MPREPLAY: ARCHITECTURE SUPPORT FOR DETERMINISTIC REPLAY OF MESSAGE PASSING PROGRAMS ON MESSAGE PASSING MANY-CORE PROCESSORS

Carl Erik-Svensson, David Kesler, Rakesh Kumar and Gilles Pokam
**Title:** MPreplay: Architecture Support for Deterministic Replay of Message Passing Programs on Message Passing Many-core Processors

**Authors:** Carl Erik-Svensson, David Kesler, Rakesh Kumar and Gilles Pokam

**Abstract:**
While a lot of work has been focused on design and programming of shared memory multi-core architectures, message passing architectures are increasingly being considered an attractive design point for many-core [10] and application-specific [2] processors. A big concern with message passing architectures, however, is programmability and debuggability on such machines and the significant overhead of providing support for the same at software level. In this paper, we take a first look at providing hardware support for debugging and replay of message passing programs on message passing architectures. We propose a hardware framework for logging races between messages to allow deterministic replay of message passing programs. One implementation of the framework is based on Netzer algorithm [19] for software-based logging and uses vector timestamps. The other implementation is based on a novel algorithm that uses scalar timestamps. We show that the two implementations have small time and space overhead. We discuss similar hardware support for allowing incremental replay of message passing programs.

**Subject Terms:** Message passing, debugging, races, bugs

**Distribution Statement:** Approved for public release; distribution unlimited.
MPreplay: Architecture Support for Deterministic Replay of Message Passing Programs on Message Passing Many-core Processors

Carl Erik-Svensson & David Kesler & Rakesh Kumar
Coordinated Science Laboratory
1308 West Main St.
Urbana, Illinois 61801
cvssens2, dkesler2, rakeshk@illinois.edu

Gilles Pokam
Microprocessor Technology Labs
Intel Corporation
Santa Clara, CA 95054
gilles.a.pokam@intel.com

Abstract

While a lot of work has been focused on design and programming of shared memory multi-core architectures, message passing architectures are increasingly being considered an attractive design point for many-core [10] and application-specific [2] processors. A big concern with message passing architectures, however, is programmability and debuggability on such machines and the significant overhead of providing support for the same at software level. In this paper, we take a first look at providing hardware support for debugging and replay of message passing programs on message passing architectures. We propose a hardware framework for logging races between messages to allow deterministic replay of message passing programs. One implementation of the framework is based on Netzer algorithm [19] for software-based logging and uses vector timestamps. The other implementation is based on a novel algorithm that uses scalar timestamps. We show that the two implementations have small time and space overhead. We discuss similar hardware support for allowing incremental replay of message passing programs.

1 Introduction

While a lot of work has been focused on architecture and programming of shared memory multi-core architectures, message passing architectures are increasingly being considered an attractive design point for many-core [10] and application-specific [2] processors. The attractiveness of message passing architectures in such domains is due to following reasons. First, for some domain specific applications, message passing provides a better programming model than shared memory. For this class of applications, using a message passing architecture would be an intuitive choice. As an example, processors from Ambric [2] that are used for transcoding and video compression (e.g., at Sorenson Media and Pyro AV) are purely message passing architectures. This is in large part due to the ease of expressing these media applications using a message passing programming model. Another reason for the attractiveness of message passing architectures is their scalability. As Moore’s Law scaling continues to hold, the number of cores on a single chip is expected to double every 18 months, on average. With shared memory architectures, the latency and power cost of maintaining hardware-based cache coherence increases superlinearly with number of cores [13]. Message-passing architectures do not have to support hardware-based cache coherence and, therefore, tend to be scalable. As an example, Intel’s Terascale prototype processor which features more than 80-cores on a single chip [12] is a message passing processor. The 64-core tiled processor from Tilera [10] has integrated hardware support for message passing. Similarly, the 256-core multi-core processor from Ambric [2] is an explicit message passing processor.

The enthusiasm for message passing processors is often dampened, however, due to concerns about programmability and debuggability on such machines. Debugging message passing programs can often be hard as they can often have many subtle/difficult-to-detect bugs. As an example, as pointed out in [9], the MPI standard includes 14 send calls and five receive calls that can be combined arbitrarily for a total of 70 ways to implement a single point-to-point communication. This can often cause bugs. Similarly, distributed processing in message passing execution can obscure the location of errors. Also, since several message passing libraries, like MPI, are not compiled languages, existing tools often do not perform static checks on messaging usage beyond correct use of prototypes. Finally, non-deterministic errors, can be common as changing to a different multi-core chip or messaging implementation (or changing problem size) can make potential or latent deadlocks and
race conditions appear. In fact, [9] points out that the most common bugs in message passing programs are indeed non-deterministic errors.

One approach that has been explored to detect non-deterministic bugs is deterministic replay debugging. Deterministic replay debugging (DRD) [19] provides the ability to replay the exact same sequence of instructions that led to the manifestation of a bug. Therefore, a requirement for DRD is to log the sources of non-determinism during a program execution in order to replay the program in the exact same order. In a message-passing architecture, a requirement for deterministic replay debugging, therefore, is to record the non-deterministic messages, i.e. all incoming messages whose arrival order is not guaranteed. Racing communication can be recorded during runtime so that the application could be replayed, synchronizing at all the same points that were recorded.

Previous work has proposed tracking racing messages in software, however software-based logging represents a major source of overhead. This overhead can account for as much as 20% of an application’s execution as indicated in [19] and can, therefore, be prohibitive in many situations. In this paper, we take a first look at providing hardware support for debugging and replay of message passing programs on message passing architectures. We make the following two contributions.

