Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs

Robert H. B. Netzer, Sairam Subramanian
and Jian Xu

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-19
April 1994

Appears in Intl Conference on Distributed Computing Systems, Poznan, Poland, June 1994

Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs

Robert H. B. Netzer rn@cs.brown.edu
Sairam Subramanian ss@cs.brown.edu
Jian Xu jx@cs.brown.edu

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

Abstract

Debugging long-running, nondeterministic message-passing parallel programs requires incremental *replay, the ability to exactly replay selected parts of an execution. To support incremental replay we must log enough messages and checkpoint processes often enough to allow any requested replay to complete quickly. We present an* adaptive *tracing strategy to keep the message-logging overhead down. We let the user specify a bound on the maximum time any replay request is allowed to take. Our algorithm tracks what each processes' critical path will be* during a replay *and logs enough messages to ensure the critical path will never exceed the bound. Overhead is kept low by not logging messages that can be recomputed during a replay. Experiments indicate that we log about 0.1−5% of the messages while still providing a reasonable bound on any replay.*

1. Introduction

Program debugging requires repeated execution, where more information about a bug is gathered each iteration. However, the nondeterminacy in message-passing parallel programs can turn bugs into moving targets: reexecuting the program does not always reproduce the bugs. Obtaining repeatability requires tracing the execution's messages, using the trace to replay as needed. In addition, it is crucial to guarantee response time to replay requests — parallel programs are so long running that it takes too long to restart an execution every time from the beginning. Processes can be periodically checkpointed to allow *incremental* replay, the restarting from intermediate checkpoints. To bound the replay response time, we present an *adaptive* message logging strategy that decides at run-time which messages can be quickly recomputed during replay and thus need *not* be logged. We let the user specify an upper bound on the amount of time they can wait for a replay to finish. Our algorithm adapts to the execution's message-passing patterns by tracking critical paths on-line and logging only messages that would allow a *replay*'s critical path to grow beyond the given bound. Experiments show that $0.1 - 5\%$ of the messages are typically logged. Our scheme simultaneously lowers logging overhead while guaranteeing a bound on the time required to replay any part of the execution.

Our approach to supporting incremental replay is to independently 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 messages that can be easily recomputed during the replay. 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 can be quickly recomputed (during the replay) by reexecuting the processes that sent them. In a previous paper we presented a heuristic to identify which messages can be quickly recomputed during replay and need not be logged[4]. The problem with this technique is that it can log more than necessary and cannot always keep the replay time bounded. In this paper we directly address the replay time problem and guarantee a replay bound.

Our main results are two adaptive algorithms that log enough messages to allow the replay of any interval of the execution to complete within a given time bound. We let the user specify a bound on how long the critical path of any replay is allowed to be. A replay is a reexecution of an interval between two checkpoints in some process. Our algorithms track during execution what each interval's critical path will be during replay. When a message is received that would require so much time to reproduce that the critical path would exceed the specified bound,

This research was partly supported by ONR Contract N00014-91-J-4052 (ARPA Order 8225), NSF grant CCR-9309311, and NSF PYI award CCR-9157620 together with PYI matching funds from Thinking Machines, Xerox, and Honeywell Corporations.

the message is logged. A logged message does not contribute to the critical path since it is retrieved directly from the trace during replay; its sending process need not be reexecuted. Our first algorithm provides a *guarantee* on the replay time but at the expense of requiring two vectors of size P (where P is the number of processes) to be piggy-backed on all messages. Our second algorithm is a simplification which piggybacks only a single integer. Experiments show that in practice it performs just as well as the first algorithm but without the overhead, usually logging $0.1-5\%$ of the messages.

This work is part of our larger effort on developing adaptive tracing strategies for debugging[3, 4].

2. Motivation and related work

Checkpointing and message logging have been used in many applications including parallel debugging[2, 8], fault-tolerant computing[7], and other areas[1]. Checkpointing supports incremental replay by saving each process' state periodically so that reexecution can start from a checkpoint instead of the execution's beginning. However, checkpointing alone is not enough. When replaying a portion of some process, we must ensure that it receives during replay the same messages it received during the original execution. To provide these messages we can either log them initially (during execution) so they are available during replay, or we can avoid logging by reproducing them during replay (by reexecuting portions of the processes that sent them). The difficult aspect of supporting incremental replay is balancing these two choices. Below we outline several options for addressing this problem and discuss what approaches past work has taken.

One simple approach proposed in the past is to log all the messages[2, 8]. However, the run-time overhead of this approach is prohibitive. On modern machines, executions of even moderate length can pass gigabytes of data in messages. Another approach at the other extreme is to log nothing and reproduce every message needing during a replay. The problem with this approach is that long chains of messages may need to be reproduced, making the replay too long. For example, consider replaying interval $S_{1,3}$ in the execution shown in Figure 1. During the replay we need to supply $S_{1,3}$ with message $m4$. If no messages were logged during the execution this leads to a chain of dependencies requiring us to replay all the intervals (to regenerate message $m4$) in order to replay $S_{1,3}$.

