Adaptive Message Logging for Incremental Replay of Message-Passing Programs

Robert H.B. Netzer and Jian Xu

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-26
September 1993

Appears in Supercomputing '93, Portland, OR, November 1993

Adaptive Message Logging for Incremental Replay of Message-Passing Programs

Robert H. B. Netzer rn@cs.brown.edu

Jian Xu jx@cs.brown.edu

Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

Cyclic debugging executes a program over and over to track down bugs. However, for message-passing parallel programs, nondeterminacy makes cyclic debugging impossible without support of special tools. To provide repeatable executions, messages must be traced for later replay. Since parallel programs are long-running, providing fast response to debugging queries requires *incremental* replay, where reexecution is started from intermediate states instead of from the beginning. To support incremental replay, processes must be checkpointed periodically and the contents of some messages traced, but the time and space cost of saving these messages can be prohibitive. This paper presents an *adaptive* message logging algorithm that keeps these costs low by logging only a fraction of the messages. Our algorithm dynamically tracks dependences among messages to determine which cause *domino effects* and must be traced. The domino effect can force a replay to start arbitrarily far back in the execution, and domino-free replay allows any part of the execution to be quickly reexecuted. Experiments on an iPSC/860 hypercube indicate that our algorithm logs only 1–10% of the messages, a 1 to 2 order of magnitude reduction over past schemes which log every message. Our experiments also show that the resulting logs provide a small bound on the amount of reexecution needed to satisfy any replay request. Our new logging algorithm thus reduces the overhead of message logging while bounding the response time to replay requests.

1. Introduction

Repeated execution is a powerful debugging strategy, allowing enough information to eventually be collected to understand bugs. However, message-passing parallel programs can be nondeterministic, and obtaining repeatable executions requires special tools — bugs cannot always be reproduced by a simple reexecution. Obtaining repeatability requires tracing the execution's messages, using the trace to replay the program as needed. In previous work we developed an *adaptive* strategy that makes run-time tracing decisions, usually tracing only $0-10\%$ of the messages, and reducing the cost of tracing by $1-2$ orders of magnitude over past schemes which trace every message. However, this technique required reexecuting the program from the beginning, and can provide unacceptable response time to replay requests of long-running programs. In this paper we present a new scheme that integrates checkpointing and adaptive message logging to provide *incremental* replay, allowing the execution to be restarted from intermediate states. Our scheme adapts to the execution's message-passing patterns to log only the contents of messages that would substantially slow replay if not logged. Experiments on an iPSC/860 show that only $1-10\%$ of the messages typically need to be logged, and that such a log is sufficient to quickly replay *any* part of the execution. Our scheme simultaneously lowers logging overhead while bounding the time required to satisfy a replay request.

Our approach to supporting incremental replay is to periodically checkpoint to disk the state of each process. We can then restart the execution of any process from one of its checkpoints instead of from its beginning. To satisfy message receive operations during the replay, we must also log the contents of some messages, and logging is a dominant cost of this scheme. We keep this cost low by not logging every message. When processes checkpoint independently of each other, we have the freedom to restart a process from any of its checkpoints, independently of where other processes are restarted. This freedom allows us to decide at run-time which messages need *not* be logged because they can be quickly recomputed (during the replay) by re-executing the processes that sent them. We log only messages whose recomputation would substantially slow replay by causing a *domino effect*. For example, a domino effect occurs when some part of a process (say p_1) must be re-executed to reproduce a message needed by another, but p_1 itself requires a message that causes part of a third process p_3 to be re-executed, and p_3 requires a message sent by an earlier part of p_1, requiring

This research was supported by ONR Contract N00014-91-J-4052 (ARPA Order 8225) and NSF grant CCR-9309311.

p_1 to be restarted even further back. In the worst case this domino effect can continue indefinitely, requiring all processes to be restarted from the beginning of the execution. Our algorithm logs enough message to completely avoid any domino effect during replay.

Our main result is an algorithm that dynamically locates and logs domino-effect-causing messages. The algorithm tracks dependences between messages, and when a message is received that introduces a domino-effect-causing dependence, it is logged. No other messages are logged. By logging enough to provide domino-free replay, only a small amount of each process must be reexecuted to satisfy any replay request. Experiments show that in each process only one or two intervals between checkpoints usually need to be reexecuted. We also present a model to exactly characterize which messages lead to domino-causing dependences, and prove that logging the minimal number of messages to avoid these dependences is NP-hard. Our algorithm logs the first message at which such a dependence can be detected, and is thus an approximation. However, experiments show that it performs well.

This work is part of our larger effort on developing adaptive tracing strategies for debugging[7, 8, 14].

2. Motivation and Related Work

There are two parts to incremental replay of message-passing programs: inter-process communication (IPC) tracing, and checkpointing and message logging. IPC tracing records enough to ensure that replay is deterministic and exhibits the same computation as the original execution. Checkpointing and message logging saves a program's intermediate states so that replay of long-running programs can start from a checkpointed state (instead of the beginning). Although in this paper we address checkpointing and message logging, below we briefly describe past work on both problems.

In a message-passing program, variations in scheduling and message latencies can cause two executions of the same program on the same input to produce different results. Although this nondeterminacy may be intended, replaying an execution of such a program requires information about how its processes communicated during that execution. By tracing the order in which messages are delivered (but not their contents), a traced execution can be replayed from its start by forcing a reexecution to deliver messages in the same order[6, 11, 13]. Since IPC tracing is necessary for replay, we assume that such a scheme is always used in addition to checkpointing and message logging (and since message contents are not recorded, its overhead is much lower).