- The first contribution of this paper is a hardware logging framework (which we call MPreplay) that records only a subset of the racing messages that are needed for deterministically replaying a message passing program. The proposed framework hinges on keeping track of a timestamp and maintaining a per-processor local counter of the number of executed synchronization operations. We present two implementations of the framework - one that is based on the Netzer algorithm [19] for software-based logging and uses vector timestamps. The second implementation is based on a novel algorithm based on scalar timestamps. With only modest hardware changes and logic to determine when to log a racing message, we show that our solutions provide significant performance improvement, with only 0.8%-4.9% overhead on 4-256 cores as compared to 20% overhead reported by prior software tools.

- Our second contribution is a hardware framework that builds on checkpointing mechanism for providing incremental replay to allow more debugging flexibility. Again, implementations are presented using scalar and vector timestamps.

The rest of this paper is organized as follows. In Section 2, we relate our work to previous research. In Section 3, we define races and outline the general approach to race detection. Our hardware framework for enabling deterministic replay of message passing programs is described in Section 4. Section 5 discusses a scalar timestamp-based implementation of the framework. Section 6 builds on MPreplay and describes more advanced mechanisms for enabling incremental replay of message passing programs. Section 7 describes our methodology, while Section 8 analyzes and presents our results. Finally, we conclude in Section 9.

2 Related Work

Although there exists significant amount of work on hardware support for enabling deterministic replay, none of this work really applies to message passing systems. Almost all recent work in this direction has focused on shared memory programs. The Flight Data Recorder (FDR) is a hardware-based full-system recorder for shared memory programs. FDR builds on a hardware implementation of the transitive reduction optimization of Netzer [19] to eliminate shared memory dependences that are implied by others and piggybacks on cache coherence messages to determine what memory races need to be logged at runtime [24]. Instead of providing a full-system replay, BugNet [18] concentrates on replaying the user code and shared libraries only. BugNet does not log information about the entire system state, but just information regarding the machine registers state and memory races in an application, thus reducing the cost of replay. Recent hardware proposals for recording memory races [17, 11, 16] improve on FDR and BugNet by logging fewer shared memory states using a coarser granularity than individual shared memory references. With Strata [17], memory races are logged simultaneously on all processors after a RAW or WAW memory dependency has occurred. ReRun [11] uses the same mechanism as in Strata to trigger logging of shared memory states, but it does so by recording an episode instead of a strata. An episode in ReRun is a per-thread series of dynamic instructions that can execute without encountering a conflict. A similar approach was explored in DeLorean [16] assuming a chunk-based execution [6].

This paper focuses on providing hardware support for enabling deterministic replay debugging of message passing programs. Hence, none of these prior proposals for shared memory programs apply directly to our work. The reason for this is that the definition of a race, which is central to any hardware replay debugging scheme, is different in a shared memory system than in a message passing system. While a race in a shared memory system can be defined by means of an ordering relation among the memory access interleavings, in a message passing system, a race is described by the ordering of message interleavings at a receive operation.
Netzer [20] is the first to define the notion of a race for message passing programs in the context of debugging. In his seminal work, Netzer proposed an optimal online tracing algorithm in software to replay message passing programs. During a receive operation, the mechanism proposed in Netzer's work capitalizes on the definition of a racing message to log only a subset of the total messages for replay. Prior to that work, software-based replay schemes for message passing programs such as [15] have attempted to log every message, but have reported significant time and space overhead. While the work of Netzer [20] significantly reduces the overhead of prior software-based replay schemes for message passing programs, the remaining overhead due to online tracing is still very high, comprising almost 20% of the program execution time. More recent studies such as [4, 7, 22] have striven to provide support for deterministic replay debugging of message passing programs. The approach taken in these studies consists of logging all communications from non-deterministic synchronization events such as wildcard receives (e.g. MPI_ANY_SOURCE, MPI_ANY_TAG, etc), waits, tests, probes, etc. While useful, these studies are very challenging to port to new systems or runtime libraries as they require instrumenting or modifying all non-deterministic synchronization events. In addition, they do not help solve the space overhead problem of prior proposals. Our work builds on the optimal tracing algorithm of Netzer [20] and proposes the first hardware support for enabling deterministic replay debugging of message passing programs. Our logging implementations include a vector timestamp-based approach that adapts the Netzer algorithm for hardware. We also have a scalar timestamp-based implementation that is based on a novel algorithm.

3 Recording Racing Messages

A necessary condition for replaying a parallel message passing program is to make sure that all sources of non-determinism, i.e. racing messages, are captured appropriately during program execution. This means that we need to address at least two issues. First, we need to define a race precisely for message passing programs. Second, we need to determine how these racing messages can be detected and how many of these races we need to log in order to deterministically replay a message passing program. This section addresses these issues.

3.1 Definition of a race

A race is defined for a message passing program in terms of the way incoming send messages are ordered at a receive operation. Intuitively, if two or more messages are in transit to the same receiving process and it is not possible to determine which one of them will be received first, then we have a race. This non-determinism can be due to reasons such as variations in network latency or sensitivity to scheduler decisions.

![Figure 1: (a) Example of a race condition. (b) Example of no race condition.](image)

We illustrate this with an example (Figure 1(a)). The figure shows three communicating processes, $P_0$, $P_1$ and $P_2$. Two messages, $m_{01}$ and $m_{21}$, originating from $P_0$ and $P_2$, respectively, are sent to $P_1$. Process $P_1$ has issued two untagged receive operations to handle these two messages without assuming a particular order (e.g. using MPI_ANY_SOURCE receives of the MPI library). If the programmer’s intention was to have $P_1$ receive $P_0$’s message first, then message $m_{21}$ should have been received before $m_{01}$. However, because of variations in network latencies, for instance, it is possible for $m_{01}$ to be received by $P_1$ first, as shown in our example in Figure 1(a), due to unwanted non-determinism during execution. For deterministic replay debugging, the order in which the messages are received at $P_1$, i.e. $m_{01}$ followed by $m_{21}$, should therefore be correctly recorded during program execution, since these two messages race.

