used for evaluating expected execution time by Ziv and Bruck also [35]. The procedure for constructing the Markov chain is presented later, we first present some preliminaries. The Markov chain has an unique *start* state and an unique *absorbing* state. For a given k, the Markov chain contains $2k+1$ states. A state transition

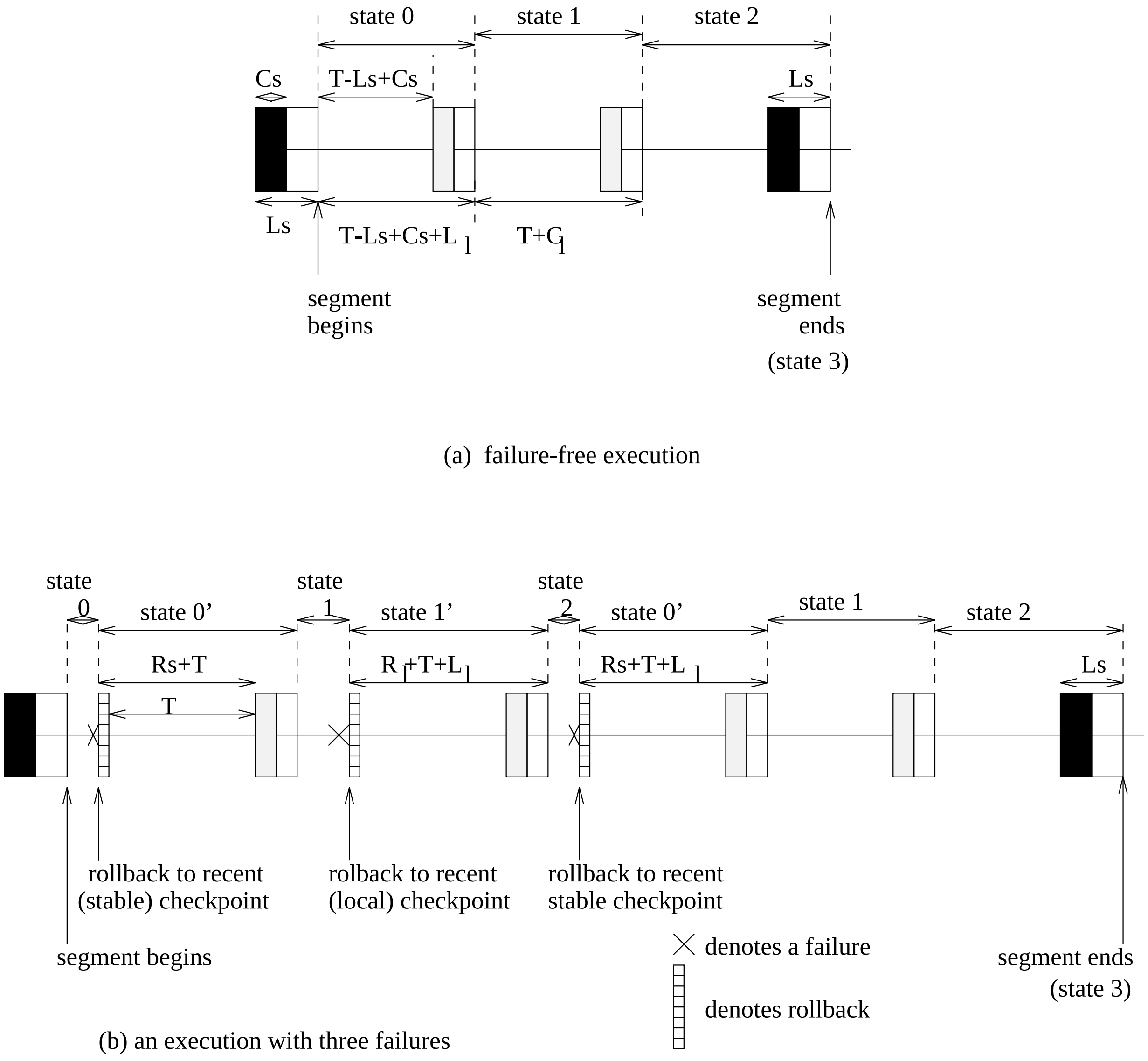

Figure 7: Execution of a type 2 segment: $k = 3$

probability is associated with each state transition in the Markov chain. In addition, we also associate a *weight* with each transition. *Weight* of a transition from state X to state Y equals the expected (average) time spent in state X before making the transition to state Y. The probability of a transition from state X to state Y is denoted as P_{XY} and the corresponding weight is denoted as W_{XY}.

The analysis for $k = 1$ and $k > 1$ has a few minor differences. In the following, we focus on $k > 1$. Figure 8 illustrates the Markov chain for $k > 1$. The Markov chain contains $(2k + 1)$ states named 0, 0', 1, 1', ..., i, i', ..., $(k - 1)$, $(k - 1)$', k. (There is no state k'.) State k is the absorbing state. The transitions out of the other states are as follows: From state i $(0 \leq i < k)$, transitions can occur to states i', $i + 1$ and 0'. Similarly, from state i' $(0 \leq i < k)$ transitions can occur to states i', $i + 1$ and 0'.

A state i is reached when the i-th checkpoint after the start of the segment is established. State i' is reached when a rollback to the i-th checkpoint occurs. (As the k-th checkpoint is the last checkpoint of the segment, we do not account for rollback to the k-th checkpoint when evaluating the execution time of this segment. These rollbacks will be taken into account when evaluating the execution time of the next segment of the task. Therefore, there is no need for a state named k'.) If a transient processor failure occurs while in state i, then a transition is made to state i' (because the system rolls back to the i-th checkpoint taken since the start of the segment). If a permanent processor failure or a local storage failure occurs while in state i, then a transition is made to state 0' (because the system rolls back to the stable checkpoint at the start of the segment). Figure 7 shows the states entered during two executions of an example task.

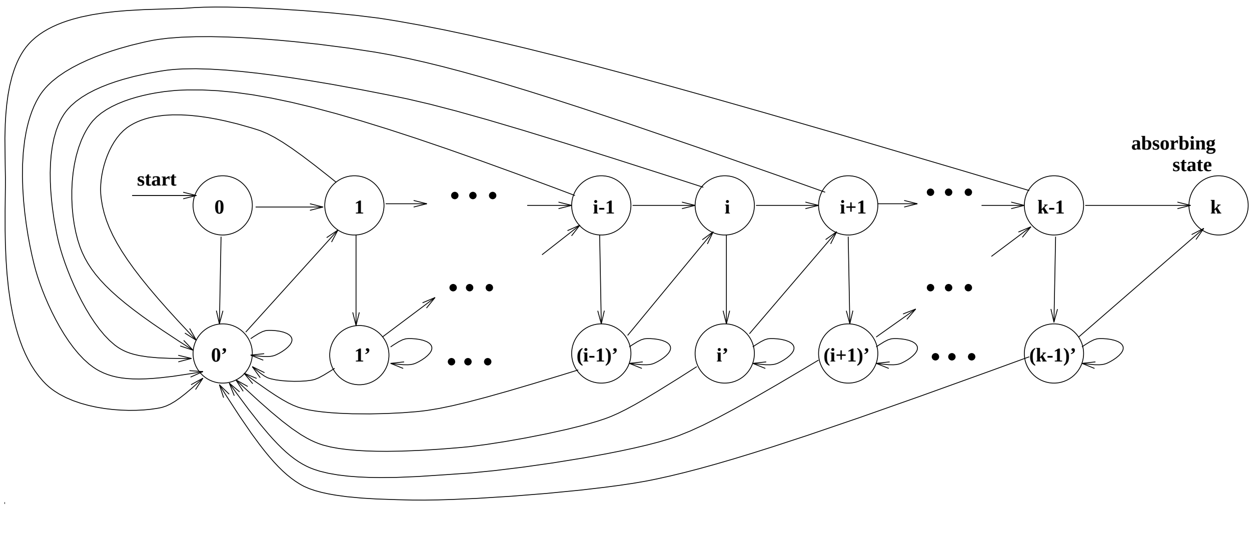

Figure 8: Markov chain for $k > 0$

Note that there are no transitions out of state k. States 0 and 0' differ from the other states because the most recent checkpoint accessible while in states 0 and 0' is a *stable* checkpoint (refer Figure 7). For all other states, the most recent checkpoint accessible while in those states is a *local* checkpoint. Similarly, states $(k-1)$ and $(k-1)$' differ from the other states because, in the absence of a failure, the checkpoint established *after* entering states $(k-1)$ and $(k-1)$' is a *stable* checkpoint. For all other states, the checkpoint established after entering the states is a *local* checkpoint. Therefore, the states are divided into four sets: (a) states i and i', $0 < i < k-1$, (b) states 0 and 0', (c) states $(k-1)$ and $(k-1)$', and (d) absorbing state k. We first consider states i and i', where $0 < i < k-1$, and obtain the transition probabilities and weights for the transitions out of these states.

Transition from state i to state $(i+1)$: State i is entered when the i-th local checkpoint $(0 < i < k-1)$ after the start of the segment is established. The system makes a transition from state i to state $(i+1)$ when the next checkpoint is established without any failure occurring after entering state i. In the absence of failures, $T + C_l$ units of time[5] is spent in state i before entering state $i+1$ (e.g., refer state 1 in Figure 7(a)). Therefore, the probability of the transition from state i to $(i+1)$ is equal to the probability that no failure occurs during $T + C_l$ units of execution. Thus, $P_{i(i+1)} = e^{-N\lambda(T+C_l)}$. (Recall that $\lambda = \lambda_p + \lambda_l$.)

The weight $(W_{i(i+1)})$ of this transition is equal to $T + C_l$, as $T + C_l$ units of time is required to reach state $(i+1)$ after entering state i, if no failure occurs.

Transition from state i to state i' : This transition takes place when a *transient processor failure* occurs after entering state i, but before the next checkpoint is established. The probability of this transition is equal to the probability that a failure occurs during $T + C_l$ units of execution and that the failure is a transient processor failure. Thus,

$$P_{ii'} = (1 - e^{-N\lambda(T+C_l)})\, \frac{(1-p)\lambda_p}{\lambda_p + \lambda_l} = (1 - P_{i(i+1)})\, \frac{(1-p)\lambda_p}{\lambda}$$

The weight of this transition is equal to the *expected* time spent in state i, until a transition to state i' is made. The *probability density function* (pdf) for the *time to failure*, given that a transient processor failure has occurred within $T + C_l$ units **and** given that the transient processor failure occurred *before* a permanent processor failure or a local storage failure could have occurred, is given by

$$\frac{N\lambda\, e^{-N\lambda t}}{1 - e^{-N\lambda(T+C_l)}}$$

[5] Of the $T + C_l$ time units, $T - L_l + C_l$ is spent in executing the $(i+1)$-th checkpoint interval and L_l in latency period of the $(i+1)$-th checkpoint. Of the latency period L_l, $L_l - C_l$ is spent executing a part of the $(i+2)$-th checkpoint interval.