Although IPC tracing alone provides information sufficient for replay, it requires restarting the execution from its beginning, which is impractical for long-running programs — tool users expect fast responses to debugging queries. To achieve quick replay, checkpoints must be taken to periodically save the execution's intermediate states. In addition, the contents of certain messages must also be logged. For example, Figure 1 shows an execution where each process takes two checkpoints. If the checkpoints are always combined as shown to form *restart lines* for replay, then message $m2$ is *lost* — it was sent before the rightmost line but received after. Thus, when replay starts from this restart line, $p1$ will not regenerate the message, so its contents must be initially logged during execution (so the message receive can be satisfied from the log during replay). If the replay will always begin from either of the two lines shown messages $m1$ and $m3$ need not be logged since they will always be recomputed during any replay.

Checkpointing and message logging has been mainly studied in the context of fault tolerant computing [4, 10, 12] and global-state evaluation[1]. Two major schemes have been proposed: *coordinated checkpointing* and *independent checkpointing*. With a coordinated checkpointing, some process must initiate the checkpointing session and run an algorithm to coordinate all other processes in saving their states[1]. Messages that cross the line formed by the local checkpoints are also saved. Although coordinated checkpoints can be used for replay, their main disadvantage is that they force all processes to checkpoint at about the same time. Such a scheme does not adapt well to the execution's message passing patterns, but instead logs *all* messages in-transit at the time

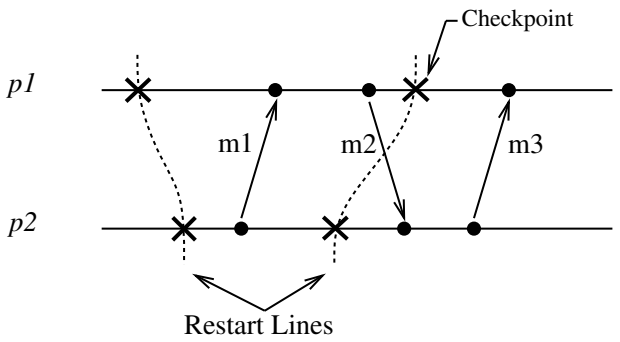

Figure 1. Example execution with checkpoints. The restart lines show ways to start replay. Message m2 is then "lost" and must be retrieved from the log.

of the checkpoint. If the coordinated checkpoint is initiated at an inopportune moment, many messages could be logged. This problem stems from the overly restrictive view that only consistent checkpoints can be taken. Replay *need not* start from a consistent checkpoint; any set of checkpoints can be combined for restart. The approach we outline in this paper exploits this observation by computing at run time how individual checkpoints are best combined to minimize the number of messages logged.

In contrast, independent checkpointing gives each process the autonomy to checkpoint independently of all others. The main disadvantage is the *domino-effect*. Since the checkpoints are no longer coordinated, replaying part of one process could require an earlier interval of another to also be re-executed. For example, if in Figure 1 the execution were started from the rightmost restart line, but $m2$ had not been logged, then process $p1$ cannot be started from the checkpoint but must be rolled back further so that $m2$ will be recomputed. In the worst case, the reexecution has to be started from the very beginning. To avoid the domino effect, each process must independently determine which messages to log. Most work on message logging has been in the fault tolerance community, where execution is restarted after a fault only from last consistent checkpoint[12], and old messages can be discarded. However, for debugging, replays are requested often (unlike failures), and they must complete quickly. Thus, we wish to be able to restart from *any* checkpoint, which need not be consistent. The fault tolerance results thus do not solve the replay problem. In the debugging community, previous work on independent checkpointing for replay simply traces the contents of all messages[3, 13], which can be prohibitively expensive.

Our work on adaptive logging exploits independent checkpointing by combining checkpoints for replay in ways that adapt to the particular execution being traced. This paper reports the first results on how to support trace-and-replay debugging with independent checkpointing without logging all messages. In Section 3 we present our model for representing program executions, and use this model in Section 4 to consider how checkpoints can be combined to minimize message logging but still avoid the domino effect. In Section 5 we present an adaptive logging algorithm that makes these decisions on-the-fly, and in Section 6 present experimental results that show our algorithm usually logs only $1-10\%$ of the messages.

3. Model

Our model is simply a notation for representing program executions. A *checkpointed program execution* is a pair, $P = \langle E, \xrightarrow{\text{RD}} \rangle$, where E is a finite set of *events* and $\xrightarrow{\text{RD}}$ is the *replay dependence* relation defined over E.

An event represents the execution instance of a send, receive, or checkpoint operation. We assume a fixed number of processes exist during execution, and denote the set of all events in process p by E_p, and the i^{th} checkpoint in process p as $C_{p,i}$. The *state interval* $S_{p,i}$ is all computation performed in process p between checkpoints $C_{p,i}$ and $C_{p,i+1}$ (and includes $C_{p,i}$ but not $C_{p,i+1}$).

