An Analysis of Communication-Induced Checkpointing

Lorenzo Alvisi[†] *Elmootazbellah Elnozahy*[*] *Sriram Rao*[†] *Syed Amir Husain*[†] *Asanka Del Mel*[†]

Abstract

Communication induced checkpointing (CIC) is a style of rollback-recovery which allows processes in a distributed computation to take independent checkpoints without susceptibility to the domino effect. This style of recovery has been subject to an increasing interest lately, but most of the work done is algorithmic in nature. This paper presents an analysis of CIC protocols based on a prototype implementation and validated simulations. For the applications we studied, our results show that:

1. CIC protocols do not scale well as the number of processes increases.
2. The occurrence of forced checkpoints at random points within the execution makes it very difficult to predict the required amount of stable storage for a particular application run. Also, this unpredictability affects the policy for placing local checkpoints and makes CIC protocols cumbersome to use in reality.
3. The benefit of autonomy in allowing processes to take local checkpoints at their convenience does not seem to hold. In all experiments, a process takes at least twice as many forced checkpoints as local, autonomous ones.
4. Within the class of CIC protocols, those that use time-stamping functions to delay taking a forced checkpoint to the extent possible seem to work better than protocols that inspect the communication and checkpoint patterns and prevent Z-cycles from forming by taking forced checkpoints as soon as possible.
5. A successful implementation of CIC requires a dynamic checkpoint placement policy for local checkpoints that adapts to the occurrence of forced checkpoints and the variations in the application communication patterns. Simple or static policies such as taking checkpoints at regular intervals do not seem to work well.
6. CIC protocols in general seem to work better for situations where the communication load is low and the communication pattern is random. Regular, or heavy-load communications make these protocols trigger too many forced checkpoints.

We would like to stress that the results are only valid for the application set that we have studied, and we lay no claim that these results would generalize to all applications. Nevertheless, we believe that there is sufficient evidence to suspect that much of the conventional wisdom about these protocols is questionable.

Citation: Technical Report TR-99-01, Department of Computer Sciences, Univ. of Texas at Austin

[†] Department of Computer Sciences, The University of Texas at Austin.
[*] IBM Austin Research Lab.

1. Introduction

There are three styles for implementing application-transparent rollback-recovery in message-passing systems, namely coordinated checkpointing, message logging, and communication-induced checkpointing (CIC) [5]. Both coordinated checkpointing and message logging have received considerable analysis in the literature [4,6,11,14,17,18,20,21,24,28], but little is known about the behavior of CIC protocols. This paper presents an experimental analysis of these protocols through a prototype implementation and reveals several of their theoretical and pragmatic characteristics.

CIC protocols are not new. The paper by Briatico et al. was perhaps the first to describe this style of checkpointing back in 1984 [2]. Other papers have described variations over their protocol [9,13,26], or used some protocol features to simplify the implementation of coordinated checkpointing [6,25]. Recently, CIC protocols have attracted an increasing interest in the research community, with new sophisticated protocols based on the Z-path theory [16]. But to our knowledge, there are no published implementation or evaluation reports of CIC.

CIC protocols piggyback special information on application messages to manage the failure-free operation of the processes in a distributed computation. This information allows the processes to take independent checkpoints without exchanging explicit control messages to ensure that their checkpoints form consistent states [3]. Furthermore, this information protects the processes from the domino effect [22] should one or more of them fail. CIC protocols are therefore believed to have several advantages over other styles of rollback-recovery. For instance, they allow processes considerable autonomy in deciding on when to take checkpoints. A process can thus take a checkpoint at times when saving the state would incur a small overhead [12,19]. CIC protocols also are believed to scale up well with a larger number of processes since they do not require the processes to participate in a global checkpoint. But there is a price to pay for these advantages. First, the protocol-specific information piggybacked on application messages occasionally "induce" the processes to take *forced* checkpoints before they can process the messages. Second, processes have to pay the overhead of piggybacking information on top of application messages, and they also need to keep several checkpoints on stable storage. These advantages and disadvantages are clearly qualitative and potentially arguable. A purpose of our work is to shed some light on these issues using quantitative metrics drawn from a real system.

To this end, we have implemented three CIC protocols and studied their behavior. These protocols are the original one by Briatico et al [2], and two recent protocols based on the Z-cycle framework [1,8]. These two protocols differ in the theoretical approach that they adopt to implement CIC. Generally, one can derive a CIC protocol by designing a function that associates a timestamp with each checkpoint, such that the timestamps increase in a manner consistent with the causal order of the checkpoints [10]. Equivalently, one can derive a CIC protocol by designing a way to prevent the occurrence of Z-cycles. Hélary et al. have proved that the two methods are equivalent [7]. That is, for every algorithm to prevent Z-cycles there is an equivalent time stamping function, and vice versa. Again, it is not clear what are the tradeoffs involved in choosing either approach. We have therefore selected a representative from each style: the protocol by Hélary et al uses a time stamping function [8], while the protocol by Baldoni et al uses an algorithm to predict and prevent the formation of Z-cycles [1].

We examine the performance of the three protocols using two metrics, namely the average number of forced checkpoints a protocol causes, in addition to the traditional running time performance. The first metric is important because forced checkpoints negate the autonomy advantage of CIC protocols. Also, they contribute substantially to the performance and resource overheads. A good CIC protocol therefore tries to limit these forced checkpoints to the extent possible. The experiments use four compute-intensive programs from the NPB 2.3 benchmark suite [15], which is a representative of a class of applications that have traditionally been the primary users of checkpointing protocols. We then use the implementation in part to validate a simulation model that we built to further study the scalability of the protocols and their behaviors under different communication patterns.

Our results reveal several important properties of CIC protocols and highlight several implementation and theoretical issues that were not addressed in the literature. But we would like to state unequivocally that while our study reports interesting results, by no means we are trying to make any sweeping conclusions based on it. We hope that our work will be a first step in investigating an area that thus far has been only subject to theoretical research.

The organization of this paper has a brief coverage of the three protocols in Section 2, a description of implementation issues in Section 3, and the performance study in Section 4. Section 5 contains a brief comparison to similar work and Section 6 concludes the paper.