This *pdf* turns out to be identical to the *pdf* for the time to failure, given that a failure has occurred within $T + C_l$ units (the type of failure is unspecified). Now that we know the *pdf*, it follows that

$$W_{ii'} \;=\; \int_0^{T+C_l} (t)\, \frac{N\lambda\, e^{-N\lambda t}}{1 - e^{-N\lambda(T+C_l)}} dt \;=\; (N\lambda)^{-1} - \frac{(T+C_l)e^{-N\lambda(T+C_l)}}{1 - e^{N\lambda(T+C_l)}} \;=\; (N\lambda)^{-1} - \frac{T + C_l}{e^{N\lambda(T+C_l)} - 1}$$

At a first glance, it may seem counter-intuitive that the above expression contains λ (= $\lambda_p + \lambda_l$) and not just λ_p. However, note that a transition is made to state *i*' only if a transient processor failure occurs *before* a local storage or permanent processor failure can occur. Therefore, weight $W_{ii'}$ is a function of λ_p as well as λ_l.

Transition from state i **to state** $0'$ **:** This transition takes place when a *permanent* processor failure or a *local storage* failure occurs after entering state i, but before the next checkpoint is established. The probability of this transition is equal to the probability that a failure occurs during $T + C_l$ units of execution and that the failure is a permanent processor failure or a local storage failure. Thus,

$$P_{i0'} = (1 - e^{-N\lambda(T+C_l)})\, \frac{p\lambda_p + \lambda_l}{\lambda_p + \lambda_l} = (1 - P_{i(i+1)})\, \frac{p\lambda_p + \lambda_l}{\lambda} = 1 - P_{i(i+1)} - P_{ii'}$$

The weight ($W_{i0'}$) of this transition is identical to $W_{ii'}$ obtained above.

The transitions out of state *i*' are similar to those out of state i. Recall that presently we assume $0 < i < k - 1$ and $k > 1$.

Transition from state *i*' **to state** $(i + 1)$ **:** The probability of this transition is equal to the probability that no failure occurs during $R_l + T + L_l$ units of execution (e.g., refer state 1' in Figure 7(b)). Thus, $P_{i'(i+1)} = e^{-N\lambda(R_l+T+L_l)}$.

The weight ($W_{i'(i+1)}$) of this transition is equal to $R_l + T + L_l$, as $R_l + T + L_l$ units of time is required (in state i) before the next local checkpoint is established, provided no failure occurs.

Transition from state i' **to state** *i*' **:** Probability of this transition is equal to the probability that a failure occurs during $R_l + T + L_l$ units of execution and that the failure is a transient processor failure. Thus,

$$P_{i'i'} = (1 - e^{-N\lambda(R_l+T+L_l)})\, \frac{(1-p)\lambda_p}{\lambda_p + \lambda_l} = (1 - P_{i'(i+1)})\, \frac{(1-p)\lambda_p}{\lambda}$$

The weight of this transition is equal to the *expected* time spent in state i', until a transition back to state i' is made. It can be seen that

$$W_{i'i'} \;=\; \int_0^{R_l+T+L_l} (t)\, \frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(R_l+T+L_l)}} dt \;=\; (N\lambda)^{-1} - \frac{R_l+T+L_l}{e^{N\lambda(R_l+T+L_l)}-1}$$

Transition from state i' to state $0'$: This transition takes place when a *permanent* processor failure or a *local storage* failure occurs after entering state i', but before the next local checkpoint is established. The probability of this transition is equal to the probability that a failure occurs during R_l+T+L_l units of execution and that the failure is a permanent processor failure or a local storage failure. Thus,

$$P_{i'0'} = (1-e^{-N\lambda(R_l+T+L_l)})\, \frac{p\lambda_p+\lambda_l}{\lambda_p+\lambda_l} = (1-P_{i'(i+1)})\, \frac{p\lambda_p+\lambda_l}{\lambda} = 1-P_{i'(i+1)}-P_{i'i'}$$

The weight $(W_{i'0'})$ of this transition is identical to $W_{i'i'}$ obtained above.

Transition probabilities and weights for the transitions out of states 0, 0', $(k-1)$ and $(k-1)$' can be obtained similarly, as summarized below. (Recall that, at present, we are analyzing a type 2 segment with $k>1$.)

$$\begin{array}{ll}
P_{01} = e^{-N\lambda(T-L_s+C_s+L_l)} & W_{01} = T-L_s+C_s+L_l \\
P_{00'} = 1-P_{01} & W_{00'} = (N\lambda)^{-1} - \frac{T-L_s+C_s+L_l}{e^{N\lambda(T-L_s+C_s+L_l)}-1} \\
P_{0'1} = e^{-N\lambda(R_s+T+L_l)} & W_{0'1} = R_s+T+L_l \\
P_{0'0'} = 1-P_{0'1} & W_{0'0'} = (N\lambda)^{-1} - \frac{R_s+T+L_l}{e^{N\lambda(R_s+T+L_l)}-1} \\
P_{(k-1)k} = e^{-N\lambda(T-L_l+C_l+L_s)} & W_{(k-1)k} = T-L_l+C_l+L_s \\
P_{(k-1)(k-1)'} = (1-P_{(k-1)k})\, \frac{(1-p)\lambda_p}{\lambda} & W_{(k-1)(k-1)'} = (N\lambda)^{-1} - \frac{T-L_l+C_l+L_s}{e^{N\lambda(T-L_l+C_l+L_s)}-1} \\
P_{(k-1)0'} = 1-P_{(k-1)k}-P_{(k-1)(k-1)'} & W_{(k-1)0'} = W_{(k-1)(k-1)'} \\
P_{(k-1)'k} = e^{-N\lambda(R_l+T+L_s)} & W_{(k-1)'k} = R_l+T+L_s \\
P_{(k-1)'(k-1)'} = (1-P_{(k-1)'k})\, \frac{(1-p)\lambda_p}{\lambda} & W_{(k-1)'(k-1)'} = (N\lambda)^{-1} - \frac{R_l+T+L_s}{e^{N\lambda(R_l+T+L_s)}-1} \\
P_{(k-1)'0'} = 1-P_{(k-1)'k}-P_{(k-1)'(k-1)'} & W_{(k-1)'0'} = W_{(k-1)'(k-1)'}
\end{array}$$

7.1.2 Evaluating the expected segment completion time

To evaluate the expected time required to execute a segment of type 2, we first reduce the number of states in the above Markov chain. Then, we evaluate the *expected number of entries* into each state in the Markov chain. The expected segment completion time can be evaluated using the expected number of entries, as elaborated below.

The first step is is to reduce the number of states by merging states i and i' for $1 \le i \le k-1$. The new Markov chain is shown in Figure 9 (recall that $k>1$). State

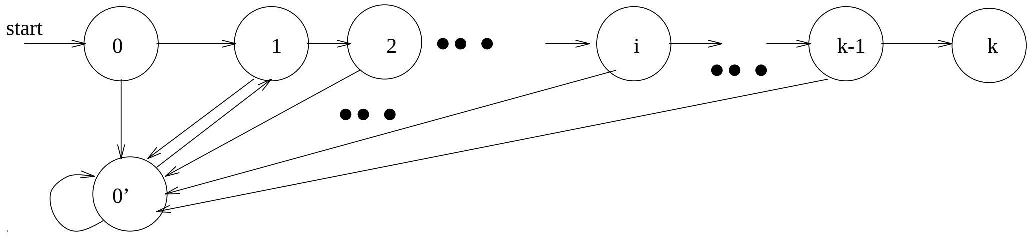

Figure 9: Reducing the number of states

i, $1 \leq i \leq (k-1)$ in the new Markov chain is obtained by merging states i and i' in the original Markov chain. (States 0 and 0' are not merged.)

Let Q_{ab} denote the state transition probability for the transition from state a to state b in the new Markov chain. Also, let V_{ab} denote the weight of the transition from state a to state b in the new Markov chain in Figure 9. Then, it can be seen that,

$$\begin{array}{ll} Q_{01} = P_{01} & V_{01} = W_{01} \\ Q_{00'} = P_{00'} & V_{00'} = W_{00'} \\ Q_{0'1} = P_{0'1} & V_{0'1} = W_{0'1} \\ Q_{0'0'} = P_{0'0'} & V_{0'0'} = W_{0'0'} \end{array}$$

To obtain $Q_{i(i+1)}$ and $V_{i(i+1)}$ $(1 \leq i \leq (k-1))$ observe that, in Figure 8, state $(i+1)$ can be reached from state i by two paths which do not go through state 0'. The first path is simply the direct transition from state i to $i+1$. The other path includes a transition from state i to i' and then a transition from state i' to $i+1$. Note, however, that once state i' is reached, on average, $P_{i'i'}/(1-P_{i'i'})$ transitions are made back to i' before a transition out of state i' occurs. Based on these observations the following expressions for $Q_{i(i+1)}$ and $V_{i(i+1)}$ are obtained. Similar observations also lead to the following expressions for $Q_{i0'}$ and $V_{i0'}$.

Also, for $1 \leq i \leq (k-1)$

$$\begin{aligned} Q_{i(i+1)} &= P_{i(i+1)} + \frac{P_{ii'}P_{i'(i+1)}}{P_{i'(i+1)} + P_{i'0'}} \\ V_{i(i+1)} &= \frac{W_{i(i+1)}P_{i(i+1)} + (W_{ii'} + W_{i'i'}P_{i'i'}/(1-P_{i'i'}) + W_{i'(i+1)})\left(\frac{P_{ii'}P_{i'(i+1)}}{P_{i'(i+1)}+P_{i'0'}}\right)}{P_{i(i+1)} + \frac{P_{ii'}P_{i'(i+1)}}{P_{i'(i+1)}+P_{i'0'}}} \\ Q_{i0'} &= P_{i0'} + \frac{P_{ii'}P_{i'0'}}{P_{i'(i+1)} + P_{i'0'}} \\ V_{i0'} &= \frac{W_{i0'}P_{i0'} + (W_{ii'} + W_{i'i'}P_{i'i'}/(1-P_{i'i'}) + W_{i'0'})\left(\frac{P_{ii'}P_{i'0'}}{P_{i'(i+1)}+P_{i'0'}}\right)}{P_{i0'} + \frac{P_{ii'}P_{i'0'}}{P_{i'(i+1)}+P_{i'0'}}} \end{aligned}$$

The next step is to calculate the *expected* (average) number of entries into each state of the Markov chain in Figure 9. Let N_s denote the expected number of entries into state s. Clearly, $N_0 = N_k = 1$. The other N_s's can be obtained using standard techniques [27], as follows.

$$N_{0'} = \frac{Q_{00'} + Q_{01}Q_{10'} + \cdots + Q_{01}Q_{12}\cdots Q_{(k-2)(k-1)}Q_{(k-1)0'}}{Q_{0'1}Q_{12}Q_{23}\cdots Q_{(k-1)k}} = \frac{Q_{00'} + \sum_{n=1}^{k-1}\left(\Pi_{j=0}^{n-1}Q_{j(j+1)}\right)Q_{n0'}}{Q_{0'1}\,\Pi_{j=1}^{k-1}Q_{j(j+1)}}$$

For $1 \le i \le (k-1)$

$$N_i = \frac{1}{Q_{i(i+1)}Q_{(i+2)(i+3)}\cdots Q_{(k-1)k}} = \frac{1}{\Pi_{j=i}^{k-1}Q_{j(j+1)}}$$

The expected number of times a transition (a,b), from state a to state b, is taken can be obtained as $Q_{ab}\,N_a$. Then, the expected (average) segment completion time can be obtained as

$$\sum_{(a,b)} V_{ab}\ Q_{ab}\ N_a$$

where, the summation is over all transitions in the Markov chain in Figure 9.