The replay dependence relation, $\xrightarrow{\text{RD}}$, shows how events depend on one another *during a replay*. An event b is said to be *replay dependent* on an event a (i.e., $a \xrightarrow{\text{RD}} b$) iff a must be reproduced during re-execution before b can be re-executed, because a precedes b in the same state interval, or because a sequence of messages was sent from a (or a following event) to b (or a preceding event). The $\xrightarrow{\text{RD}}$ relation is identical to Lamport's happened-before relation[5] except that events in a state interval $S_{p,i}$ do *not* necessarily depend on events in $S_{p,i-1}$ (the previous state interval), even though they both belong to the same process. This definition reflects the independence during replay of a state interval on earlier ones; since $S_{p,i}$ can be re-executed starting from $C_{p,i}$, no events in earlier intervals in the same process need be executed. In Lamport's relation, an event is dependent on all earlier events in the same process. The $\xrightarrow{\text{RD}}$ relation is defined as the irreflexive transitive closure of the union of two other relations: $\xrightarrow{\text{RD}} = (\xrightarrow{\text{SO}} \cup \xrightarrow{\text{M}})^+$. The $\xrightarrow{\text{SO}}$ relation shows the order in which events in the same state interval execute. The i^{th} event in a state interval in process p (denoted $e_{p,i}$) always executes before the $i+1^{\text{st}}$ event in the same interval: $e_{p,i} \xrightarrow{\text{SO}} e_{p,i+1}$. Note that the last event in the interval is not ordered before the first event in the following interval by $\xrightarrow{\text{SO}}$. The $\xrightarrow{\text{M}}$ relation shows the message deliveries: $a \xrightarrow{\text{M}} b$ means that a sent a message that b received (and we write $a \xrightarrow{\text{M}} b$ to denote the message).

Figure 2 illustrates for an example checkpointed program execution the *execution graph*, an equivalent representation of $P = \langle E, \xrightarrow{\text{RD}} \rangle$. The events in E are the nodes of this graph, and $a \xrightarrow{\text{RD}} b$ iff a path exists from a to b in the graph. In our examples, we implicitly assume that intra-process edges are directed from left to right (but we omit the arrowheads from the figures). Note that there is no edge entering any of the checkpoints (denoted by the "×"s), indicating that the checkpoints are not replay dependent on any event.

4. Domino-Free Replay

Our goal is to be able to replay any single state interval in the execution. We can easily replay multiple intervals by simply concurrently replaying several single in-

tervals at once (discussed in Section 5.2). To replay some interval $S_{p,i}$, we must restart the execution of process p from checkpoint $C_{p,i}$, and then ensure that the messages $S_{p,i}$ originally received (during execution) are reproduced (during replay). To reproduce a message we can either log it initially during execution (so we can retrieve it from the log during replay), or we can recompute its contents by also replaying the state interval that sent it. In this section we present results showing which messages need to be logged and which can be timely recomputed. We log only the *domino-effect* messages, which if not logged could cause arbitrarily many other intervals to be replayed (to recompute messages) when satisfying a replay request, thus slowing replay. We first define precisely which messages need to be reproduced to allow any state interval to be replayed, and characterize what domino-effect-free replay means. We then prove that by logging enough domino-effect messages, the replay of any state interval will never exhibit a domino effect. In Sections 5 and 6 we present an on-line algorithm for detecting and logging these messages and show its effectiveness.

To replay a state interval $S_{p,i}$, any message $a \xrightarrow{M} b$ on which an event in $S_{p,i}$ depends must either be recomputed during replay or be initially logged during execution. To recompute messages during replay, we must choose a *restart line*, checkpoints at which to restart each process (illustrated in Figure 2). Any message on which $S_{p,i}$ depends that is sent from the right of the restart line will be recomputed during replay (and thus need not be logged). Any message on which $S_{p,i}$ depends that crosses the line from left to right must initially be logged so it can be retrieved from the log during replay. Otherwise, it will be unavailable during replay, since its sender (which is to the left of the line) is not being re-executed. If $S_{p,i}$ does not depend on a message $a \xrightarrow{M} b$, it means that $S_{p,i}$ either depends in no way on the state interval X that sent the message, or that it depends only on events in X *before* $a \xrightarrow{M} b$ was sent.

Figure 2 illustrates these cases. If we wish to replay interval $S_{1,3}$, we need to reproduce messages m4 and m5. They can be either logged or recomputed. If we choose the restart line as shown, message m5 will be recomputed (by also replaying interval $S_{2,3}$), and message m4 will be logged (message m4 is a domino-effect message (described below) and is logged by our algorithm). Since $S_{2,3}$ is also being replayed, the message it requires (m7) must also be reproduced, and is recomputed by replaying $S_{3,2}$. Even though m6 crosses the restart line, it is *not* logged, since $S_{1,3}$ does not depend on m6 — it is received by $p3$ *after* $p3$ sends the message on which the other intervals depend (m7). In general, after a process is

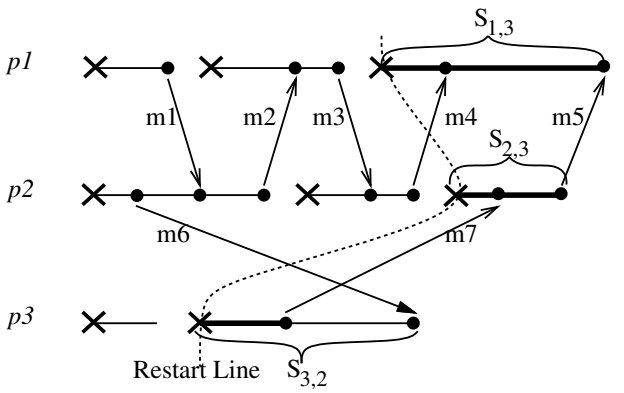

Figure 2. Restart line for replaying interval $S_{1,3}$. The "×"s indicate checkpoints. Only message m4 is logged, and only bold parts of state intervals are re-executed.

restarted, it may be stopped and then restarted from a later checkpoint to skip over receives of messages that are not needed. This start/stop replay allows us to avoid logging some messages that need not be reproduced (such as m6). The bold portions of processes $p1$, $p2$, and $p3$ in Figure 2 are re-executed to replay $S_{1,3}$.

