Application-Driven Coordination-Free Distributed Checkpointing*

Adnan Agbaria William H. Sanders
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
{adnan, whs}@crhc.uiuc.edu

Abstract

Distributed checkpointing is an important concept in providing fault tolerance in distributed systems. In today's applications, e.g., grid and massively parallel applications, the imposed overhead of taking a distributed checkpoint using the known approaches can often outweigh its benefits due to coordination and other overhead from the processes. This paper presents an innovative approach for distributed checkpointing. In this approach, the checkpoints are obtained using offline analysis based on the application level. During execution, no coordination is required. After presenting our approach, we prove its safety and present a performance analysis of it using stochastic models.

1. Introduction

In an era in which global computation is possible, e.g., via the Internet, and in which approaches such as grid computing are still growing and attracting more attention in distributed settings, fault tolerance is becoming more crucial in distributed systems. By being fault-tolerant, a commercial distributed system may continue to provide its services in the presence of faults, and a long-running message-passing application may continue its progress even if some processes may fail during the course of its execution.

Checkpoint/Restart (C/R) is a way to provide persistence and fault tolerance in both uniprocessor and distributed systems [10, 20]. *Checkpointing* is the act of saving an application's state to stable storage during its execution, while *restart* is the act of restarting the application from a checkpointed state. If checkpoints are taken, then when an application fails, it may be possible to restart it from its most recent checkpoint. This limits the amount of computation lost because of a failure to the computation performed between the last checkpoint and the failure.

One of the main challenges in implementing C/R mechanisms is that of maintaining low overhead, since otherwise the cost of taking checkpoints will outweigh its potential benefit. In a distributed system, if each process saves its state in a completely independent manner, it is possible that no collection of checkpoints, one from each process, will correspond to a consistent application state. A distributed application's state is *inconsistent* if it represents a situation in which some message m is received by a process, but the sending of m is not in the checkpoint collection. A collection of checkpoints that corresponds to a consistent distributed state forms a *recovery line*. Therefore, the main research focus in this area was and remains on devising techniques that guarantee the existence of a recovery line while minimizing the coordination between processes. This coordination overhead includes the amount of control information exchanged between processes and the number of times some process p is forced to take a checkpoint to ensure that a recovery line exists. Such checkpoints are called *forced* checkpoints.

Over the past decades, intensive research work has been done on providing efficient C/R protocols in traditional distributed computing [10]. There are three main kinds of distributed checkpointing protocols. The first approach is *coordinated checkpointing*, in which all the processes coordinate to produce a recovery line. The two common techniques for coordinated checkpointing are to synchronize and stop (SaS) the execution of all processes until every one has taken a checkpoint [2], or to use Chandy-Lamport's (C-L) distributed snapshots protocol [7], which produces a recovery line on-the-fly. The second approach is *uncoordinated checkpointing*, in which every process takes a checkpoint independently. That approach does not impose any coordination overhead, but runs the risk of never generating a recovery line due to the *domino effect*. The third approach is called *communication-induced checkpointing*, in which processes take checkpoints in an uncoordinated manner. However, based on the communication patterns that can be learned from control information, a process may occasionally be forced to take a checkpoint in order to guarantee the existence of a recovery line. From our point of view, all three approaches impose different coordination overheads, and determination of the optimal approach that

*This material is based upon work supported by the National Science Foundation under Grant No. CNS-0406351.

imposes the minimum overhead could be difficult given the various types of distributed settings. For example, for a grid application in which communication is costly, the uncoordinated approach could be the best where there is no need to exchange messages, but the rollback propagation during restart could be unbounded. On the other hand, the coordinated checkpointing approach may behave better than others if the failure rate is high, in cases where a distributed application needs to roll back only to the latest checkpoint. Even considering the coordination overhead, rollback propagation, failure rate, and other parameters, it is not clear which checkpointing approaches are most appropriate for different distributed settings [2].

In this paper, we introduce an innovative approach for constructing recovery lines without any coordination overhead. Like the other well-known coordinated checkpointing protocols (e.g., SaS and C-L), our approach ensures that the recovery line is known and exists as the collection of the latest checkpoints in every process.

In a nutshell, our approach consists of three main phases that are applied offline. The first phase is to use the application level to gain knowledge about expected running time and the communication pattern to insert checkpoint statements in the code. The second phase is to represent the message-passing program using *control flow graphs* [18], and then try to match every receive statement with its corresponding send statement in the control flow graph. The third phase is to change the location of the checkpoint statements in the code such that in any possible execution, we always obtain recovery lines for a specific set of checkpoints. Although the idea behind the first phase has been used for serial programs, our approach involves several innovations, including the application of the first phase in message-passing applications, the introduction of new techniques for applying the second and third phases, and the combination of these three phases together to achieve a coordination-free distributed checkpointing approach.

2. System Model and Definitions

We consider a distributed system consisting of n processes, denoted by $P_1, P_2, \cdots, P_n$, connected by a network (a process P_p may be denoted by p). Processes communicate via asynchronous reliable message passing. Each process P_i is modeled as an automaton with a predefined initial state e_i and a deterministic transition function from its current state to the next state based on the current state and the occurring *event*. The possible events are **computation**, **send**, **receive**, or **checkpoint**. A *local history* of a process is a sequence of such events. An *execution* is a collection of local histories, one for each process.

For a message m in the execution, **Send**(m) denotes the send event of m, and **Recv**(m) denotes the receive event of m. Events in an execution are related by the *happened before* relation [13], denoted by $\xrightarrow{\text{hb}}$. This relation is defined as the transitive closure of the process order and the relation between the send and receive events of the same message. Moreover, we assume that the network delivers messages reliably, in FIFO order.