As elaborated below, analysis of the segments of type 1, 3 and 4 is similar to the above analysis of a type 2 segment. Once we know the expected execution time for each segment of the task, the expected task completion time can be obtained by summing the expected completion time for all the segments of the task.

7.2 Average Execution Time of a Type 1 Segment

Analysis of a type 1 segment is similar to that of a type 2 segment. The Markov chain for a type 1 segment is essentially identical to that for type 2. The only difference is in the state transition probabilities and the weights for the transitions out of state 0 in Figure 8. The reason for this difference is that a segment of type 1 does not begin after a stable checkpoint, instead, this segment begins with the beginning of the task. The segment terminates with a stable checkpoint, similar to a segment of type 2.

For a type 1 segment, The state transition probabilities and the weights for the transitions out of state 0 are as follows:

$$P_{01} = e^{-N\lambda(T+L_l)}$$

$$
\begin{aligned}
P_{00'} &= 1 - P_{01} \\
W_{01} &= T + L_l \\
W_{00'} &= (N\lambda)^{-1} - \frac{T + L_l}{e^{N\lambda(T+L_l)} - 1}
\end{aligned}
$$

The rest of the analysis for a type 1 segment is identical to that for a type 2 segment.

7.3 Average Execution Time of a Type 3 Segment

Analysis of a type 3 segment is also similar to that of a type 2 segment. The Markov chain for a type 3 is essentially identical to that for type 2. The only difference is in the state transition probabilities and the weights for the transitions out of states $(k-1)$ and $(k-1)$' in Figure 8. The reason for this difference is that a segment of type 3 does not terminate with a stable checkpoint, instead this segment terminates with the completion of the task. The type 3 segment begins after a stable checkpoint, similar to the segment of type 2.

For a type 3 segment, The state transition probabilities and the weights for the transitions out of states $(k-1)$ and $(k-1)$' in Figure 8 are as follows ($k > 1$):

$$
\begin{aligned}
P_{(k-1)k} &= e^{-N\lambda(T-L_l+C_l)} \\
P_{(k-1)(k-1)'} &= (1 - P_{(k-1)k})\frac{(1-p)\lambda_p}{\lambda_p + \lambda_l} \\
P_{(k-1)0'} &= 1 - P_{(k-1)k} - P_{(k-1)(k-1)'} \\
P_{(k-1)'k} &= e^{-N\lambda(R_l+T)} \\
P_{(k-1)'(k-1)'} &= (1 - P_{(k-1)'k})\frac{(1-p)\lambda_p}{\lambda_p + \lambda_l} \\
P_{(k-1)'0'} &= 1 - P_{(k-1)'k} - P_{(k-1)'(k-1)'} \\
W_{(k-1)k} &= T - L_l + C_l \\
W_{(k-1)(k-1)'} &= (N\lambda)^{-1} - \frac{T - L_l + C_l}{e^{N\lambda(T-L_l+C_l)} - 1} \\
W_{(k-1)0'} &= W_{(k-1)(k-1)'} \\
W_{(k-1)'k} &= R_l + T \\
W_{(k-1)'(k-1)'} &= (N\lambda)^{-1} - \frac{R_l + T}{e^{N\lambda(R_l+T)} - 1} \\
W_{(k-1)'0'} &= W_{(k-1)'(k-1)'}
\end{aligned}
$$

The rest of the analysis for a type 1 segment is identical to that for a type 2 segment. When μ is not an integral multiple of k, the number of checkpoint intervals in a type 3 segment is less than k, say c. In that case, the expected execution time can be obtained by replacing k by c in the above analysis.

7.4 Average Execution Time of a Type 4 Segment

Analysis of a type 4 segment is also similar to that of a type 2 segment. In this case, the transition probabilities and weights for states 0, $(k-1)$ and $(k-1)$' are different from those for type 2. The new probabilities and the weights are identical to those listed above for type 1 and 3 segments. The remaining analysis is identical to a type 2 segment.

In the above, we assume $k > 1$. The analysis for $k = 1$ is much simpler, and is omitted here for brevity.

8 Numerical Results

In this section, we present numerical results to determine optimal values of k and μ for a given set of parameter values. Significant effort has been devoted in the past for analytically determining optimal checkpoint intervals for checkpointing and rollback recovery schemes [3, 7, 12, 25, 33, 34]. Due to the complexity of the expressions for the two-level recovery scheme under consideration, an analytical approach for determining optimal k and μ is not very attractive. Instead, we choose to determine the optimal values numerically.

Two goals of the analysis:

- To demonstrates that sometime a two-level recovery scheme can perform better than one-level schemes. We show this by evaluating the average task completion time for an example task.
- To analyze the impact of checkpoint *latency* on the performance overhead.

In this report, we present analytical results for a hypothetical task. The chosen parameter values are motivated by the experimental results presented in Section 10, and reference [16]. Conclusions drawn from the numerical results presented in this section are applicable to a wide range of parameters. Assuming the parameters shown in Table 1, Figure 10 plots the average task completion time for various values of k and μ. Presently we assume $L_s = C_s$ and $L_l = C_l$. We will consider the situations where $L_s > C_s$ and $L_l > C_l$ later in Section 9.

Due to the limitations of our graph-plotting software, μ is denoted as `mu` in the graphs. Similar convention is followed for other greek letters also.

$\lambda_p = 0.0001$	$\lambda_l = 0.00001$	$p = 0.05$
$N = 256$	$\Upsilon = 80$	
$C_s = L_s = R_s = 2.0$		
$C_l = L_l = R_l = 0.6$		

Table 1: Example parameters

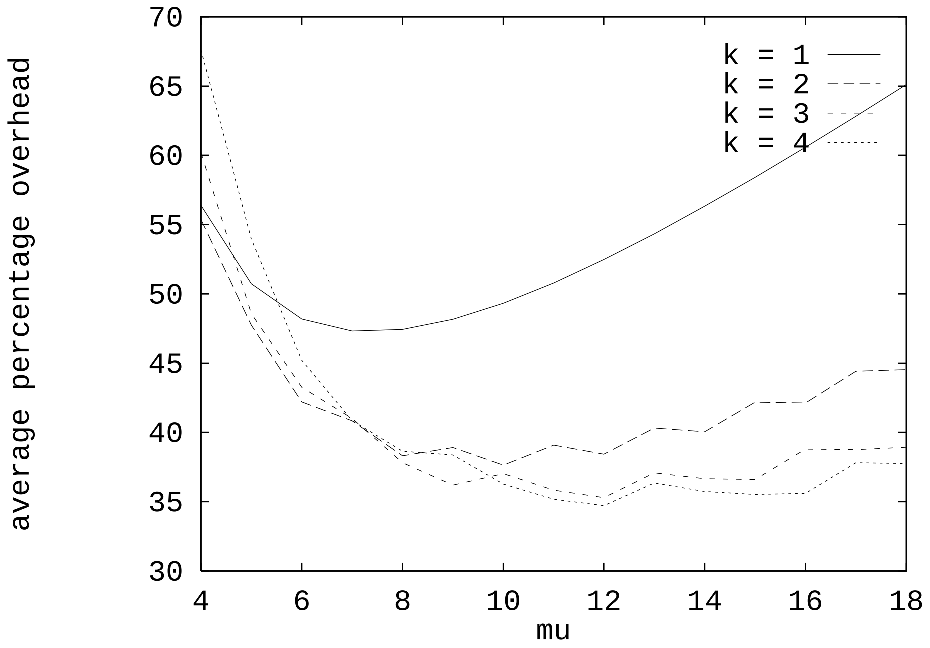

Figure 10: The curves are not always convex

The first interesting feature of the two-level recovery scheme is that the curves for average overhead may have multiple minimas – this is illustrated by the curve for $k = 2$ in Figure 10. These curves are not always convex, unlike the traditional checkpointing and rollback schemes (e.g., [3]).

The curve for $k = 1$ is also shown in Figure 10. When $k = 1$, all the checkpoints are stable checkpoints, and the two-level recovery scheme reduces to traditional checkpointing schemes. Therefore, as shown previously [3], the curve for $k = 1$ is convex and has exactly one minimum.

As seen in Figure 10, $k = 4$ can achieve a lower overhead as compared to $k = 1, 2, 3$. Figure 11 shows the curves for $k = 4, 5, 6, 7$. Observe that $k = 4$ can achieve a lower overhead than $k = 5, 6, 7$ also. In fact, our numerical search indicates that the average overhead is minimized when $k = 4$ and $\mu = 12$.

The fact that the average overhead is minimized when $k = 4$ implies that $k = 1$ does

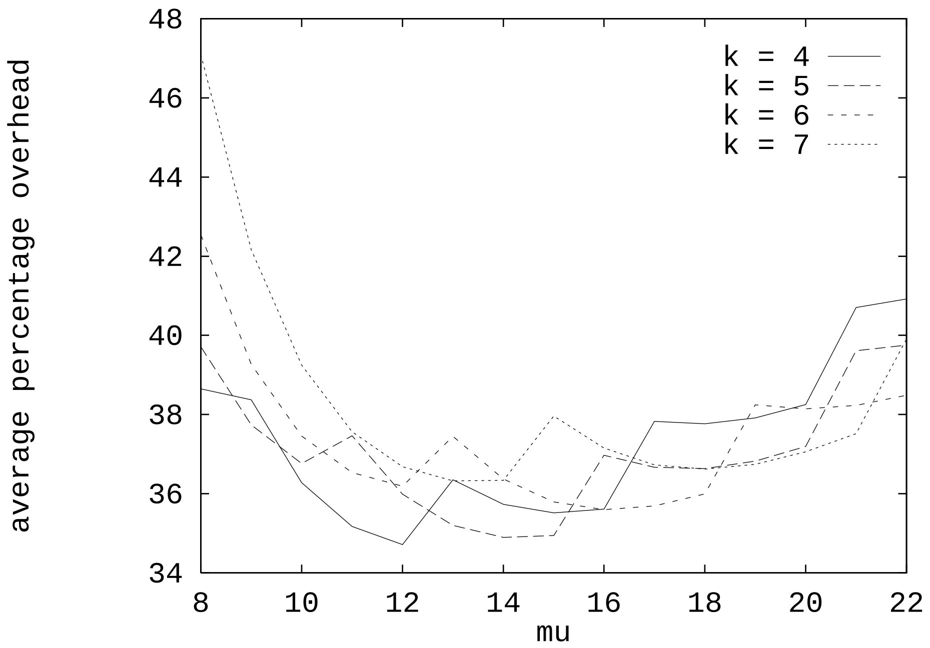

Figure 11: Minimum achieved when $k = 4$ and $\mu = 12$

not yield optimal performance. As $k = 1$ corresponds to taking only *stable* checkpoints, the graphs imply that a "one-level" recovery scheme that takes only stable checkpoints is not always optimal. Also, in this example, note that the optimal overhead is achieved when $k \neq \mu$. As the recovery scheme that takes only local checkpoints is obtained when $k = \mu$, the above results imply that taking only *local* checkpoints is also not optimal. In summary, the two-level scheme can achieve a better performance than the one-level recovery schemes that take only stable checkpoints or only local checkpoints.