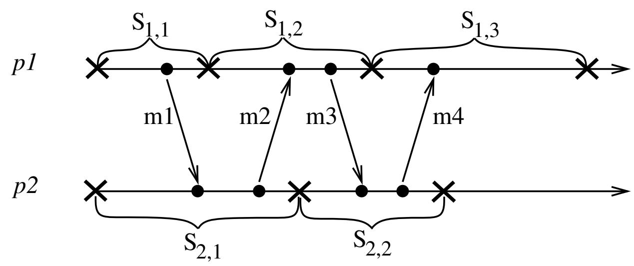


Figure 1. A long dependence chain that must be recreated to replay $S_{1,3}$. The $\times$'s are checkpoints.

To balance the message logging overhead and the replay response time, we require a logging mechanism that only logs those messages that cannot be quickly reproduced due to long dependence chains. The type of scheme we can use depends upon the checkpointing strategy we adopt. Two general choices exist: *coordinated* and *independent* checkpointing. To take a coordinated checkpoint, a distributed algorithm is run that coordinates all processes in saving their states1. Messages that cross the line formed by connecting the i^{th} checkpoints of each process are also detected and logged. The main disadvantage of coordinated checkpointing is that it gives processes no autonomy, forcing them all to checkpoint at about the same time and such a scheme does not adapt well to the execution's message passing patterns. For some message-passing patterns, many messages can be logged. In addition, forcing all processes to write their checkpoints to disk at about the same time degrades the I/O-performance of the program.

In contrast, independent checkpointing lets each process checkpoint on its own, thus avoiding the overhead of a coordination algorithm. It can adapt to the execution's message-passing patterns by selectively logging only those messages that cannot be quickly reproduced. Furthermore, it can also be used with other adaptive schemes (like *incremental* checkpointing) since each process is free to checkpoint at opportune moments[5].

In a previous paper we presented a method that adaptively logs messages to cut what we call *domino dependences* (chains of messages between two checkpoints of the same process see Figure 1 for an example)[4]. Although this strategy is effective at reducing message logging (only $1-10\%$ of messages are typically logged), its disadvantage is that it does not directly address the trace-size *vs* replay-time balance. On one hand, some domino dependences are short and could be quickly recomputed during replay; thus logging all domino dependences is too conservative. On the other hand, even if all domino dependences are eliminated (by logging), the remaining chains of messages could still be long enough to cause an unacceptable delays during replay.

To directly address the problem, our goal in this paper is to let the user specify the amount of time they are willing to let any replay take, and exploit this given time bound in two ways. First, we log enough messages to ensure that any replay can finish within this bound. Second, we take advantage of the bound by logging *only* enough messages to meet the bound, letting most of the messages be recomputed during replay.

In the fault tolerance area, work has been done to avoid the domino effect during the recovery from faults. However, this work does not solve the incremental replay problem for debugging. Recovering from faults requires only the ability to replay from the most recent checkpoint, and past work has focused on discarding old messages and checkpoints[6]. In contrast to recoveries, replays are requested often, and we must be able to quickly replay *any* part of the execution (old messages and checkpoints cannot be discarded).

3. Model

First we outline a notation for representing program executions. A *checkpointed program execution* is a pair, $P = \langle E, \xrightarrow{\mathrm{RD}} \rangle$, where E is a finite set of *events* and $\xrightarrow{\mathrm{RD}}$ is the *replay dependence* relation defined over E. The graphical representation of P is called the *replay dependence graph* (RDG); its nodes correspond to the events in E, and its edges represent the $\xrightarrow{\mathrm{RD}}$ relation.

An event represents the execution instance of a send, receive, or checkpoint operation. The *interval* $S_{p,i}$ represents all events and computation performed in process p between the i^{th} and $i+1^{\mathrm{st}}$ checkpoints. Intra-process edges in the *RDG* represent computation performed between the connected events. Some edges are labeled with non-negative weights to represent the total time taken by the action represented by the edge. Weights on computation edges indicate their cpu time; weights on messages indicate the delivery time (although for simplicity we implicitly assume 0 weight for messages in our examples). We use the weights to track critical paths; the weight of any path is the cpu + message-delivery time required to perform the computation on the path. We also assume an upper-bound of T time in between any two checkpoints in the same process. Since the user is specifying the maximum allowable time for a replay, it is reasonable to know this maximum inter-checkpoint time (it must be less than the allowable replay time).

The replay dependence relation, $\xrightarrow{\mathrm{RD}}$, 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., $a \xrightarrow{\mathrm{RD}} b$) iff a must be reproduced during reexecution before b can be reexecuted, because a precedes b in the same interval, or because a sequence of *unlogged* messages was sent from a (or a following event) to b (or a preceding event). The $\xrightarrow{\mathrm{RD}}$ relation is defined as the irreflexive transitive closure of two other relations: $\xrightarrow{\mathrm{RD}} = (\xrightarrow{\mathrm{SO}} \cup \xrightarrow{\mathrm{M}})^{+}$. The i^{th} event in an interval in process p (denoted $e_{p,i}$) always executes before the $i+1^{\mathrm{st}}$ event in the same interval: $e_{p,i} \xrightarrow{\mathrm{SO}} e_{p,i+1}$. Note that the last event in the interval is not ordered before the first event in the following interval by $\xrightarrow{\mathrm{SO}}$ (since $S_{p,i}$ can be immediately restarted from $C_{p,i}$). The $\xrightarrow{\mathrm{M}}$ relation shows where the *unlogged* messages are delivered: $a \xrightarrow{\mathrm{M}} b$ means that a sent a message that b received, and that the message was *not* logged (we write $a \xrightarrow{\mathrm{M}} b$ to denote the message). This definition differs slightly from that presented in our earlier work[4]. Messages that are logged can be retrieved from the log during replay, so they do not introduce replay dependences.