Obviously, any execution E is created as a result of the execution of a message-passing program on a distributed system. Every event in E occurs as the result of the execution of a particular statement written in the code of the program. Formally, given a message-passing program $\mathcal{P}$, we say that an execution $E(\mathcal{P})$ *obeys* $\mathcal{P}$ if all the events in $E(\mathcal{P})$ are created as a result of the execution of $\mathcal{P}$ on a distributed system. Particularly, the send, receive, and checkpoint events in an execution happen because of the invocation of send, receive, and checkpoint statements in the program code. The execution is denoted by E if $\mathcal{P}$ is implicity known.

We assume that different executions of the same program are identical for the same input. Moreover, we assume that the receive events in the execution are *blocking*. In other words, if a **Recv**(m) is occurring in process p, then the process is blocked until m is received.

Each checkpoint taken by a process is assigned a unique sequence number. The ith checkpoint of process p is denoted by $C_{p,i}$. The ith *checkpoint interval* of process p, denoted by $I_{p,i}$, is the sequence of all events performed between p's $(i-1)$th and ith checkpoints, including the $(i-1)$th checkpoint, but not the ith.

When a failure occurs in a distributed system, we need to recover from a *cut* of checkpoints (i.e., a set of checkpoints consisting of one checkpoint from each process). However, not all cuts of checkpoints are *consistent*, i.e., correspond to a state that could have been reached in the execution. A consistent cut of checkpoints is called a *recovery line*. More precisely,

Definition 2.1: Given an execution E and a cut of checkpoints $S \in E$, S is a *recovery line* if there are no checkpoints $C, C' \in S$ such that $C \xrightarrow{\text{hb}} C'$.

In this paper we are concerned with only a subset of all the possible cuts in the execution. Our interest is on ensuring that every cut of checkpoints in the subset is a recovery line. The subset consists of all the cuts that have the same checkpoint sequence number in every process.

Definition 2.2: Given an execution E and an integer i, the collection of all the ith checkpoints of every process in E represents a *straight cut of checkpoints*, denoted by $S_i = \{C_{1,i}, C_{2,i}, \cdots, C_{n,i}\}$.

In our approach, we show a method to transform an arbitrary message-passing distributed program so that every

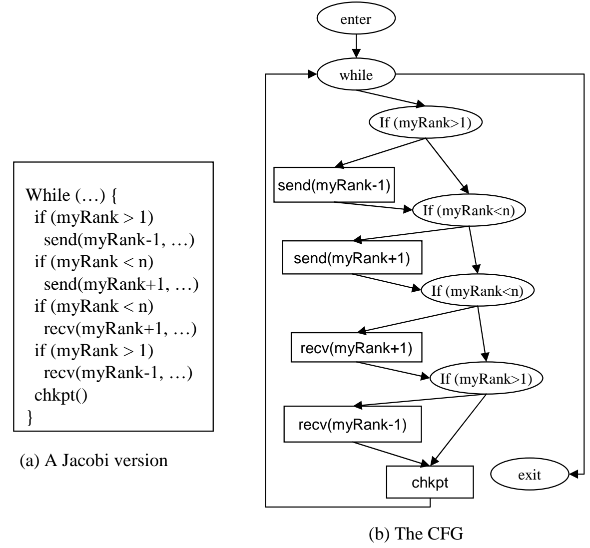
straight cut of checkpoints in the program is a recovery line. Given a message-passing program $\mathcal{P}$, we construct its control flow graph (hereafter, CFG), which is constructed during compilation [18]. Then, we apply our techniques on the CFG to ensure that every straight cut of checkpoints is a recovery line.

Given a message-passing program $\mathcal{P}$, the CFG of $\mathcal{P}$ is the directed graph $G = \langle N, E \rangle$ with node set N and edge set $E \subseteq N \times N$, where a directed edge is denoted by $\langle a, b \rangle$. As defined in [18], a CFG should contain nodes representing loops and conditions in the program. In addition, we build a CFG with nodes representing the send, receive, and checkpoint statements that generate the events described in the system model. In addition, a CFG G has another two nodes: the *entry* and *exit* to indicate the start and termination nodes in G.

Given a directed edge $\langle a, b \rangle$ in G, we say that a is a *predecessor* node of b, and b is a *successor* node of a. A *branch* node is a node that has more than one successor, and a *join* node is a node that has more than one predecessor. Notice here that a branch node in G corresponds to a condition expression statement in the program. Loops in $\mathcal{P}$ are identified in G through the use of dominators. A node a in G *dominates* another node b if every path from the entry node to b includes a. An edge $e = \langle a, b \rangle$ is said to be a *backward* edge in G if b dominates a. Given a backward edge $\langle a, b \rangle$, a loop in G consists of all the nodes that exist in the path between b to a, including a and b.

Figure 1(a) presents a message-passing version of a program that solves a system of linear equations using the Jacobi method. Figure 1(b) presents the corresponding CFG of the program. Notice that the edge from the **chkpt** node to the **while** node in the CFG is a backward edge. In this example, since the checkpoints of all the processes are taken in the same place in the code, it is easy to show that any possible execution of the program has the property that every straight cut of checkpoints is a recovery line.

However, if we change the program in Figure 1 such that the **chkpt** statement does not appear in the same place for each process, a straight cut of checkpoints may not be a recovery line. For example, consider another version of the program, whose corresponding CFG is presented in Figure 2. In this example, we change only the checkpoint statement. Notice here that the CFG has different paths inside the **while** loop for odd and even process ID numbers. A process with an even ID number takes the checkpoint before sending and receiving messages. However, a process with an odd ID number takes a checkpoint after sending and receiving messages. Figure 3 presents a possible execution of this version of the program. Since there are exchange messages between neighbor processes, causality exists between checkpoints of any connected neighbor processes. Therefore, in the execution shown in Figure 3, not every straight



(a) A Jacobi version

(b) The CFG

Figure 1. The Jacobi program

cut of checkpoints is a recovery line.