To be fair, we should note that whether the two-level recovery scheme can achieve better performance or not depends completely on the parameter values. The above example illustrates that the two-level approach can sometimes perform better. Now, we present examples of parameters where the one-level schemes perform better.

For example, if $\Upsilon = 20$ (other parameters being the same as in Table 1), then the average overhead is minimized when $k = \mu = 3$. This means that, in this case, taking only *local* checkpoints minimizes the average overhead. The reason for this result is that the task length is sufficiently small compared to the mean time to a failure that *affects* a local storage (i.e., a local storage failure or a permanent processor failure). Therefore, the probability of such a failure is small. Thus, it is acceptable to restart the task on such a failure. Local checkpoints are taken, however, to minimize re-execution overhead in presence of *transient* processor failures, as they occur with a relatively higher probability. Figure 12 shows the curves for a few selected values of k, and also for $k = \mu$.

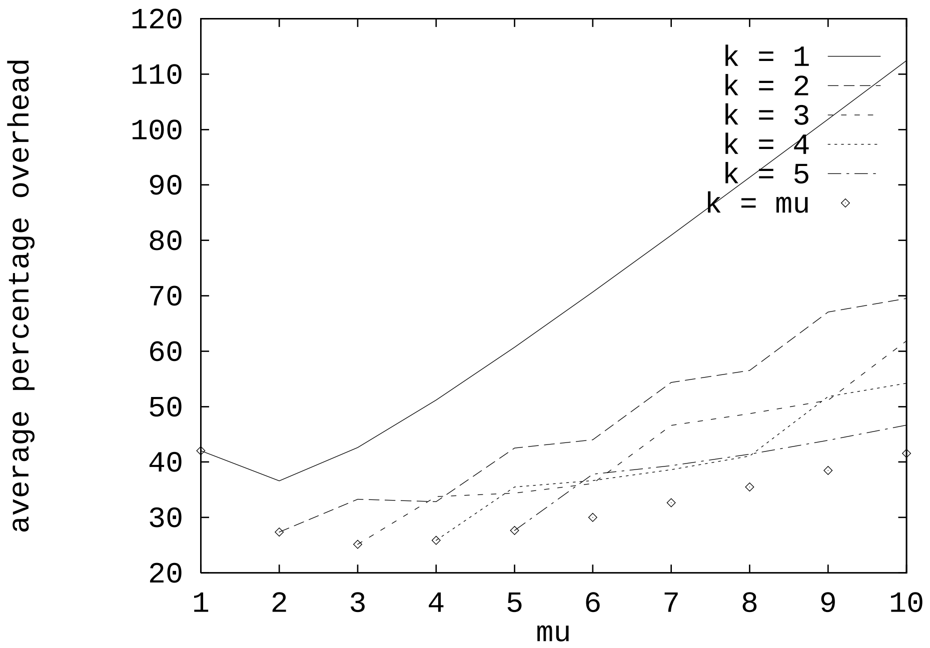

Figure 12: Example: Taking only *local* checkpoints ($k = \mu$) is optimal

If $C_l = L_l = R_l = 1.6$ (other parameters being the same as in Table 1), then the average overhead is minimized when $k = 1$ and $\mu = 7$. This means that, in this case, taking only *stable* checkpoints minimizes the average overhead. The reason for this result, in this case, is that the overhead of taking a local checkpoint is not sufficiently small compared to a stable checkpoint. Therefore, it is preferable to take a stable checkpoint, as it provides protection against all failures. Figure 13 shows the curves for a few selected values of k.

In general, it seems that, for most applications, either the two-level scheme will achieve lowest overhead, or the scheme that takes only local checkpoints will achieve lowest overhead (as, in practice, C_l is much smaller than C_s). Taking only *stable* checkpoints is not likely to be optimal in most cases. This is interesting, as most past implementations take only stable checkpoints (e.g., [5]), and therefore are often sub-optimal.

9 Impact of Checkpoint Latency on Performance

Increasing the checkpoint latency *does not* increase the *failure-free* execution time of the application. However, it does increase the "window of vulnerability" of a given checkpoint. To be more specific, larger checkpoint latency can increase the *rollback distance* upon a failure, i.e., the amount of computation lost due to a failure is likely to increase due to a larger checkpoint latency. We illustrate this with a comparison of two scenarios:

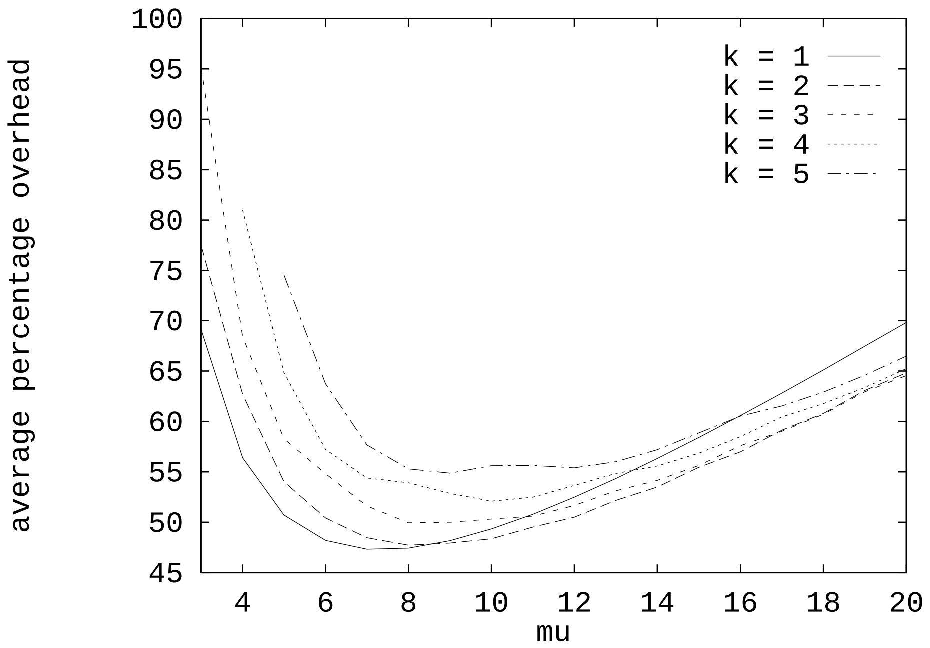

Figure 13: Example: Taking only *stable* checkpoints ($k = 1$) is optimal

Scenario 1: Consider the case when the checkpoint latency is identical to the checkpoint overhead. Figure 14(a) illustrates a scenario where a failure has occurred and the system has rolled back to the recent local checkpoint CP1. To avoid another rollback, no failure should occur until the next checkpoint is established, i.e., no failure should occur within $R_l + T + C_l$ time units after the first failure (we assume that the next checkpoint is a local checkpoint).

Scenario 2: Now consider the case when the checkpoint latency is greater than the checkpoint overhead (all other parameters being identical to scenario 1). Figure 14(b) illustrates a scenario where a failure has occurred and the system has rolled back to the recent local checkpoint CP1. To avoid another rollback, no failure should occur until the next checkpoint is established, i.e., no failure should occur within $R_l + T + L_l$ time units after the first failure.

L_l in scenario 2 is larger than C_l in scenario 1. The *window of vulnerability* in scenario 2 is $(R_l + T + L_l)$, which is larger than the *window of vulnerability* in scenario 1 $(R_l + T + C_l)$. This implies that there is a greater chance of another rollback in scenario 2, as compared to scenario 1.

The above discussion suggests that larger checkpoint latency may result in worse performance. To evaluate the impact of checkpoint latency on system performance, we evaluate the performance overhead as a function of checkpoint latency. We limit the values

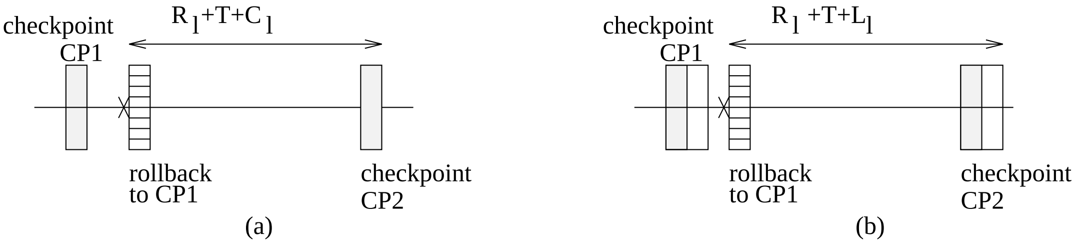

Figure 14: Performance impact of checkpoint latency

of k and μ such that two checkpoint latency periods do not overlap.

Assume that $\frac{L_s}{C_s} = \frac{L_l}{C_l} = \alpha$. (Thus, for the parameters values used in Section 8, $\alpha = 1$.) In practice, $\frac{L_s}{C_s}$ and $\frac{L_l}{C_l}$ may not be identical. We limit the number of graphs by considering only the case of $\frac{L_s}{C_s} = \frac{L_l}{C_l}$.

Consider a task for which all parameters, except L_s and L_l, are as in Table 1. Figure 15 is plotted assuming that $L_s = \alpha C_s$ and $L_l = \alpha C_l$, for various values of α, k and μ. Observe that, with all other parameters being fixed, the average overhead increases almost linearly with increasing α. Also, the increase is significant for the chosen parameter values. The increase in the average overhead with increasing α becomes less significant for smaller failure rates. Figure 16 is plotted for $\lambda_p = 0.00005$, $\lambda_l = 0.000005$, $L_s = \alpha C_s$ and $L_l = \alpha C_l$. (All other parameters are as in Table 1). Observe that the average overhead again increases almost linearly with α, however, the rate of increase is smaller as compared to Figure 15.