2. Background

This section reviews the three protocols used in the experiments, along with some necessary definitions. The description is inevitably terse and covers only the features necessary to follow the experimental work described later. The interested reader may wish to consult the references for full details. Readers familiar with the Z-path theory and its use in CIC protocols may wish to move directly to Section 3.

2.1 Definitions

Local checkpoints: A process may take a *local* checkpoint any time during the execution. The local checkpoints of different processes are *not* coordinated to form a global consistent checkpoint [3].

Forced checkpoints: To guard against the domino effect, a CIC protocol piggybacks protocol-specific information to application messages that processes exchange. Each process examines the information, and occasionally, is forced to take a checkpoint according to the protocol.

Useless checkpoints: A useless checkpoint of a process is one that will never be part of a global consistent state [27]. In Figure 1, checkpoint $C_{2,2}$ is an example of a useless checkpoint. Useless checkpoints are not desirable because they do not contribute to the recovery of the system from failures, while consume resources and cause performance overhead.

Checkpoint intervals: A checkpoint interval is the sequence of events between two consecutive checkpoints in the execution of a process.

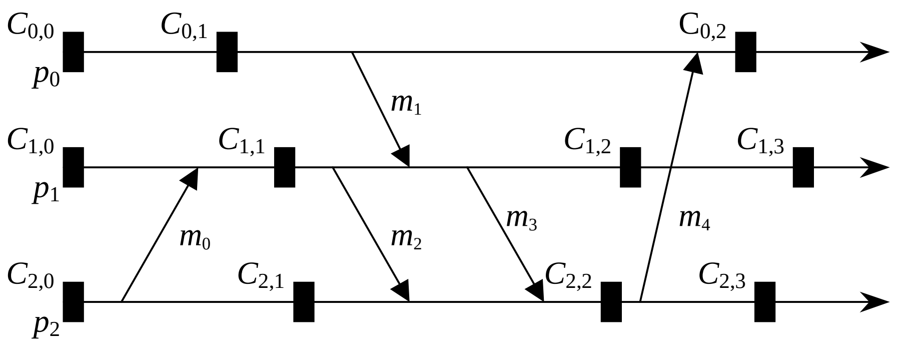

Figure 1: A distributed computation. $C_{i,j}$ denotes the j^{th} checkpoint of process p_i.

2.2 Z-paths and Z-cycles

Z-paths: A Z-path (zigzag path) is a special sequence of messages that connects two checkpoints. Let ξ denote Lamport's happen-before relation [10]. Given two local checkpoints $C_{i,m}$ and $C_{j,n}$, a Z-path exists between $C_{i,m}$ and $C_{j,n}$ if and only if one of the following two conditions holds:

1. $m < n$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_z]$, $z \mu 0$, such that:
   (a) $C_{i,m} \xi send_i(m_0)$;
   (b) … $l < z$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \xi send_k(m_{l+1})$; and
   (c) $deliver_j(m_z) \xi C_{j,n}$

where $send_i$ and $deliver_i$ are communication events executed by process p_i. In Figure 1, $[m_1, m_2]$ and $[m_1, m_3]$ are examples of Z-paths.

Z-cycles: A *Z-cycle* is a Z-path that begins and ends with the same checkpoint. In Figure 1, the Z-path $[m_4, m_1, m_3]$ is a Z-cycle that involves checkpoint $C_{2,2}$.

2.3 Z-cycles and CIC

CIC protocols do not take useless checkpoints. These protocols recognize that the creation of useless checkpoints depends on the occurrence of specific patterns in which processes communicate and take checkpoints [7]. Informally, these protocols recognize potentially dangerous patterns, and they break them before they occur. This intuition has been formalized in an elegant theory based on the notion Z-cycles. A key result in this theory is that in order for a local checkpoint to be useless it has to be involved in a Z-cycle [7,16]. Hence, to avoid useless checkpoints it is sufficient to guarantee that no Z-path ever becomes a Z-cycle. Enforcing the no-Z-cycle (*NZC*) condition may require that a process save additional forced checkpoints in addition to the local checkpoints taken on its own initiative.

There are two approaches to avoiding Z-cycles. The first approach uses a function that associates a timestamp with each checkpoint. The protocol guarantees, through forced checkpoints if necessary, that (*i*) if there are two checkpoints $C_{i,m}$ and $C_{j,n}$ such that $C_{i,m} \xi C_{j,n}$ then $ts(C_{j,n}) \geq ts(C_{i,m})$, where $ts(C)$ is the timestamp associated with checkpoint C, and (*ii*) consecutive local checkpoints of a process have increasing timestamps.

The second approach relies instead on preventing the formation of specific checkpoint and communication patterns that may lead to the creation of a Z-cycle. Protocols that follow this approach do not adopt a specific function for associating timestamps with checkpoints. However, for protocols that follow this approach, there always exists an equivalent time stamping function that would cause the same forced checkpoints [7].

Of the three protocols that we study in this paper, the first two adopt the first approach, while the third is an instance of the second approach. We briefly review them below.

2.4 The Protocol by Briatico, Ciuffoletti and Simoncini (BCS)

In BCS, each process p_i maintains a logical clock lc_i that functions as p_i's checkpoint timestamp. The timestamp is an integer variable with initial value 0 and is incremented according to the following function:

1. lc_i increases by 1 whenever p_i takes a local checkpoint.
2. p_i piggybacks on every message *m* it sends a copy of the current value of lc_i. We denote the piggybacked value as *m.lc*.
3. Whenever p_i receives a message *m*, it compares lc_i with *m.lc*. If $m.lc > lc_i$, then p_i sets lc_i to the value of *m.lc* and takes a forced checkpoint before it can process the message.

The set of checkpoints having the same timestamps in different processes is guaranteed to be a consistent state. Therefore, this protocol guarantees that there is always a recovery line corresponding to the lowest timestamp in the system, and the domino effect cannot happen.