3.2 Detecting racing messages

Now that we know how to define a race, the next issue we need to address is figuring out how to detect these racing messages at runtime by using as small amount of information as possible. The key for detecting whether a message is potentially involved in a race with another message is to check for a race condition at the moment of a receive operation. Let us consider again the example shown in Figure 1(a). Recall that, in that figure, the programmer’s intention is to receive $m_{21}$ first, before message $m_{01}$ can be processed by $P_1$. Since the receive order of these two messages is inverted in the figure, this is actually an unintended message race that could potentially lead to a bug. In order to detect this race, when the receive operation $recv_i^{(1)}$ in $P_1$ is processed, we should check if there exists a previous receive operation in the same process whose message could potentially race with the current one. In Figure 1(a), when $recv_i^{(1)}$ executes in $P_1$, we note that message $m_{01}$ received...
by the previous receive operation $recv_0^{(1)}$ on the same process races with the current message $m_{21}$ because the previous receive could also have been a potential recipient of the current message. The race between the two messages would not have existed had we considered the example shown in Figure 1(b). In this example, the previous message $m_{01}$ received earlier by $recv_1^{(1)}$ is ordered by a happened before relation with respect to the current message $m_{21}$ received by $recv_1^{(3)}$. This happened before relation is implied by the fact that message $m_{12}$ orders the previous receive operation $recv_0^{(1)}$ in $P_1$ before the send operation $s_2^{(2)}$ in $P_2$; hence the messages $m_{01}$ and $m_{21}$ are ordered by transitivity and could not be involved in a race. Whenever we detect messages involved in a race, we only need to log the first racing message among them. Netzer has shown that the set of race messages recorded in this manner is usually optimal. We refer the interested reader to [20] for a proof of this algorithm.

4 Hardware Support for Logging Racing Messages

In this section, we first describe a baseline processor architecture that will be used for our studies. Then we describe the Netzer's vector timestamp-based algorithm for logging racing messages. We then propose a framework that efficiently implements this algorithm in hardware. Finally, we discuss the design of the replayer.

4.1 Baseline Processor Model

The baseline message passing architecture for this study is a tiled many-core architecture that consists of multiple cores arranged in a 2-D mesh connected by a network-on-chip. Each core has a private memory inaccessible by other cores except through explicit messages. The ISA is extended to include explicit send messages that write to an output FIFO buffer (sends are considered as writes to memory-mapped IO). We choose our send instructions from the MDP instruction set [8] (e.g., SEND, SEND2, SEND2E). An explicit instruction (SEND2E) marks the end of a message. The packetization hardware (present in the network interface) waits until a SEND2E instruction is received before a message is packetized (packetization is also initiated when the FIFO output buffer that the send instructions write to is full). The message packets are then put in the output queues (one corresponding to each link) from which they are injected into the network. The packets travel to the destination where they are de-packetized into a message in a form that can be processed by the destination node. The de-packetized message is put into the input FIFO buffer (FIFOn). We do not consider DMA in this study. Figure 2 shows the baseline message passing architecture.

4.2 Vector Timestamp-based Logging Algorithm

The tracing algorithm is based on Netzer's algorithm[20] and relies on each processor having a vector of counters that comprise a time stamp. This vector has $p$ entries, where $p$ is the number of processors. Each entry, $i$, stands for the local counter of processor $i$ at the time of the most recent instruction from $i$ that happens before the current instruction in processor $j$. This number at position $i$ of $j$'s vector is updated when a receive occurs in $j$ which came from $i$. Processor $j$'s timestamp is updated in any entries where processor $i$'s timestamp has greater values. Messages are logged when a processor determines that it has received a message which could have arrived at an earlier receive by noting that the sender is unaware of any intervening sends between the two racing receives. It has been shown [20] that the optimal logging is equivalent to computing a vertex cover, an intractable problem, however experiments reveal that the optimal trace is generated most of the time. Non-optimal traces are generated when there are non-transitive races. These races can occur when receives occur on a subset of the possible nodes instead of all possible nodes.

4.2.1 An Example

Figure 3(b) shows some code that produces a race. One possible execution is depicted in Figure 3(a). P1 starts out with a local time stamp of $[7, X, X]$, indicating that its local counter reads 7 after executing a send to P2. P1 has yet to receive from P2 or P3, so the remainder of its time stamp is undefined (X). Likewise, P2 has an initial time stamp value
void process IQ {
    void processZQ { ini data * 1;
        int dote = 2; Send(dote, p2); } 
    SendCdato, p3); }
}

Figure 3: (a) Logging example with three processors. (b) Code for the three processes that generate the race seen in (a)

of [X,55,X] when it sends to P3. P3’s local counter, after receiving from P2, is incremented from 10 to 11. As dictated by the logging mechanisms, P2 sent its entire time stamp to P3 along with its message. This causes the corresponding entry in P3 to be updated to P2’s local counter value of 55. Time stamps are updated in this manner as the remaining sends and receives execute.