To decide where to place the restart line, we attempt to log as few messages as possible as long as messages can be quickly recomputed. If we were to always recompute the messages on which $S_{p,i}$ depends, the *domino effect* may force arbitrarily many state intervals in the past to be re-executed[9]. Our approach is to log enough messages to provide *domino-free replay*. Figure 3 illustrates a domino effect when replaying $S_{1,3}$. If no messages are initially logged, we must replay all the state intervals in $p1$ and $p2$ to recompute message m4; interval $S_{2,2}$ requires a message from $S_{1,2}$, which requires a message from $S_{2,1}$, which finally requires a message from $S_{1,1}$. In general, the domino effect can force replay to start at the very beginning of the execution. The *domino-effect messages* cause this effect by introducing replay dependences from earlier state intervals in a process to later intervals.

Definition 4.1

A *domino dependence* is a replay dependence from an event in an interval $S_{p,i}$ to an event in a later interval $S_{p,j}$ in the same process ($j > i$).

The *domino-effect messages* are those messages that lie on any path in the execution graph that comprises a domino dependence.

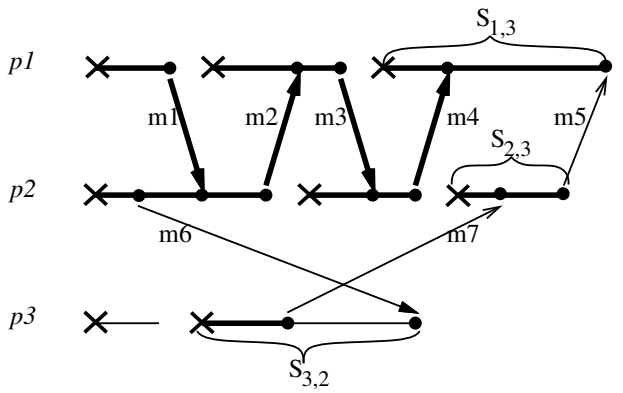

Figure 3. Domino effect when replaying interval $S_{1,3}$, causing $p1$ and $p2$ to start at the beginning of the execution. Domino-effect messages are in bold, and bold parts of state intervals must be re-executed.

Intuitively, the domino-effect messages are those causing replay dependences that make it impossible to replay $S_{p,i}$ without also re-executing earlier state intervals in the same process. Our goal is to provide *domino-free* replay.

Definition 4.2

We say that a state interval $S_{p,i}$ will have *domino-free replay* iff it can be replayed without reexecuting any previous state interval $S_{p,j}$ $(j < i)$ in the same process.

The execution P will have domino-free replay iff all of its state intervals will have domino-free replay.

Theorem 1 states that logging enough domino-effect messages to break the domino dependences guarantees that replay is domino-free (proofs appear in the Appendix). In Section 6 we show that domino-free replay allows any state interval to be replayed by typically reexecuting at most one or two state intervals in each other process.

Theorem 1 (Domino-Free Replay Theorem)

We say that a replay dependence $a \xrightarrow{M} b$ is *broken* if the message $a \xrightarrow{M} b$ is logged. By logging enough domino-effect messages to break all domino dependences, the execution will have domino-free replay.

If we consider the execution graph, this theorem states that if we remove enough edges from the graph (by logging messages) to break all paths from a state interval to future state intervals in the same process, then any state interval can be replayed with no domino effect.

Although the next theorem states that computing the minimal number of messages needed to provide domino-free replay is NP-hard, we present an on-line approximation algorithm in the next section that works by logging any message that completes a domino dependence.

Theorem 2 (Message-Logging Complexity Theorem)

Computing the minimal number of messages that must be logged to obtain domino-free replay is an NP-hard problem.

5. Adaptive Logging and Replay Algorithms

Theorem 2 shows that the problem of logging the minimal number of messages to provide domino-free replay is NP-hard. In this section, we present an on-line approximation algorithm, and discuss how to replay from the message log. Although we concentrate on replaying a single state interval, the replay scheme can be extended to replay multiple state intervals concurrently.

5.1. Adaptive Message-Logging Algorithm

We now present our message logging algorithm, shown in Figure 4. Our approach is to dynamically track the replay dependences among messages and checkpoints to determine which messages *complete* a domino-dependence, and log only these messages. A message completes a domino dependence if it is the first message that completes a path from an earlier state interval to a later interval in some process. To track dependences, each process keeps count of its current state interval and maintains a *replay dependence vector* (*RDV*). The *RDV* is

```
1: Send = the event that sent Msg;
2: if (e --RD--> Send for any event e in any earlier state
3:     interval of process p) then
4:     /* Msg completes a domino dependence */
5:     log Msg;
6: else
7:     /* Msg does not complete a domino dependence */
8:     update this process' replay dependence vector(RDV)
9:           from the RDV piggybacked on Msg;
```

Figure 4. Adaptive message logging algorithm, invoked after receiving *Msg* in process p.

similar to a vector timestamp[2] (used to maintain the happened-before relation, discussed in Section 3), except that it is a vector of state interval indices, one per process. The current *RDV* for process p contains the index of the earliest state interval in every other process on which p has a replay dependence. Process p's *RDV* entry for p itself always contains its current state interval number. Each process appends its *RDV* onto outgoing message, and upon receiving a message, updates its *RDV* as described below. After taking a checkpoint, a process reinitializes its *RDV* by nullifying all components except the one for itself, indicating that the newly taken checkpoint is not replay dependent on any other event. At any point, the *RDV* shows the checkpoints that comprise the current restart line.