The above results suggest that when the failure rate ($N\lambda$) is large, a large checkpoint latency can have a detrimental effect on performance. Note, however, that the above graphs hold the checkpoint overhead constant while increasing the latency. In practice, an increase in the latency is typically associated with a decrease in the overhead. Therefore, a higher latency can in fact result in an *improvement* in performance, if the checkpoint overhead is reduced adequately. A study of the relationship between checkpoint overhead and checkpoint latency is a subject of further research.

Similar to our observation in Section 8, when checkpoint latency is larger than checkpoint overhead also, the two-level scheme can perform better than the schemes that take only stable checkpoints or only local checkpoints (numerical results are omitted for brevity). Also, taking only *stable* checkpoints seems to be sub-optimal for many parameter values.

10 Experimental Evaluation

To get an estimate of the relative values of checkpoint latency and checkpoint overhead for *local* and *stable* checkpoints, we implemented the proposed two-level recovery scheme on a

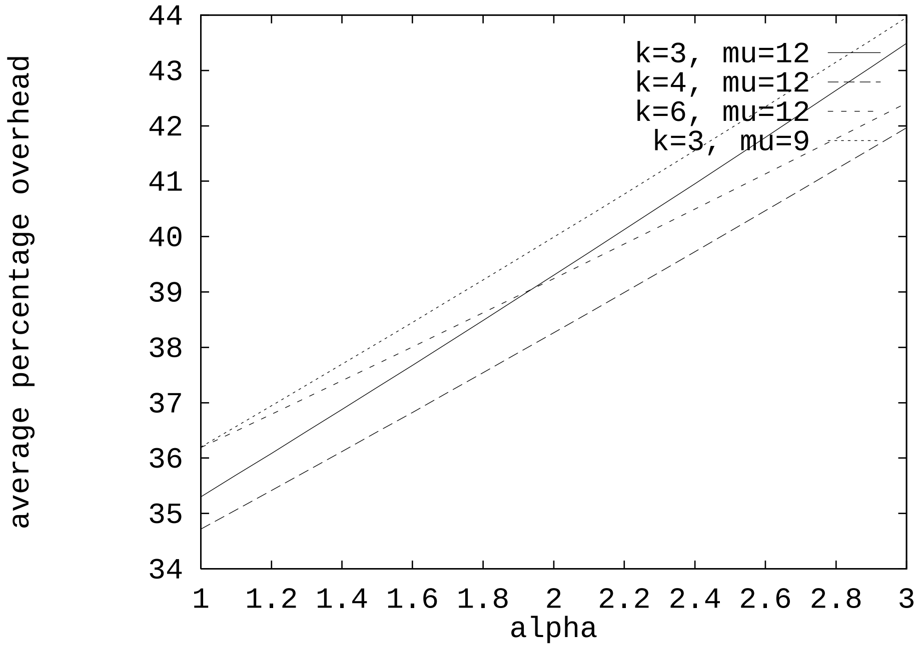

Figure 15: Dependence of average overhead on checkpoint latency

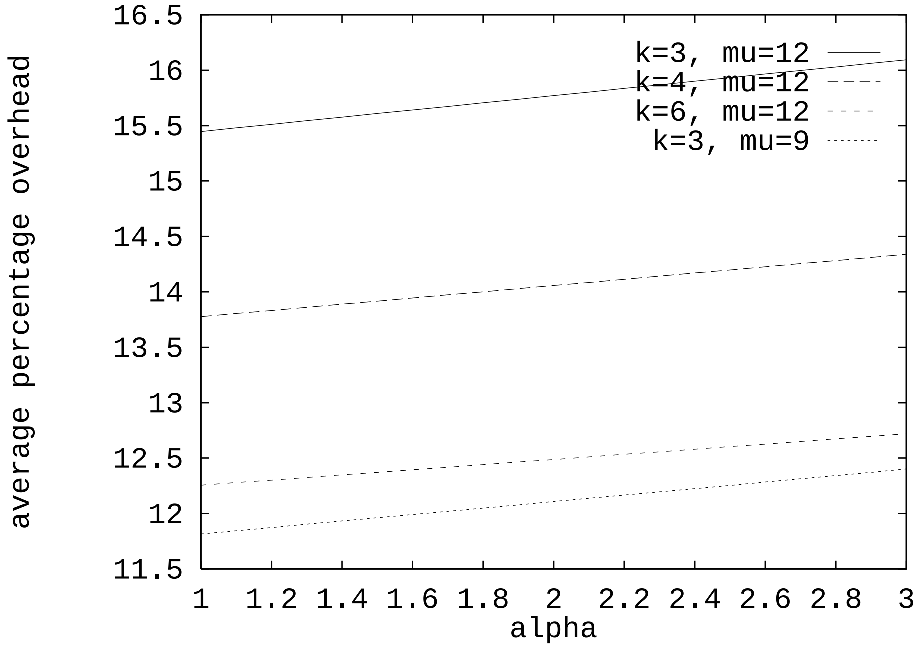

Figure 16: Dependence of average overhead on checkpoint latency – with smaller failure rates than Figure 15

network of workstations, each workstation having a local disk. The checkpointing scheme is implemented on top of Unix (at user level). The *local* checkpoints are stored on the local disks, whereas the *stable* checkpoints are stored on a disk that is accessed over the network. The application processes can communicate with each other using a message-passing library that we have developed. (The application does not directly access *sockets*.) A software layer has been implemented that provides additional support to transparently establish logical communication *channels* between any pair of application processes.

In this report, we present measurements for four *uni-process* applications. Results for *multi-process* applications will be included in a future revision of this report.[6] Specifically, the uni-process applications were executed on a Sun SPARCstation-10 workstation (with SunOS 4.1.3) using a Sun Sparc1000 server (over 10 Mbps Ethernet) as the stable storage. The SPARCstation-10 workstation has about 17 Mbyte RAM memory free to be used by an application. The page size on this machine is 4096 bytes (or 1024 words).

Two methods for taking individual process checkpoints are presently evaluated. (Both the methods have been previously used by other researchers also [5, 14, 19].)

- "Sequential-checkpoint" : In this method, when a process wants to take a checkpoint, it saves its state on the storage, and then proceeds with the computation. The computation is *not* overlapped with the checkpointing operation. With this method, for uni-process applications, checkpoint *overhead* and checkpoint *latency* may be expected to be identical.

  For multi-process applications, the latency can be larger than the overhead, as the *consistent* checkpointing algorithm may require the processes to take checkpoints at somewhat different times, i.e., the processes may not start (or complete) checkpoints at exactly the same physical time. For a consistent checkpoint, the latency is defined as the time from the instant when the first process initiated its checkpoint, till the instant when the last process established its checkpoint. Even if each process uses sequential-checkpointing, the latency of the *consistent* checkpoint is likely to be larger than the overhead.

- "Forked-checkpoint" : In this method, when a process wants to take a checkpoint, it forks a child process. The child process then saves its state on the storage to establish a checkpoint. The original process (i.e., parent) continues computation while the child process is saving the checkpoint. In this method, computation is overlapped with checkpointing, therefore, checkpoint overhead may be expected to be smaller than the checkpoint latency. Also, the checkpoint overhead with this method may be expected to be smaller than sequential-checkpointing.

[6] Plank [20] presents measurements of checkpoint latency and overhead for applications executed on an iPSC/860 multicomputer. These measurements, however, do not provide information regarding the relationship between *local* and *stable* checkpoint overheads and latencies.

In either method, the executable code is *not* saved as a part of the checkpoint.

We present experimental results for four applications. The first application (MAT) performs matrix multiplication on square matrices. We measured performance of each application with different checkpoint sizes. The checkpoint size for MAT was varied by changing the matrix size. The second application (FFT-3) performs the Cooley-Tukey fast Fourier transform (FFT) algorithm on three sets of data points. The checkpoint size was varied by changing the size of each set of data points. The remaining two applications were synthetic. In the present-day Unix implementations (and its variants), the *fork* command is implemented using the *copy-on-write* technique [23]. Hence, the checkpoint overhead of *forked-checkpoint* may be expected to depend on the program's *locality of reference*. Therefore, we implemented two types of locality:

- "Low-locality" (LL) : The pseudo-code for this application is presented in Figure 17(a). In this case, the program accesses memory locations from different memory pages in a rapid succession. The program consists of a `for` loop that updates one memory location in each iteration. To get "low-locality", each iteration accesses a location in a different page.

  The page size on our machine is 1024 words (4096 bytes), while the lower dimension of the `state` matrix in program LL is 16384 ($= 16 \times 1024$). Therefore, the `for` loop in program LL accesses the memory pages in sixteen parts – each part accesses one-sixteenth of the pages. Thus, at any time, memory accesses in the `for` loop are localized to one-sixteenth of the memory pages containing `state` matrix.

  To further minimize the locality of reference, the lower dimension of the `state` matrix can be made equal to 1024. (Such a program may be said to have "zero-locality", as it accesses *all* pages in a rapid succession.) We plan to perform experiments with "zero-locality" as well, however, the results for LL program suffice to illustrate the effect of variations in the locality of access.

- "Full-locality" (FL) : The pseudo-code for this application is presented in Figure 17(b). In this case, the program accesses only one location within the `state` array, although the program state is much bigger. To avoid defeating the purpose behind "full-locality", we <u>*do not*</u> perform incremental checkpointing. (In fact, incremental checkpointing is not used in any of our experiments.)

In the above two cases, the size of the checkpoint was varied by changing parameter XLEN in Figure 17.

Figure 18 illustrates how the checkpoint overhead and checkpoint latency were measured for the two checkpointing methods. Figure 18(a) shows an execution of a program *without* any checkpoints. The execution time for this program is obtained as the difference

```
    Low-locatity (LL)                       Full-locality (FL)
    -----------------                       ------------------

float state[XLEN][16384];               float state[XLEN][16384] ;

repeat N times {                        repeat N time {
 for (k=0; k<16384; k++)                 for (k=0; k<16384; k++)
  for (j=0; j<XLEN; j++                   for (j=0; j<XLEN; j++)
   state[j][k] = state[j][k] + 1.2;        state[0][0] = state[0][0] + 1.2;
 }                                        }
```

Figure 17: C-like pseudo-code for programs LL and FL

between the start time and the end time. To estimate the correct execution time, we executed each program at least 10 times and calculated the average execution time. Let the estimated (average) execution time for a program without checkpoints be denoted as E.

Figure 18(b) shows an execution of a program that takes sequential-checkpoints. Again, the total execution time is obtained as the difference between the start time and the end time. The total execution time is estimated by averaging over at least 10 executions of the program. Let the estimated execution time for the program with sequential checkpoints be E_s. Then, the overhead of a sequential checkpoint is calculated as $(E_s - E)/n$, where n is the number of checkpoints taken during each execution of the program. For sequential checkpointing, we expect the checkpoint overhead to be identical to the checkpoint latency. To validate this, we also measured the checkpoint latency, as shown in Figure 18(b). The checkpoint latency of a sequential checkpoint is calculated as the difference between the time when the checkpoint is initiated and the time when the checkpoint is completed. Average checkpoint latency is calculated by taking an average over all checkpoints taken during all executions of the application. (In our measurements, the average was calculated over at least 60 samples.)