To illustrate an interesting iteration of the logging algorithm, consider the second receive in P2. Upon executing this receive, the logging algorithm will begin its execution. First, P2 reads the time stamp for the previous receive instruction, [7,56,X]. Each process keeps track of the time stamp for its previous receive, to enable this lookup. It then compares that to the time stamp that it is currently receiving from P3, [X,55,12]. The operation involves comparing the entry for P2 in each time stamp. If the previous receive has a greater time stamp entry than what it just received from P3, then the previous receipt of P2 is logged. Intuitively, this means that there were no messages from P2 after P2’s first receive (and before the second receive) that arrived before P3 sent the message currently being received by P2. If such a message were present, then the receives in P2 would be implicitly ordered by this message, and no logging would occur.

4.3 Logging Hardware

The above online tracing algorithm can be implemented in hardware simply by providing support for timestamping the messages at the sender, updating the logs and updating the timestamps at the destination. Figure 4 presents an overview of our architecture, MPreplay. The shaded components represent the new addition to the baseline processor model for providing hardware logging capability.

4.3.1 Timestamping the messages

Each message that is generated in the system by a node i is appended with a vector timestamp which is a vector of size P (where P is the number of processors) such that each element of the vector is a local counter corresponding to a node in the system. The local counter, similar to a local Lamport clock [14], gets incremented at every send or receive event. For example, the counter for node i is incremented when node i issues a send. For our study, we assume local counters as well as the network packets to be the size of an integer. So, packetization of a message with a vector timestamp results in P extra packets injected into the system. A message is also assumed to be appended with the sender ID. Upon receipt of a message, the receive logic of MPreplay extracts the sender ID and timestamp vector fields of the message and stores it into the Receive Buffer. Assuming a 4 cores processor system, the format of an entry in the Receive Buffer is as follows: (send.PID, < local.IC1, local.IC2, local.IC3, local.IC4,... >), where local.ICi is the local synchronization instruction counter for processor i.

4.3.2 Updating the log

Every node is augmented with a Race Log Buffer that is responsible for storing the log generated as a result of the online tracing algorithm. Every node also has a Receive Buffer that stores the local counter of the previous receive as well as information about the most recent receive’s source.

When a message is received at destination, the vector timestamp and the sender ID (or PID) are extracted (note that a vector timestamp is simply P contiguous packets each representing a counter) from the message by the MPreplay receive logic (R-Logic). The local counter of the previous receive (read from the receive buffer) is then compared against the sender’s time stamp entry for the receiving node (i.e., the corresponding counter in the timestamp vector). The comparison is a way to detect if the previous receive happened before the sending of the current message. If the
local counter for the previous receive is greater than the sender's time stamp, then there is a potential for a race, and logging must be done. Logging simply involves writing the sender ID of the previous receive (obtained from the sender ID field in the Receive Buffer) and its corresponding local instruction count (the local.IC entry at position send.ID in the Receive Buffer) to the race log buffer. An entry in the Race Log Buffer has the following format: (recv.IC, < send.PID, send.IC >). The first field in the log, recv.IC, denotes the local instruction count of the receive operation involved in the race. The next field, < send.PID, send.IC >, simply identifies the sender of the racing message. The sender is uniquely identified by its PID and the local instruction count of the send operation in that process. Note that the output of the comparator doing the comparison can simply act as write enable for the race log buffer. The Receive Buffer is then updated with the new information from the last receive. The contents of the race log buffer is periodically written to memory or when it is full.

4.3.3 Updating the timestamp vector

After the logging decision is made (and log created whenever applicable), the time stamp at the destination node j is updated with the component-wise maximum value between the time stamp in destination node and the time stamp received from the sender. Additionally, the local counter (the ith entry in the time stamp vector) is incremented in order to indicate that an event has occurred. Finally, the receive buffer is updated with the sender ID, the sender's local counter, as well as the receiver's local counter.

4.4 Replay

Replaying execution using the logs generated by the hardware is simple. During replay, messages may arrive at a process in any order, however, we must make sure that they are processed in the same order as the log dictates. To do so, we must update the local instruction counter in each process exactly the same as done during execution. Therefore, during replay, each outgoing message is appended with information related to the sender ID and the local synchronization instruction count so that they can be extracted by the replayer at the destination. At the destination, before a receive happens, the replay environment checks the Race Log Buffer to see if the next receive is one of the logged, racing receives. This is the case only if the local instruction count of the current receive is included in the Race Log Buffer. If it is, then the receive operation will only accept message from process send.PID with an instruction count of send.IC as encoded in the RLB entry for that process. Messages sent to this process that do not match the send.PID and send.IC fields of the corresponding log entry are buffered until the correct message is received.

The messages sent to a racing receive may become a problem. If the buffer size is limited, and many messages are sent to a racing receive before the logged message arrives, then there is the possibility for buffer overflow. It is possible to circumvent the use of buffers altogether by enforcing the ordering of messages with additional control messages. During replay, a control message can be placed between two racing messages, thus ordering the previously racing receives. This approach can greatly reduce the amount of parallelism achievable during replay, as additional orderings must now be enforced. Also, there is additional overhead for generating and processing the control messages.

4.5 Multiple Processes

Note that the framework described above assumes that a single process is running on a core. When the number of processes is higher than the number of cores, some extra support would be required either at the hardware level to store state corresponding to multiple processes simultaneously or at the OS level where it can be augmented with some capabilities of handling multiple software threads by saving/restoring the log states upon a context or task switch. We think this problem should be addressed at this level if we need to keep hardware cost low. Also, note that this problem is common to all replay schemes and is not specific to our approach.

5 Using Scalar Timestamps for Message Logging

The vector timestamp-based approach in the previous section can have high bandwidth and latency overhead when the number of cores is large. In this section, we discuss a scalar timestamp-based implementation of the MP-Replay framework which is expected to be more scalable.