To determine whether a message completes a domino dependence, the algorithm in Figure 4 is invoked after each receive operation. When process p receives a message, it checks whether the sender is replay dependent on any previous state interval in p (lines $1-3$ in Figure 4). This check is performed by determining if the p^{th} component of the piggybacked vector is less than the current state interval number of p. If so, then the sender depends on some event in an earlier interval of p; the message thus completes a domino dependence and is logged. Otherwise, the message can be reproduced during replay by properly choosing the restart line (discussed later) and is not logged. When the message does not complete a domino dependence, the *RDV* must be updated to reflect the new dependence it introduces; the *RDV* is updated by a component-wise minimum with the vector appended to the message (except that if an entry for one of the vectors is null, then the element from the other is used as the result). Updating *RDV*'s by a component-wise minimum ensures that they always show the current restart line.

To illustrate our algorithm, consider the execution of Figure 2, which is reproduced in Figure 5 with messages annotated with their piggybacked *RDV*'s. We consider why messages $m2$ and $m4$ are logged. When $p1$ takes its first checkpoint, $C_{1,1}$, its *RDV* is initialized to [1,–,–]. When $p1$ sends $m1$, it appends this *RDV* to the message. When $p2$ receives $m1$, the algorithm tests whether the second element of the piggybacked *RDV* is less than the current checkpoint interval number. It is not (since it is null), meaning that the sender does not depend on an earlier interval in $p2$, so message $m1$ is not logged. Consequently, $p2$'s *RDV* is updated to [1,1,–] by taking a component-wise minimum with the piggybacked vector. The new *RDV* indicates the restart line needed for replaying interval $S_{2,1}$ up to the current event. When $p1$ receives $m2$, the first component of the piggybacked *RDV* is 1, less than $p1$'s current state interval number (2). The

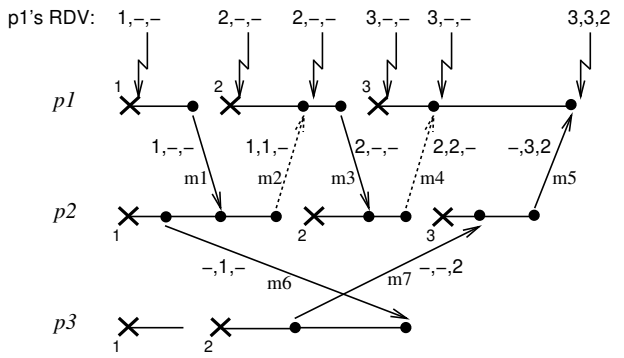

Figure 5. Message logging algorithm on example in Figure 2. Each dashed message completes a domino dependence and is logged. The replay dependence vector (RDV) piggybacked on each message is shown.

send event thus depends on an earlier state interval in $p1$ ($S_{1,1}$, by the dependence introduced by $m1$), so $m2$ is logged. Since $p1$ will not depend during replay on $m2$ (because it is logged), $p1$'s *RDV* is left unchanged.

Similarly, our algorithm logs $m4$, the message that completes the domino dependence from $S_{1,2}$ to $S_{1,3}$. Messages $m2$ and $m4$ (the dashed edges in Figure 5) are the only messages logged. After removing the dashed edges from the execution graph, we can see there is no path from an event in an earlier state interval to an event in a later interval of the same process. By Theorem 1, we can replay all state intervals with no domino effect. Note that although our algorithm logs the minimal number of messages in this example, in general it is not optimal when a domino-effect message lies on more than one path (where logging one message can break many domino dependences).

5.2. Execution Replay

We now outline how to replay a state interval using the message logs generated by our algorithm. Our algorithm guarantees that, for any state interval, all messages on which it depends have either been logged or can be regenerated during replay with no domino effect. The crux of replay is computing the proper restart line and determining which state intervals must be reexecuted. As discussed in Section 2, we must also ensure replay is deterministic by forcing messages to be delivered in the same order as during the original execution. Enforcing this order requires using the message dependence trace (which we assume is always recorded), and for this we rely on

past schemes and do not address this problem here. We focus only on how to determine which state intervals need be replayed. We consider how to replay a single state interval, $S_{p,i}$, although multiple state intervals can easily be replayed concurrently using the same approach.

To determine where to start replay for $S_{p,i}$, the restart line is computed by constructing the execution graph and locating the earliest checkpoint in each process on which any event in $S_{p,i}$ depends. Process p itself is always restarted from checkpoint $C_{p,i}$. The execution graph is constructed from the message dependence trace, which shows to which event each message was delivered (discussed in Section 2).

Once the restart line is known, we must then compute which portions of the state intervals to the right of the line should be reexecuted. Recall that only those portions of intervals on which $S_{p,i}$ depends need be reexecuted (as illustrated in Figure 2), and this start/stop replay allows us to avoid logging messages that $S_{p,i}$ does not require (such as message m6 in Figure 2). We must reexecute at least part of each interval on which $S_{p,i}$ depends, and these parts can easily be computed by examining the execution graph; we find the last event in each interval from which a path to an event in $S_{p,i}$ exists (this event will always be a send). During replay each such interval is started from its checkpoint, and stopped (by setting a local breakpoint) at this send. Reexecuting all such intervals will ensure that all messages required by $S_{p,i}$ (and not logged) will be recomputed.