2.5 The Protocol by Hélary, Mostefaoui, Netzer and Raynal (HMNR)

The HMNR protocol uses the observation that if checkpoints' timestamps always increase along a Z-path (as opposed as simply non-decreasing, as required by rule (*i*) above), then no Z-cycle can ever form. It is thus possible to design functions that take advantage of this observation. Hélary et al start with the following simple scheme which would require each process to maintain a logical clock, as in BCS, and to apply the following rules:

1. lc_i increases by 1 whenever p_i takes a local *or forced* checkpoint.
2. Whenever p_i sends a message *m*, it piggybacks on *m* a copy of lc_i, and we denote this value by *m.lc* as before.
3. Whenever p_i receives a message *m*, it compares lc_i with *m.lc*. If $m.lc > lc_i$, then p_i sets lc_i to the value of *m.lc*.

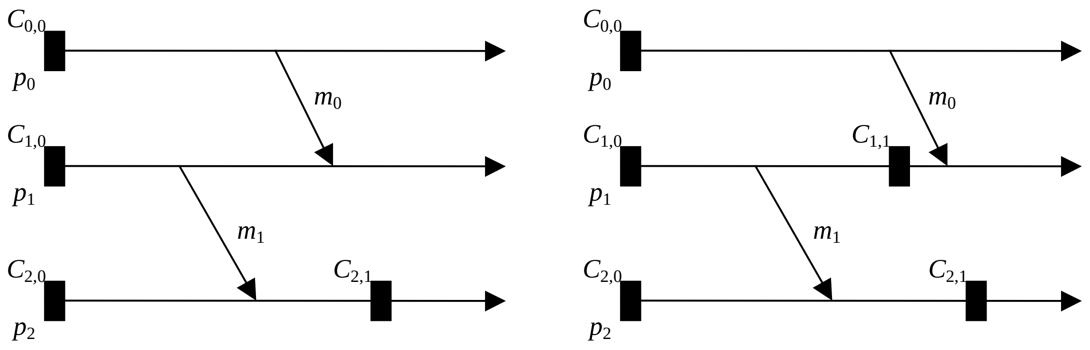

Figure 2: To checkpoint or not to checkpoint?

Then, they refine the protocol using more sophisticated observations and requiring that processes append more information. Figure 2, adapted from [8], shows how this simple timestamp function can be used to decide when to take a checkpoint. Let ts(m) denote the timestamp piggybacked on message m. When process p_1 receives message m_0, if $ts(m_0) \leq ts(m_1)$ then certainly $ts(C_{0,0}) \leq ts(C_{2,1})$ and there is no need for a forced checkpoint. If $ts(m_0) > ts(m_1)$, however, delivering m_1 may create the possibility of generating a Z-path along which the timestamps do not increase. A straightforward way to avoid this risk is to force p_1 to take a checkpoint before delivering m_0, thereby breaking the Z-path.

It may be possible, however, for p_1 to avoid taking a forced checkpoint by using a more sophisticated timestamp function. For instance, a function that piggybacks on application messages information about the logical clocks of all processes may give process p_1 more information to decide if a forced checkpoint is really necessary. In Figure 2 (b), if p_1 knows that the value of p_2's local clock is at least x when it is about to deliver m_0, then even if $ts(m_0) > ts(m_1)$, p_1 does not need to take a forced checkpoint if $ts(C_{0,0}) \leq ts(m_1) \leq x < ts(C_{2,1})$.

HNMR uses this observation and more sophisticated ones to reduce the number of forced checkpoints while still ensuring that the timestamps always increase along a Z-path [8]. They in fact present several CIC algorithms. For our experiments, we have used the most sophisticated one, which reduces as much as possible the number of forced checkpoints.

2.6 The Protocol by Baldoni, Quaglia and Ciciani (BQC)

The BQC protocol does not prevent forced checkpoints by using an explicit function to time stamp checkpoints [1]. Rather, this protocol enforces $\mathcal{NZC}$ by preventing the formation of patterns of checkpoints and communication that *may* result in the creation of a Z-cycle. In particular, BQC prevents the creation of *suspected* Z-cycles.

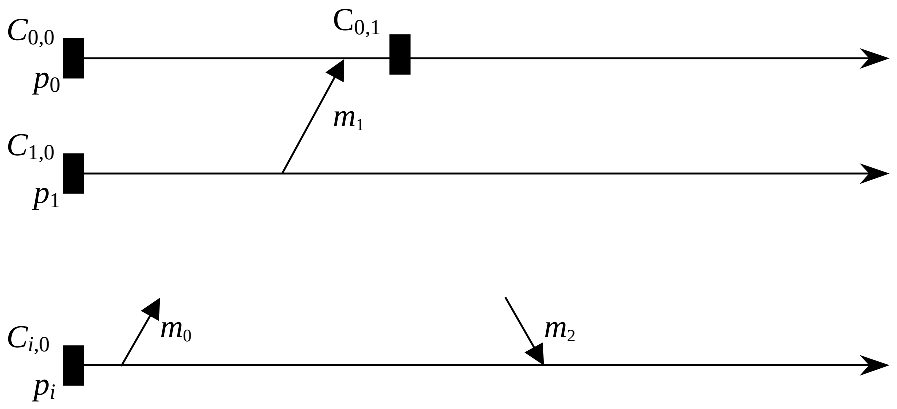

Figure 3: A suspect Z-cycle involving checkpoint $C_{0,1}$

Figure 3 shows the structure of a suspected Z-cycle. In this example, process p_i would take a checkpoint before delivering message m_2, in order to avoid a potential Z-cycle that includes m_0 and m_1 and involves checkpoint $C_{0,1}$. Note that a suspected Z-cycle is not necessarily a Z-cycle. For instance, there is no Z-cycle in Figure 3 unless there is an actual Z-path starting with m_0 and ending with m_1 between $C_{i,0}$ and $C_{0,1}$. This information may or may not be available to process p_i when it receives m_2. If the information is not available, the protocol opts for safety and takes a forced checkpoint before delivering m_2. If the information is available, however, the protocol refrains from taking a forced checkpoint. The actual protocol propagates n^2 values on each application message to help processes detect suspected Z-cycles and suppress them using forced checkpoints [1]. The reference also contains a formal characterization of the notion of suspected Z-cycle and a proof that a protocol that prevents suspected Z-cycle also satisfies (NZC) [1].