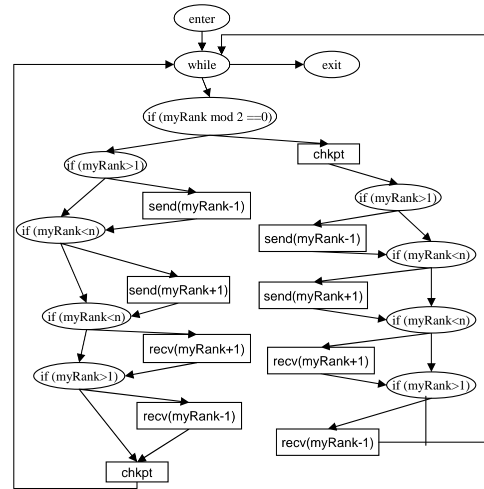


Figure 2. Another version of Jacobi

Since an inconsistent cut in an execution is caused by messages that cause causality between checkpoints, and since a CFG does not have any information about messages, then based on the CFG of a message-passing program, it is difficult to predict whether a straight cut of checkpoints will form a recovery line. Therefore, we extend a CFG representation to include more information about the expected communication patterns that may happen during any execution. In addition, we show that such information is useful for predicting whether every straight cut of checkpoints will be a recovery line. Briefly, we extend the CFG to include *message edges* that represent the communication between

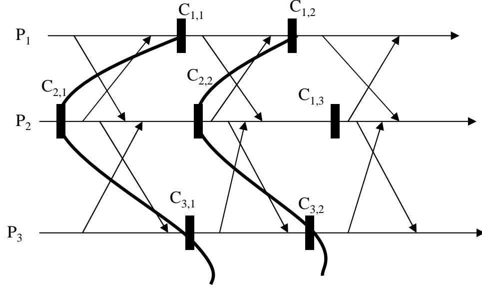


Figure 3. An execution of the CFG of Figure 2

every two corresponding send and receive nodes. In Section 3, we show the determination of the message edges for a given CFG. For example, Figure 4 presents the *extended* CFG of Figure 2, with message edges added between the corresponding send and receive nodes. For a CFG G, we use $\hat{G}$ to denote the extended CFG of G.

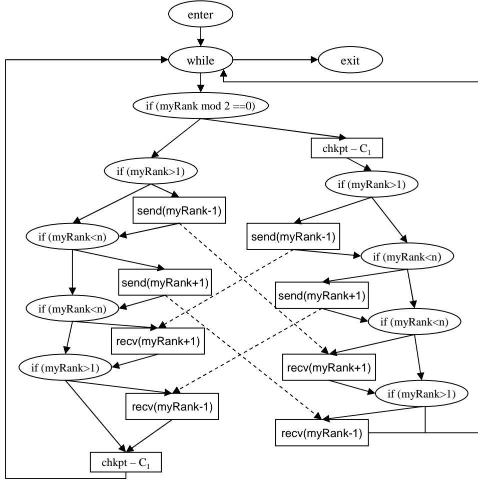


Figure 4. The extended CFG of Figure 2

Given a message-passing program $\mathcal{P}$ and its CFG G, we enumerate the checkpoint nodes along every path from the **entry** node to the **exit** node. We use C_i^γ to denote the node in CFG that contains the ith checkpoint node from the **entry** node to the **exit** node along the γ path. We only use C_i if the path is not mentioned explicitly. For example, since the two checkpoint nodes in the extended CFG of Figure 4 exist in different paths in the corresponding CFG, they are both denoted by C_1. In addition, we define S_i to be the collection of C_i's in a CFG for every path.

Previously, we used $C_{p,i}$ to represent the occurrence of the ith checkpoint event in process p in a given execution. We refine the definition of $C_{p,i}$ in the execution to match the definition of C_i in the CFG. Every invocation of the checkpoint statement corresponding to C_i by a process p is denoted by $C_{p,i}$ in the execution. Notice here that if a checkpoint statement is in a loop, then in every iteration of the loop, the checkpoint created due to the invocation of this statement will be assigned to the same index. Therefore, a process p may contain several checkpoints indexed as $C_{p,i}$. Consequently, we refine the definition of a straight cut of checkpoints to deal with multiple $C_{p,i}$'s as follows.

Definition 2.3: Given a message-passing program $\mathcal{P}$ and an integer i, in any execution E that is driven by the CFG of $\mathcal{P}$, we define the *straight cut of the ith checkpoints* in E to be $R_i = \{C_{p,i} | C_{p,i}$ is the latest ith checkpoint in $p\}$.

3. Offline-Based Analysis Recovery Line

In our approach, a recovery line is determined during offline analysis before a message-passing application is executed. In two sentences, our approach works as follows: if the code does not have checkpoint statements, we insert checkpoint statements in the application code. Then we move the checkpoint statements that represent a straight cut of checkpoints to ensure a recovery line for any further execution. As a result of this approach, during an execution of the message-passing application, each process takes *local* checkpoints due to the checkpoint statements without any extra communication or coordination with the other processes.

To simplify our offline analysis, we assume that a message-passing program belongs to the Single Program Multiple Data (SPMD) type, in which the whole program is represented in one source file. This assumption helps us to identify the corresponding send statement for every receive statement. In addition, if all the files of the source code of a message-passing program are presented for offline analysis, our approach works for Multiple Program Multiple Data (MPMD) as well.