Note that sometimes a message will be recomputed that no interval requires (e.g., a message sent to an interval on which $S_{p,i}$ does not depend). These unneeded messages pose no problem since the mechanism for forcing the original message delivery order (to keep replay deterministic) will ensure that they are never delivered to the wrong receive operation.

To illustrate replay, suppose that interval $S_{1,3}$ in the example of Figure 2 is to be replayed. The restart line is shown, and consists of the earliest checkpoints on which $S_{1,3}$ depends. Note that m4 is logged, so $S_{1,3}$ does not depend on $C_{2,2}$ for replay. Replay is begun from the restart line, and all of interval $S_{2,3}$ is reexecuted, since $S_{1,3}$ depends on its last event (which sends m5). However, interval $S_{3,2}$ is stopped after sending m7, since $S_{1,3}$ depends on no subsequent events in this interval.

Although starting and stopping the replay incurs some overhead, it allows us to avoid logging unnecessary messages and further reduce run-time overhead. The experiments in Section 6 indicate that not many intervals usually need be reexecuted to satisfy a replay request, so the starting and stopping cost should be low. In addition, skipping unneeded portions of the execution further reduces replay response time.

This method of replaying a single interval can be directly extended to replay multiple intervals concurrently (e.g., to replay up to some consistent global state). Given a set of intervals to be replayed, we form the restart line by computing a minimum over the restart lines for those intervals. Any message not logged on which one of the intervals depends will be regenerated.

6. Experimental Results

To measure the effectiveness of our adaptive message logging strategy, we analyzed executions of several message-passing programs. We measured the percentage of messages our algorithm logged, and the number of state intervals needing reexecution to satisfy a replay request. These measures indicate how well we adaptively reduce trace size, and how effectively domino-free replay keeps replay time low. We computed these measures as we varied the frequency with which each process takes a checkpoint. To compare the results as this frequency varied, we simulated our algorithm from a message trace of a single execution of each program. Since the test programs were nondeterministic, simulation was necessary to allow us to vary checkpoint frequency and still analyze the same execution (otherwise, different runs would produce different executions). Our experiments show that only $1-10\%$ of the messages were typically logged, a substantial reduction over past schemes which log the contents of every message. In addition, we found that only one to two state intervals in each process usually required reexecution to satisfy any replay request, indicating that logging the domino-effect messages is effective at keeping replay times low.

Our first experiment investigated what percentage of messages are typically logged. We analyzed five test programs (provided by colleagues) on a 128-node Intel iPSC/860 hypercube. Each program was run once on a 16-node subcube with an instrumented message-passing library to provide traces for simulation. These programs ran between 10 and 934 seconds and passed between 300 and 370,000 messages. Since the checkpoint frequency has an impact on our algorithm's performance (e.g., no messages are logged if no checkpoints are taken), we varied the period with which each process takes its checkpoint as we simulated the algorithm. We varied the period from 1% of the total execution time to 50% (taking a checkpoint every 1% results in 100 checkpoints per process). Data for checkpoint periods longer than 50% are omitted since then only two checkpoints are taken (per process) and the data for 50% is representative. Figure 6

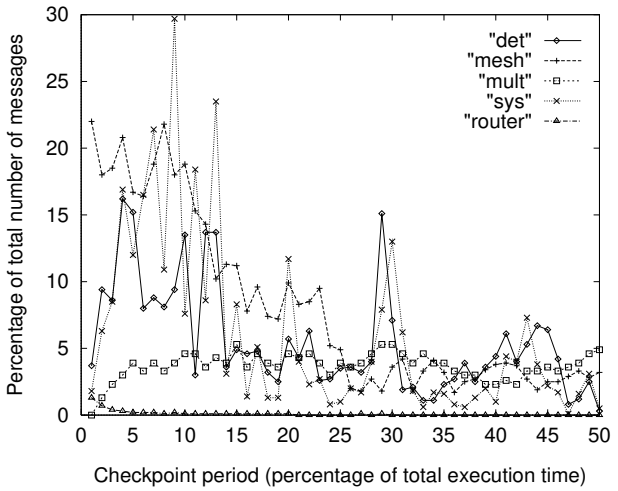

Figure 6. Percentage of messages logged for varying checkpoint intervals.

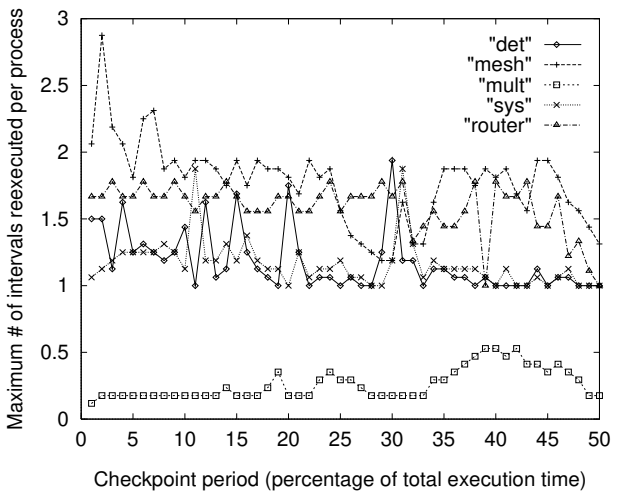

Figure 7. Maximum state intervals (per process) that must be reexecuted to satisfy any replay request.

shows the percentage of messages logged as the checkpoint period varies. For most cases, our algorithm logs less than 10% of the messages. The performance tends to improve as the checkpoint period increases, indicating the decreased likelihood of domino-effect messages as the number of distinct state intervals decreases. In practice, checkpoints should be taken as far apart as possible (to reduce checkpointing overhead), and the data points for small checkpoint periods do not represent cases that would appear in practice. We thus expect our algorithm to log typically $1-10\%$ of the messages. For *router* (the longest-running program), less than 0.1% of the messages were usually logged.