3. Implementation Issues

An implementation of CIC must deal with several pragmatic issues that are typically left out of protocol specifications. We describe how we resolved these issues in our implementation.

3.1 Autonomy in Local Checkpoints

A stated advantages of CIC protocols is that they prevent the domino effect while allowing processes considerable autonomy in deciding when to take local checkpoints. An efficient implementation of CIC must therefore adopt a checkpointing policy that exploits this autonomy and translates it into a benefit. In general, this requires a good understanding of the application and of the execution environment. For example, detailed knowledge of the application may allow processes to take checkpoints when the size of the live variables is small [12,19].

In our study, we have chosen a set of compute-intensive, long-running applications that have traditionally been the main beneficiary of checkpointing protocols. We have found that either the complexity of the application program precludes investing the effort in defining a reasonable checkpoint placement policy, or that the application structure does not reveal points within the execution where taking a checkpoint is more advantageous than others. Therefore, we have resolved to use a probabilistic distribution to emulate what an autonomous application would do in deciding on where to place the local checkpoints. In addition, we have also used the traditional policy of taking checkpoints at regular intervals. The results were identical in both cases, and in fact the decision of how to take local checkpoints turned out to have no effect on the results of our study. Indeed, one of the first conclusions that we reached in our implementation is that even with a deep understanding of the applications' structure, no policy for taking local checkpoints can reasonably be adopted without considering the effect of the forced checkpoints that occur because of communication events. For example, a pre-run analysis may decide on the execution points during which it is most "convenient" to schedule local checkpoints within a particular process. But if this policy ignores the forced checkpoints, then it is possible to schedule a local checkpoint immediately after a forced checkpoint. In this case, taking the local checkpoint will result in additional work and overhead, with no substantial reduction of the amount of work at risk. A more reasonable decision may then be to postpone taking the local checkpoint. In any case, the autonomy of deciding when to take checkpoints seems to be limited by the occurrence of forced checkpoints due to interactions with other processes.

3.2 Non-blocking Checkpointing in CIC

The benefits of non-blocking checkpointing in reducing the performance overhead of checkpointing protocols have been clearly established [6,14,20]. Non-blocking checkpointing allows the application to resume computation as soon as possible and to schedule the actual writing of the checkpoint concurrently with the application execution. The result is that saving the checkpoint to stable storage does not become a bottleneck that impedes application progress. However, using non-blocking checkpointing in CIC introduces potential inconsistencies with the specification of the protocols. To understand why, consider the situation where a process p_i takes a non-blocking checkpoint (local or forced) and then sends a message *m* to another process p_j; assume furthermore that the CIC protocol being used requires p_j to take a forced checkpoint before delivering *m*. Recall that all CIC protocols maintain the invariant that no Z-cycles may form and no useless checkpoint is ever taken: hence, the forced checkpoint of p_j should never be useless and no Z-cycles can include it. However, suppose that p_i fails after sending *m*, but before p_i's checkpoint has been entirely written to stable storage. The failure of p_i makes the checkpoint taken by p_j useless, thereby violating the protocol invariant.

The problem is not just cosmetic, because several such communication events may occur while several non-blocking checkpoints are being written to stable storage according to an implementation of non-blocking checkpointing. Therefore, it may be possible for many checkpoints on stable storage to be rendered useless because a failure of some process in the system occurred before one or more of its checkpoints were saved on stable storage. Indeed, in situations like this, Z-cycles do form and useless checkpoints are taken.

There are two solutions to this problem. The first one blocks any outgoing messages from a process until all its non-blocking checkpoints have been written to stable storage. The messages are buffered and released only when a process receives notification from the checkpointing agent that the checkpoint has been saved. There is a penalty to pay for this modification, but it is preferable to disallowing non-blocking checkpointing altogether. It is interesting to note here that this solution shows how a pragmatic consideration may require a modification in the protocol itself to maintain its invariant.

The second solution that we considered is to simply allow these temporary Z-cycles to form, and hope that the checkpoints will be written before a failure actually occurs. In a sense, this is an optimistic implementation of CIC, which may allow Z-cycles to form temporarily while some checkpoints are being written to stable storage. If the optimistic assumption holds, then the invariant of the protocol is preserved and no useless checkpoints are ever taken. However, if the assumption is violated because of a failure, then some of the checkpoints that would have otherwise be part of the recovery line will have to be discarded. The benefit of this optimistic alternative is that the overhead is small and does not require any modification to the protocol as specified. After an interesting debate among the authors, it was resolved to use the second solution for its simplicity and because failures are supposedly rare. It is important however for future implementers to understand the subtle issues involved with the pragmatic choices described here and how they may affect the protocol implementation.

4. Experiments and Analysis

The testbed for this study consists of four 300-MHz Pentium-II based workstations connected by a 100MB/s Ethernet. Each workstation has two processors, 512MB of RAM, and a 4GB disk used to implement stable storage. The machines ran Solaris 2.6, and used Sun's f77 and C compilers. The testbed is part of the Egida tool [23], which includes support for incremental checkpointing and implements non-blocking checkpointing by forking off a child process that writes the checkpoint to stable storage. The applications under study consist of four programs from the NPB 2.3 benchmark suite [15]. These programs represent common computational loads in fluid dynamics applications and typify the type of applications that have traditionally benefited from checkpointing; their characteristics are given in Table 1.

Application	NPB Specific Info	Per-Process Avg. Message Rate		Exec. Time
		Messages/sec	KB/sec	(sec)
BT	Class A, 200 iterations	7	352.3	1554
CG	Class B, 75 iterations	15	752.0	1508
LU	Class A, 250 iterations	64	233.9	1018
SP	Class A, 400 iterations	17	738.8	1350

Table 1: Characteristics of the benchmarks used in the experiments.

The performance metrics we report are the number of forced checkpoints that a protocol causes and the performance overhead. We use a combination of experiments on the prototype implementation, and then we use the prototype itself to validate a simulator that we built to further study CIC protocol under different communication patterns and environments.