In this section, we describe our approach in three phases. In the first phase, we insert application-level checkpoints if needed. Then, in the second phase, we describe how to determine the message edges in the CFG in order to generate its extended CFG $\hat{G}$. Lastly, in the third phase, we state and prove a necessary and sufficient condition ensuring that every collection of checkpoint statements S_i will produce a recovery line in any further execution. Then, based on this condition, using $\hat{G}$, we examine every collection of checkpoint statements S_i.

3.1. Phase I: Static Checkpoint Insertion

In this phase, we apply known static techniques for checkpoint insertion [8, 15, 16, 22] in serial code. Note that this phase is optional, and is only needed if the code does not contain checkpoint statements.

The checkpoint insertion techniques provide an efficient way to insert checkpoint statements in serial applications. Some of the techniques require users to insert checkpoint statements in the application code [8, 22]; others depend on the compiler to analyze the program and insert the checkpoint statements in the target code after compilation [15, 16]. In general, these techniques are based on analyzing the code and inserting checkpoint statements in the source code to ensure optimal checkpoint intervals. For example, Chandy and Ramamoorthy [8] used a *graph model* to describe a program. They claimed that after estimating the bounds of the program instruction, they would be able to insert checkpoint entries in the program such that they satisfy optimal checkpoint intervals. On the other hand, in [14], Li et al. built the CATCH compiler, which takes program code and inserts the checkpoint statements in a way that tries to preserve optimal checkpoint intervals.

In Phase I, we use one of the known techniques for checkpoint insertion, and insert checkpoint statements in the code of a message-passing program. The difference between a message-passing program and a serial program is that the message-passing program contains message-passing statements that may perform differently with different processes. Therefore, before applying this phase, we estimate the message delay in the network [5, 12]. By estimating the delay, we help the used technique to determine the optimum checkpoint interval in every process for further execution of the message-passing application.

Notice here that we may add/remove some of the checkpoints to ensure that every path of the CFG has the same number of checkpoint nodes.

3.2. Phase II: Generating the Extended CFG

Given a CFG, in this phase we show how we add the message edges to the CFG that match every receive node with its corresponding send node. Informally, we scan the CFG. Then, for each receive node that has not yet been matched, we scan the CFG again, trying to identify all the candidate send nodes for the matching. We claim that a send node can be matched with a given receive node if they exist in different paths and their parameters (source and destination parameters) do not present any contradiction.

Usually, a typical message-passing program contains two types of communication statements that can eventually be reduced to send/receive statements. The two types are collective communication and point-to-point communication [1].

In *collective communication*, the same statement should exist in the code of all the processes in which the *source* and *distention* parameters are indicated explicitly. Usually, the source and distention parameters are computation expression patterns that indicate the process IDs of the sender(s) and the receiver(s), respectively. For example, the MPI called **MPI_Bcast** should be called by all the processes that participate in the multicast operation [1]. Furthermore, using any message-passing compiler, every collective communication statement can be reduced to send/receive statements. For that reason, it is easy to determine the message edges in a CFG between the send and receive nodes that are driven from a collective communication statement.

On the other hand, the *point-to-point communication* type consists of the basic send/receive statements, e.g., **MPI_Send** and **MPI_Recv**. Since a send and its corresponding receive statement may exist in different places in the code, it might be difficult to determine the match between them during compilation. Another difficulty that we may encounter is that the source and destination parameters could have irregular computation patterns [18]. A computation pattern is called *irregular* if it depends on input data. Notice here that in a statement of collective communication type, irregular computation patterns do not bother to match, since the statement exists in the code of each process. However, for the point-to-point communication type, we observe that since the send and receive statements are supposed to be called by different processes in any further execution, the corresponding send and receive nodes should exist in different paths in the CFG. Notice that those different paths, which include the matched send and receive statements, should be created due to a condition expression that depends on process IDs. We say that a branch node in the CFG is *ID-dependent* if its condition expression in the program depends on process IDs.

A branch node in a CFG represents a condition expression in the program. Obviously, every control path in the CFG from the branch node is characterized by an *attribute* that is driven from the condition expression. Therefore, for every branch node, we can determine the attribute of every control path from the node during compilation. For example, in Figure 2, the first branch node is ID-dependent, since it depends on **myRank**. The right path after this branch node has the attribute of even process ID numbers, and the left path has the attribute of odd process ID numbers.

Using any data flow analysis technique [4, 18], we can specify whether each branch is ID-dependent or not. Specifically, we first determine the variables and constants that depend on process IDs, and then use the technique of data flow analysis to determine whether each condition expression is ID-dependent or not. To simplify the presentation of our mechanism for matching the send and receive nodes, we ignore all the non ID-dependent branches in the CFG. For that purpose, without loss of generality, we assume that all the branch nodes are ID-dependent.

Our algorithm depends on the fact that the matched send and receive nodes exist in different paths that are obtained via ID-dependent branches. Relying on the assumption that

every send/receive statement has the source and destination parameters written explicitly, our algorithm tries to match every receive statement with the send statement such that there are no contradictions between the source and destination parameters of the receive statement and the destination and source parameters of the send statement, respectively. Therefore, for every send/receive statement, we determine the driven attributes from the source and destination parameters. Our algorithm of matching works as follows:

Algorithm 3.1 : Scan the CFG graph using the DFS (Depth-First Search) method from the **entry** node. Encounter the following nodes:

Branch: analyze the condition expression corresponding to this node and determine the attributes for every fan-out control path.