Figure 18(c) shows an execution of a program that takes forked-checkpoints. The total execution time is estimated by averaging over at least 10 executions of the program. Let the estimated execution time for the program with forked-checkpoints be E_f. Then, the overhead of a forked-checkpoint is calculated as $(E_f - E)/n$, where n is the number of checkpoints taken during each execution of the program. To determine the checkpoint latency, we noted the time when a child is forked, and the time when the child exits *after* saving the checkpoint. The checkpoint latency was calculated as a difference between the two time observations. (The time at which the child exits is determined by a signal handler for SIGCHLD signal.) Similar to sequential-checkpoints, the average checkpoint latency is calculated as an average over at least 60 samples.

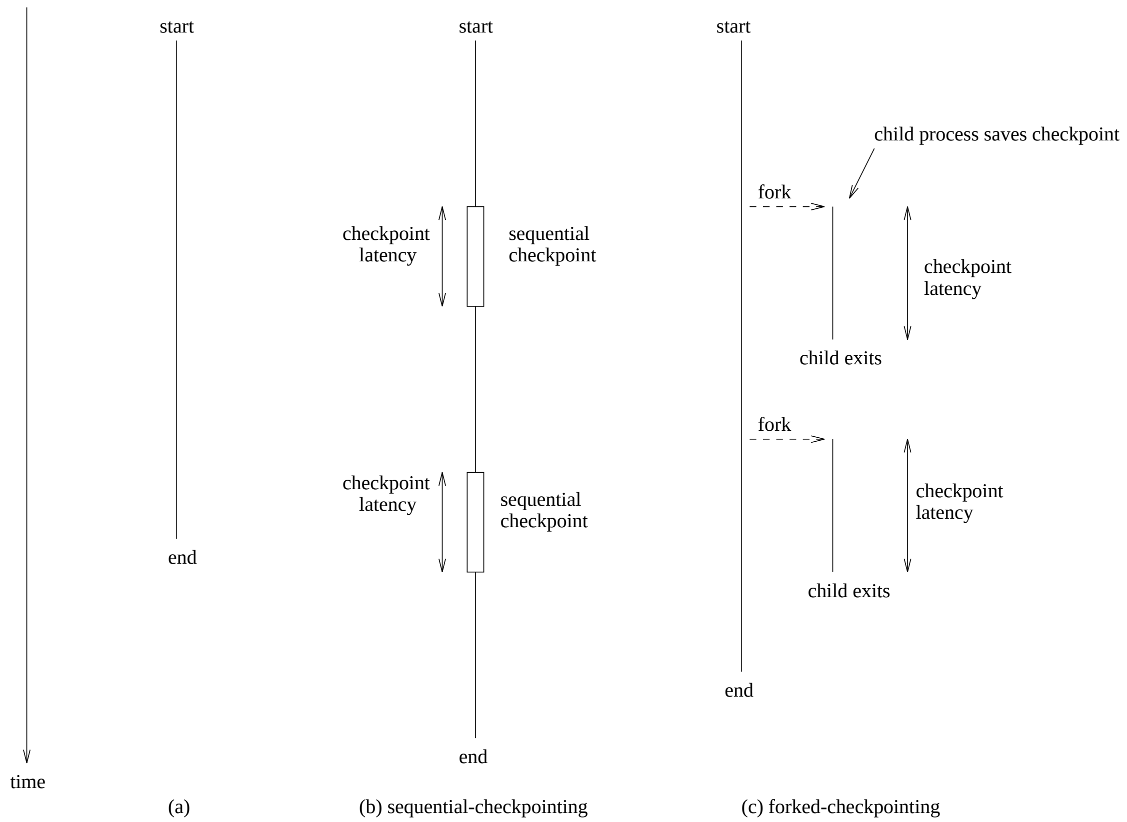

Figure 18: Measurement scheme

We now present the experimental results, followed by a discussion of the results. Tables 2 through 5 present checkpoint overhead and checkpoint latency measurements. In these tables, a `no` in the `chkpt` column implies that no checkpoints are taken. Also, a `yes` in the `fork` column implies that forked-checkpoints are taken, a `no` in the `fork` column implies that sequential-checkpoints are taken. The `local/stable` column in the tables indicates whether the checkpoints are stored on local storage or stable storage. Figures 19 through 22 plot checkpoint overheads as a function of checkpoint size. As noted earlier, checkpoint sizes were varied by changing the data sizes for the various applications. Observe that, in most cases, the checkpoint overheads increase almost linearly with checkpoint size – the rate of increase depends on where the checkpoint is stored and whether the checkpoint is *sequential* or *forked*.

For most measurements of average execution time presented in the following, the *standard deviation* [27] of the execution time is less than 0.4% of the average execution time. For some measurements the standard deviation is larger than 0.4%, but never exceeds 1% of the average execution time.

In the remainder of this section, we present some observations based on the above

MAT

chkpt size (bytes)	number of chkpts	chkpt	fork	local/ stable	execution time (milisec)	chkpt overhead (milisec)	chkpt latency (milisec)	latency/ overhead (ratio)
759748	8	no	–	–	37642	–	–	–
	8	yes	no	local	42156	564	561	0.994
	8	yes	yes	local	38823	147	595	4.04
	8	yes	no	stable	57205	2445	2444	1
	8	yes	yes	stable	40279	329	2536	7.71
2045892	13	no	–	–	173067	–	–	–
	13	yes	no	local	188866	1215	1219	1
	13	yes	yes	local	177944	375	1271	3.39
	13	yes	no	stable	257432	6489	6473	0.997
	13	yes	yes	stable	184604	887	6695	7.54
3962820	18	no	–	–	473901	–	–	–
	18	yes	no	local	513055	2175	2162	0.994
	18	yes	yes	local	487342	746	2225	2.98
	18	yes	no	stable	705039	12841	12797	0.996
	18	yes	yes	stable	506092	1788	13129	7.34
5822404	22	no	–	–	847251	–	–	–
	22	yes	no	local	917342	3185	3147	0.988
	22	yes	yes	local	873037	1172	3265	2.79
	22	yes	no	stable	1279125	19630	19552	0.996
	22	yes	yes	stable	903099	2538	19913	7.84

Table 2: Measurements for application MAT

experimental data.

Sequential-Checkpointing: As seen from the tables, the ratio of checkpoint latency and checkpoint overhead for (local or stable) sequential checkpoints is very close to 1 for most cases (only exception is of local checkpoints for program LL with checkpoint size 8.4 Mbyte). Observe that the measured value of latency is often slightly smaller than the overhead. However, in most cases, the difference is too small to be statistically significant – in such cases, latency and overhead should be considered to be, essentially, identical. Therefore, our analysis in Section 7 is applicable to sequential-checkpointing.

The case of program LL with checkpoint size 8.4 Mbyte is an exception. In this case, for local checkpoints, the L_l/C_l ratio is 0.816 (much smaller than 1). We do not completely understand the reason for this phenomenon, and we are investigating it at the present. We can envisage one reason that may cause this behavior:

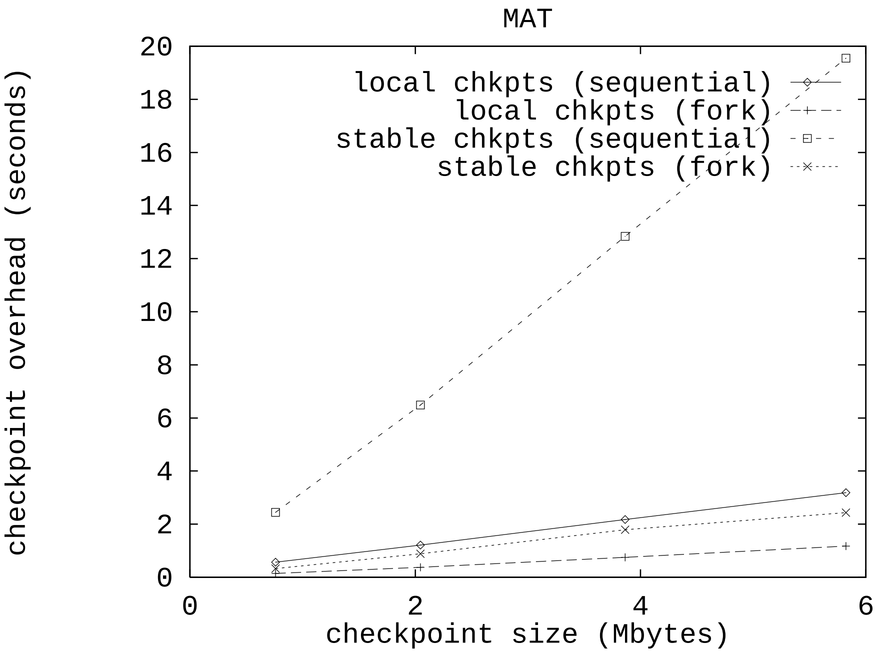

Figure 19: Measurements for application MAT

- When the process takes a checkpoint, it uses the `write` system call. The `write` system call usually copies the data into a buffer, and returns. (The `write` call can return before the data is actually written to the disk – we use the `fsync` system call to ensure the data is actually written to disk.) When checkpoint size is larger than half the available memory, the process state and the buffer require all the available memory. This may cause some virtual memory pages to be written to secondary storage (i.e., local disk). After the checkpoint is completed, these pages must be read back to the main memory (as program LL eventually needs to modify these pages). These page-ins will result in an overhead *after* the checkpoint is completed.

  The above reasoning also explains why high overhead is not observed for program FL with checkpoint size 8.4 M. Program FL accesses only two pages of memory, one page containing `state[0][0]` and another containing `j` and `k`. (Even if other pages are paged-out to the local disk, they do not have to be paged-in, unlike program LL.)

Unfortunately, the information presently available to us does not support the above explanation. We plan to perform further experiments to resolve this issue.