4.1 The Measured Performance of CIC Protocols

The first set of experiments consists of running each of the four applications under the three protocols and for two local checkpoint placement policies. The first policy triggers local checkpoints according to an exponential distribution with a mean checkpoint interval (μ) set to 360 seconds, while the second policy uses the same probabilistic distribution but with the mean checkpoint interval set to 480 seconds. Table 2 shows the results of the experiments. It reports the execution time of the entire application, in addition to the per-process average number of local and forced checkpoints. For convenience, the average per-process total number of checkpoints is also reported. The table also reports the per-process average checkpoint size (either local or forced).

Application	Protocol	μ	Number of Checkpoints			Avg. Ckp. Size (MB)	Exec. Time (sec.)
			Local	Forced	Total		
BT	BCS	360	6	15	21	69.1	2181
		480	4	10	14	70.2	2203
	HMNR	360	6	14	20	69.2	2190
		480	4	9	13	70.2	2187
	BQC	360	6	63	70	39.2	2400
		480	4	41	45	42.4	2298
CG	BCS	360	5	11	16	14.1	1667
		480	4	9	13	17.5	1672
	HMNR	360	5	12	17	13.7	1634
		480	4	9	13	17.5	1613
	BQC	360	5	24	29	8.6	1752
		480	4	19	23	10.7	1720
LU	BCS	360	4	10	14	11.0	1150
		480	3	6	9	11.2	1161
	HMNR	360	4	10	14	10.8	1147
		480	3	6	9	11.2	1175
	BQC	360	4	34	38	5.7	1284
		480	3	17	20	6.9	1253
SP	BCS	360	5	11	16	21.0	1572
		480	4	9	13	21.5	1583
	HMNR	360	5	11	16	20.8	1574
		480	4	9	13	21.5	1585
	BQC	360	5	43	48	11.6	1615
		480	4	33	37	11.0	1601

Table 2: Performance of three CIC protocols for two checkpoint intervals and four applications.

Analysis: The results reveal a few issues. In BCS and HMNR, the number of forced checkpoints is essentially the same. In contrast, the BQC protocol is showing a comparatively larger number of forced checkpoints when compared to the other two. To understand the reason for this, we examined the protocol behavior of BQC to see what is the cause of this anomaly. The reason can be ascribed to the communication pattern of the applications under study. These applications use a common iterative structure to solve a computationally intensive problem in which processes exchange partial results and resume. This leads to a communication pattern that mimics a periodic broadcast. Under this pattern, the BQC protocol seems to be too "eager" in preventing Z-cycles compared to BCS and HMNR, and also it seems that two situations occur:

1. Many suspected Z-cycles end up causing forced checkpoints without actually being a menace.
2. It is often the case that more than one process "volunteer" *in parallel* to break the same suspected Z-cycle by forcing checkpoints.

Consider Figure 4. In this example, we see process p_2 take a forced checkpoint because of message $m_{3,}$ not knowing that process p_1 has already broken the suspected Z-cycle (part (2) of the figure). Similarly, process p_0 takes a forced checkpoint because of message m_2, not knowing that process p_1 has already broken the Z-cycle using checkpoint $C_{1,1}$. The pattern continues for a while under the communication pattern used by the application.

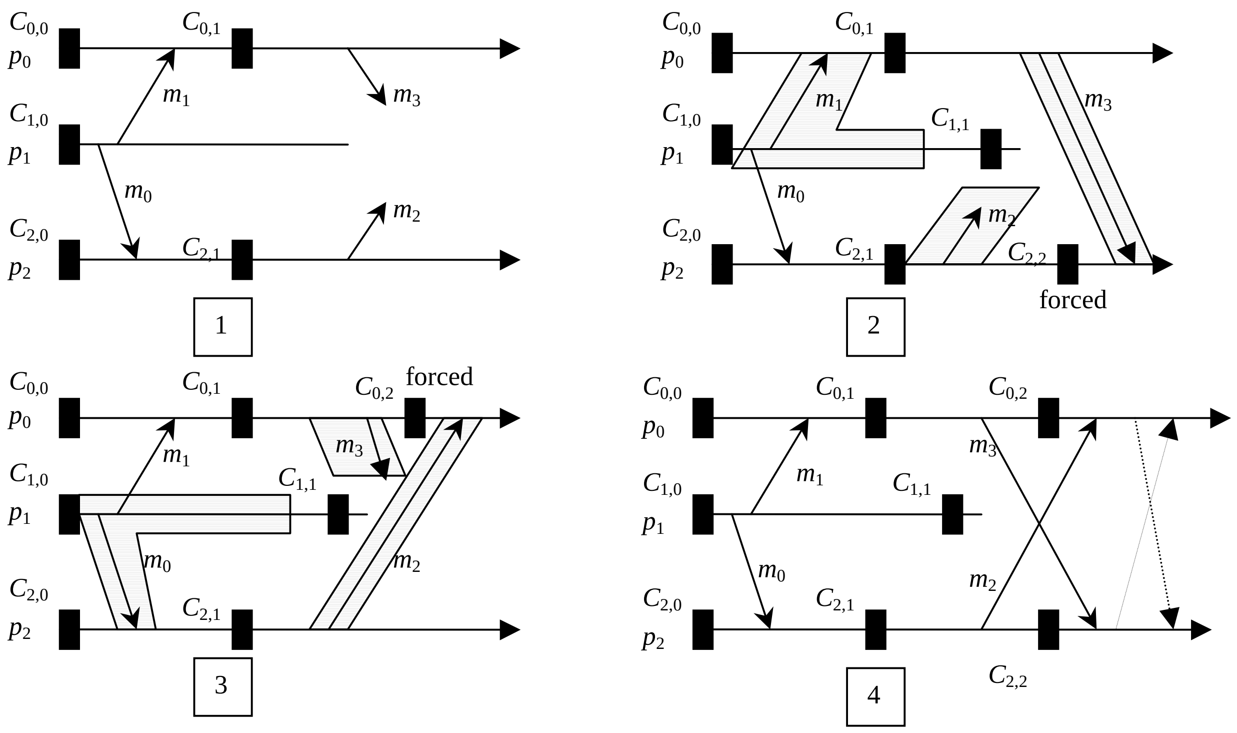

Figure 4: Anomalies in detecting suspect Z-cycles

This result suggests that there is a disadvantage to using CIC protocols that use suspected Z-cycles, or ones that are eager to prevent a Z-cycle from forming before it is actually clear that one is indeed forming. In contrast, protocols that use the time stamping functions seem to adopt a lazy approach, preventing a Z-cycle from forming at the last moment that is possible, and therefore work better.