Figure 2 shows an example *RDG*. Intra-process edges represent the computation in between events (these edges are directed left to right but we omit the arrowheads), and their weights denote the cpu time required. A path exists from a to b in the graph iff $a \xrightarrow{\mathrm{RD}} b$. We draw logged messages as dashed lines to indicate that they introduce no replay dependences. Note that no edges enter any of the checkpoints, as checkpoints are not replay dependent on any event.

4. Problem formulation

We now formulate the problems of logging messages to guarantee bounded-time replay and replaying from the log. Our goal is to log few messages while still allowing any interval $S_{p,i}$ to be quickly replayed. As discussed in Section 2 we should only log messages that create long replay dependence chains and reproduce (during the replay) the ones that do not, achieving a balance between low logging and fast replay.

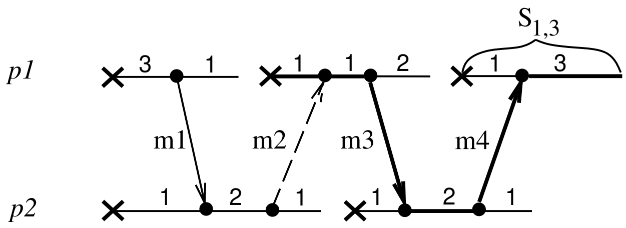


Figure 2. A replay dependence graph. Dotted lines are logged messages; weights are computation time. The critical path for replaying $S_{1,3}$ is in bold.

To guarantee a bound on the replay time we must make several assumptions about how quickly events will execute during replay. We assume that when a computation is reexecuted it will require the same cpu time as during the original execution. We also assume that a message will require the same amount of time to be delivered. The weights on the *RDG* thus show the expected *replay* time. These assumptions are not crucial to the correctness of the algorithm; if there is reason to expect different replay or message delivery times, the weights could be adjusted.

4.1. Logging to meet bounded-time replay

To see which messages must be logged, consider how long it takes to replay an interval $S_{p,i}$, given that certain messages were logged during execution. In addition to reexecuting $S_{p,i}$ itself, we must reexecute other intervals to reproduce the messages $S_{p,i}$ needs that were not logged. These are exactly the intervals that are connected by a directed path to $S_{p,i}$ in the replay dependence graph. $S_{p,i}$ depends on each of these intervals (directly or through other intervals) for a message. For example, Figure 2 shows an execution where message $m2$ is logged (and thus introduces no replay dependence). To replay interval $S_{1,3}$, messages $m3$ and $m4$ must be recomputed.

Replaying the intervals on which $S_{p,i}$ depends cannot take less time than the weight, C, of the longest directed path to $S_{p,i}$ (which we call $S_{p,i}$'s *critical path* during replay). C gives us the cpu time required to perform all the computation on the critical path; it is a lower bound on the replay time (since no two events on this path can be executed concurrently). In Figure 2, $S_{1,3}$'s critical path is 7 (shown in bold); it takes 7 time units to replay the portions of intervals on which $S_{1,3}$ depends. If message $m2$ had not been logged, $S_{1,2}$ would have a critical path of length 8 consisting of portions of $S_{1,1}$, $S_{2,1}$, and $S_{1,2}$ along with messages $m1$ and $m2$.

To establish the upper bound on the replay time, consider the total amount of computation required for $S_{p,i}$'s replay: the sum of weights of *all* edges with a path to $S_{p,i}$ (not just those on the critical path). If we let W denote the total execution time of all such intervals, and we could evenly divide this work among the P processors, the total replay time would be W/P. However, dividing the work evenly is not always possible. We may have many intervals in one process that need to be replayed but few in other processes. In the worst case the replay takes W time. The actual replay time is always between C ($S_{p,i}$'s critical path) and W.

One way to bound the replay time more tightly is to log enough messages so that the number of intervals per process that need be reexecuted is limited. If we log enough so that at most M intervals in any process ever have a path to $S_{p,i}$, then the following theorem states that the replay time is bounded by CM. We call M the *interval multiplicity* of $S_{p,i}$.

Theorem 1 (Bounded-Time Replay)
If in a *RDG* an interval $S_{p,i}$ has

(1) a critical path of size at most C (the longest path to $S_{p,i}$ has weight at most C), and
(2) an interval multiplicity of M (at most M intervals in the same process with paths to $S_{p,i}$), then

$S_{p,i}$ can be replayed in time at most CM.

To achieve this replay bound we allow the replay of a process' intervals to be interleaved. When one interval blocks to receive a message, we are free to run another interval in the same process (if one exists that must also be reexecuted). This interleaving makes efficient use of processors and allows us to achieve a quick replay bound. To get the replay bound of Theorem 1. processors cycle through their intervals executing one time unit of each (whenever possible). In such a replay scenario it can be shown that after M time units the critical path remaining is at least one unit smaller than before. Therefore after CM time units all the intervals have been replayed (we omit the proof details for lack of space). We could relax this assumption but at the cost of slower replay.

C and M must be specified by the user to reflect the maximum time they are willing to wait for any replay to complete. However, our experimental results (discussed in Section 6) show that in practice the replay bound is virtually independent of M, and setting M to be infinity results in a maximum replay time of at most $2C$. In practice the user need only specify C.