Scalar timestamps are feasible only if each processor knows the destination of the messages it sends. As in the vector case, processors keep track of a local vector, prevRecv, which stores local counters from other processors. Processor i's entry j in prevRecv corresponds to j's local counter the last time j sent to i. An implementation of scalar timestamps would have the sender sending it's local counter, along with the previous receive from the processor it is sending to, as shown in Figure 5(a).

The logging comparison remains mostly the same as seen in Figure 5(b). The only difference is that now instead of having to index a vector to find the value from the sender to compare with myLastRecv, we have explicitly received what value to compare against (msg.prevRecvFromI). We
Figure 5: (a) Necessary updates to the send function for implementing scalar timestamps in software. Here we are sending msg to processor j. (b) Necessary updates to the receive function for implementing scalar timestamps in software. Here we are receiving msg from processor j.

Also still use the value of the sender’s local counter to update the processor’s prevRecv vector.

Since we no longer have to handle parsing a vector from the sender, the hardware complexity can be slightly reduced when dealing with scalar timestamps. Additionally, we don’t have to update every entry of prevRecv: we only need to update the one corresponding to the sender. This should further reduce the hardware complexity when compared to the vector timestamp case. Note that this technique will log at least as many messages as the vector-based approach.

5.1 Hardware

The hardware necessary for implementing scalar timestamp logging is very similar to that needed for traditional Netzer logging with vector timestamps. Each processor still needs to maintain a vector of timestamps locally, so that it can keep track of the timestamps it receives from each processor. However, two simplifications can be made to the required hardware.

Since scalar timestamps only consist of one number, the send hardware now only has to append a single integer, rather than one integer for every processor. Additionally, upon receipt of a message, a processor no longer has to worry about taking component-wise maximums with the received timestamp vector and its local timestamp vector. Instead, in the scalar case, the processor can simply replace the corresponding entry in its local timestamp vector with the received value. This will significantly reduce the area required for a hardware implementation.

5.2 Limitations

Scalar timestamps have the obvious benefit that very little additional information needs to be sent with each message in comparison with vector timestamps. This will significantly reduce network bandwidth requirements, as the number of processors increase. This benefit, however, comes with the cost of increased log size.

In particular, if we send scalar timestamps, instead of vector timestamps, along with each message, we lose the ability to fully update the receiving processor’s timestamp vector. We can only update this vector with the scalar quantity received from the sender. This means that if the sender has more recent timestamp entries for other processors, these entries will not be updated in the receiving processor. To illustrate this, consider Figures 6(a) and 6(b).

Figure 6: (a) A program’s execution with vector timestamps. (b) The same program’s execution with scalar timestamps. A value of X implies minus infinity.

Note how timestamps are updated in the leftmost diagram. Since each message comes with a full, vector timestamp, the receiving process can update its timestamp with the component-wise maximum of the two timestamps. For instance, When P3 receives a message from P2, it is able to update its entries for both P1 and P2 counters, resulting in the timestamp [10,53,102]. This is important because it allows us to send this updated timestamp to P1, which will in turn avoid having to log a message.

Now, consider sending scalar timestamps, denoted by the right of Figure 6(b). Since we only update one local counter with each message, we do not accumulate as much information as the program executes. In particular, each send updates the scalar value of its own local counter. This will give us enough information to infer some orderings, but, as Figure 6(b) shows, this is not enough information to prevent logging when ordering is transitively implied. Even though this inability to infer some transitive orderings may cause the scalar timestamp technique to log additional messages, it will not affect the correctness of the technique. The scalar technique will log at least as many messages as the vector technique, and any additional messages logged cannot be races (as the vector technique has been proven to detect all races) and thus they will occur deterministically regardless of replay intervention. At worst, this additional logging could affect performance during logging and replay.
6 Incremental Replay Hardware

Figure 7: MPreplay-I: architecture extensions to MPreplay to allow incremental replay

Given the logging hardware described in the previous section, we can deterministically replay a parallel message passing program by re-executing a program from its beginning and synchronizing at each racing message according to the information logged during execution. The resulting replay process is simple, but it is mostly impractical for one main reason. Because replay has to start from the beginning of a program, it does not allow a programmer to time travel back and forth across a program execution. In this section, we show that, with only a modest addition to MPreplay, we could provide incremental replay capability to allow replay to start anywhere in the program. We call this new architecture MPreplay-I.

6.1 Incremental Replay Algorithm

In order to time travel back and forth along a program execution, we need to create checkpoints to resume replay from any intermediate program state. In addition to that, we also need to be able to reproduce any synchronization operation that was taken after a checkpoint was created. In this paper, we assume our baseline processor model is already equipped with such a checkpointing mechanism, making our first requirement straightforward. To understand the issue with the second requirement, consider the example shown in Figure 8(a).

The figure shows two processes, P0 and P1. The replay starts at checkpoint c01 in P0, while in P1 it can potentially start at checkpoint c10 or c11. These checkpoints define two potential replay frontiers, denoted by \{c01, c10\} and \{c01, c11\} in the figure. We could not have been able to reproduce the message m10, had we resumed replay from the replay frontier \{c01, c11\}. This is because sender send10 issued message m10 before checkpoint c11 was created, while the recipient of the sent message, recv01, was issued after that checkpoint c01 was taken. If, instead, we had replayed the program from the first frontier, \{c01, c10\}, we would not have run into this problem because message m10 would have been reproduced accordingly. Alternatively, we could have avoided this problem by simply logging all incoming messages following the creation of a checkpoint. Obviously, this solution has the potential of increasing the log size significantly. Therefore, an appropriate solution is to log only the messages for which we can not guarantee reproduction during replay, as discussed in [25].