Our second experiment studied the response time of replaying a state interval using the message logs. We computed the maximum number of state intervals (per process) that required reexecution to satisfy any replay request (as discussed in Section 5.2). This maximum indicates at most how long a typical replay request will take, and shows how effectively domino-free replay keeps this time low. Figure 7 shows the results as the checkpoint period varies. In almost all cases, only $1-2$ state intervals (per process) required reexecution, even when the checkpoint period was small (and thus when the execution was divided into many state intervals).

We conclude that our strategy of logging enough messages to eliminate the domino effect is effective. It not only requires logging a small percentage of messages, but the absence of domino effects results in (essentially) bounded replay time.

7. Conclusions and Future Work

In this paper we show how a checkpoint-and-replay scheme can *adaptively* decide which messages to log, instead of logging every message as proposed by past schemes. By formulating the problem in terms of reachability in a graph, we prove that only messages causing domino-dependences need be logged to provide domino-free replay. Although computing the minimal set of messages that must be logged is an NP-hard problem, we present an on-line approximation algorithm. Our algorithm tracks dependences among messages and logs the first message that completes each domino dependence. Experiments indicate that our algorithm typically logs only $1-10\%$ of the messages. In addition, these logs are sufficient to replay any state interval in a process, and normally require only one or two additional intervals in each other process to also be reexecuted. Our work thus achieves both low message logging and quick replay.

Future work includes combining adaptive message logging with adaptive message dependence tracing, investigating adaptive checkpointing, and improvements to our on-line algorithm. First, we assumed that all dependences among messages were traced (e.g., by writing a trace for every message that shows to which event it was delivered). Netzer and Miller have shown how to significantly reduce the size of dependence traces (by 1 to 2 orders of magnitude) by tracing only *racing* messages, which can be located on-the-fly[7]. We wish to combine this approach with adaptive message logging. Second, we assumed that processes periodically take checkpoints. We are investigating how checkpointing *and* message logging can adapt to the execution. Finally, better on-line approx-

imation algorithms for logging may exist.

Acknowledgements

We thank Prith Banerjee of the Univ. of Illinois for the router program, and Mark Hill of the Univ. of Wisconsin–Madison for the other test programs.

References

[1] K. Mani Chandy and Leslie Lamport, ‘‘Distributed Snapshots: Determining Global States of Distributed Systems,’’ *ACM Trans on Comp Syst* **3**(1) pp. 63-75 (Feb, 1985).

[2] C. J. Fidge, ‘‘Partial Orders for Parallel Debugging,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[3] Arthur P. Goldberg, Ajei Gopal, Andy Lowry, and Rob Strom, ‘‘Restoring Consistent Global States of Distributed Computations,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 144-154 Santa Cruz, CA, (May 1991).

[4] David B. Johnson and Willy Zwaenepoel, ‘‘Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,’’ *Proc. of the 7th Annual ACM Symp. on Principles of Dist Computing*, (1988).

[5] Leslie Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *CACM* **21**(7) pp. 558-565 (July 1978).

[6] Thomas J. LeBlanc and John M. Mellor-Crummey, ‘‘Debugging Parallel Programs with Instant Replay,’’ *IEEE Trans. on Computers* **C-36**(4) pp. 471-482 (April 1987).

[7] Robert H.B. Netzer and Barton P. Miller, ‘‘Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs,’’ *Supercomputing ’92*, pp. 502-511 Minneapolis, MN, (November 1992).

[8] Robert H.B. Netzer, ‘‘Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1-11 San Diego, CA, (May 1993).

[9] B. Randell, ‘‘System Structure for Software Fault Tolerance,’’ *IEEE Trans. on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

[10] R. E. Strom and S. Yemini, ‘‘Optimistic Recovery in Distributed Systems,’’ *ACM Trans. on Computer Systems* **3** pp. 204-226 (August 1985).

[11] Kuo-Chung Tai and Sanjiv Ahuja, ‘‘Reproducible Testing of Communication Software,’’ *IEEE COMPSAC ’87*, pp. 331-337 (1987).

[12] Y. M. Wang and W. K. Fuchs, ‘‘Optimistic message logging for independent checkpointing in message-passing systems,’’ *IEEE Symp. on Reliable Distributed Syst.*, pp. 147-154 (Oct 1992).

[13] Larry D. Wittie, ‘‘Debugging Distributed C Programs by Real Time Replay,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 57-67 Madison, WI, (May 1988).

[14] Jian Xu and Robert H.B. Netzer, ‘‘Adaptive Independent Checkpointing for Reducing Rollback Propagation,’’ *IEEE Symp. on Parallel and Distributed Processing*, Dallas, TX, (Dec 1993).

Appendix. Proofs of Theorems

Theorem 1 (Domino-Free Replay Theorem)
By logging enough domino-effect messages to break all domino dependences, the execution will have domino-free replay.

Proof. Let $\xrightarrow{\text{RD'}}$ be the replay dependence relation that results by removing all dependences broken by message logging. We will prove that for all state intervals $S_{p,i}$, there exists a restart line that passes through $C_{p,i}$, and one checkpoint in each other process, such that no message $a \xrightarrow{\text{M}} b$ crosses it left to right where $a \xrightarrow{\text{RD'}} x$ (for some x in $S_{p,i}$). That is, each message $a \xrightarrow{\text{M}} b$ crossing the line left to right is either logged or does not affect $S_{p,i}$. The existence of such a restart line implies that $S_{p,i}$ will have domino-free replay since no previous state intervals in the same process then need to be replayed.