Our goal therefore is to log the minimum number of messages required to ensure that every interval has a critical path at most C, and to determine dynamically which messages to log. However, by performing a reduction from the vertex-cover problem we can show that finding the minimum set of messages to log is NP-complete and therefore intractable. In the next section we present approximation algorithms that perform well in practice.

4.2. Replaying from the log

Once the execution is over and the message logs created, we replay any interval $S_{p,i}$ as follows. Replay of multiple intervals can be handled in a similar fashion.

To replay $S_{p,i}$ we first scan the log to locate the set of intervals necessary for $S_{p,i}$'s replay (those with a directed path to $S_{p,i}$ in the *RDG*). For each such interval $S_{q,j}$ we also determine its last event (called the *requisite event*) on which $S_{p,i}$ depends. The requisite event is the last event that has a path to $S_{p,i}$. For example, to replay $S_{1,3}$ in Figure 2 we only need to reexecute $S_{1,2}$ until it sends message $m3$ and $S_{2,2}$ until it sends message $m4$.

After identifying the intervals that must be replayed, we reexecute each interval on the same processor on which it originally executed. Each processor q cycles through intervals assigned to it. When an interval blocks on a receive, q switches over to a different interval. In addition, each interval is reexecuted only up to its requisite event. The advantage of this aggressive replay is that processors can avoid wasting time blocked on receives. Interleaving makes efficient use of processors during replay and allows us to achieve a quick replay bound. Also, by replaying intervals only up to the requisite event, processors do not spend time replaying portions of the interval unnecessary for the replay. Aggressive replay could be implemented by setting up the replay of each interval as a separate process.

5. Adaptive logging algorithms

In this section we present two adaptive logging algorithms that allow any interval to be replayed within a specified time. Both algorithms track the critical path of each interval on-the-fly and log any message that would cause the critical path to exceed the user-specified bound C. The first algorithm simply tracks the critical path and does not bound the interval multiplicity of each interval. Although this algorithm does not *theoretically* provide the replay time guarantee given in Theorem 1, our experiments indicate that in practice it performs well. We then show how the algorithm can be extended (at additional run-time cost) to bound the interval multiplicity and thus guarantee the worst-case bound. In Section 6 we present experimental results indicating that the overall performance of the simpler algorithm is best.

5.1. Adaptive logging by tracking the critical path

Our first algorithm tracks in each process the critical path of the current interval as the execution progresses. After receiving a message, it determines whether *not* logging the message would increase the critical path beyond C (the allowable bound). The algorithm (shown in Figure 3) is run after either a checkpoint or synchronization event. Every intervals' critical path will stay below the desired amount. The advantage of this approach is its low run-time overhead; the critical path checks are quick numerical comparisons, requiring only a single integer to be piggybacked onto each message.

In each process p we dynamically maintain two quantities, *cp* and *remaining*, about the currently executing interval ($S_{p,i}$). The variable *cp* is the cost of $S_{p,i}$'s current critical path and is the weight of the longest path to $S_{p,i}$ in the *RDG* so far (due to past logging decisions). The variable *remaining* is an estimate of the maximum amount of time left in $S_{p,i}$ (it must be estimated since it may contribute to the critical path). It is computed by subtracting from T the time spent in the interval so far (recall that T is the maximum time any interval will take; see Section 3). These variables are updated at each checkpoint or synchronization event in $S_{p,i}$.

When p checkpoints and begins a new interval, *cp* is initialized to 0 since the new interval can be replayed starting from the checkpoint just taken.

When p issues a send operation, the current critical path size *cp* is piggy-backed onto the outgoing message. This value tells the receiving process how long it will take (during replay) to reproduce the message if it is not logged. Before sending the message, the cpu time spent since the last checkpoint or synchronization event is added to the current value of *cp* to reflect amount of computation done so far in $S_{p,i}$.

```
if (e is a checkpoint event) {
    cp = 0;
}
if (e is a send event) {
    cp = cp + time since previous event;
    remaining = T − time spent in this interval;
    piggy-back cp onto the message to be sent;
}
if (e is a receive event) {
    cp = cp + time since previous event;
    remaining = T − time spent in this interval;
    cp_m = critical path on incoming message;
    if (cp_m + remaining > C)
        log the message;
    else
        cp = max (cp, cp_m);
}
```

Figure 3. Critical-path based message logging, run after each synchronization or checkpoint event e. T is the maximum time ever between checkpoints. C is the user-specified bound on the critical path.

After p receives a message, it uses its current critical path size cp and the information piggy-backed on the message to decide whether to log the message. Let cp_m be the critical path size piggy-backed onto the message; cp_m was the critical path of the sender's interval just before the message was sent (and indicates at least how long it would take to reproduce the message). The message will be logged if $cp_m + remaining > C$. Recall that *remaining* is an upper bound on how much time is left before interval $S_{p,i}$ finishes (i.e., before the next checkpoint); it needs only to be an upper bound and need not be completely accurate (although more messages than necessary will be logged if it is too large). In the worst case the entire remainder of the interval will be spent performing useful computation (and not blocked on a receive), so this remainder might contribute to the critical path. Thus, if $cp_m + remaining > C$, then *not* logging the message could potentially increase the critical path size beyond C. If the message is not logged we update the value of cp; the current interval's critical path is now the maximum of cp (the time required to replay the interval up to the receive) and cp_m (the time required to reproduce the message just received). Figure 2 gives an example of the way our algorithm would work. Here C was set to be equal to 7 and T was equal to 4. Message $m2$ was logged because otherwise we would have had a critical path of size 8 or more.