Receive: if the node has not yet been matched, then

1. Based on the source parameter, determine the attribute, and call it SA.
2. Scan G to determine all the successor nodes $e_1, \cdots, e_r$ in different paths that have attributes that do not contradict SA.
3. For every successor node e_i, scan the CFG starting from e_i. Consider all the send nodes in the subgraph from e_i and determine their attributes that are determined by the destination parameters; call them $DA_1, \cdots, DA_k$. For every j, $1 \leq j \leq k$,
   - If SA and/or DA_j has an irregular pattern, then match the receive node and the corresponding send node if SA and DA_j do not contradict.
   - Otherwise, if the corresponding send node has not yet been matched and SA and DA_j do not contradict, then match the receive node and the corresponding send node.

Notice here that since the algorithm uses DFS for scanning the graph, it will visit all the nodes in the CFG. Therefore, every send/receive node should be visited by the algorithm. In addition, since our algorithm does the match only if there are no contradictions between any attributes, then if a receive node has a parameter that is an irregular computation pattern, the node may be matched with more than one send node. The following lemma states the safety of our algorithm. Because of space restrictions, we eliminate the proof.

Lemma 3.1: Given a CFG G, for every **receive** node in $\hat{G}$, at least one of the matched **send** node(s) due to Algorithm 3.1 is the corresponding send statement of the receive statement in the program.

3.3. Phase III: Ensuring Recovery Lines

In the third phase, we consider the extended CFG $\hat{G}$ of a message-passing program that has the checkpoint nodes in it. First, we give a necessary and sufficient condition on $\hat{G}$ to ensure that for every i and any further execution, R_i is a recovery line. In order to guarantee that this condition is satisfied, we may need to move some checkpoint nodes in $\hat{G}$, and consequently in the program code. For example, if a checkpoint statement is before a send statement, we may move the checkpoint statement to appear after the send statement.

Obviously, the $\stackrel{\text{hb}}{\to}$ relation is created by an application message from $I_{p,i+1}$ to $I_{q,i}$ in E. For every process p, any checkpoint $C_{p,i}$ is created as a result of calling a checkpoint statement in S_i (S_i contains all the checkpoint nodes C_i^γ for every path $\gamma \in \hat{G}$). Then, to avoid the $\stackrel{\text{hb}}{\to}$ relation between any two checkpoints in R_i, we need to ensure that there are no edge messages between any two checkpoint nodes in S_i.

Consider the extended CFG $\hat{G}$ presented in Figure 5 and its corresponding execution example. Assume that process P_1 executes along path A and P_2 executes along path B. It is easy to verify that due to the path between C_i^A and C_i^B in $\hat{G}$, the straight cut of checkpoints is inconsistent. Therefore, it is clear that in order to avoid such inconsistent cuts, we need to avoid any path between any two members of S_i.

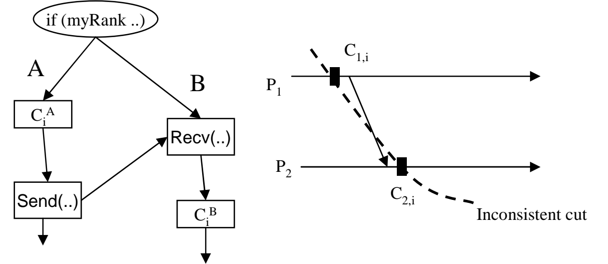


Figure 5. A CFG $\hat{G}$ and its execution

Now consider the example presented in Figure 6. In this example we can see that there is a path from C_1^B to C_1^A. However, the path contains the drawback edge $\langle$**Recv**, **while**$\rangle$ of the loop in path A. More precisely, the path is $\langle C_1^B$, **Send**, **Recv**, **while**, $C_1^A\rangle$. If P_2 fails right after sending the message in the second iteration of the loop, R_1 is not a recovery line.

We now state a sufficient condition for ensuring recovery lines.

Condition 1: If for every integer i and every collection of checkpoint nodes S_i, there is no path in the extended CFG between any two checkpoint nodes in S_i, then in any further execution, R_i is a recovery line.

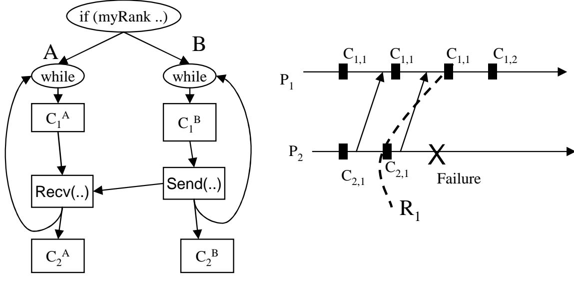


Figure 6. Another CFG $\hat{G}$ and its execution

Given an extended CFG $\hat{G}$, based on Condition 1, we now show an algorithm that examines every S_i in the CFG to ensure that S_i produces a recovery line in any further execution.

Algorithm 3.2: Given an extended CFG $\hat{G}$, determine $\{S_1, S_2, \cdots, S_m\}$ in $\hat{G}$ (assume that every path has m checkpoint nodes). For every S_i, $1 \leq i \leq m$, do the following steps:

Step 1 For every two checkpoint nodes $C_i^A, C_i^B \in S_i$, check whether there is a path between them in $\hat{G}$. To verify whether there is a path, we can use any graph theory algorithm for finding paths, e.g., Dijkstra's algorithm [9].

Step 2 If there is a path γ between C_i^A and C_i^B in $\hat{G}$ such that $C_i^A \overset{\gamma}{\rightsquigarrow} C_i^B$, we move C_i^B back in G to avoid γ. Let $\langle a, b\rangle$ be the edge in G such that a and b dominate C_i^B in G (notice here that G is the CFG without the message edges). Let $C_i^A \overset{\gamma}{\rightsquigarrow} b$ in $\hat{G}$; however, if there is no path from C_i^A to a in $\hat{G}$, then we move C_i^B to appear between a and b in G. Notice here that b is the first node of the path $\langle \text{ENTRY}, \cdots, C_i^B\rangle$ that is in γ.