To prove that such a restart line exists, consider the execution graph with edges corresponding to all logged messages deleted. Since all domino dependences are broken, this graph has no path from an event in any state interval $S_{p,i}$ to an event in a later interval $S_{p,j}$ $(j > i)$ in the same process. Below we show that for each interval $S_{p,i}$, a restart line exists such that no path crosses the line left to right from any event a to an event in $S_{p,i}$.

Consider any state interval $S_{p,i}$. We construct its restart line by including $C_{p,i}$ and for each other process, q, the earliest checkpoint $C_{q,k}$ in q on which any event x in $S_{p,i}$ depends (i.e., $C_{q,k} \xrightarrow{\text{RD}^*} x$). We claim that no path exists from an event a to the left of the line to any event x in $S_{p,i}$ (which must be to the right of the line, since the line contains $C_{p,i}$). To establish a contradiction, assume that such a path exists, and further assume that a is the only event in the path to the left of the line (i.e., the edge out of a crosses the line).

Case 1: a belongs to process p. In this case, a is in a state interval that precedes $S_{p,i}$, so the path from a to x is a domino dependence. However, the existence of this dependence contradicts the assumption that all domino dependences were broken by logging messages.

Case 2: a belongs to process q, where $q \neq p$. Recall that the restart line was constructed to contain $C_{q,k}$, the earliest checkpoint in q on which $S_{p,i}$ depends. However, the path from a to $S_{p,i}$ implies that $S_{p,i}$ depends on a, and because a is to the left of the restart line, $C_{q,k}$ is not the *earliest* checkpoint on which $S_{p,i}$ depends, a contradiction.

Thus, no path can exist from an event a to the left of the line to any event in $S_{p,i}$. QED

Theorem 2 (Message-Logging Complexity Theorem)
Computing the minimal number of messages that must be logged to break all domino dependences is an NP-hard problem.

Proof. We use a reduction from the vertex cover problem, known to be NP-complete: given an undirected graph, $G = (V, E)$, does G have a vertex cover with k or fewer vertices? A vertex cover is a subset V' of the vertices such that every edge is connected to some vertex in V'. Given G, we reduce the problem of determining whether it has a vertex cover with k or fewer vertices to the problem of determining whether all domino dependences in a program execution, P, can be removed by logging k or fewer messages.

From the graph G we construct P as follows. For every $v \in V$, P contains a two-process gadget, $gad(v)$, and two messages between $gad(a)$ and $gad(b)$ iff an edge exists between a and b in G, as illustrated in Figure 8. The right-most process performs two checkpoints; the left-most performs none. Each gadget consists of six message operations (three per process), and one message sent between its two processes. When two nodes a and b are connected, we construct a message from node 4 of $gad(a)$ to node 1 of $gad(b)$, and another message from node 5 of $gad(b)$ to node 6 of $gad(a)$ (or, we could construct them from node 4 of $gad(b)$ to node 1 of $gad(a)$, and from node 5 of $gad(a)$ to node 6 of $gad(b)$, and we arbitrarily pick one of these two directions). These messages introduce a path from state interval 1 (S_1) to state interval 2 (S_2) in the right-most process of the gadget for b. If a node has more than one connecting edge, we add additional events (just below nodes 1,4,5 and 6) to send and receive these messages. Thus, whenever an edge connects two nodes, there is a path from S_1 to a later interval (S_2) in some process.

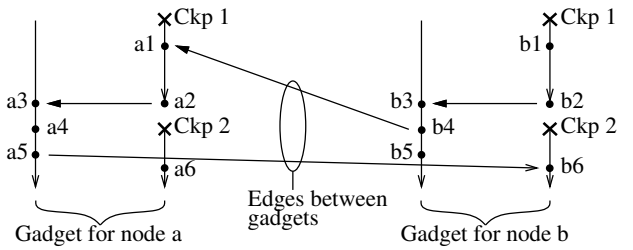

Figure 8. Gadgets, for nodes a and b, and intergadget messages (that exist iff a and b are connected in G).

We claim that G has a vertex cover of k or fewer vertices iff all domino dependences in P can be removed by logging k or fewer messages.

If Part. Assume that all domino dependences in P can be removed by logging k or fewer messages; i.e., by deleting k or fewer edges in P's execution graph, we remove all paths from S_1 to S_2 in any process. Let n be the number of deleted edges. We claim that these paths can be removed by deleting only the message inside each of n gadgets without deleting any inter-gadget messages. If a message from $gad(a)$ to $gad(b)$ is deleted, we can instead delete the message inside $gad(b)$ (or $gad(a)$), since it lies on more paths. Thus, we can assume that all paths from S_1 to S_2 in any process can be removed by logging n intra-gadget messages (one per gadget). G then has a vertex cover consisting of the nodes corresponding to these gadgets. These nodes form a vertex cover since we must have logged at least one message in each pair of connected gadgets.

Only-If Part. Assume G has a vertex cover of k or fewer nodes. We can delete k or fewer edges in P's execution graph and remove all paths from S_1 to S_2 in any process. Since gadgets are connected exactly when their corresponding nodes are connected, any two connected gadgets will have at least one of their messages logged. Deleting the edges corresponding to these logged messages will ensure no path exists from S_1 to S_2 in any process. QED