5.2. Tracking interval multiplicities

The above algorithm does not limit the interval multiplicity, M; it allows $S_{p,i}$ to be replay dependent on any number of intervals from any one process. Fortunately, Section 6 shows that algorithm performs very well: in practice the amount of logging and the replay time are independent of M. However, for completeness we show how to extend our first algorithm to limit M. Our second algorithm achieves the guaranteed replay time of CM in the worst case (given by Theorem 1), but incurs more runtime overhead. This algorithm might be used in practice when the replay bound must be guaranteed in even pathological cases and its additional overhead is acceptable.

We extend our first algorithm by tracking the interval multiplicity, M, of the current interval ($S_{p,i}$) as well as its critical path. To track M, process p maintains a set of intervals we call the *replay dependence set* (*RDS*). The *RDS* is the set of all intervals that must be replayed for $S_{p,i}$ to run to completion. When p sends a message, it piggy-backs its current *RDS* onto the message (as well as the value of the variable cp, as shown in Figure 3). When p receives a message, it checks to see if not logging the message would violate either the critical path or the interval multiplicity constraints. The multiplicity constraint is checked by determining if the union of p's current *RDS* and the piggy-backed *RDS* contains more than M intervals from any one process. If the message is not logged, p's *RDS* is unioned with the piggy-backed *RDS* to indicate the total set of intervals on which $S_{p,i}$ now depends.

The disadvantage is the run-time overhead associated with piggy-backing and unioning the replay dependence sets. One optimization to reduce this overhead is to approximate the *RDS* of process p. Instead of exactly remembering every interval on which p depends, we can store only the earliest and latest intervals in each process on which p depends (and assume that every interval in between also belongs to the *RDS*). This approximation still guarantees a replay bound, and experiments show this it has no impact on the number of messages logged. This technique involves piggy-backing two vectors of size P (the earliest and latest intervals in each process) on each message. In contrast, our first algorithm only piggy-backs a single integer (the current critical path).

We conducted extensive experiments to determine whether the extra overhead of the second algorithm was worthwhile. Section 6 shows that in practice it is unnecessary to use the second algorithm and bound the interval multiplicity. We conclude that in practice the first algorithm is preferable; limiting M is not necessary to achieve a desired bound on the replay time.

6. Experimental results

To evaluate our ideas, we analyzed executions of several message-passing programs. We measured the number of messages logged and the time required to replay any interval from the logs. Our experiments show that our critical-path based algorithm logs only about $0.1 - 5\%$ of the messages and that these logs are sufficient to meet almost any desired replay time. We also found that in most cases checkpoint placement has little effect on performance. These results suggest that our critical-path based technique bounded replay time with low message logging overhead. Because it does not depend on when checkpoints are taken (for reasonable checkpoint periods), it can be integrated with other adaptive schemes, such as incremental checkpointing[5] to further reduce the total overhead of supporting incremental replay.

We simulated our algorithm from a message trace of a single execution of each program. Simulation allowed us to vary parameters and still analyze the same execution (different runs produce different executions since the programs are nondeterministic). Due to space limitations we only present the data for our critical-path based algorithm. Our second algorithm, where M is explicitly bounded, typically logs less than 2% more messages.

We analyzed six programs developed by colleagues on an Intel iPSC/860. Each program was run once with an instrumented message-passing library to collect traces for simulation. *det* computes the determinant of a matrix, and was run on a randomly generated 600x600 matrix. *mesh* computes finite differences over a grid to solve a differential equation, and was run on a 900x900 grid. *fft* performs an FFT on randomly generated data points. *tester* derives a set of input patterns that distinguish the fault-free circuit from a faulty one. *cell* is a *VLSI* cell placement program which optimally layouts circuit cells to minimize total wire length. *router* is a *VLSI* channel router program, and was run on a data file containing 131 nets. All traces (except for *router*) were obtained on a 16-node subcube (*router* was run on an 8-node subcube). The programs ran between 48 and 1358 seconds and passed a total of $7-245$ megabytes of message data.

6.1. Performance of algorithm

Our first experiment investigated what percentage of messages are typically logged when processes take checkpoints not skewed in time with respect to each other (we later discuss the effect of varying the checkpoint placement). We had each process independently take a checkpoint every T seconds by using a local real-time clock (and did not synchronize the local clocks). We measured how many messages were logged as we varied C (the critical path) and T (the checkpoint period). Since no two programs had the same execution time, for uniformity we express T as a percentage of the total execution time and C as a multiple of T.

Figure 4 shows the percentage of messages logged for four different checkpointing periods: 5%, 10%, 15% and 20% (checkpointing every 20% gives five checkpoints per process). For each value of T, the amount of logging decreases as the critical path C increases. The performance gain is small as C increases beyond $2T$ (or $3T$ for the tester program). In most cases less than 5% of the messages are logged. When the curves stabilize, we log less than $1-2\%$ of the messages. For *router*, the longest running program, less than 0.1% of messages are logged. Our algorithm's performance improves as the checkpoint period increases; with larger checkpoint periods we automatically allow larger critical paths (since C is expressed in multiples of T).