To illustrate Algorithm 3.2, consider the extended CFG $\hat{G}$ presented in Figure 6. As we pointed out, $C_1^A, C_1^B \in S_1$, but $C_1^B \overset{\gamma}{\rightsquigarrow} C_1^A$ such that $\gamma = \langle C_1^B, \textbf{Send}, \textbf{Recv}, \textbf{while}, C_1^A\rangle \in \hat{G}$. By Algorithm 3.2, the edge $\langle \textbf{if}, \textbf{while}\rangle$ plays the role of the edge $\langle a, b\rangle$ of Step 2. Thus, we avoid γ by moving C_1^A to appear between the **if** node and the **while** node.

A significant drawback of Algorithm 3.2 is that some of the checkpoints have been moved out of loops. For many programs, such as the program presented in Figure 1, loops are considered the main code of the application. One optimization that we can do in our algorithm to avoid such movement out of loops is to ensure the following condition. If a checkpoint node C_i^A is in a loop and there is another checkpoint C_i^B such that $C_i^B \overset{\gamma}{\rightsquigarrow} C_i^A$ and γ has a backward edge (C_i^B could be inside a loop or not), then in any execution we need to guarantee that the $C_{i,p}$ that occurs due to C_i^A completes before the $C_{i,q}$ that occurs due to C_i^B for any two processes p and q. Furthermore, if $C_i^B \overset{\gamma}{\rightsquigarrow} C_i^A$ and $C_i^B \overset{\gamma'}{\rightsquigarrow} C_i^A$, then we guarantee the completion of one checkpoint before the other according to the order of the message edges that are involved on the path between them.

We now show that after Phase III has been applied, in any further execution E and for every integer i, R_i is a recovery line. We state a theorem showing that Condition 1 is a necessary and sufficient condition. Then, we show that Algorithm 3.2 ensures Condition 1.

Theorem 3.2: Given a CFG G, for every execution E from G and for every integer i, R_i is a recovery line iff there is no path in $\hat{G}$ between any two members of S_i.

Proof sketch: It is easy to show that after Algorithm 3.2 is applied on a graph $\hat{G}$, for every S_i in $\hat{G}$, there is no path between any two members of S_i. Therefore, the condition of Theorem 3.2 will hold, and then in any further execution E, every $R_i \in E$ is a recovery line. □

To prove the correctness of Algorithm 3.2, we need to show the existence of the edge $\langle a, b\rangle$ in G as mentioned in Step 2 of the algorithm. Indeed, such an edge should exist in G because of the ENTRY node. In any case, the ENTRY node can take the role of node a, where there are no incoming edges to the ENTRY node.

4. Performance Analysis

Since our approach does not impose any coordination among the processes, it promises better performance than any other approach. However, since there are many parameters that should be considered in evaluating performance, it is not easy to compare our approach with the other known approaches. We present here an analysis of the performance *overhead ratio* as presented in [2, 21, 25]. Using this analysis and considering many parameters, we try to evaluate our approach by comparing it to the other approaches.

In the analysis, we consider all the following parameters: λ denotes the failure rate of the process (following an exponential distribution) of Poisson distribution, Γ denotes the expected execution time of a checkpoint interval, o denotes the *checkpoint overhead* which is the increase in the execution time of a process p because of a single checkpoint, l denotes the *checkpoint latency* which is the time required to take a single checkpoint, R denotes the *recovery overhead* which is the time required to restart a process from a checkpoint, M denotes the *recovery overhead* which is the time required to restart a process from a checkpoint, C denotes the *coordination overhead* which is the overhead due to process coordination that may happen in taking a cut of checkpoints, O denotes the *total checkpoint overhead* which is the increase in the execution time (because of o, M, and C), and

L denotes the *total latency overhead* which is the increase in the execution time (because of l, M, and C). Lastly, we define the overhead ratio, denoted by r, as $r = \frac{\Gamma}{T} - 1$.

Any checkpoint interval is completed either with or without failure. Therefore, the expected execution time Γ can be computed using the 3-state Markov chain presented in Figure 7. Process p starts the interval with the start state i (related to the checkpoint $C_{p,i}$). A transition from state i to the sink state $i+1$ occurs if $I_{p,i+1}$ is completed without failures. If a failure occurs during $I_{p,i+1}$, then p recovers from $C_{p,i}$. In that case, we have a transition from state i to state R_i. After state R_i is entered, a transition is made to state $i+1$ if no further failure occurs in $I_{p,i+1}$ after a recovery. Otherwise, a transition is made from state R_i to itself. Let s, t be states in a Markov chain in which there is a transition from s to t. $P_{s,t}$ denotes the *probability* of the transition from s to t and $W_{s,t}$ denotes the expected execution time spent in state s before moving to state t. In the Markov chain, Γ is the *expected cost* of reaching the sink state $i+1$ from state i. Since there are only two possible paths from state i to state $i+1$, then $\Gamma = P_{i,R_i}\left(W_{i,R_i} + \frac{P_{R_i,R_i}}{1-P_{R_i,R_i}} W_{R_i,R_i} + W_{R_i,i+1}\right) + P_{i,i+1} W_{i,i+1}$.