The incremental replay algorithm therefore enforces the following constraint: in order to replay a program starting from a given checkpoint, we must also replay all checkpoints belonging to the processes from which there exists a potential replay dependence with synchronization operations included in the current checkpoint. In [25], the authors define the replay dependence relation between two synchronization operations \(a\) and \(b\) by \(a\) preceding \(b\) in the same process when no checkpoint has occurred, or by the existence of a sequence of unlogged messages between \(a\) and \(b\). The authors use the expression replay set to capture the set of checkpoints that need to be replayed as a consequence of the replay dependence relation. We illustrate the operations of the algorithm with the example shown in Figure 8(b).

6.2 Replay Set Example

Figure 8(b) shows the message traffic with 4 processors. To illustrate the operations involved in building the replay set, consider the receive events (a-f) of P2. At the start of checkpoint 1 for P2 (we will refer to this interval as \(2,1\) from now on), the replay set of P2 only contains its own interval, \(2,1\). Once the receive operation \(a\) is executed, P2’s replay set is unioned with P1’s replay set: \[\{(1,1)\}\]. Likewise, after receive \(b\), P2’s replay set is unioned with P3’s replay set: \[\{(3,1)\}\].

Receive \(c\) illustrates a slightly more complex case. Note that we still union the replay set of P3 with that of P2, but
now the replay set of P3 is \([3,1] (4,1)\] because of P3's receive from P4. This is the set that is unioned with P2's replay set to result in \([(2,1) (1,1) (3,1) (4,1)\]. Also notice that the entry for \((3,1)\] appears only once. This is a set union.

Receives \(a\) and \(e\) proceed just like \(a\) and \(b\); simply unioning the replay set of the sender with that of the receiver at each receive. By the time receive \(f\) is executed, however, P2's replay set has already reached its maximum given by the bound parameter of 7. Since there are 7 elements in P2's replay set, there are no entries added to the replay set of P2, and instead, the message at receive \(f\) is logged in its entirety. Once checkpoint 2 is reached for P2, the replay set of P2 will be written out to the log and reset to \([(2,2)\].

### 6.3 MPreplay-I Hardware

The hardware components representing the new addition to MPreplay are illustrated in Figure 7. These new components essentially include additional logic at the receive node, and buffers for maintaining the Replay Set (RS) and the incoming messages (Message Log Buffer).

As a message is injected into the network, in addition to appending the timestamp and the sender ID, MPreplay-I also appends the Replay Set (RS) of the current checkpoint. As discussed in the previous section, the RS encodes the checkpoint intervals that need to be replayed in order to reproduce all replay dependent messages of a current replay interval. MPreplay-I models the RS as a FIFO queue of \(b\) checkpoint identifiers, where \(b\) denotes the bound, i.e. the number of entries in the RS. A checkpoint identifier, denoted by \(CID\), uniquely identifies a checkpoint in a program. It is composed of the sender ID (or Process ID) and a local checkpoint counter, checkpt.ID. The local checkpoint counter is initialized to zero and is incremented by 1 after each newly created checkpoint. The format of an entry in the RS is shown below.

\[
RS : CID_0, CID_1, ..., CID_{n-1}
CID : \langle send.PID, checkpt.ID \rangle
\]

The number of bits used to represent checkpt.ID will depend on the number of checkpoints that a process can create. This paper assumes a local checkpoint counter of \(m\) bits. If the counter overflows, we simply terminate recording since we may have missed some checkpoints. With \(N\) concurrent processes, a maximum message payload of \(b(m + \log(N))\) bits can be appended to each outgoing message due to sending RS information.

As messages arrive at destination, they are placed into the FIFOin buffer by the de-packetization hardware, which then automatically triggers the receive logic, R-Logic. The R-Logic then strips off each CID entry from the incoming RS message, adding it into the RS FIFO queue if not present or if there is still enough space available. When the RS queue fills up, the R-Logic instructs MPreplay-I to buffer messages directly into the Message Log Buffer (MLB) for the rest of the duration of the checkpoint interval. Note that since we can not reproduce the incoming message as a result of not being able to store the sender's \(CID\) into RS, we need to log the entire message into MLB. An entry in the MLB therefore identifies a receive operation with its matching incoming message and is represented by \(\langle recv.IC, send.Message \rangle\).

The Message Log Buffer is memory backed to allow more data to be stored than the available buffer size. This can happen whenever the MLB is full or if an incoming message that needs to be stored into the MLB is larger than the available space. In such a case, we need to occasionally write the content of the buffer to memory. Similar to Section 4.4, we assume that the content of the MLB is written to memory periodically or when full. When a new checkpoint is about to be created, the MLB and the RS also need to be memory backed. The MLB and the RS are stored alongside checkpoint data so that they can be efficiently retrieved during replay.

When a new checkpoint is created, we need to clear both the MLB and the RS to start recording information for that particular checkpoint interval. In addition, we also reset the local synchronization instruction counter so that instruction count is maintained relative to the start of a checkpoint interval. This is consistent with MPreplay-I mode since replay is done relative to the start of a checkpoint interval. As a consequence, entries in the RLB, MLB and RS can be more compactly represented since they can be encoded using fewer number of bits than MPreplay.