Additionally, the results show that the stated benefit of process autonomy in placing local checkpoints does not materialize in practice. Under the best circumstances, a process takes as twice as many forced checkpoints as local ones. The curious notion of process autonomy in distributed systems where all processes become inter-dependent seems to be on shaky ground.

The results also point out to another serious problem with CIC protocols in general, which is unpredictability of the checkpointing rate. In all experiments, the protocols ended up taking more checkpoints than could be anticipated based on the local distribution of checkpoint placement. For BCS and HMNR, the number of forced checkpoints was generally twice as many as the number of local ones. For BQC, the ratio was worse. The ratio in itself is a function of the application, the number of processes, and the checkpoint placement. The fact that it is unpredictable makes the protocols cumbersome to use in practice, because it is difficult to plan ahead of time the actual stable storage requirements and the mean checkpointing interval. Contrast this with consistent checkpointing protocols where the number of checkpoints and required stable storage can be estimated with great certainty beforehand [6].

The table also points to another negative aspect of using CIC protocols. The performance overhead when considering the running time was relatively bad, reaching between 5 to 40% of the execution time. This anomaly is actually common in systems where the checkpoints are not coordinated and the processes communicate frequently [6]. In these situation, when a process takes a local checkpoint independent of the others, it inevitably slows down due to the state saving and memory copying that occur during the checkpoint. This in turn delays the production of the expected partial result that the process will send to others in the next communication round. As a result, the slowdown affects other processes as well even if they are not taking checkpoints in the meantime. The resulting slowdowns stagger quickly and have a cumulative effects because of having many of these independent checkpoints occurring at different times [6].

Finally, we would like to note that incremental checkpointing seems to mitigate some of the effects of having to take so many checkpoints (forced or local). The results show that the average per-process checkpoint size goes down as the frequency of checkpointing increases. This is an expected result.

In summary, lazy protocols for breaking Z-cycles based on time-stamping are shown to perform better than eager protocols that take forced checkpoints as soon as they suspect a Z-cycle. The unpredictability of the actual number of checkpoints to be taken (forced and local) make these protocols cumbersome to use in practice because no reasonable planning of resources and checkpointing frequency can be made without understanding the application and its communication patterns. Also, it seems that any notion of a benefit of allowing the processes to take independent checkpoints is thwarted by the fact that a process ends up taking at least twice as many forced checkpoints than local ones. And finally, CIC protocols share some of the negative performance properties of independent checkpointing when used in computations where the processes are tightly coupled and communicate frequently.

4.2 Scalability and Effects of Communication Patterns

To measure the effect of increasing the number of processes on the protocol performance, we constructed a simulator to measure the number of forced checkpoints for each of the three protocols. We validated the simulator using the measured number of forced checkpoints for 4 processes. We then used the simulator to estimate the number of forced checkpoints under different numbers of processors and different communication patterns.

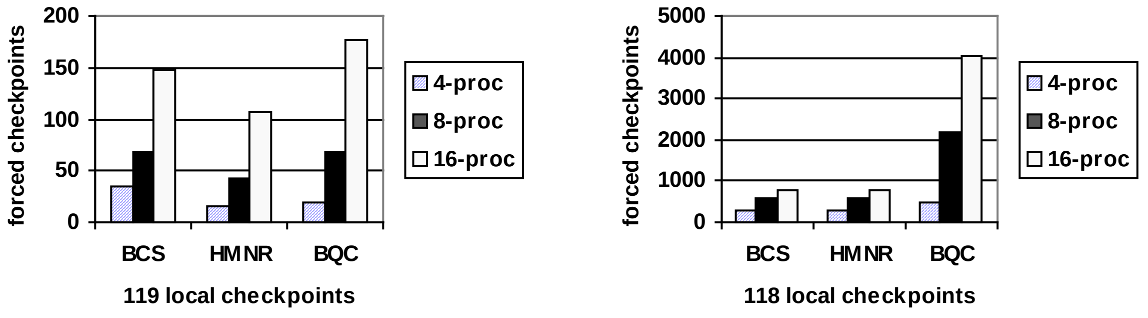

Figure 5: Effect of communication patterns and number of processors on CIC protocols.

Figure 5 shows the results for the three protocols with varying the number of processes in the computation. Two different sets of measurements are reported, the first is for a random distribution of messages with a relatively low load, where each process sends an average of 10 messages between each two consecutive local checkpoints. That is, processes do not communicate much in this simulation and communicate with different processes equally at random and at random intervals. During that simulation, 119 local checkpoints were taken on average. The second set of measurements show the same results but with a different communication pattern in which each process talks to two designated neighbors at uniform intervals. A process sends about 500 messages between each two consecutive local checkpoints. This communication pattern is representative of those that occur in distributed over-relaxation algorithms.

Analysis: The results show that in general, the communication pattern strongly affects the behavior of the CIC protocols. This is expected. But the results also show that CIC protocols do not scale very well. In both cases, there is an almost linear increase in the number of forced checkpoints per process as the number of processes increase. For this set of applications, at least, it is clear that the conventional wisdom that these protocols scale better because they do not resort to global coordination is not true. The results also show that CIC protocols seem to favor random patterns of communications with low loads.