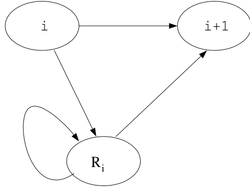


Figure 7. A Markov chain represents $I_{p,i+1}$

The probability that there is no failure during $T+O$ units of time is $P_{i,i+1} = e^{-\lambda(T+O)}$. Notice here that $W_{i,i+1} = T+O$. However, if a failure occurs during $I_{p,i+1}$, then the rollback is made to the latest checkpoint $C_{p,i}$. Therefore, a transition is made from state i to one of the states R_i. The probability of such a transition is equal to $1 - P_{i,i+1}$. The cost of this transition, W_{i,R_i}, is the expected execution time for $I_{p,i+1}$ until a failure occurs. Given that a failure occurs in the interval $[0, T+O)$ during the execution of $I_{p,i+1}$, the *time to failure* (TTF) is a random variable x in the interval $[0, T+O)$ [23]. Moreover, its *probability density function* (PDF) is $\lambda e^{-\lambda x}$ for $0 \leq x < T+O$, and its conditional density function is $f(x) = \frac{\lambda e^{-\lambda x}}{P_{i,R_i}}$, where $0 \leq x < T+O$. This implies that $W_{i,R_i} = \int_0^{T+O} x \cdot f(x)dx = \frac{1}{\lambda} - \frac{(T+O)e^{-\lambda(T+O)}}{1-e^{-\lambda(T+O)}}$.

After state R_i is entered, a transition to state $i+1$ is made if no further failure occurs before $I_{p,i+1}$ is completed. As pointed out in [2], the execution time required to reach state $i+1$ is $W_{R_i,i+1} = T+O+R+L-o \cong T+R+L$. Therefore, the probability that no additional failure will occur is $P_{R_i,i+1} = e^{-\lambda(T+R+L)}$. However, if another failure occurs after the system has been in state R_i, a transition is made from state R_i to itself. For that transition we define $P_{R_i,R_i} = 1 - P_{R_i,i+1} = 1 - e^{-\lambda(T+R+L)}$. As discussed above, W_{R_i,R_i} can be obtained in the same way as W_{i,R_i}. The PDF of the TTF is $\lambda e^{-\lambda x}$ for $x \in [0, T+R+L)$.

After replacing the variables in the equation for Γ with the obtained values, and then doing some mathematical simplification, we obtain $\Gamma = \lambda^{-1}(1 - e^{-\lambda(T+O)})e^{\lambda(T+R+L)}$. Then, after plugging that value of Γ into the equation of r, we obtain $r = \frac{\lambda^{-1}e^{\lambda(R+L-O)}(e^{\lambda(T+O)}-1)}{T} - 1$.

With any checkpointing and recovery mechanisms, T and n are the only parameters that a user can program. We report results using local checkpointing and recovery mechanisms as presented in [3]. In previous experimental work, we ran a matrix multiplication application in Starfish [3] with checkpointing. From that experimental work we obtained the following parameters: $o = 1.78$, $l = 4.292$, and $r = 3.32$ seconds. In addition, we assume that the failure rate of a single process is $1.23 \cdot 10^{-6}$ [21, 24]. Under the assumption that process failures occur randomly and independently with a probability of $p = 1.23 \cdot 10^{-6}$, the probability of a failure in a system with n processes is $1 - (1-p)^n$. Therefore, the overhead ratio increases proportionally with n. In addition, we assume that $T = 300$ seconds.

4.1. Comparison with other Protocols

We compare our approach to other coordinated checkpointing protocols. Because of space limitations, we choose to compare our approach only to coordinated protocols, because our approach has the coordinated checkpointing property that the recovery line exists as the last checkpoint of every process. The most well-known coordinated protocols are the SaS protocol [19] and the C-L protocol [7]. We do not give a description of the protocols, but we determine the value of the parameters for them.

In SaS, since all the processes stop during checkpointing, the collection of checkpoints constructs a recovery line. The message overhead results from the fact that in each phase of SaS, the coordinator broadcasts three messages, and the other $n-1$ processes send two reply messages. Notice that the protocol needs an 8-bit program message. Therefore, $M(SaS) = 5(n-1)(w_m + 8 \cdot w_b)$, where w_m is the "setup" time for sending a message, and w_b is the additional per-bit delay associated with sending a message. On the other hand, in C-L with a fully connected network with n nodes, C-L generates $2n(n-1)$ messages per checkpoint. Also, as the marker is assumed to be 8-bit, we have $M(C\text{-}L) = 2n(n-1)(w_m + 8 \cdot w_b)$.

After determining all the parameters of our approach, SaS, and C-L, we compute and compare their overhead ratio. Figure 8 shows a comparison between our approach

and the other protocols. Since our approach ("appl-driven" in the figure) does not require any coordination, its overhead ratio is smaller than that of the other protocols. Notice here that the overhead ratio increases proportionally with the number of processes because the failure rate λ increases proportionally with the number of processes.

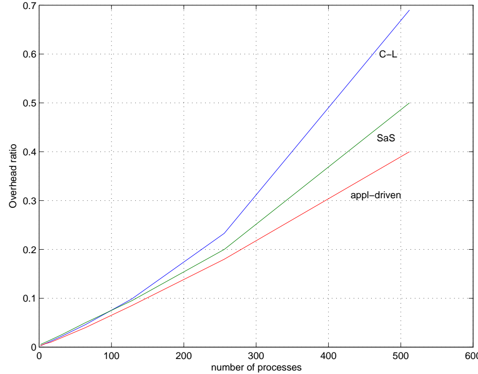


Figure 8. Comparing protocols

Figure 9 shows that while the overhead ratio of the C-L and SaS protocols gets worse as the communication parameter increases, the overhead ratio of our protocol remains unchanged. For example, in the figure, we vary the setup time (w_m) it takes to send a message. Notice here that parameters that depend on the network can be varied during execution depending on the network status. For instance, due to congestion in the network, w_m may increase significantly.