### 6.4 Replay

Replaying a program using the logs generated by MPreplay-I is straightforward. Given a starting checkpoint interval, \(CID\), to launch a replay execution the replay engine has to resume execution of each \(CID\) contained in the Replay Set (RS) of the current checkpoint interval. The RS is obtained easily by retrieving the data stored along the checkpoint information. The local counter in each replayed process is reset at the beginning of the execution and incremented each time a send or receive operation is encountered. As local receive operations are executed during replay, the replay engine has to make three decisions. If the receive operation to be executed is contained in the Race Log Buffer (RLB), then we have identified a racing message. In this case, the replay engine proceeds in a manner similar to the description in Section 4.4. If the receive operation is contained in the MLB instead, then the replay engine knows that the corresponding message must be consumed from the MLB log. The corresponding message is extracted from the log and execution proceeds to next instruction. However, if
none of the above applies, then the replay engine knows that the corresponding message must arrive from the network. This case corresponds to a sender reproducing a message awaited by a receiver. No synchronization needs be performed by the replay engine in this case since the message is not a race.

7 Evaluation Methodology

This paper explores the hardware support needed to allow deterministic full and incremental replay for message passing programs on message passing architectures. For our studies, we consider 4, 8, 16, 32, 64, 128, and 256 core many-core architectures. The cores are assumed to be connected using a 2-D mesh with 32-bit links clocked at 2GHz. Link latency is assumed to be 5 clock cycles for all our experiments. Cores themselves are assumed to be clocked at 2GHz and implement the Alpha ISA.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>2GHz</td>
<td>LI Cache</td>
<td>32KB, 2-way, 1 cycle</td>
</tr>
<tr>
<td>LI Cache</td>
<td>64KB, 2-way, 1 cycle</td>
<td>Private L2</td>
<td>4MB/CORE, 6 way, 10 cycle</td>
</tr>
<tr>
<td>Execution</td>
<td>In-order</td>
<td>Number of Processors</td>
<td>N</td>
</tr>
<tr>
<td>Checkpoint Frequency</td>
<td>50ns</td>
<td>Timestamp</td>
<td>N words</td>
</tr>
<tr>
<td>Receive Buffer</td>
<td>N+1 words</td>
<td>Race Log Buffer</td>
<td>32KB</td>
</tr>
<tr>
<td>Message Log Buffer</td>
<td>8KB</td>
<td>Replay Set</td>
<td>8 to 128 words</td>
</tr>
</tbody>
</table>

Table 1: Individual core specifications

The message passing instructions were modeled after MDP [8], specifically the send instructions and the sequence of instructions used to do a receive. The cores are assumed to have 256-entry input queues where each entry is 4 bytes wide. The packets are 4 bytes long. For our experiments, we modified M5 [5] to model the various many-core architectures.

Table 2 shows the benchmarks that we used for our evaluations. While we used a standard sequential implementation of these benchmarks as starting points, we wrote the message passing version of these benchmarks ourselves. Our message passing implementations were done such that minimal amount of change was made to the original algorithm. A component-oriented programming model [23] was used for our implementations where each software component shares no state with other components. All communication is through explicit messages. Message passing was done through MPI-like extensions. We wrote our own C++-based MPI-like message passing library because of the difficulty of porting the original MPI library onto the simulator in non-system-level simulation mode. Our message passing library compiles into C++ Alpha binaries (with messaging extensions that were inserted using asm). Each software component is compiled into its own binary. Each binary is mapped to a separate core during execution. We used an Alpha cross-compiler with -O3 flag turned on for compilation.

Stress and Cycle are synthetic benchmarks which require further explanation. In Stress, when the application begins, each process enters a cycle in which it checks to see if a new message has arrived, and if it has, it receives it. Regardless of whether a message was received or not, the process chooses a random core and sends a 1 word message. After sending a set number of messages, the process signals that it is done and loops continuously receiving messages. When all processes signal they are done, the application terminates. This benchmark is completely network bound and thus acts as a stress test of the network.

Cycle is similar, only processes will only send a message after receiving a message. The system starts out with a single message which is passed from core to core. Since there is only one message active in the system at a time and a process sends after each receive, there are no races due to this message. This benchmark was designed to feature transitive relationships between sends and subsequent receives so as to highlight the tradeoffs between the Scalar and Vector implementations.

Software-based logging was implemented using extensions to our message passing library, so the overheads of implementing the online tracing algorithm (mentioned in Section 4.4) completely in software are carefully modeled. Hardware-based logging was implemented by modeling the log buffers, the receive buffer, and other hardware structures described in Section 4.3.
8 Analysis and Results

Figures 9(a) and 9(b) show the effects of logging overhead on the speedup of our benchmarks. Each figure shows the effects of software and hardware overheads for both vector and scalar timestamps, as well as a baseline speedup for comparison. In terms of performance, being closest to the "No Logging" trend is best. If a logging technique has similar speedup to the no logging baseline, then the overhead of that logging technique is very small.

There are several things to note in the graphs in Figure 9(a). First, the absolute overhead of providing support for deterministic replay depends on the benchmark. This is not surprising considering that our tracing algorithm is invoked only on a message receive. The fewer the number of messages received in a program, the smaller the overhead of providing system support for replay. It is clear, especially as parallelism increases, that the hardware approaches to logging, whether vector or scalar, produce a greater speedup than their software counterparts. This indicates that the hardware logging techniques have less overhead. In fact, discounting the synthetic benchmark Cycle, vector timestamp logging in hardware is never any worse than 90% of the speedup in the no logging case. Scalar timestamp logging in hardware performs even better: it is never worse than 99% of the speedup in the no logging case except for Cycle.

Incremental logging shows the same trend. This is not surprising, since incremental logging requires some form of message racing logging in order to enable deterministic replay. We now simply have some additional overhead for keeping track of replay sets. In our experiments, these overheads did not significantly contribute to the trends, and we can see that incremental logging produces roughly the same speedups as logging racing messages alone.