Choosing C to be about $2T$ works well. With this choice less than 5% (and in most cases less than 2%) of the messages are logged. Our algorithm logs about $15-35\%$ of the messages when $C = T$ (not shown in the figures), much more than when C is even slightly larger than T. When the checkpoint period equals T we require at least T time just to replay any interval, so it is likely that many messages will cause paths of size greater than T. This result suggests that we should avoid choosing a value of C that is close to T.

6.2. Replay response time

Our goal is to not only log few messages but also bound the replay time. Our second experiment studied how long replay requests would take given the message logs. We computed the time required to replay each interval, and averaged this time over all intervals. This average indicates how long a typical replay request will take, and shows how effective our logging strategy keeps this time bounded. This experiment is important because the critical-path based algorithm is not guaranteed in the *worst* case to provide the *CM* replay time bound given by Theorem 1, but the results show that in practice a bound of less than C is obtained.

Figure 5 shows the average replay times. We represent the replay time in multiples of C to show how near the critical path an average replay takes. The average replay time is *always* below $0.8C$. This time is smaller than C because after all the messages that violate the critical path constraint are logged, no paths of length greater than C are left, allowing most intervals be replayed in time less than C. We also found that the *maximum* time required to replay any interval was always less than $2C$.

The results of these experiments show that critical-path based logging effectively keeps the replay time bounded, even though the algorithm does not bound the interval multiplicity. In practice the user need specify only the critical path C and use the critical-path based logging algorithm (Figure 3), since even the maximum replay time is no more than $2C$. Several factors explain this result. Bounding the critical path seems to automatically bound the multiplicity of the intervals. Our experiments showed that the interval multiplicity is always less than $2C$. However, if M were $2C$, Theorem 1 states that the worst-case replay time could still be $O(C^2)$. A closer look at the proof of Theorem 1 shows that to obtain the worst-case $C * M$ bound requires all intervals assigned to any processor q to remain unfinished until the very end of the replay. Thus at any point in the replay q would have to cycle through all M intervals to perform one unit of work on each of them. However, our experiments showed that many intervals do not persist until the end of the replay. As the replay progresses, more intervals finish so q has to cycle through fewer intervals; thus the replay time becomes a geometric progression and the total replay time is proportional to C instead of C^2.

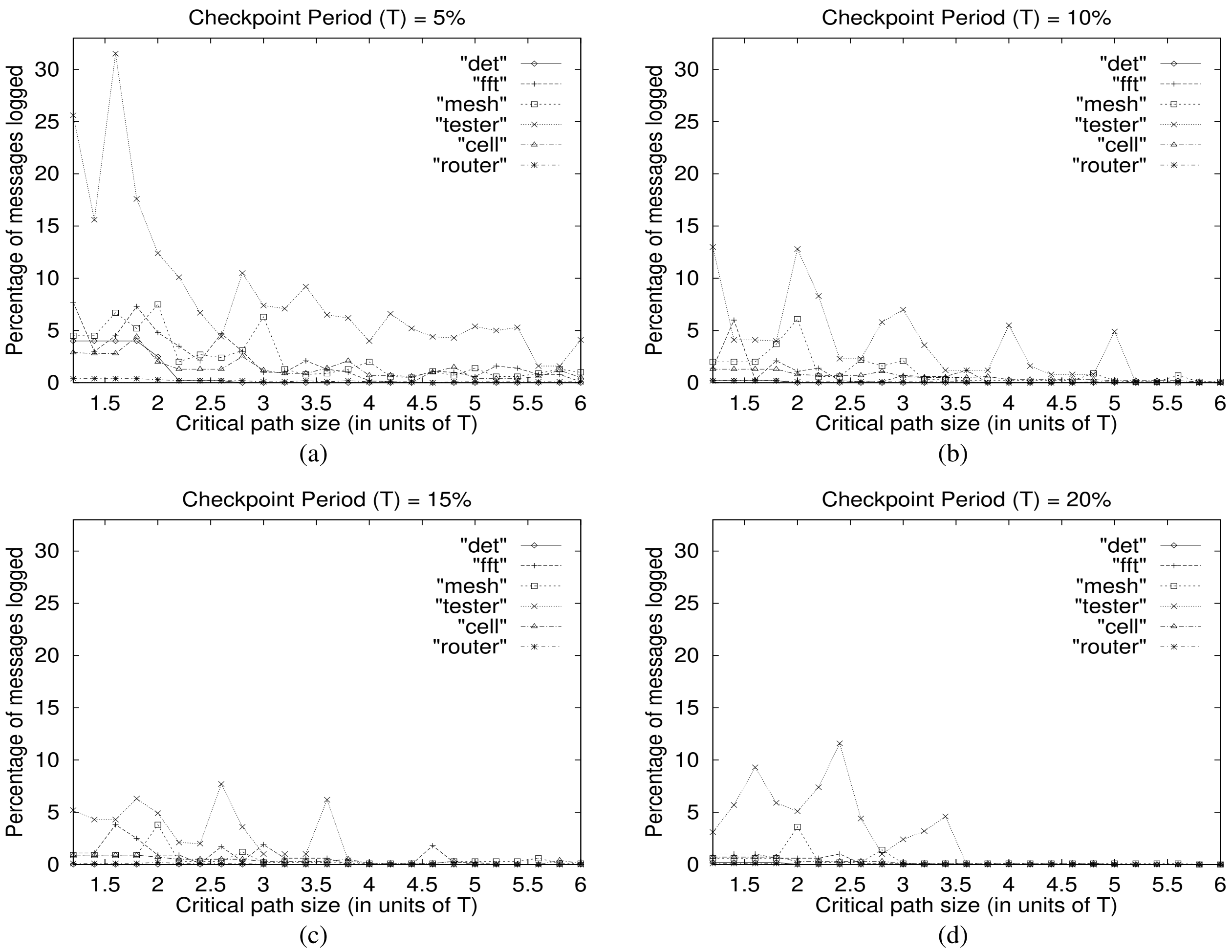


Figure 4. Percentage of messages logged for different checkpoint periods, T=5,10,15,20%
(*T* is expressed as a percentage of total execution time).