Forked-Checkpointing: As should be expected, the latency of a forked-checkpoint is (in most cases) much larger than the overhead of a forked-checkpoint. (Thus, the analysis in Section 7 is applicable.) Also, observe (in the tables) that the latency of forked-checkpoints

FFT-3

chkpt size (bytes)	number of chkpts	chkpt	fork	local/ stable	execution time (milisec)	chkpt overhead (milisec)	chkpt latency (milisec)	latency/ overhead (ratio)
1096428	6	no	–	–	42186	–	–	–
	6	yes	no	local	46740	759	757	0.997
	6	yes	yes	local	43944	293	811	2.76
	6	yes	no	stable	63230	3507	3496	0.997
	6	yes	yes	stable	45558	562	3580	6.37
2145004	6	no	–	–	91005	–	–	–
	6	yes	no	local	98764	1293	1278	0.988
	6	yes	yes	local	94125	520	1352	2.6
	6	yes	no	stable	132615	6935	6935	1
	6	yes	yes	stable	97377	1062	7107	6.7
4242156	6	no	–	–	194368	–	–	–
	6	yes	no	local	208547	2363	2310	0.978
	6	yes	yes	local	200922	1092	2422	2.2
	6	yes	no	stable	279943	14262	14227	0.997
	6	yes	yes	stable	207468	2183	14492	6.6
8436460	6	no	–	–	417782	–	–	–
	6	yes	no	local	445697	4652	4637	0.997
	6	yes	yes	local	429606	1970	4962	2.5
	6	yes	no	stable	598410	30104	30127	1.0007
	6	yes	yes	stable	443729	4324	31233	7.2

Table 3: Measurements for application FFT-3

is a little larger than the latency (and overhead) of corresponding sequential-checkpoints. This is reasonable, because the child process cannot possibly save the checkpoint any faster than the parent process can (when it takes sequential-checkpoints). The overhead of forked-checkpoints is significantly smaller than that for the corresponding sequential-checkpoints. (Program LL with checkpoint size 8.4 M is an exception, as before.)

Let *overhead ratio* be the ratio of the overhead of a sequential-checkpoint and that of the corresponding forked-checkpoint. The overhead ratio can be seen to be between 2 and 4 for local checkpoints, and between 6 and 9 for stable checkpoints (in most cases). For example, for FFT-3 with checkpoint size 4.2 Mbyte, the overhead ratio for local checkpoints is (2363/1092) = 2.16 and that for stable checkpoints is (14262/2183) = 6.53. Different ratios for local checkpoints and stable checkpoints are obtained due to the differences in access rates for the local and stable storages.

The overhead ratio for stable checkpoints of program LL with checkpoint size 8.4

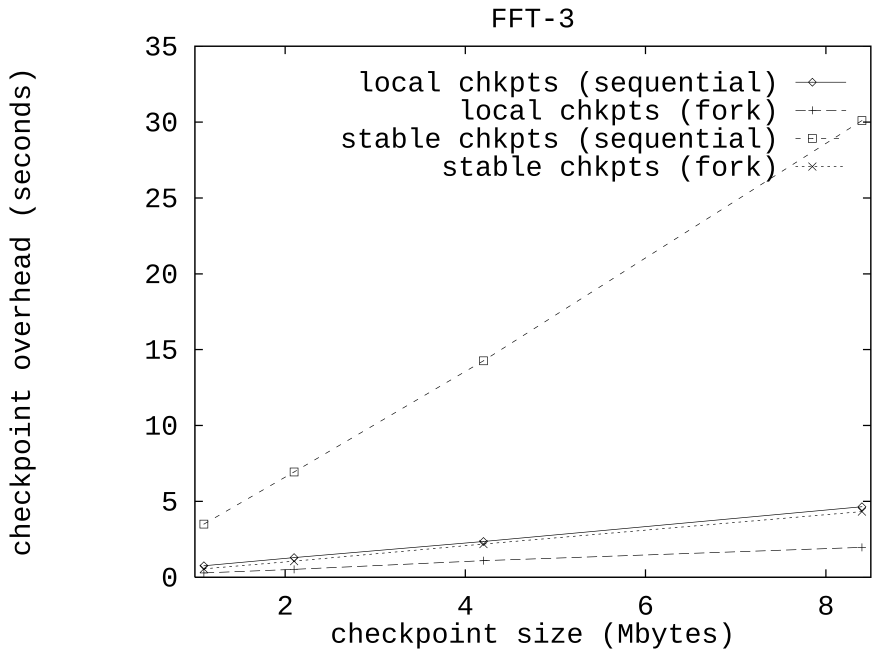

Figure 20: Measurements for application FFT-3

M is (30619/7004)=4.37, significantly smaller than other programs (or smaller checkpoint sizes). This behavior seems to be caused because the checkpoint size is large. Note that the checkpoint size is about half the size of available memory (17 M). As the program has no locality of reference, when forked-checkpointing is used, the parent and child together require all the available memory. This implies that when the child process tries to `write` the state, adequate buffer space is not available to make a copy of the state. We believe that this causes an increase in the overhead of the forked-checkpoint, resulting in a lower overhead ratio.

Low-locality and Full-locality: As should be expected, the FL program has a smaller execution time than the LL program, as FL can cache all necessary data at all times. Our interest, however, is in the checkpoint overheads and latency. The overhead or latency of sequential-checkpointing is largely a function of checkpoint size (locality should not affect sequential checkpointing). This is reflected in the experimental measurements for sequential-checkpoints of the two programs. The exception to this is, again, LL with checkpoint size 8.4 M.

As expected, the overhead of forked-checkpoints is smaller for full-locality (FL) than low-locality (LL). For program LL, whenever the parent modifies a page, a copy of the original page must be made for the child process. As LL modifies different pages in a quick succession, a large number of pages must be copied – the parent must wait each time a page low-locality (LL)

chkpt size (bytes)	number of chkpts	chkpt	fork	local/ stable	execution time (milisec)	chkpt overhead (milisec)	chkpt latency (milisec)	latency/ overhead (ratio)
1087404	6	no	–	–	32992	–	–	–
	6	yes	no	local	37423	739	731	0.989
	6	yes	yes	local	34838	308	776	2.5
	6	yes	no	stable	53849	3476	3435	0.988
	6	yes	yes	stable	36455	577	3529	6.1
2135980	6	no	–	–	67567	–	–	–
	6	yes	no	local	74934	1227	1246	1.015
	6	yes	yes	local	70376	468	1310	2.8
	6	yes	no	stable	108574	6835	6879	1.006
	6	yes	yes	stable	73680	1019	7056	6.9
4233132	6	no	–	–	170850	–	–	–
	6	yes	no	local	184575	2287	2278	0.996
	6	yes	yes	local	177131	1046	2397	2.3
	6	yes	no	stable	256466	14269	14162	0.993
	6	yes	yes	stable	185172	2387	14383	6
8427436	6	no	–	–	358335	–	–	–
	6	yes	no	local	392112	5629	4596	**0.816**
	6	yes	yes	local	394739	6067	4971	**0.819**
	6	yes	no	stable	542054	30619	30097	0.983
	6	yes	yes	stable	400361	7004	31171	4.45

Table 4: Measurements for application LL (low locality)

is being copied, adding to the overhead. For FL, only two pages are modified by the parent, therefore, all other pages can be shared by the parent and child (and at most two pages need to be copied). Therefore, the overhead of forked-checkpoints is smaller for FL as compared to LL.

Interestingly, the latencies of forked-checkpoints for LL and FL are comparable. We had expected the latency of forked-checkpoints for FL to be larger than LL.

11 Related Work

We define *two-level* recovery schemes as those that tolerate the *more probable* failures with a low overhead, while the *less probable* failures may incur a higher overhead. This definition can also be extended to *multi-level* recovery schemes.

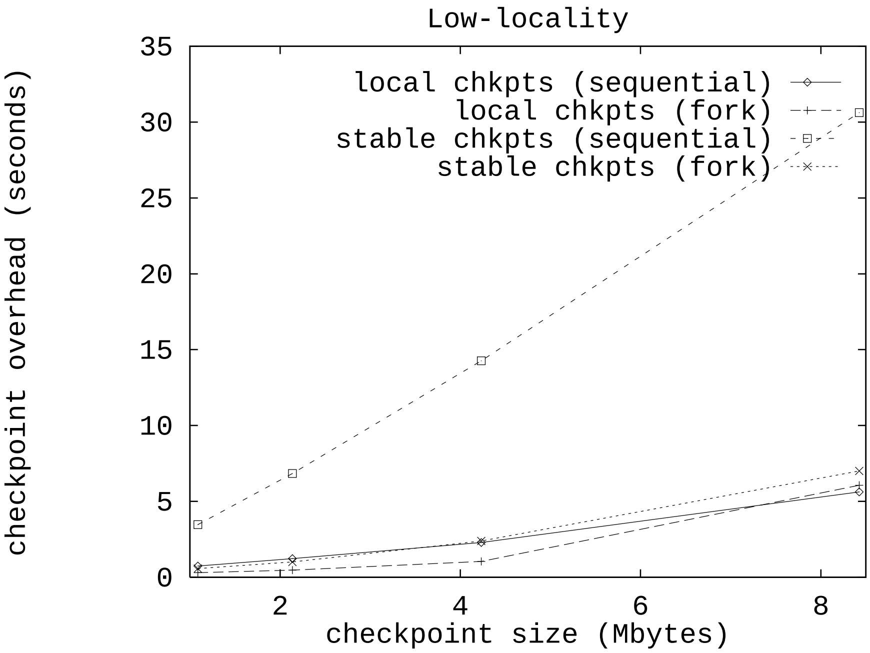

Figure 21: Measurements for application LL (low locality)

It was recently brought to our attention [2] that Gelenbe [8] previously proposed a "multiple checkpointing" approach that is very similar to the "multi-level" approach that we advocate in this report. Gelenbe divides system failures into multiple (n) categories according to their severity. The system takes n types of checkpoints, each type of checkpoint designed for one type of failures. Each type of failure is assumed to be governed by a Poisson process. Although Gelenbe's analysis focuses on transaction-oriented systems, the fundamental idea behind *multiple checkpoints* and *multi-level recovery* is the same – minimize the overhead by designing different approaches for tolerating different types of failures. We characterize a "type" of failure according to the probability of its occurrence, while Gelenbe characterizes a "type" of failure according to how "difficult" it is to recover from the failure. (A failure of type 1 is less "difficult" than a failure of type 2 if a checkpoint for failure type 2 can be used to recover from a type 1 failure [8].) To our knowledge, Gelenbe did not present specific multi-level schemes for distributed systems. His analysis as such may not be applicable to the multi-level schemes of our interest, for two reasons:

- Gelenbe assumes the failures of different types to be governed by Poisson process. This may not be true, in general, even if the failure of each processor is governed by a Poisson process. For instance, this assumption will **not** apply for the two-level scheme presented in Section 2, while it will apply to the scheme presented in Section 4.

- Gelenbe considers transaction-oriented systems. The analysis for a distributed system executing a long-running parallel application may differ (depending on the multi-level full-locality (FL)

chkpt size (bytes)	number of chkpts	chkpt	fork	local/ stable	execution time (milisec)	chkpt overhead (milisec)	chkpt latency (milisec)	latency/ overhead (ratio)
1087404	6	no	–	–	17151	–	–	–
	6	yes	no	local	21625	746	736	0.987
	6	yes	yes	local	18445	216	778	3.6
	6	yes	no	stable	38418	3545	3540	0.999
	3	yes	yes	stable	18581	477	3854	8
2135980	6	no	–	–	33745	–	–	–
	6	yes	no	local	41250	1251	1246	0.996
	6	yes	yes	local	36194	408	1297	3.2
	6	yes	no	stable	76286	7090	7084	0.999
	3	yes	yes	stable	36449	901	7708	8.6
4233132	6	no	–	–	66996	–	–	–
	6	yes	no	local	80712	2286	2283	0.999
	6	yes	yes	local	71814	803	2363	2.9
	6	yes	no	stable	154029	14506	14498	0.999
	3	yes	yes	stable	72336	1780	15659	8.79
8427436	6	no	–	–	133364	–	–	–
	6	yes	no	local	161330	4661	4606	0.988
	6	yes	yes	local	142820	1576	4701	3
	6	yes	no	stable	317538	30696	30689	0.9997
	3	yes	yes	stable	144167	3601	32760	9.1

Table 5: Measurements for application FL (full locality)

scheme under consideration).