For the two synthetic benchmarks Stress and Cycle, Figure 9(a) shows the running time (not speedup) of the various logging techniques. These benchmarks are meant to be illustrative of worst-case scenarios for vector and scalar timestamps. In Cycle, we can see that initially VectorHW outperforms ScalarHW, since VectorSW is able to deduce those transitive orderings and therefore logs far fewer messages. However, at 128 and 256 cores, the trend reverses as the network overhead of sending vector timestamps outweighs the additional logging overhead that ScalarSW incurs. It is interesting to note that ScalarHW always outperforms VectorHW in our experiments. This indicates that the advantage of a hardware implementation minimizing the overhead of logging messages leads to bandwidth becoming the next bottleneck.

In Stress, we can clearly see that scalar timestamps outperform vector timestamps at every point. This benchmark is meant to illustrate the effects of the network overhead as the number of processors increases. Here both scalar and vector timestamps will result in similar log sizes, however vector timestamps will be using far more network bandwidth to send its timestamp data. The effect of this becomes clearest at 128 and 256 cores.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>Logged Messages</th>
<th>Log Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darts</td>
<td>1048576</td>
<td>3-255</td>
<td>36-3060</td>
</tr>
<tr>
<td>Grep</td>
<td>64000</td>
<td>13-1022</td>
<td>156-12264</td>
</tr>
<tr>
<td>Energy</td>
<td>1024</td>
<td>15-1023</td>
<td>180-12276</td>
</tr>
<tr>
<td>Parser</td>
<td>512</td>
<td>7-511</td>
<td>84-6132</td>
</tr>
<tr>
<td>Sort</td>
<td>65536</td>
<td>7-511</td>
<td>84-6132</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>128x128</td>
<td>11-707</td>
<td>132-9204</td>
</tr>
<tr>
<td>Jacobi</td>
<td>512x512</td>
<td>63-5355</td>
<td>756-64290</td>
</tr>
<tr>
<td>Stress</td>
<td>4096</td>
<td>4074-4044</td>
<td>4888-48528</td>
</tr>
</tbody>
</table>

Table 3: Size of the Log Generated while Providing Support for Deterministic Replay on 4-256 Cores

For most of the benchmarks shown here do not contain transitive relationships between threads, and thus there is no difference in log sizes between the Scalar and Vector implementations. Additionally, the communication pattern of most of the benchmarks is data-size independent, so those benchmarks see a simple linear relationship between number of cores and number of messages logged. In each of those cases, the number of logged messages varies from 1-4x the number of cores, where each logged message consists of 3 integers. Darts, Grep, Energy, Parser, Sort, and MatrixMult all follow this pattern, with log sizes varying from 4 messages to 1024 messages for a maximum necessary storage of 12 KB regardless of the input size. A summary of log sizes is shown in Table 3. Jacobi, like most other benchmarks is data size dependent, but does not feature a linear relationship between number of cores and number of messages logged because the communication pattern of a 2D block Jacobi algorithm is more complicated than...
the previous benchmarks, however it still sees no difference between Scalar and Vector implementations. Because virtually every message in Stress is a race, both the Vector and Scalar implementations log nearly the same amount of messages, differing by less than 1% at 256 cores. Cycle is the only benchmark which shows an interesting difference between the Scalar and Vector Implementations. In Cycle, the only true races are artifacts of the messages sent when starting and ending the application because every receive must transitively follow the receiving core’s previous send. Thus the Vector implementation logs only a number of messages equal to the number of cores. The Scalar algorithm, however, is facing a somewhat pathological case as it is incapable of detecting these transitive relationships. Thus it ends up logging a significant fraction of the total messages. Figure 10 shows the difference between the Scalar and Vector implementations for 8192 total messages on a varying number of cores.

9 Summary and Conclusions

The emphasis toward parallel programming is likely to break the sequential nature of software programming by introducing a lot of non-determinism. For this reason, software productivity tools for improving a programmer’s productivity are gaining a lot of traction in academia as well as in the industry. There already exists a significant amount of research devoted to shared memory architectures in order to achieve this goal. Recently, a lot of emphasis has been placed on providing hardware support for deterministic replay debugging of shared memory programs on shared memory architectures. Our work advances the state of the art of hardware debugging for message passing programs on message passing architectures. We anticipate that with the pace at which CMP architectures are evolving, with Moore’s Law helping, we will soon hit a critical limit in terms of core counts on a chip, making the message passing paradigm an attractive alternative for many class of applications. We took a first look at hardware support for providing a software production stack with deterministic replay debugging capability for message passing programs. To the best of our knowledge, this is the first work that investigates hardware support for debugging message passing programs on message passing many-core architectures. We described MPreplay, an architecture support that builds on happened before relations among messages to log only a subset of the messages involved in race. MPreplay enables high performance software tools for deterministic replay debugging, achieving an overhead of less than 5% of program execution time on 256 cores. We also described MPreplay-I, an architecture that builds on MPreplay to provide incremental replay capability in software, enabling a programmer to replay any part of a program. Our results showed that log sizes of about 32KB and 8KB are enough for capturing races and messages for enabling deterministic replay debugging and incremental replay, respectively. This is similar to the hardware complexity of prior proposals for shared memory architectures. Both the low performance overhead of hardware recording and the small log size demonstrate that an MPreplay-like architecture support for debugging message passing programs is a viable option.

References

[12] Intel Corp. Intel’s Teraflops Research Chip.


Figure 9: Speedup results for all benchmarks