6.3. Sensitivity to checkpoint placement

In the above experiments each process checkpointed every T seconds. Since one of our goals is an algorithm that can be integrated with other adaptive strategies (such as adaptive checkpointing), our last experiment examined the effect of varying the checkpoint placement. This experiment allows us to predict the effect of different checkpointing schemes. To vary the relative placement of checkpoints in different processes, we skewed the time each process checkpoints relative to the others. We allowed the first checkpoint in each process to be taken randomly at a time between 0 and S (called the *skew parameter*). Subsequent checkpoints were taken periodically at intervals of T. Due to the random choice of the first checkpoint, the i^{th} checkpoint of any two processes experience a random relative time difference between 0 and S.

Figure 6 shows the percentage of messages logged for the tester program and the *maximum* replay time as C and S vary (here T is 10% of the execution time). These curves are typical of what we saw in the other programs and for other values of T not more than 10%. Figure 6a shows that as the checkpoints become more skewed, more messages are logged, but the trend is only slight. In general, with skewed checkpoints more messages can pass through several intervals, adding up the time for regenerating them and forcing the algorithm to log more. However, the effect of skew on logging is marginal and for most cases the increase in logging is less than 5%, indicating our algorithm is not very sensitive to checkpoint placement for T less than 10%. In addition, the skew has little affect on the replay time. Figure 6b shows that the *maximum* time required to satisfy any replay request is typically less than $2C$. This result again shows that in

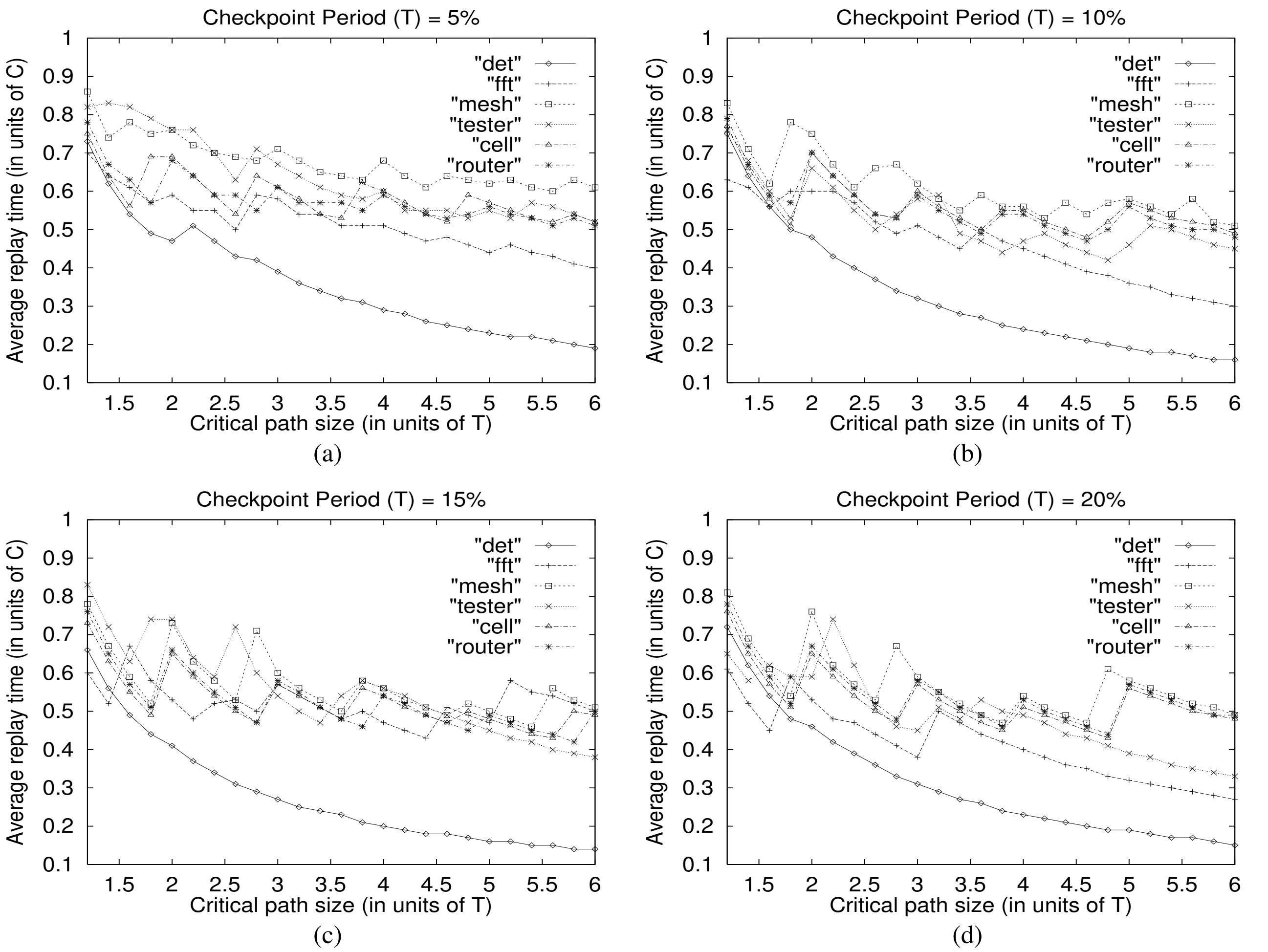


Figure 5. Average replay time (in units of C) for different checkpoint periods (T=5, 10, 15, 20%).

practice the critical path is the dominant factor in determining the replay time.