4.3 An Adaptive Local Checkpointing Policy

The results of the experiments so far suggest that a flurry of forced checkpoints occur throughout the system as a result of one process taking a local checkpoint. It is plausible that if forced checkpoints are not taken into account, a local checkpointing policy may take a local checkpoint shortly after a forced checkpoint has been taken. Such a local checkpoint advances the recovery point of the process by a very short amount compared to the previous forced checkpoint. Furthermore, this local checkpoint will likely trigger more forced checkpoints in other processes, escalating the phenomenon even further. It may be argued that the resulting overhead can be limited by using incremental checkpointing, and therefore the local checkpoint will not have to save a lot of state on stable storage if a forced checkpoint has been taken recently. But we contend that taking a checkpoint, however small, always has an overhead associated with it, if only to compute the state that must be saved and arranging for the copy-on-write to implement non-blocking checkpointing. However it may be, the overhead cannot be ignored. Therefore, there is very little to gain by taking this local checkpoint, while there is a potential for larger overhead.

To fix this problem, we experimented with an adaptive local checkpointing policy that refrains from taking a scheduled local checkpoint if a forced checkpoint has occurred during the last T seconds, where T is a tunable parameter. Figure 6 shows the resulting number of local and forced checkpoints for the four applications and the three protocols under study. We report three measurements, one with the adaptive policy disabled (T = 0), and two for different values of T (60 and 90 seconds). The table shows for each T, the number of local and forced checkpoints under each of the three protocols. The measurements for different applications are reported separately.

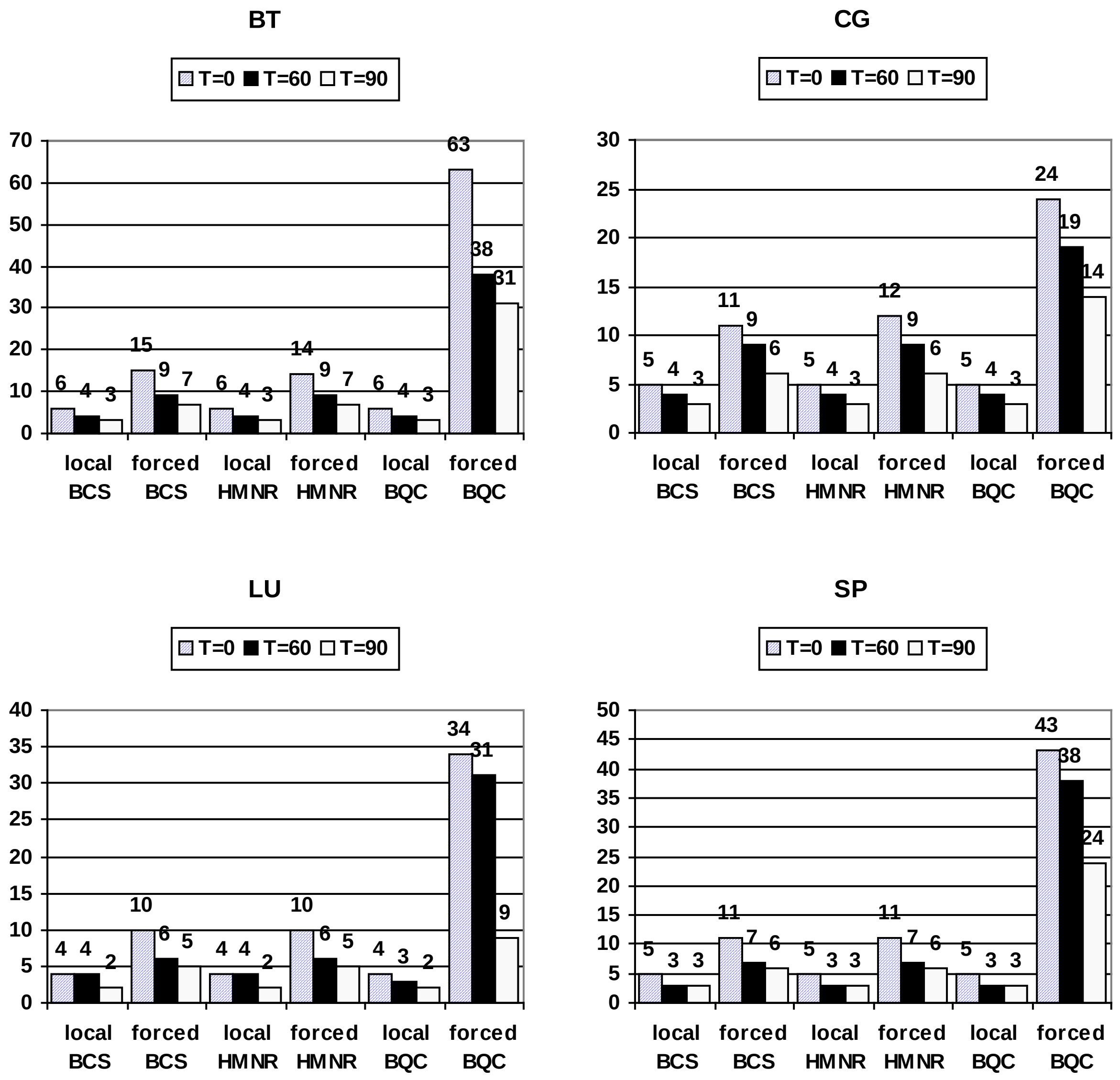

Figure 6: The effect of adaptive local checkpointing: Number of local and forced checkpoints for the 3 protocols under different values of T for the four applications under study.

Analysis: The results show that taking forced checkpointing into account reduces the number of local checkpoints that have to be taken, and in turn this reduces the number of forced checkpoints. The results were more pronounced for the BQC protocol. This result shows two things:

1. A successful local checkpoint placement policy must have a dynamic element in it that takes into account the occurrence of forced checkpoints. The simple "let us checkpoint ever *x* seconds" does not work well.
2. A successful local checkpoint placement policy must adapt to the application communication patterns if they change during execution. This would allow the frequency to be reduced during times where the communication load is heavy and frequency of forced checkpoints is high, and vice versa.

Our recommendations once more outline the unpredictability that faces a user of these protocols in practice, though they outline plausible solutions. It is perhaps possible to come up with better placement policies than the one we outlined here, but this is out of the paper's scope.