Ziv and Bruck [36] present a checkpointing and rollback scheme for *duplex* systems. Although it does not satisfy the above definition of two-level schemes, their scheme also takes two types of checkpoints (similar to the schemes we have proposed). They assume that the duplex system is formed by a pair of workstations connected by a local area network. It is assumed that the state of the two processors in a duplex system must be compared to detect failures (i.e., fail-stop assumption is not made). To compare the checkpoints, the processors must send the checkpoints over a local area network. The overhead of checkpoint comparison, therefore, is high as compared to saving the checkpoints (the checkpoints are saved on the local disk of each workstation). Ziv and Bruck propose a scheme where checkpoint comparison is carried out only at a subset of the checkpoints, thus giving rise to two *types* of checkpoints. Checkpoint comparison may be performed at every k-th checkpoint (for some k). If a failure is detected, then the previous k checkpoints are compared (sequentially) until

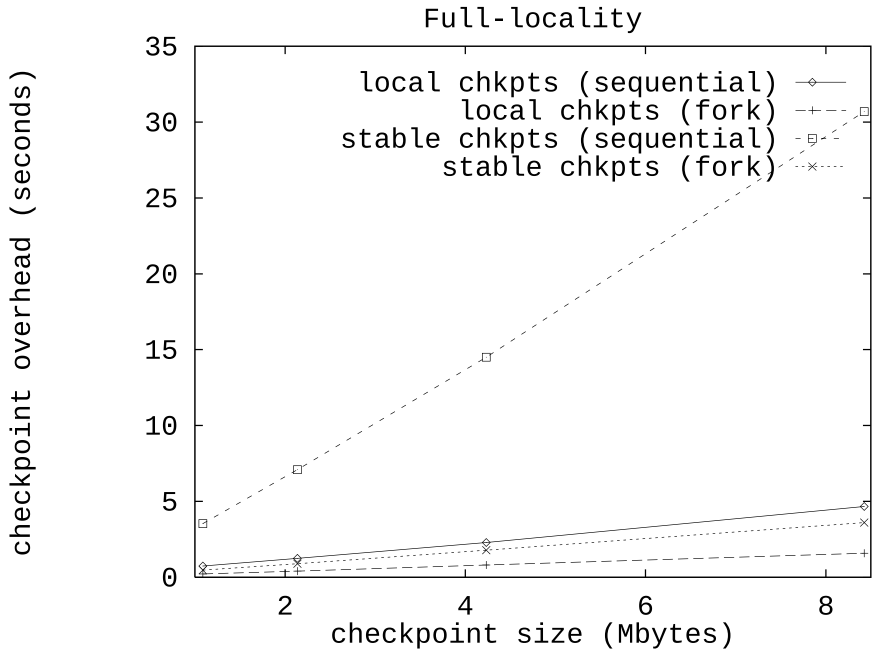

Figure 22: Measurements for application FL (full locality)

an error-free checkpoint is found. The duplex system then rolls back to this checkpoint. By restricting checkpoint comparison (during failure-free operation) to every k-th checkpoint, [36] reduces overhead of the recovery scheme, as compared to a scheme that compares the states at *each* checkpoint.

Our approach differs from [36] in that we attempt to minimize the average overhead by distinguishing between *more probable* and *less probable* failures. [36] improves the overhead (for duplex systems) by decoupling *checkpoint saving* and *checkpoint comparison*. The two approaches are similar, however, in that they both take two types of checkpoints.

We previously proposed a *roll-forward* recovery scheme [21, 29] for duplex systems that tolerates single processor failures with a low overhead, and multiple failures with a high overhead. This is achieved by taking different actions during recovery, depending on the number of failures – the actions taken during the failure-free operation are independent of the number of expected failures. Although this scheme satisfies our definition of *two-level* recovery schemes, in our present research, we are interested in recovery schemes that take *explicit* actions during *failure-free* operation that are designed to minimize the overhead for the *more probable* failures.

12 Conclusions

A "two-level" recovery scheme can tolerate *more probable* failure scenarios with low overhead and the *less probable* failure scenarios with a higher overhead. Most existing recovery schemes are *"one-level"* in the sense that their actions during *failure-free* execution are designed to tolerate the worst case failure scenario. This report presented a *two-level* scheme that can tolerate a *transient* processor failure with low overhead, while permanent processor failures and local storage failures incur a higher overhead. This is achieved using two types of checkpoints – *local* checkpoints and *stable* checkpoints. Local checkpoints are stored on the local storage (e.g., local disk of a workstation), while stable checkpoints are stored on stable storage.

The report presents an analysis to determine the expected completion time of a task using the two-level recovery scheme. The analysis takes into account the fact that, for most implementations, the checkpoint *latency* is larger than the checkpoint *overhead.* (No previous work has taken checkpoint latency into account.) The analytical results indicate that the two-level recovery scheme can achieve a better performance than the traditional *one-level* schemes.

The report also evaluates the impact of checkpoint *latency* on the performance of a recovery scheme. The analysis shows that large checkpoint latency can have a detrimental effect on the performance, particularly with high failure rates. When failure rates are small, increase in the checkpoint latency has a relatively small impact on the performance.

The report presents experimental data on checkpoint overhead and latency of four uni-process applications. Measurement of checkpoint latency for multi-process applications is a subject of ongoing work. Also, study of the relationship between checkpoint latency and the checkpoint overhead, and its impact on performance, is a subject of further research.

Acknowledgements

The checkpointer used in our experimental evaluation is based on a checkpointer written by Bennet Yee and David Applegate of Carnegie Mellon University during 1986-88. The MAT multiplication program is similar to that presented by Plank et al. [19]. The FFT program was provided by Akhilesh Kumar of Texas A&M University. We thank the anonymous referee of one of our papers [2] who pointed us to the related work by Gelenbe [8].

References

[1] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and orphan-free message logging protocols," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, pp. 145–154,

1993.

[2] Anonymous referee's comments on a 1995 SIGMETRICS paper [32], January 1995.

[3] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic models for rollback and recovery strategies in data base systems," *IEEE Trans. Softw. Eng.*, vol. 1, pp. 100–110, March 1975.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 63–75, February 1985.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Symposium on Reliable Distributed Systems*, 1992.

[6] S. Garg and K. F. Wong, "Analysis of an improved distributed checkpointing algorithm," Tech. Rep. WUCS-93-37, Dept. of Comp. Sc., Washington University, June 1993.

[7] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *Comm. ACM*, vol. 21, pp. 493–499, June 1978.

[8] E. Gelenbe, "A model for roll-back recovery with multiple checkpoints," in *2nd Int. Conf. on Software Engineering*, pp. 251–255, October 1976.

[9] E. Gelenbe, "On the optimum checkpointing interval," *J. ACM*, vol. 2, pp. 259–270, April 1979.

[10] V. Grassi, L. Donatiello, and S. Tucci, "On the optimal checkpointing of critical tasks and transaction-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 72–77, January 1992.

[11] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 23–31, January 1987.

[12] V. G. Kulkarni, V. F. Nicola, and K. S. Trivedi, "Effects of checkpointing and queueing on program performance," *Commun. Statist.-Stochastic Models*, vol. 4, no. 6, pp. 615–648, 1990.

[13] P. L'Ecuyer and J. Malenfant, "Computing optimal checkpointing strategies for rollback and recovery systems," *IEEE Trans. Computers*, vol. 37, pp. 491–496, April 1988.

[14] J. León, A. L. Fisher, and P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery," Tech. Rep. CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, February 1993.

[15] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Par. Distr. Syst.*, vol. 5, pp. 874–879, August 1994.

[16] D. Long, J. Carroll, and C. Park, "A study of the reliability of internet sites," in *Proc. Symp. Rel. Distr. Systems*, pp. 177–186, 1991.

[17] V. F. Nicola and J. M. van Spanje, "Comparative analysis of different models of checkpointing and recovery," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 807–821, August 1990.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, 1994.

[19] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter 1995 Technical Conference, New Orleans*, January 1995.

[20] J. S. Plank, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Dept. of Computer Science, Princeton University, June 1993.

[21] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Trans. Computers*, vol. 43, pp. 1163–1174, October 1994.

[22] A. Reuter, "Performance analysis of recovery techniques," *ACM Trans. Database Systems*, vol. 9, pp. 526–559, December 1984.

[23] C. Schimmel, *UNIX Systems for Modern Architectures.* Addison-Wesley, 1994.

[24] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comp. Syst.*, vol. 1, pp. 222–238, August 1983.

[25] K. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal checkpointing of real-time tasks," *IEEE Trans. Computers*, vol. 36, pp. 1328–1341, November 1987.

[26] A. N. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Trans. Comp. Syst.*, vol. 2, pp. 123–144, May 1984.

[27] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Prentice-Hall, 1988.

[28] S. J. Upadhyaya and K. K. Saluja, "An experimental study to determine task size for rollback recovery schemes," *IEEE Trans. Computers*, vol. 37, pp. 872–877, July 1988.

[29] N. H. Vaidya, *Low-Cost Schemes for Fault Tolerance.* PhD thesis, University of Massachusetts-Amherst, February 1993. Available from UMI Dissertation Services, Ann Arbor, Michigan. Order number 9316722.

[30] N. H. Vaidya, "A case for multi-level distributed recovery schemes," Tech. Rep. 94-043, Computer Science Department, Texas A&M University, College Station, May 1994. Available via anonymous ftp from ftp.cs.tamu.edu in directory /pub/vaidya.

[31] N. H. Vaidya, "Some thoughts on distributed recovery," Tech. Rep. 94-044, Computer Science Department, Texas A&M University, College Station, June 1994. Available via anonymous ftp from ftp.cs.tamu.edu in directory /pub/vaidya.

[32] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *SIGMETRICS/Performance*, May 1995.

[33] K. Wong and M. Franklin, "Distributed computing systems and checkpointing," in *Proc. 2nd Int. Symp. High Perf. Distr. Comp., Spokane, Washington*, pp. 224–233, July 1993.

[34] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. ACM*, vol. 17, pp. 530–531, September 1974.

[35] A. Ziv and J. Bruck, "Analysis of checkpointing schemes for multiprocessor systems," Tech. Rep. RJ 9593, IBM Almaden Research Center, November 1993.

[36] A. Ziv and J. Bruck, "Efficient checkpointing over local area network," in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, College Station*, June 1994.