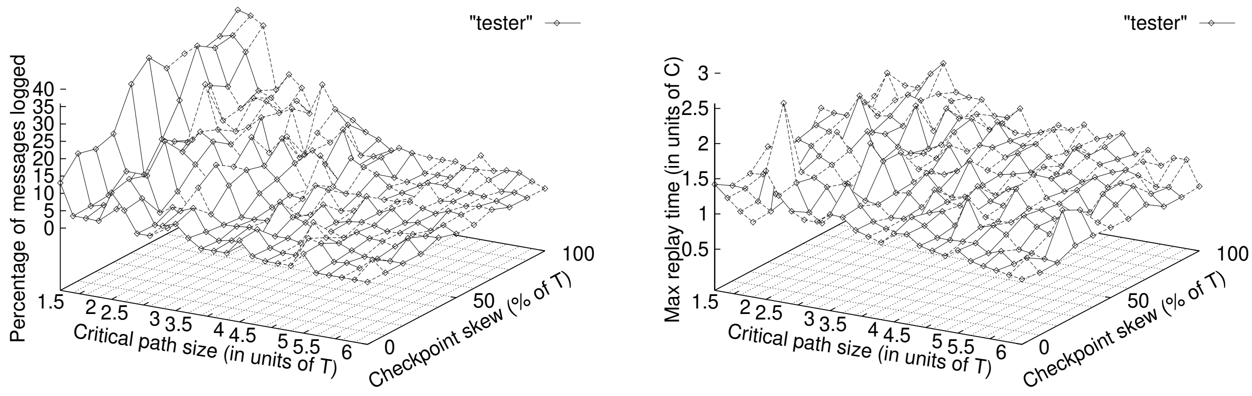
For T larger than $15-20\%$ the effect of skew was more apparent. For example, when the skew was maximum ($S = T$), 10% more messages were logged for $T = 15\%$, and 15% more for $T = 20\%$. But even for these larger values of T, the increase in logging was below 5% when S was below about 45% of T. For larger values of skew the logging could be reduced by letting the critical path size increase (to $3T$) as the skew increases. These results indicate that our algorithm can tolerate a reasonable amount of skew without much deterioration in performance.

Since incremental replay is required only for long-running programs (e.g., hours or days of execution time), we do not expect skew to have any effect in practice. For such programs, checkpoint periods above 10% would probably incur replay delays too great to be accepted by users. We there expect the above results for T not greater than 10% to be typical.

6.4. Discussion

To summarize, we discuss how to choose the checkpoint period (T) and the critical path (C) to meet a user-specified response time requirement (δ). The user should choose δ to balance checkpointing overhead with the maximum tolerable response time. Small δ provides fast replay but incurs high overhead since the checkpoint period must then be no more than δ. For example, setting δ at less than ten seconds is probably impractical on modern machines even when there is a dedicated co-processor for scanning and outputting memory pages.

The experimental results indicate that given δ, we should chose T and C as follows. According to Section 6.2, the average response time for replaying an interval is



(a) (b)

Figure 6. The effect of checkpoint placement: (a) Percentage of messages logged. (b) ***Maximum*** **replay time required to satisfy any replay request.**

less than $0.8C$, so we should choose C to be 1.25δ. According to Section 6.1, the number of messages logged stabilizes when $C = 2T$, so we should choose T to be $0.5C$, or 0.625δ. With this setting, we can expect to log less than 5% of the messages while still satisfying most replay requests within time δ.

7. Conclusions

We have presented adaptive logging schemes to provide bounded-time incremental replay. Our critical-path based logging algorithm, when used with independent checkpointing, logs $0.1 - 5\%$ of the messages while providing almost any replay bound specified by the user. In this paper, we have reported the first results on how to provide bounded replay with low message logging for independent checkpointing. We advocate independent checkpointing since other adaptive schemes can then be used to curb the total overhead. Future work involves integrating our algorithm with adaptive checkpointing to help keep the total tracing overhead low (of both messages and checkpoints).

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] 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).

[3] 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).

[4] Robert H.B. Netzer and Jian Xu, “Adaptive Message Logging for Incremental Program Replay,” *IEEE Parallel and Distributed Technology*, (November 1993). Also appears in Supercomputing ’93, Portland, OR (November 1993).

[5] Robert H.B. Netzer and Mark H. Weaver, “Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs,” *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Orlando, FL, (June 1994).

[6] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs, “Checkpoint Space Reclamation for Independent Checkpointing in Message-passing Systems,” *Tech. Rep. CRHC-92-06, University of Illinois CSL.*, (1992).

[7] 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).

[8] 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).