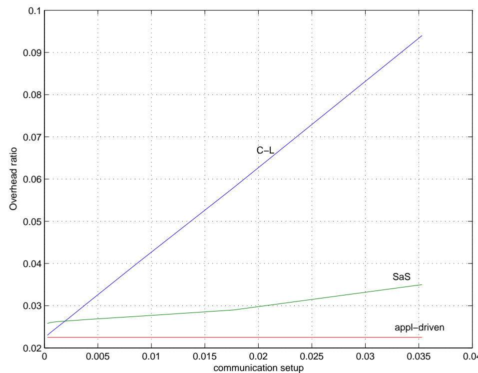


Figure 9. The communication setup effect

5. Related Work

Much work has been done on distributed checkpointing, such as [10, 17, 19]. Those papers presented summaries of different checkpointing protocols. To the best of our knowledge, all the checkpointing protocols impose overhead due to additional communication (e.g., program messages) and rollback recovery. However, our approach is the first approach for distributed settings that does not impose any overhead from those parameters [2, 21].

Some work has been done on application-driven checkpointing, especially in distributed settings. Bronevetsky et al. [6] presented an application-level checkpointing for message-passing application that was based on the C-L (Chandy-Lamport) protocol, with the change that the checkpoints are initiated at the application level instead of at the system level as in the C-L protocol. The only thing that the Bronevetsky et al. protocol takes from the application is information on where to apply the checkpoint calls at the application level to capture a global snapshot of the distributed application. Therefore, that protocol still imposes the same overhead as the C-L protocol. On the other hand, for serial applications, there are a number of compiler- and application-driven checkpointing mechanisms. Most of them were developed to insert checkpoint calls at the application level to obtain an optimal checkpoint interval. Chandy and Ramamoorthy [8] were the first people to show that by analyzing the program code, we can insert the checkpoint calls to achieve an optimal checkpoint interval. The problem was then solved by others [16, 22]. Also, for serial applications, much of the research has relied on the application level in introducing new features of fault tolerance using checkpointing. For example, many checkpointing protocols use compilers and code analysis to achieve heterogenous checkpointing [11, 15], incremental checkpointing [20], and other types of checkpointing.

6. Conclusions

In our previous work [2], we showed that coordinated checkpointing approaches are the most efficient among a set of known distributed checkpointing protocols. Concentrating on distributed checkpointing for providing fault tolerance, we came up with a new approach that has better performance than the previous approaches that we know about.

To the best of our knowledge, our approach is the first checkpointing protocol that produces recovery lines without coordination, additional checkpoints, or rollback propagation. In addition to presenting the protocol itself, we stated and discussed the safety and termination of the protocol. As a proof of concept, we used stochastic analysis to compare the overhead ratio of our approach with other well-known approaches for distributed checkpointing.

Our approach has three phases. In the first phase, we solve the problem of inserting checkpoint calls in the application code. Then, in the second phase, we construct the control flow graph of the message-passing program, try to match send and receive statements, and enumerate the checkpoint statements. Lastly, in the third phase, we elabo-

rate the CFG to ensure that in every further execution of the CFG, the straight cut of checkpoints is a recovery line.

We believe that this approach is the first mechanism for obtaining coordination-free recovery lines that is based on use of the application level. This approach is especially useful for high-performance applications in which reliability can be provided without imposing much overhead on the performance.

Acknowledgments

We would like to thank Vikram Adve for discussions related to compilers and Jenny Applequist for her editorial assistance.

References

[1] MPI: Message Passing Interface. http://www.mpi-forum.org.

[2] A. Agbaria, A. Freund, and R. Friedman. Evaluating Distributed Checkpointing Protocols. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 266–273, Providence, Rhode Island, May 2003.

[3] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 167–176, August 1999.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.

[5] S. Biaz and N. Vaidya. Is the Round-Trip Time Correlated With the Number of Packets in Flight? In *Proceedings of ACM Internet Measurement Conference*, Miami, Florida, October 2003.

[6] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 84–94, 2003.

[7] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[8] K. M. Chandy and C. V. Ramamoorthy. Rollback and Recovery Strategies for Computer Programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1986.

[10] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[11] F. Karablieh, R. Bazzi, and M. Hicks. Compiler-Assisted Heterogeneous Checkpointing. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 56–65, New Orleans, LA, USA, October 2001.

[12] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems (TOCS)*, 9(4):364–373, November 1991.

[13] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] C. C. Li and W. K. Fuchs. CATCH: Compiler-Assisted Techniques for Checkpointing. In *Proceedings of the 20th IEEE International Symposium on Fault-Tolerant Computing*, pages 74–81, 1990.

[15] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-Assisted Full Checkpointing. *Software: Practice and Experience*, 24(10):871–886, October 1994.

[16] J. Long, W. K. Fuchs, and J. A. Abraham. Compiler-Assisted Static Checkpoint Insertion. In *Proceedings of the 22nd IEEE International Symposium on Fault-Tolerant Computing*, pages 58–65, July 1992.

[17] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[19] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, January 1993.

[20] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

[21] J. S. Plank and M. G. Thomason. Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, November 2001.

[22] S. Toueg and O. Babaoglu. On the Optimum Checkpoint Selection Problem. *SIAM Journal on Computing*, 13(3):630–649, August 1984.

[23] K. S. Trivedi. *Probability and Statistics with Reliablity, Queuing, and Computer Science Applications*. Prentice-Hall, USA, 1982.

[24] N. Vaidya. On Checkpoint Latency. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, December 1995.

[25] N. H. Vaidya. Another Two-Level Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency. Technical Report TR94-068, Department of Computer Science, Texas A&M University, 1994.