5. Related Work

The earliest work reported on CIC is due to Briatico et al [2]. Several variations on this protocol were subsequently published [9,26]. Recently, there has been a growing interest in reinvestigating these protocols using the Z-cycle framework [1,7,8]. We are not aware, however, of any experimental work to investigate CIC along the lines we followed here. We would like to point out though that two implementations of coordinated checkpointing have used the idea of time-stamping a message with the checkpointing interval as suggested by Briatico [6,23]. There are also several experimental evaluations that were performed on other styles of rollback-recovery such as message logging [4,17], and coordinated checkpointing [6,14,18,20,24,28], but comparing these efforts with the work presented here is out of the scope of this paper.

6. Conclusions

We have conducted several experiments to analyze the behavior and characteristics of communication-induced checkpointing. We studied a class of compute intensive distributed applications, and our results for that class show that:

1. CIC protocols that use an eager approach to preventing Z-cycles by taking forced checkpoints whenever they suspect the formation of a Z-cycle are bound to perform worse

than lazy protocols that use a time stamping function to prevent a Z-cycle at the last possible second.

2. CIC protocols do not scale well with a larger number of processes. We have found that the number of forced checkpoints increase almost linearly with the number of processes.
3. A process takes at least twice as many forced checkpoints as local ones. Therefore, the touted benefit of autonomy of CIC protocols in allowing the processes to take independent checkpoints does not seem to materialize in practice.
4. There is a considerable unpredictability in the way CIC protocols behave in practice. The amount of stable storage required, performance overhead, and number of forced checkpoints depend greatly on the number of processes, the application, and the communication pattern. This unpredictability makes the use of CIC protocols in practice more cumbersome than other alternatives.
5. A successful placement policy of local checkpoints must be dynamic, and must take into account the occurrences of forced checkpoints, and adapt to the change in the application behavior.
6. CIC protocols seem to perform best for situations where the communication load is low and the pattern is random. Regular, heavy load communication patterns seem to fare worse.

Again, we would like to stress that the results are only valid for the application set that we have studied, and we lay no claim that these results would generalize to all applications. Nevertheless, we believe that there is sufficient evidence to suspect that much of the conventional wisdom about these protocols is questionable, and certainly there is need for more work along the lines we followed in this paper to investigate these protocols further.

References

1. R. Baldoni, F. Quaglia, and B. Ciciani. A *VP*-accordant checkpointing protocol preventing useless checkpoints. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems,* pp. 61—67, Oct. 1998.
2. D. Briatico A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the IEEE International Symposium on Reliability, Distributed Software, and Databases,* pp. 207—215, Dec. 1984.
3. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computing Systems*, 3(1):63—75, Aug. 1985.
4. E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing (FTCS-24),* pp. 298—307, Jun. 1994.
5. E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Sep. 1996.

6. E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 39—47, Oct. 1992.
7. J.M. Hélary, A. Mostefaoui, and M. Raynal. Virtual precedence in asynchronous systems: concepts and applications. In *Proceedings of the 11th workshop on distributed algorithms*, LNCS press, 1997.
8. J.M. Hélary, A. Mostefaoui, R.H. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 183—190, Oct. 1997.
9. T.H. Lai and T.H. Yang. On distributed snapshots. In *Information Processing Letters*, vol. 25, pp. 153—158, 1987.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):588—565, Jul. 1978.
11. J. Leon, A.L. Ficher, and P. Steenkiste. Fail-safe PVM: A protable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Feb. 1993.
12. C.C. Li and W.K. Fuchs. CATCH: Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant computing*, pp. 74—81, Jun. 1990.
13. D. Manivannan and M. Singhal. A low-overhead recovery technique using synchronous checkpointing. In *Proceedings of International Conference on Distributed Computing Systems*, pp.100—107, May 1996.
14. G. Muller, M. Hue and N. Peyrouz. Performance of consistent checkpointing in a modular operating system: results of the FTM experiment. In *Proceedings of the First European Dependable Computing Conference* (*EDCC-1*) , pp. 491—508, Oct. 1994.
15. NASA Ames Research Center. NAS parallel benchmarks. Available through the World Wide Web, 1997.
16. R.H. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. Technical Report 93-32, Department of Computer Sciences, Brown University, Jul. 1993.
17. N. Neves and W. K. Fuchs. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Proceedings Proceedings of the 27th IEEE Fault-Tolerant Computing Symposium*, pp., Jun. 1998.
18. J.S. Plank and K. Li. Faster checkpointing with $N + 1$ parity. In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp. 288—297, Jun. 1994.
19. J.S. Plank, M. Beck and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. In the *IEEE Technical Committee on Operating Systems Newsletter, Special Issue on Fault Tolerance*, pp. 62—67, Dec. 1995.
20. J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Technical Conference*, pp. 213—214, Jan. 1995.
21. B. Ramamurthy and S.J. Upadhyaya and R.K. Iyer. An object-oriented testbed for the evaluation of checkpointing and recovery Systems. In *Proceedings of the 27th IEEE Fault-Tolerant Computing Symposium*, pp. 194—203, Jun. 1997.
22. B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220—232, Jun. 1975.
23. S. Rao and L. Alvisi and H. M. Vin. Egida: An Extensible Toolkit for Low-overhead Fault-tolerance. Technical Report TR 98-29, Dec. 1998.
24. L.M. Silva. Checkpointing mechanisms for scientific parallel applications. Ph.D. thesis, University of Coimbra, Portugal, Mar. 1997.
25. L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pp. 155—162, Oct. 1992.
26. K. Venkatesh, T. Radakrishnan, and H.L. Li. Optimal checkpointing and local recording for domino-free rollback-recovery. *Information Processing Letters*, vol. 25, pp. 295—303, 1987.
27. Y.M Wang and K. Fuchs. Optimistic message logging for independent checkpointing in message passing systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 147-154, Oct. 1992
28. Y.M Wang, P.Y. Chung , Y. Huang, and E.N. Elnozahy. Integrating checkpointing with transaction processing. In *Proceedings of the 27th IEEE Fault-Tolerant Computing Symposium*, pp. 304—308, Jun. 1997.